Burkhard Höfling

CRISP

Version 1.4.3

Computing with Radicals, Injectors Schunck classes and Projectors of finite soluble groups

Contents

Copyright

1 Introduction

The GAP package CRISP provides algorithms for computing subgroups of finite soluble groups related to a group class $\mathscr{C}$. In particular, it allows to compute $\mathscr{C}$-radicals and $\mathscr{C}$-injectors for Fitting classes (and Fitting sets) $\mathscr{C}$, $\mathscr{C}$-residuals for formations $\mathscr{C}$, and $\mathscr{C}$-projectors for Schunck classes $\mathscr{C}$. In order to carry out these computations, the group class $\mathscr{C}$ must be represented by an algorithm which can decide membership in the group class. Moreover, additional information about the class can be supplied to speed up computations, sometimes considerably. This information may consist of other classes (such as the characteristic of the class), or of additional algorithms, for instance for the computation of residuals and local residuals, radicals, or for testing membership in related classes (such as the basis or boundary of a Schunck class).

Moreover, the present package contains algorithms for the computation of normal subgroups belonging to a given group class, including an improved method to compute the set of all normal subgroups of a finite soluble group, and methods to compute the socle and p-socles of a finite soluble group, as well as the abelian socle of any finite group. CRISP also provides basic support for classes (in the set theoretical sense). The algorithms used are described in [Höf01].

$\mathscr{C}$-projectors and $\mathscr{C}$-injectors of finite soluble groups arise as generalisations of Sylow and Hall subgroups, and have attracted considerable interest. They were first studied by Gaschütz [Gas63], Schunck [Sch67], and Fischer, Gaschütz and Hartley [FGH67]. In particular, $\mathscr{C}$-injectors only exist in any finite soluble group if the group class $\mathscr{C}$ is a Fitting class. Similarly, $\mathscr{C}$-projectors exist in any finite group G if and only if $\mathscr{C}$ is a Schunck class. An extensive account of the subject can be found in [DH92].

In the case when the class $\mathscr{C}$ in question is a local formation (which is a special kind of Schunck class), algorithms for dealing with $\mathscr{C}$-projectors and related subgroups of finite soluble groups are available also in the GAP package FORMAT by Eick and Wright; see also [EW02]. In order to use their methods, $\mathscr{C}$ has to be described in terms of algorithms for the computation of residuals with respect to an integrated local function for $\mathscr{C}$.

The author would like to thank J. Neubüser and the Lehrstuhl D für Mathematik, RWTH Aachen, for an invitation, which made it possible to develop a first version of the algorithm for the computation of projectors. He is indebted to the GAP team, particularly Bettina Eick and Alexander Hulpke, for its advice, and to the anonymous referee, J. Neubüser, and C. R. B. Wright for their detailed comments on previous versions of CRISP.

2 Set theoretical classes

In CRISP, a class (in the set-theoretical sense) is usually represented by an algorithm which decides membership in that class. Wherever this makes sense, sets (see GAP Reference Manual, 30.3.7) may also be used as classes.

2.1 Creating set theoretical classes

1 ▶ `IsClass(`*C*`)` C

returns true if *C* is a class object. The category of class objects is a subcategory of the category `IsListOrCollection`.

2 ▶ `Class(`*rec*`)` O
▶ `Class(`*func*`)` O

returns a class *C*. In the first form, *rec* must be a record having a component `\in` and an optional component `name`. The values of these components, if present, are bound to the attributes `MemberFunction` and `Name` (see GAP Reference Manual, 12.8.2) of the class created. The value bound to `\in` must be a function *func* which returns `true` if a GAP object belongs to *C*, and `false` otherwise; cf. 2.2.2 below. The second form is equivalent to `Class(rec(\in :=` *func*`))`. It is the user's responsibility to ensure that *func* returns the same result for different GAP objects representing the same mathematical object (or element, in the GAP sense; see GAP Reference Manual, 12 in the GAP reference manual).

```
gap> FermatPrimes := Class(p -> IsPrime(p) and p = 2^LogInt(p, 2) + 1);
Class(in:=function( p ) ... end)
```

3 ▶ `View(`*class*`)`

If the class does not have a name, this produces a brief description of the information defining *class* which has been supplied by the user. If the class has a name, only its name will be printed.

```
gap> View(FermatPrimes);
Class(in:=function( p ) ... end)
```

4 ▶ `Print(`*class*`)`

`Print` behaves very similarly to `View`, except that the defining information is being printed in a more explicit way if possible.

```
gap> Print(FermatPrimes);
Class(rec( in = function( p )
    return IsPrime( p ) and p = 2 ^ LogInt( p, 2 ) + 1;
end))
```

5 ▶ `Display(`*class*`)`

For classes, `Display` works exactly as `Print`.

6 ▶ *obj* `in` *class*

returns true or false, depending upon whether *obj* belongs to *class* or not. If *obj* can store attributes, the outcome of the membership test is stored in an attribute `ComputedIsMembers` of *obj*.

7 ► *C1* = *C2*

Since it is not possible to compare classes given by membership algorithms, two classes are equal in GAP if and only if they are the same GAP object (see GAP Reference Manual, 12.5.1 in the GAP reference manual).

8 ► *C1* < *C2*

The operation < for classes has no mathematical meaning; it only exists so that one can form sorted lists of classes.

2.2 Properties of classes

1 ► `IsEmpty(`*C*`)` P

This property may be set to `true` or `false` if the class *C* is empty resp. not empty.

2 ► `MemberFunction(`*C*`)` A

This attribute, if bound, stores a function with one argument, *obj*, which decides if *obj* belongs to *C* or not, and returns `true` and `false` accordingly. If present, this function is called by the default method for `\in`. `MemberFunction` is part of the definition of *C* and should not be called directly by the user.

2.3 Lattice operations for classes

1 ► `Complement(`*C*`)` O

returns the unary complement of the class *C*, that is, the class consisting of all objects not in *C*. *C* may also be a set.

```
gap> cmpl := Complement([1,2]);
Complement([ 1, 2 ])
gap> Complement(cmpl);
[ 1, 2 ]
```

2 ► `Intersection(`*list*`)` F
► `Intersection(`*C1*`,` *C2*`, ...)` F

returns the intersection of the groups in *list*, resp. of the classes *C1*, *C2*, ... If one of the classes is a list with fewer than `INTERSECTION_LIMIT` elements, then the result will be a sublist of that list. By default, `INTERSECTION_LIMIT` is 1000.

```
gap> Intersection(Class(IsPrimeInt), [1..10]);
[ 2, 3, 5, 7 ]
gap> Intersection(Class(IsPrimeInt), Class(n -> n = 2^LogInt(n+1, 2) - 1));
Intersection([ Class(in:=function( N ) ... end),
  Class(in:=function( n ) ... end) ])
```

3 ► `Union(`*C*`,` *D*`)` F

returns the union of *C* and *D*.

```
gap> Union(Class(n -> n mod 2 = 0), Class(n -> n mod 3 = 0));
Union([ Class(in:=function( n ) ... end), Class(in:=function( n ) ... end)
 ])
```

4 ► `Difference(`*C*`,` *D*`)` O

returns the difference of *C* and *D*. If *C* is a list, then the result will be a sublist of *C*.

```
gap> Difference(Class(IsPrimePowerInt), Class(IsPrimeInt));
Intersection([ Class(in:=function( n ) ... end),
  Complement(Class(in:=function( N ) ... end)) ])
gap> Difference([1..10], Class(IsPrimeInt));
[ 1, 4, 6, 8, 9, 10 ]
```

3 Generic group classes

In this chapter, we describe how group classes can be defined by assigning basic attributes and properties, in particular closure properties.

A class (see 2) is a **group class** if it consists of groups, and if it is closed under isomorphisms of groups. Thus if G and H are isomorphic groups, then G is in the group class *grpclass* if and only if H is. Groups belonging to the same group class may be regarded as sharing a group theoretical property (a property shared by isomorphic groups), and groups sharing a group theoretical property form a class of groups. It not empty, group classes are genuinely infinite objects, so GAP sets can never form group classes. Some authors require every group class to contain the trivial groups. Here we do not make this assumption; in particular every empty class is a group class.

The following sections describe how to create group classes and declare some of their basic properties.

Note that, for common types of group classes, there are additional functions available to accomplish this task; see the following Chapters 4 and 5. There are also a number of pre-defined group classes; see Chapter 6.

3.1 Creating group classes

Group classes can either be defined by a function deciding membership, or alternatively by a (finite) list of groups containing at least one representative of each isomorphism type of groups contained in the class.

1 ► `GroupClass(`*rec*`)` O
► `GroupClass(`*func*`)` O
► `GroupClass(`*group-list*`)` O
► `GroupClass(`*group-list*`,` *iso-func*`)` O

The function `GroupClass` returns a new group class *class*, specified by its argument(s).

In the first form, *rec* must be a record which has a component `\in`, and may have further components `name`, and `char`. `\in` must be a function having one argument. When called with a group G as its argument, it must return `true` or `false`, depending upon whether G is in *class* or not. It is the user's responsibility to ensure that the function supplied returns the same value when called with isomorphic groups. If *rec* has components `name` or `char`, their values are stored in the attributes `Name` (see GAP Reference Manual, 12.8.2) and `Characteristic` (see 3.4.1) of *class*.

`GroupClass(`*func*`)` is a shorthand for `GroupClass(rec(\in := func))`.

In the other cases, `GroupClass` returns the group class consisting of the isomorphism classes of the groups in the list *group-list*. If *iso-func* is given, *iso-func* is used to check whether a given group G is isomorphic with one of the groups in the defining list. *iso-func* must have two arguments, and must return true if two groups, one of which is in *group-list*, passed as arguments are isomorphic, and false otherwise. If *iso-func* is not given, the GAP function `IsomorphismGroups` is used for the isomorphism test. Note that even for relatively small groups, `IsomorphismGroups` tends to be very slow.

```
gap> GroupClass(IsNilpotent);
GroupClass(in:=<Operation "IsNilpotent">)
gap> GroupClass([CyclicGroup(2), CyclicGroup(3)]);
GroupClass([ <pc group of size 2 with 1 generators>,
  <pc group of size 3 with 1 generators> ])
gap> AbelianIsomorphismTest := function(A,B)
>     if IsAbelian(A) then
>         if IsAbelian(B) then
>             return AbelianInvariants(A) = AbelianInvariants(B);
>         else
>             return false;
>         fi;
>     elif IsAbelian(B) then
>         return false;
>     else # this will not happen if called from GroupClass
>         Error("At least one of the groups <A> and <B> must be abelian");
>     fi;
> end;
function( A, B ) ... end
gap> cl := GroupClass([AbelianGroup([2,2]), AbelianGroup([3,5])],
> AbelianIsomorphismTest);
GroupClass([ <pc group of size 4 with 2 generators>,
  <pc group of size 15 with 2 generators> ], function( A, B ) ... end)
gap> Group((1,2),(3,4)) in cl;
true
```

2 ▶ `Intersection(`*list*`)` F

▶ `Intersection(`*C_1*`,` *C_2*`, ...,` *C_n*`)` F

The intersection of a list *list* of group classes resp. of the group classes *C_1*, *C_2*, ..., *C_n* is again a group class. The intersection automatically has those closure properties (see 3.2) which all of the intersected classes have.

3.2 Properties of group classes

Since nonempty group classes are infinite, CRISP cannot, in general, decide whether a group class has a certain property. Therefore the user is required to set the appropriate properties and attributes. See Sections GAP Reference Manual, 13.5 and GAP Reference Manual, 13.7. To facilitate this task, there are special functions available to create common types of group classes such as formations (see 4.4), Fitting classes (see 5.1), and Schunck classes (see 4.1).

However, CRISP knows about the implications between the closure properties listed below; for instance it knows that a group class which has `IsResiduallyClosed` also has `IsDirectProductClosed`, and that a class having `IsSchunckClass` also has `IsDirectProductClosed` and `IsSaturated`. Moreover, the intersection of group classes all having one of the closure properties in common also has that closure property.

The following basic properties are defined for group classes.

1 ▶ `IsGroupClass(`*grpclass*`)` P

A generic class (see Chapter 2) is considered a group class if it has the property `IsGroupClass`. There is no way for CRISP to **know** that a given class defined by a membership function is a group class, i. e., consists of groups and is closed under group isomorphisms.

2 ▶ `ContainsTrivialGroup(`*grpclass*`)` P

This property, if bound, indicates whether *grpclass* contains the trivial group or not.

3 ► `IsSubgroupClosed(`*grpclass*`)` P

if true, then for every G in *grpclass*, the subgroups of G likewise belong to *grpclass*.

4 ► `IsNormalSubgroupClosed(`*grpclass*`)` P

if true, then for every G in *grpclass*, the (sub)normal subgroups of G likewise belong to *grpclass*.

5 ► `IsQuotientClosed(`*grpclass*`)` P

if true, then for every G in *grpclass*, the factor groups of G likewise belong to *grpclass*.

6 ► `IsResiduallyClosed(`*grpclass*`)` P

if true and G is a group such that G/N_1 and G/N_2 belong to *grpclass* for two normal subgroups N_1 and N_2 of G **which intersect trivially**, then G belongs to *grpclass*.

7 ► `IsNormalProductClosed(`*grpclass*`)` P

if true and G is a group which is generated by subnormal subgroups in *grpclass*, then G belongs to *grpclass*.

8 ► `IsDirectProductClosed(`*grpclass*`)` P

if true and the group G is the direct product of N_1 and N_2 belonging to *grpclass*, then G likewise belongs to *grpclass*.

9 ► `IsSchunckClass(`*grpclass*`)` P

if true, then G belongs to *grpclass* if and only if its primitive factor groups lie in *grpclass*. A (finite) group is primitive if it has a faithful primitive permutation representation, or equivalently, if it has a maximal subgroup with trivial core. A Schunck class contains every trivial group.

10 ► `IsSaturated(`*grpclass*`)` P

if true, G belongs to X whenever G/`FrattiniSubgroup(`G`)` belongs to X.

3.3 Additional properties of group classes

Note that the following “properties” are not properties but only filters in the GAP sense (cf. GAP Reference Manual, 13.7 and GAP Reference Manual, 13.2 in the GAP reference manual).

1 ► `HasIsFittingClass(`*obj*`)` F

is true if *obj* **knows** if it is a Fitting class, that is, if it lies in the filters `HasIsGroupClass`, `HasContainsTrivialGroup`, `HasIsNormalSubgroupClosed` and `HasIsNormalProductClosed`.

2 ► `IsFittingClass(`*obj*`)` F

is true if *obj* is a Fitting class, that is, if it has the properties `IsGroupClass`, `ContainsTrivialGroup`, `IsNormalSubgroupClosed` and `IsNormalProductClosed`.

3 ► `SetIsFittingClass(`*group class*`, `*bool*`)` F

If *bool* is `true`, this fake setter function sets the properties `IsNormalSubgroupClosed` and `IsNormalProductClosed` of *group class* to *true*. It is the user’s responsibility to ensure that *group class* is indeed a Fitting class.

```
gap> nilp := GroupClass(IsNilpotent);
GroupClass(in:=<Operation "IsNilpotent">)
gap> SetIsFittingClass(nilp, true);
gap> nilp;
FittingClass(in:=<Operation "IsNilpotent">)
```

4 ► `HasIsOrdinaryFormation(`*obj*`)` F

is true if *obj* **knows** if it is a formation, that is, if it lies in the filters `HasIsGroupClass`, `HasContainsTrivialGroup`, `HasIsQuotientClosed` and `HasIsResiduallyClosed`.

5 ► `IsOrdinaryFormation(`*obj*`)` F

is true if *obj* is a formation, that is, if it has the properties `IsGroupClass`, `ContainsTrivialGroup`, `IsQuotient-Closed` and `IsResiduallyClosed`.

6 ► `SetIsOrdinaryFormation(`*class*`, `*bool*`)` F

If *bool* is `true`, this sets the attributes `IsQuotientClosed`, `ContainsTrivialGroup`, and `IsResiduallyClosed` of *class*, making it a formation.

7 ► `HasIsSaturatedFormation(`*obj*`)` F

returns true if *obj* **knows** if it is a saturated formation, that is, if it lies in the filters `HasIsOrdinaryFormation` and `HasIsSaturated`.

8 ► `IsSaturatedFormation(`*obj*`)` F

returns true if *obj* is a saturated formation, that is, if it has the properties `IsOrdinaryFormation` and `IsSaturated`

9 ► `SetIsSaturatedFormation(`*class*`, `*bool*`)` F

If *bool* is `true`, this sets the attributes `IsQuotientClosed`, `ContainsTrivialGroup`, and `IsResiduallyClosed` and `IsSaturated` of *class*, making it a saturated formation.

10 ► `HasIsFittingFormation(`*obj*`)` F

returns true if *obj* **knows** whether it is a Fitting formation, that is, if it lies in the filters `HasIsOrdinaryFormation` and `HasIsFittingClass` (see 3.3.4 and 3.3.1).

11 ► `IsFittingFormation(`*obj*`)` F

returns true if *obj* is both a formation and a Fitting class.

12 ► `SetIsFittingFormation(`*class*`, `*bool*`)` F

If *bool* is `true`, this function sets the attributes of *class* to indicate that it is a Fitting formation.

13 ► `HasIsSaturatedFittingFormation(`*obj*`)` F

returns true if *obj* **knows** whether it is a saturated Fitting formation, that is, if it lies in the filters `HasIsSaturated-Formation` and `HasIsFittingClass` (see 3.3.7 and 3.3.1).

14 ► `IsSaturatedFittingFormation(`*obj*`)` F

returns true if *obj* is both a saturated formation and a Fitting class, that is, if it lies in the filters `IsSaturatedForma-tion` and `IsFittingClass` (see 3.3.8 and 3.3.2).

15 ► `SetIsSaturatedFittingFormation(`*class*`, `*bool*`)` F

If *bool* is `true`, this sets the attributes of *class* to indicate that it is a saturated Fitting formation.

3.4 Attributes of group classes

In addition to the attribute `MemberFunction` which has the same meaning as for generic classes, a group class may have the following attribute.

1 ► `Characteristic(`*grpclass*`)` A

This attribute, if present, stores a class containing all primes p such that *grpclass* contains a cyclic group of order p. There is a pre-defined class `AllPrimes` which should be assigned to `Characteristic` if *grpclass* contains a cyclic group of order p for every prime p.

4 Schunck classes and formations

In principle, any group class can be created as generic (group) class, followed by setting the required properties and attributes described in the preceding chapters. For certain standard kinds of group classes, there are additional functions available to accomplish this task, which are described in this and the following chapter.

4.1 Creating Schunck classes

A class $\mathscr{C}$ of finite groups is a **Schunck class** if a finite group G belongs to $\mathscr{C}$ if and only if all its primitive factor groups belong to $\mathscr{C}$. In particular, a Schunck class is nonempty and closed with respect to factor groups. By definition, a Schunck class $\mathscr{C}$ is determined by the primitive groups which it contains (the basis of $\mathscr{C}$), or by the primitive groups not in $\mathscr{C}$ but all of whose proper factor groups belong to $\mathscr{C}$ (the boundary of $\mathscr{C}$).

1 ► `SchunckClass(`*rec*`)` O

returns a Schunck class defined by the information stored in the record *rec*. Note that it is the user's responsibility to ensure that *rec* really defines a Schunck class. *rec* may have the following components: `\in`, `proj`, `bound`, `char`, and `name`. The values bound to these entries, if present, are stored, respectively, in the attributes `MemberFunction`, `ProjectorFunction`, `BoundaryFunction`, `Characteristic`, and `Name`, see 2.2.2, 4.2.6, 4.2.5m, 3.4.1, and GAP Reference Manual, 12.8.2 for the meaning of these attributes.

At least one of the attributes `MemberFunction`, `ProjectorFunction`, or `BoundaryFunction` must be present in order to be able to compute with a Schunck class.

```
gap> nilp := SchunckClass(rec(bound := G -> not IsCyclic(G),
>        name := "class of all nilpotent groups"));
class of all nilpotent groups
gap> DihedralGroup(8) in nilp;
true
gap> SymmetricGroup(3) in nilp;
false
```

4.2 Attributes and operations for Schunck classes

In addition to the attributes and operations for generic group classes, for Schunck classes also the following are available:

1 ► `Boundary(`*class*`)` A

computes the boundary of *class*, i. e., the class of all primitive groups which do not belong to *class* but whose proper factor groups do. The result is a group class.

2 ► `Basis(`*class*`)` A

The basis of *class* consists of the primitive soluble groups in *class*. The result is a group class.

3 ► `Projector(`*grp*`, `*class*`)` O

This function returns a *class*-projector of *grp*. Note that, at present, methods are only available for finite soluble groups *grp*, or when *class* has an attribute `ProjectorFunction`.

A subgroup H of the group G is a *class*-projector of G if HN/N is *class*-maximal in G/N for all normal subgroups N of G. A subgroup H of G is *class*-maximal in G if H belongs to *class*, and there is no subgroup L of G which contains H and lies in *class*. Note that if *class* consists of finite soluble groups, then *class*-projectors exist in every finite soluble group if and only if *class* is a Schunck class, and in this case all *class*-projectors of G are conjugate. See [DH92], III, 3.21.

```
gap> H := SchunckClass(rec(bound := G -> Size(G) = 6));
SchunckClass(bound:=function( G ) ... end)
gap> Size(Projector(GL(2,3), H));
16
gap> # H-projectors coincide with Sylow subgroups
gap> U := SchunckClass(rec( # class of all supersoluble groups
>    bound := G -> not IsPrimeInt( Size(Socle(G)))
> ));
SchunckClass(bound:=function( G ) ... end)
gap> Size(Projector(SymmetricGroup(4), U));
6
gap> # the projectors are the point stabilizers
```

4 ► `CoveringSubgroup(`*grp*`, `*class*`)` O

A subgroup H of G is a *class*-covering subgroup of G if H is a *class*-projector of L for every subgroup L with $H \leq L \leq G$. Note that projectors and covering subgroups coincide for Schunck classes of finite soluble groups. At present, methods are only available for finite soluble groups *grp*.

5 ► `BoundaryFunction(`*grpclass*`)` A

If bound, this attribute stores a function *func* which has been set by the user to define *grpclass*. *func* must be a function taking one argument. If the argument is a finite primitive soluble group G, *func* must return `true` if G is in the boundary of *grpclass*, and `false` if G belongs to *grpclass*. The behaviour for arguments which are not primitive soluble groups, or which belong neither to *grpclass* nor to the boundary of *grpclass* need not be defined. Note that `BoundaryFunction` should **not** be used to test whether a given group belongs to the boundary of *grpclass*. `Boundary` and/or `Basis` (see 4.2.1 and 4.2.2), which are defined independently of the way *grpclass* is defined and will work for all finite soluble groups.

6 ► `ProjectorFunction(`*grpclass*`)` A

If bound, `ProjectorFunction` stores a function *func* supplied by the user as part of the definition of *grpclass*. *func* must be a function taking a group G as the only argument, and returns a *grpclass*-projector of G. Note that `Projector` (see 4.2.3), rather than `ProjectorFunction`, should be used by the user to compute *grpclass*-projectors.

4.3 Additional attributes for primitive soluble groups

1 ► `IsPrimitiveSolubleGroup(`*grp*`)` P
► `IsPrimitiveSoluble(`*grp*`)` P
► `IsPrimitiveSolvableGroup(`*grp*`)` P
► `IsPrimitiveSolvable(`*grp*`)` P

returns true if *grp* is primitive and soluble, and false otherwise. An abstract finite group G is **primitive** if it has a faithful primitive permutation representation, or equivalently, if it has a maximal subgroup M with trivial core. If G is soluble, M complements the unique minimal normal subgroup N of G. Therefore N is the socle as well as the Fitting subgroup of G. A permutation group may be primitive as an abstract group while it is not primitive as a permutation group (cf. GAP Reference Manual, 41.10.7).

2 ► `SocleComplement(`*grp*`)` A

If present, this attribute stores a complement of the socle of *grp*. Currently, there is only a method available for `SocleComplement` if *grp* has the property `IsPrimitiveSoluble`.

4.4 Creating formations

A nonempty group class is a formation if it is closed with respect to factor groups and residually closed. A saturated formation is, of course, a formation which is saturated. Note that by the Gaschütz-Lubeseder-Schmid theorem (see e. g. [DH92], IV, 4.6), every saturated formation is a local formation. Moreover, every saturated formation is a Schunck class. Therefore a saturated formation admits the operations `Boundary`, `Basis`, and `Projector`.

1 ► `OrdinaryFormation(`*rec*`)` O

creates a formation from the record *rec*. Note that it is the user's responsibility to ensure that *rec* really defines a formation. *rec* may have components `\in`, `res`, `char`, and `name`, whose values are stored in the attributes `MemberFunction`, `ResidualFunction`, `Characteristic`, and `Name`, respectively, of the new formation. See 2.2.2, 4.5.2, 3.4.1, and GAP Reference Manual, 12.8.2, respectively, for the meaning of these attributes.

The following example shows how to construct the formations of all groups of derived length at most 3 and of all groups of exponent dividing 6.

```
gap> der3 := OrdinaryFormation(rec(
>    res := G -> DerivedSubgroup(DerivedSubgroup(DerivedSubgroup(G)))
> ));
OrdinaryFormation(res:=function( G ) ... end)
gap> SymmetricGroup(4) in der3;
true
gap> GL(2,3) in der3;
false
gap> exp6 := OrdinaryFormation(rec(
>    \in := G -> 6 mod Exponent(G) = 0,
>    char := [2,3]));
OrdinaryFormation(in:=function( G ) ... end)
```

2 ► `SaturatedFormation(`*rec*`)` O

creates a saturated formation from the record *rec*. Note that it is the user's responsibility to ensure that *rec* really defines a saturated formation. *rec* may have any components admissible for formations (see 4.4.1) or Schunck classes (see 4.1.1), that is, `\in`, `res`, `char`, `proj`, `bound`, `locdef`, and `name`, whose values, if bound, are stored in the attributes `MemberFunction`, `ResidualFunction`, `Characteristic`, `ProjectorFunction`, `BoundaryFunction`, `LocalDefinitionFunction`, and `Name`, respectively. Please refer to 2.2.2, 4.5.2, 3.4.1, 4.2.6, 4.2.5, 4.5.3, and GAP Reference Manual, 12.8.2 for the meaning of these attributes.

There are also functions `FittingFormation` and `SaturatedFittingFormation` to create Fitting formations and saturated Fitting formations; see 5.2.1 and 5.2.2 below for details.

The following example shows how to construct the saturated formations of all finite nilpotent groups and of all nilpotent-by-abelian groups whose order is not divisible by a prime congruent 3 mod 4, and whose 2-chief factors are central. In the first case, we choose $f(p) = (1)$ for all primes p, so that the $f(p)$-residual of G is generated by a generating set of G (see 4.5.3 below). In the second example, we let $f(2) = 1$, if $p \equiv 3 \pmod 4$, we define $f(p) = \mathscr{A}$, the class of all finite abelian groups, and set $f(p) = \emptyset$ otherwise.

```
gap> nilp := SaturatedFormation(rec(
>      locdef := function(G, p)
>          return GeneratorsOfGroup(G);
>      end));
SaturatedFormation(locdef:=function( G, p ) ... end)
gap> form := SaturatedFormation(rec(
>    locdef := function(G, p)
>        if p = 2 then
>           return GeneratorsOfGroup(G);
>        elif p mod 4 = 3 then
>           return GeneratorsOfGroup(DerivedSubgroup(G));
>        else
>           return fail;
>        fi;
>     end));
SaturatedFormation(locdef:=function( G, p ) ... end)
gap> Projector(GL(2,3), form);
Group([ [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ],
  [ [ Z(3)^0, Z(3) ], [ 0*Z(3), Z(3)^0 ] ],
  [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ] ])
```

3 ► `FormationProduct(`*form1*`, `*form2*`)` O

The formation product *prod* of two formations *form1* and *form2* consists of the groups G such that the *form2*-residual of G belongs to *form1*. The product *prod* is again a formation. If *form1* and *form2* are saturated formations, the result is a saturated formation. The same is true if the characteristic of *form2* is contained in that of *form1*. This is automatically recognised if the characteristic of *form1* is `AllPrimes` (see 6.3.1). In all other cases, you will have to set the attribute `IsSaturated` manually, if applicable. Note that in general it is not possible for CRISP to determine if two classes are contained in each other.

```
gap> nilp := SaturatedFormation(rec(\in := IsNilpotent, name := "nilp"));
nilp
gap> FormationProduct(nilp, der3); # no characteristic known
FormationProduct(nilp, OrdinaryFormation(res:=function( G ) ... end))
gap> HasIsSaturated(last);HasCharacteristic(nilp);
false
false
gap> SetCharacteristic(nilp, AllPrimes);
gap> FormationProduct(nilp, der3); # try with characteristic
FormationProduct(nilp, OrdinaryFormation(res:=function( G ) ... end))
gap> IsSaturated(last);
true
```

4 ► `FittingFormationProduct(`*fitform1*`, `*fitform2*`)` O

If *fitform1* and *fitform2* are Fitting formations, the formation product equals the Fitting product (see 5.1.2) of *fitform1* and *fitform2*, which, in turn, equals the class product of *fitform1* and *fitform2*. The latter consists of all groups G having a normal subgroup N in *fitform1* such that G/N belongs to *fitform2*.

Note that if *fitform1* and *fitform2* are Fitting formations, then `FormationProduct(`*fitform1*`, `*fitform2*`)`, `FittingProduct(`*fitform1*`, `*fitform2*`)` and `FittingFormationProduct(`*fitform1*`, `*fitform2*`)` all return the same mathematical object (but distinct GAP objects), which is, again, a Fitting formation.

```
gap> nilp := FittingFormation(rec(\in := IsNilpotent, name := "nilp"));;
gap> FormationProduct(nilp, nilp);
FittingFormationProduct(nilp, nilp)
gap> FittingProduct(nilp, nilp);
FittingFormationProduct(nilp, nilp)
gap> FittingFormationProduct(nilp, nilp);
FittingFormationProduct(nilp, nilp)
```

4.5 Attributes and operations for formations

In addition to those available for generic group classes, formations also admit the following attributes and operations. See also 4.2 for additional operations for saturated formations.

1 ► `Residual(`*grp*`,` *form*`)` O
► `Residuum(`*grp*`.` *form*`)` O

return the *form*-residual (also called *form*-residuum) of the group *grp*. This is the smallest normal subgroup *res* of *grp* such that *grp*/*res* belongs to *form*. Note that, unless *form* has an attribute `ResidualFunction`, there are presently only methods available for finite soluble groups.

2 ► `ResidualFunction(`*form*`)` A

This attribute is part of the definition of *form* supplied by the user. If present, it must contain a function which computes the *form*-residual of a given group. In general, such a residual only exists if *form* is residually closed; cf. 3.2.6. Note that `ResidualFunction`, if present, is called by `Residual` (see 4.5.1). Therefore `Residual`, which also works for formations without `ResidualFunction`, should be used by the user to compute *form*-residuals.

3 ► `LocalDefinitionFunction(`*form*`)` A

Let *form* be a saturated formation with local function f. This attribute, if present, stores a function *func* supplied by the user as part of the definition of *form*. *func* must be a function taking two parameters, a group G and a prime p. If p is in the characteristic of *form*, this function must return a list *list* of elements of G, such that the smallest normal subgroup of G containing *list* is the $f(p)$-residual of G. If p is not in the characteristic of *form*, then *func*`(`G`,` p`)` must return `fail` for any group G. `LocalDefinitionFunction` is part of the definition of *form* and should not be called by the user.

4.6 Low level functions for normal subgroups related to residuals

1 ► `OneInvariantSubgroupMinWrtQProperty(`*act*`,` *grp*`,` *pretest*`,` *test*`,` *data*`)` O

Let *act* be a list or group whose elements act on *grp* via the caret operator, such that every subgroup of *grp* invariant under *act* is normal in *grp*. Assume that $\mathscr{X}$ is a set of *act*-invariant subgroups of *grp* containing *grp*, and such that whenever M and N are *act*-invariant subgroups with $M \in \mathscr{X}$ and M contained in N, then also $N \in \mathscr{X}$. Then `OneInvariantSubgroupMinWrtQProperty` computes an *act*-invariant subgroup $M \in \mathscr{X}$ such that no *act*-invariant subgroup of *grp* contained in M belongs to $\mathscr{X}$. At present, there exist only methods for finite soluble groups *grp*.

The class $\mathscr{X}$ is described by two functions, *pretest* and *test*.

pretest is a function taking four arguments, U, V, R, and *data*, where *data* is just the argument passed to `OneInvariantSubgroupMinWrtQProperty` (see below for examples). U/V is a chief factor of *grp*, and R is an *act*-invariant subgroup of *grp* containing U which is known to belong to $\mathscr{X}$.

pretest may return the values `true`, `false`, or `fail`. If it returns `true`, every *act*-invariant subgroup N of *grp* such that V is contained in N and R/N is G-isomorphic with U/V must belong to $\mathscr{X}$. If it returns `false`, no such *act*-invariant subgroup N may belong to $\mathscr{X}$.

test is a function taking three arguments, *S*, *R*, and *data*, where data has been described above. *R* is a *act*-invariant subgroup of *grp* belonging to $\mathscr{X}$, and R/S is a chief factor of *grp*. The function must return true if *S* belongs to $\mathscr{X}$, and false otherwise.

Note that whenever *test*(*S*, *R*, *data*) is called, *pretest*(*U*, *V*, *R*, *data*) has been called before, and has returned `fail`, where U/V is a chief factor which is *G*-isomorphic with R/S. Thus *test* need not repeat tests which have been performed by *pretest*. In particular, if *pretest* always returns `true` or `false`, *test* will never be called.

data is never used or changed by `OneInvariantSubgroupMinWrtQProperty`, but exists only as a means for passing additional information to or between the functions *pretest* and *test*.

For example, if $\mathscr{C}$ is a group class which is closed with respect to factor groups and $\mathscr{X}$ is the set of all *act*-invariant subgroups *N* of *grp* with $grp/N \in \mathscr{C}$, then $\mathscr{X}$ satisfies the above properties. In particular, if $\mathscr{C}$ is a formation, then `OneInvariantSubgroupMinWrtQProperty` will return the $\mathscr{C}$-residual of *grp*.

The following example shows how to use `OneInvariantSubgroupMinWrtQProperty` to compute the derived subgroup of a group *G*. (Note that in practice, this is not a particularly efficient way of computing the derived subgroup.)

```
gap> G := DirectProduct(SL(2,3), CyclicGroup(2));;
gap> data := rec(gens := GeneratorsOfGroup(G),
>    comms := List(Combinations(GeneratorsOfGroup(G), 2),
>       x -> Comm(x[1],x[2])));;
gap> OneInvariantSubgroupMinWrtQProperty(
>    G, G,
>    function(U, V, R, data) # test if U/V is central in G
>        if ForAny(ModuloPcgs(U, V), y ->
>           ForAny(data.gens, x -> not Comm(x, y) in V)) then
>           return false;
>        else
>           return fail;
>        fi;
>     end,
>     function(S, R, data)
>        return ForAll(data.comms, x -> x in S);
>     end,
>     data) = DerivedSubgroup(G); # compare results
true
```

2 ► `AllInvariantSubgroupsWithQProperty(`*act*`, `*grp*`, `*pretest*`, `*test*`, `*data*`)` O

`AllInvariantSubgroupsWithQProperty` returns a list consisting of all *act*-invariant subgroups in $\mathscr{X}$, where $\mathscr{X}$ is the class defined by *pretest*, *test*, and *data*, as described for `OneInvariantSubgroupMinWrtQProperty` (see 4.6.1). At present, there exist only methods for finite soluble groups *grp*.

```
gap> G := GL(2,3);
GL(2,3)
gap> normsWithSupersolubleFactorGroups :=
> AllInvariantSubgroupsWithQProperty(G, G,
>    function(U, V, R, data)
>       return IsPrimeInt(Index(U, V));
>    end,
>    ReturnFail, # pretest is sufficient
>    fail); # no data required
[ GL(2,3),
  Group([ [ [ Z(3)^0, Z(3) ], [ 0*Z(3), Z(3)^0 ] ],
      [ [ Z(3), Z(3)^0 ], [ Z(3)^0, Z(3)^0 ] ],
```

```
        [ [ 0*Z(3), Z(3)^0 ], [ Z(3), 0*Z(3) ] ],
        [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ] ]),
    Group([ [ [ Z(3), Z(3)^0 ], [ Z(3)^0, Z(3)^0 ] ],
        [ [ 0*Z(3), Z(3)^0 ], [ Z(3), 0*Z(3) ] ],
        [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ] ]) ]
```

3 ► `OneNormalSubgroupMinWrtQProperty(`*grp*`,` *pretest*`,` *test*`,` *data*`)` O

`OneNormalSubgroupMinWrtQProperty` is a special case of `OneInvariantSubgroupMinWrtQProperty` (see 4.6.1) where *act* = *grp*.

4 ► `AllNormalSubgroupsWithQProperty(`*grp*`,` *pretest*`,` *test*`,` *data*`)` O

`AllNormalSubgroupsWithQProperty` is a special case of `AllInvariantSubgroupsWithQProperty` (see 4.6.2) where *act* = *grp*.

5 Fitting classes and Fitting sets

In this chapter, you will find information on how to create Fitting classes and Fitting sets (see 5.1 and 5.3 below), and how to compute injectors and radicals with respect to these; see 5.4.

5.1 Creating Fitting classes

Recall that a Fitting class is a nonempty group class which is closed with respect to normal subgroups and joins of subnormal subgroups.

1 ▶ `FittingClass(`*rec*`)` O

returns the Fitting class *fitclass* defined by the entries of the record *rec*. Note that it is the user's responsibility to ensure that *rec* really defines a Fitting class. *rec* may have components `\in`, `inj`, `rad`, `char`, and `name`. The functions assigned to the components are stored in the attributes `MemberFunction`, `InjectorFunction`, `RadicalFunction`, `Characteristic`, and `Name`, of *fitclass*. Please refer to 2.2.2, 5.4.4, 5.4.3, 3.4.1, and GAP Reference Manual, 12.8.2 for the meaning of these attributes.

The third example below shows how to construct the Fitting class $L_2(\mathscr{N})$ (see [DH92], IX, 1.14 and 1.15), where $\mathscr{N}$ is the class of all finite nilpotent groups.

```
gap> myNilpotentGroups := FittingClass(rec(\in := IsNilpotentGroup,
>    rad := FittingSubgroup));
FittingClass(in:=<Property "IsNilpotentGroup">, rad:=<Attribute "FittingSubgr\
oup">)
gap> myTwoGroups := FittingClass(rec(
>    \in := G -> IsSubset([2], Set(Factors(Size(G)))),
>    rad :=  G -> PCore(G,2),
>    inj := G -> SylowSubgroup(G,2)));
FittingClass(in:=function( G ) ... end, rad:=function( G ) ... end, inj:=func\
tion( G ) ... end)
gap> myL2_Nilp := FittingClass(rec(\in :=
>     G -> IsSolvableGroup(G)
>          and Index(G, Injector(G, myNilpotentGroups)) mod 2 <> 0));
FittingClass(in:=function( G ) ... end)
gap> SymmetricGroup(3) in myL2_Nilp;
false
gap> SymmetricGroup(4) in myL2_Nilp;
true
```

2 ▶ `FittingProduct(`*fit1*`,` *fit2*`)` O

returns the Fitting product *prod* of the Fitting classes *fit1* and *fit2*, i. e., the class of all groups G such that G/R is a *fit2*-group, where R is the *fit1*-radical of G. *prod* is again a Fitting class. Note that if *fit1* and *fit2* are also formations, then *prod* equals the formation product of *fit1* and *fit2*; see 4.4.3 and 4.4.4.

```
gap> FittingProduct(myNilpotentGroups, myTwoGroups);
FittingProduct(FittingClass(in:=<Property "IsNilpotentGroup">, rad:=<Attribu\
te "FittingSubgroup">), FittingClass(in:=function( G ) ... end, rad:=function\
( G ) ... end, inj:=function( G ) ... end))
gap> FittingProduct(myNilpotentGroups, myL2_Nilp);
FittingProduct(FittingClass(in:=<Property "IsNilpotentGroup">, rad:=<Attribu\
te "FittingSubgroup">), FittingClass(in:=function( G ) ... end))
```

5.2 Creating Fitting formations

Fitting formations are Fitting classes which are also formations.

1 ► `FittingFormation(`*rec*`)` O

creates a Fitting formation from the record *rec*. Note that it is the user's responsibility to ensure that *rec* really defines a Fitting formation. *rec* may have any components admissible for saturated formations (see 4.4.2) or Fitting classes (see 5.1.1), that is, `\in`, `res`, `rad`, `inj`, `char`, and `name`, whose values are stored in the attributes `MemberFunction`, `ResidualFunction`, `RadicalFunction`, `InjectorFunction`, `Characteristic`, and `Name`, respectively. Please refer to 2.2.2, 4.5.2, 5.4.3, 5.4.4, 3.4.1, and GAP Reference Manual, 12.8.2, respectively, for the meaning of these attributes.

2 ► `SaturatedFittingFormation(`*rec*`)` O

creates a saturated Fitting formation from the record *rec*. Note that it is the user's responsibility to ensure that *rec* really defines a saturated Fitting formation. *rec* may have any components admissible for saturated formations (see 4.4.2) or Fitting classes (see 5.1.1), that is, `\in`, `res`, `proj`, `bound`, `locdef`, `rad`, `inj`, `char`, and `Name`, whose values are stored in the attributes `MemberFunction` (see 2.2.2), `ResidualFunction` (see 4.5.2), `ProjectorFunction` (see 4.2.6), `BoundaryFunction` (see 4.2.5), `LocalDefinitionFunction` (see 4.5.3), `RadicalFunction` (see 5.4.3), `InjectorFunction` (see 5.4.4), `Characteristic` (see 3.4.1), and `Name` (see GAP Reference Manual, 12.8.2), respectively.

5.3 Creating Fitting sets

A nonempty set $\mathscr{F}$ of subgroups of a group G is a **Fitting set of** G if it satisfies the following properties:

(1) if H belongs to $\mathscr{F}$ and K is normal in H, then K belongs to $\mathscr{F}$;

(2) if H and K belong to $\mathscr{F}$, and H and K are normal in $\langle H,K\rangle$, then $\langle H,K\rangle = HK$ belongs to $\mathscr{F}$;

(3) if H is in $\mathscr{F}$ and $g \in G$, then H^g also belongs to $\mathscr{F}$.

Note that a Fitting set *fitset* of the group G is a subset of the set of all subgroups of G. Therefore it is not closed under group isomorphisms, hence is **not** a group class. If H is a subgroup of G, then the subgroups of G in *fitset* which are contained in H form a Fitting set of H. We will not distinguish between *fitset* and the arising Fitting set of H. Moreover, if *fit* is a Fitting class and *grp* is a group, then the set of all subgroups of *grp* which belong to *fit* is a Fitting set of *grp*.

1 ► `IsFittingSet(`G`, `*fitset*`)` O

tests whether *fitset* (or, more precisely, the set of all subgroups of G which are contained in *fitset*) is a Fitting set of the group G. Thus if *fitset* is a Fitting class, or if G is a subgroup of the group H and *fitset* is a Fitting set of H, then `IsFittingSet(`G`, `*fitset*`)` will return `true`.

2 ► `FittingSet(`G`, `*rec*`)` O

returns the Fitting set *fitset* of the group G, defined by the entries of the record *rec*. Note that, although it would be possible to test whether *rec* defines a Fitting set, such a test is not performed, since it would be extremely expensive, even for relatively small groups.

rec may have components `\in`, `inj`, `rad`, and `name`. The functions assigned to the components are stored in the attributes `MemberFunction`, `InjectorFunction`, `RadicalFunction`, and `Name`, of *fitset*. Please see 2.2.2, 5.4.4 and 5.4.3 for the meaning of these arguments.

Note that at present, every Fitting set has to be a class (see 2). The second example below shows how to define a Fitting set from a list of subgroups.

```
gap>  fitset := FittingSet(SymmetricGroup(4), rec(
>        \in := S -> IsSubgroup(AlternatingGroup(4), S),
>        rad := S -> Intersection(AlternatingGroup(4), S),
>        inj := S -> Intersection(AlternatingGroup(4), S)));
FittingSet(SymmetricGroup(
[ 1 .. 4 ] ), rec(in:=function( S ) ... end, rad:=function( S ) ... end, inj:\
=function( S ) ... end))
gap> FittingSet(SymmetricGroup(3), rec(
>       \in := H -> H in [Group(()), Group((1,2)), Group((1,3)), Group((2,3))]));
FittingSet(SymmetricGroup( [ 1 .. 3 ] ), rec(in:=function( H ) ... end))
```

3 ► `ImageFittingSet(`*alpha*`, `*fitset*`)` O

returns the image *F_1* of the Fitting set *fitset* under the group homomorphism *alpha*, i.e. the Fitting set *F_1* of `Image(`*alpha*`)` which consists of all subgroups *alpha*`(`*S*`)` of `Image(`*alpha*`)` such that *S* is a *fitset*-injector of `PreImage(`*alpha*`, `*S*`)`. *fitset* must be a Fitting set of `PreImage(`*alpha*`)` or a Fitting class. Note that the image of a Fitting class is a Fitting set but not a Fitting class.

```
gap> alpha := GroupHomomorphismByImages(SymmetricGroup(4), SymmetricGroup(3),
>  [(1,2),(1,3),(1,4)], [(1,2),(1,3),(2,3)]);;
gap> im := ImageFittingSet(alpha, fitset);
FittingSet(Group( [(1,2),(1,3),(2,3)
 ] ), rec(inj:=function( G ) ... end))
gap> Radical(Image(alpha), im);
Group([ (), (), (1,2,3), (1,3,2) ])
```

4 ► `PreImageFittingSet(`*alpha*`, `*fitset*`)` O

returns the preimage *fitset_0* of the Fitting set *fitset* of `Image(`*alpha*`)` under the group homomorphism *alpha*. It consists of all subgroups *S* of `PreImage(`*alpha*`)` which are subnormal in `PreImage(`*alpha*`, `*T*`)` for some *T* in *fitset*. *fitset* must be a Fitting set of `Image(`*alpha*`)` or a Fitting class.

Note that the preimage of a Fitting class is just a Fitting set but not a Fitting class.

Moreover, `ImageFittingSet(PreImageFittingSet(`*fitset*`, `*alpha*`), `*alpha*`)` equals *fitset* but in general, *fitset* is not contained in `PreImageFittingSet(ImageFittingSet(`*fitset*`, `*alpha*`), `*alpha*`)`; see e.g. Example VIII, 2.16 of [DH92].

```
gap> pre := PreImageFittingSet(alpha, NilpotentGroups);
FittingSet(SymmetricGroup( [ 1 .. 4 ] ), rec(inj:=function( G ) ... end))
gap> Injector(Source(alpha), pre);
Group([ (1,2,3), (1,2)(3,4) ])
```

5 ► `Intersection(`*fitset1*`, `*fitset2*`)`

Let *fitset1* and *fitset2* be Fitting sets of the groups *G1* and *G2*. Then the intersection of *fitset1* and *fitset2* will be a Fitting set of the intersection of *G1* and *G2*. You will run into an error if GAP cannot compute the intersection of *G1* and *G2*.

```
gap> F1 := FittingSet(SymmetricGroup(3),
> rec(\in := IsNilpotentGroup, rad := FittingSubgroup));
FittingSet(SymmetricGroup(
[ 1 .. 3 ] ), rec(in:=<Property "IsNilpotentGroup">, rad:=<Attribute "Fitting\
Subgroup">))
gap> F2 := FittingSet(AlternatingGroup(4),
> rec(\in := ReturnTrue, rad := H -> H));
FittingSet(AlternatingGroup(
[ 1 .. 4 ] ), rec(in:=function( arg ) ... end, rad:=function( H ) ... end))
gap> F := Intersection(F1, F2);
FittingSet(Group(
[ (1,2,3) ] ), rec(in:=function( x ) ... end, rad:=function( G ) ... end))
gap> Intersection(F1, PiGroups([2,5]));
FittingSet(SymmetricGroup(
[ 1 .. 3 ] ), rec(in:=function( x ) ... end, rad:=function( G ) ... end))
```

5.4 Attributes and operations for Fitting classes and Fitting sets

In addition to operations applicable to classes, both Fitting sets and Fitting classes admit the following attributes and operations. Of course, Fitting classes, being group classes, also admit all properties and attributes for group classes.

1 ► `Radical(`*G*`, `*fitset*`)` O

returns the *grpclass*-radical of the group *G*, where *fitset* is a Fitting set of *G* (see 5.3.1), or a Fitting class. The *fitset*-radical of *G* is the unique largest normal subgroup of *G* belonging to *fitset*. Note that `Radical(`*G*`)` returns the soluble radical of a group *G* (see GAP Reference Manual, 39.12.9 in the GAP reference manual). The class `myL2_Nilp` in the example below has been defined in 5.1.1.

```
gap> Radical(SymmetricGroup(4), FittingClass(rec(\in := IsNilpotentGroup)));
Group([(1,4)(2,3),(1,3)(2,4) ])
gap> Radical(SymmetricGroup(4), myL2_Nilp);
Sym( [ 1 .. 4 ] )
gap> Radical(SymmetricGroup(3), myL2_Nilp);
Group([ (1,2,3) ])
```

2 ► `Injector(`*G*`, `*fitset*`)` O

returns a *fitset*-injector of the group *G*, where *fitset* is a Fitting set of *G* (or a group containing *G*), or a Fitting class. A subgroup *H* of *G* is a *fitset*-injector of *G* if $S \cap H$ is *fitset*-maximal in *S* for every subnormal subgroup *S* of *G*. Note that by [DH92], VIII, 2.9, all *fitset*-injectors of *G* are conjugate in *G*, and it is not hard to see that every subgroup of *G* has *fitset*-injectors if and only if *fitset* is a Fitting set of *G*. In particular, if *fitset* is a group class, then every finite soluble group has *fitset*-injectors if and only if *fitset* is a Fitting class; see [DH92], IX, 1.4.

```
gap> Injector(SymmetricGroup(4), FittingClass(rec(\in := IsNilpotentGroup)));
Group([ (1,4)(2,3), (1,3)(2,4), (3,4) ])
```

3 ► `RadicalFunction(`*class*`)` A

This attribute, if present, forms part of the definition of *class* supplied by the user. It must contain a function which takes one argument, a group *G*, and returns the *class*-radical of *G*. This function will be used during subsequent calls to `Radical`. Therefore `Radical` (see 5.4.1), which is guaranteed to work for arbitrary Fitting sets *class*, should always be called by the user to compute *class*-radicals.

4 ► `InjectorFunction(`*class*`)` A

This attribute constitutes part of the definition of *class* supplied by the user. If present, it must contain a function taking a group *G* as the only argument and returning a *class*-injector of *G*. This function will then be used by `Injector` (see

5.4.2). Since `Injector` will work for arbitrary Fitting sets, it should always be called by the user to compute *class*-injectors.

5.5 Low level functions for normal subgroups related to radicals

1 ▶ `OneInvariantSubgroupMaxWrtNProperty(`*act*`,` *grp*`,` *pretest*`,` *test*`,` *data*`)` O

Let *act* be a list or group whose elements act on *grp* via the caret operator, such that every subgroup of *grp* invariant under *act* is normal in *grp*. Assume $\mathscr{X}$ is a set of subgroups of *grp* such that $\mathscr{X}$ contains the trivial group, and if M and N are *act*-invariant subgroups with $M \in \mathscr{X}$ and M containing N, then also $N \in \mathscr{X}$. Then `OneInvariantSubgroupMaxWrtNProperty` computes an *act*-invariant subgroup $M \in \mathscr{X}$ such that no *act*-invariant subgroup of *grp* properly containing M belongs to $\mathscr{X}$.

For example, every Fitting set $\mathscr{X}$ satisfies the above properties, where $act = G$. In this case, `OneInvariantSubgroupMaxWrtNProperty` will return the $\mathscr{X}$-radical of *grp*.

The class $\mathscr{X}$ is described by two functions, *pretest* and *test*.

pretest is a function taking four arguments, U, V, R, and *data*, where *data* is just the argument passed to `OneInvariantSubgroupMaxWrtNProperty`. U/V is an *act*-composition factor of *grp*, and R is an *act*-invariant subgroup of *grp* contained in V which is known to belong to $\mathscr{X}$.

pretest may return the values `true`, `false`, or `fail`. If it returns `true`, every *act*-invariant subgroup N of *grp* contained in U such that N/R is G-isomorphic with U/V must belong to $\mathscr{X}$. If it returns `false`, no such N may belong to $\mathscr{X}$.

test is a function taking three arguments, S, R, and *data*, where data has been described above. R is an *act*-invariant subgroup of *grp* belonging to $\mathscr{X}$, and S/R is an *act*-composition factor of *grp*. The function must return true if S belongs to $\mathscr{X}$, and false otherwise.

Note that *test*`(`*S*`,` *R*`,` *data*`)` is only called if `pretest(`*U*`,` *V*`,` *R*`,` *data*`)` has returned `fail` for a chief factor U/V which is G-isomorphic with S/R. Therefore *test* need not repeat tests already performed by *pretest*. In particular, if *pretest* always returns `true` or `false`, *test* will not be called at all.

data is never used or changed by `OneInvariantSubgroupMaxWrtNProperty`, but exists only as a means for passing additional information to or between the functions *pretest* and *test*.

2 ▶ `AllInvariantSubgroupsWithNProperty(`*act*`,` *grp*`,` *pretest*`,` *test*`,` *data*`)` O

returns a list consisting of all *act*-invariant subgroups of *grp* belonging to the class $\mathscr{X}$ described by *pretest*, *test*, and *data*. See the documentation of `OneInvariantSubgroupMaxWrtNProperty` (see 5.5.1) for details.

```
gap> D := DihedralGroup(8);;
gap> AllInvariantSubgroupsWithNProperty(
> D, D,
>     ReturnFail,
>     function(R, S, data)
>         return IsAbelian(R);
>     end,
>     fail);
[ Group([ f3 ]), <pc group with 2 generators>, <pc group with 2 generators>,
  Group([ f1, f3 ]), Group([  ]) ]
```

3 ▶ `OneNormalSubgroupWithNProperty(`*grp*`,` *pretest*`,` *test*`,` *data*`)` O
▶ `AllNormalSubgroupsWithNProperty(`*grp*`,` *pretest*`,` *test*`,` *data*`)` O

are the same as `OneInvariantSubgroupMaxWrtNProperty` (see 5.5.1) and `AllInvariantSubgroupsWithNProperty` (see 5.5.2), where $act = grp$, and thus the *act*-invariant subgroups of *grp* are just the normal subgroups of *grp*.

6 Examples of group classes

This chapter describes some pre-defined group classes, namely the classes of all abelian, nilpotent, and supersoluble groups. Moreover, there are some functions constructing the classes of all p-groups, π-groups, and abelian groups whose exponent divides a given positive integer.

The definitions of these group classes can also serve as further examples of how group classes can be defined using the methods described in the preceding chapters.

6.1 Pre-defined group classes

1 ▶ `TrivialGroups` V

The global variable `TrivialGroups` contains the class of all trivial groups. It is a subgroup closed saturated Fitting formation.

2 ▶ `NilpotentGroups` V

This global variable contains the class of all finite nilpotent groups. It is a subgroup closed saturated Fitting formation.

3 ▶ `SupersolubleGroups` V
▶ `SupersolvableGroups` V

This global variable contains the class of all finite supersoluble groups. It is a subgroup closed saturated formation.

4 ▶ `AbelianGroups` V

is the class of all abelian groups. It is a subgroup closed formation.

5 ▶ `AbelianGroupsOfExponent(`*n*`)` F

returns the class of all abelian groups of exponent dividing n, where n is a positive integer. It is always a subgroup-closed formation.

6 ▶ `PiGroups(`*pi*`)` F

constructs the class of all *pi*-groups. *pi* may be a non-empty class or a set of primes. The result is a subgroup-closed saturated Fitting formation.

7 ▶ `PGroups(`*p*`)` F

returns the class of all p-groups, where p is a prime. The result is a subgroup-closed saturated Fitting formation.

6.2 Pre-defined projector functions

1 ▶ `NilpotentProjector(`*grp*`)` A

This function returns a projector for the class of all finite nilpotent groups. For a definition, see 4.2.3. Note that the nilpotent projectors of a finite soluble group equal its a Carter subgroups, that is, its self-normalizing nilpotent subgroups.

2 ▶ `SupersolubleProjector(`*grp*`)` A
▶ `SupersolvableProjector(`*grp*`)` A

These functions return a projector for the class of all finite supersoluble groups. For a definition, see 4.2.3.

6.3 Pre-defined sets of primes

1 ► `AllPrimes` V

is the set of all (integral) primes. This should be installed as value for `Characteristic(`*grpclass*`)` if the group class *grpclass* contains cyclic groups of prime order p for arbitrary primes p.

7 Lists of normal subgroups

The algorithms in CRISP can also be used to compute certain normal subgroups of a finite soluble group efficiently. In particular, CRISP provides fast methods for computing all normal subgroups, all minimal normal subgroups, and the socle of a finite soluble group.

7.1 Functions for normal and characteristic subgroups

1 ► `NormalSubgroups(`*grp*`)` A

For finite soluble groups *grp*, CRISP provides an efficient method to compute `NormalSubgroups` (see GAP Reference Manual, 39.19.8).

2 ► `CharacteristicSubgroups(`*grp*`)` A

returns a list containing all characteristic subgroups of the finite soluble group *grp*. `CharacteristicSubgroups` calls `AllInvSgrsWithQPropUnderAction`.

3 ► `MinimalNormalSubgroups(`*grp*`)` A

CRISP provides an efficient method to compute a list of all minimal normal subgroups of *grp* (see GAP Reference Manual, 39.19.10).

4 ► `MinimalNormalPSubgroups(`*grp*`, `*p*`)` A

For a prime *p*, this function computes a list of all *p*-subgroups which are minimal among the nontrivial normal subgroups of *grp*.

5 ► `AbelianMinimalNormalSubgroups(`*grp*`)` A

This computes a list of all minimal normal subgroups of *grp* which are abelian. If *grp* is soluble, this list coincides with the list of all minimal normal subgroups of *grp*.

7.2 Functions for the socle of finite groups

1 ► `Socle(`*grp*`)` A

CRISP provides a method for `Socle` (see GAP Reference Manual, 39.12.10) for which works for all finite soluble groups *grp*. The socle of a group *grp* is the subgroup generated by all minimal normal subgroups of *grp*. See also 7.2.2 and 7.2.5 below.

```
gap> Size(Socle( DirectProduct(DihedralGroup(8), CyclicGroup(12))));
12
```

2 ► `AbelianSocle(`*grp*`)` A
► `SolubleSocle(`*grp*`)` A
► `SolvableSocle(`*grp*`)` A

This function computes the soluble socle of *grp*. The soluble socle of a group *grp* is the subgroup generated by all minimal normal soluble subgroups of *grp*.

3 ► `SocleComponents(`*grp*`)` A

This function returns a list of minimal normal subgroups of *grp* such that the socle of *grp* (see 7.2.1) is the direct product of these minimal normal subgroups. Note that, in general, this decomposition is not unique. Currently, this function is only implemented for finite soluble groups. See also 7.2.4 and 7.2.6.

4 ► `AbelianSocleComponents(`*grp*`)` A
► `SolubleSocleComponents(`*grp*`)` A
► `SolvableSocleComponents(`*grp*`)` A

This function returns a list of soluble minimal normal subgroups of *grp* such that the socle of *grp* (see 7.2.1) is the direct product of these minimal normal subgroups. Note that, in general, this decomposition is not unique.

5 ► `PSocle(`*grp*`, `*p*`)` A

If p is a prime, the p-socle of a group *grp* is the subgroup generated by all minimal normal p-subgroups of *grp*.

6 ► `PSocleComponents(`*grp*`, `*p*`)` A

For a prime p, this function returns a list of minimal normal p-subgroups of *grp* such that the p-socle of *grp* (see 7.2.5) is the direct product of these minimal normal subgroups. Note that, in general, this decomposition is not unique.

7 ► `PSocleSeries(`*grp*`, `*p*`)` A

For a prime p, this function returns an ascending *grp*-composition series of the p-socle of *grp*.

Bibliography

[DH92] K. Doerk and T. O. Hawkes. *Finite soluble groups.* Walter de Gruyter, 1992.

[EW02] B. Eick and C. R. B. Wright. Computing subgroups by exhibition in finite solvable groups. To appear in *J. Symbolic Comp* 33:129–143, 2002.

[FGH67] B. Fischer, W. Gaschütz, W., B. Hartley. Injektoren endlicher auflösbarer Gruppen. *Math. Z.*, 102:337–339, 1967.

[Gas63] W. Gaschütz. Zur Theorie der endlichen auflösbaren Gruppen. *Math. Z.*, 80:300 – 305, 1963.

[Höf01] B. Höfling. Computing projectors, injectors, residuals and radicals of finite soluble groups. *J. Symbolic Comp.* 32:499–511, 2001.

[Sch67] H. Schunck. $\mathscr{H}$-Untergruppen in endlichen auflösbaren Gruppen. *Math. Z.*, 97:326–330, 1967.

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".