GAP

Release 4.3
05 May 2002

New Features for Developers

The GAP Group

http://www.gap-system.org

Acknowledgement

We would like to thank the many people who have made contributions of various kinds to the development of GAP since 1986, in particular:

Isabel M. Araújo, Robert Arthur, Hans Ulrich Besche, Thomas Bischops, Oliver Bonten, Thomas Breuer, Frank Celler, Gene Cooperman, Bettina Eick, Volkmar Felsch, Franz Gähler, Greg Gamble, Willem de Graaf, Burkhard Höfling, Jens Hollmann, Derek Holt, Erzsébet Horváth, Alexander Hulpke, Ansgar Kaup, Susanne Keitemeier, Steve Linton, Frank Lübeck, Bohdan Majewski, Johannes Meier, Thomas Merkwitz, Wolfgang Merkwitz, Jürgen Mnich, Robert F. Morse, Scott Murray, Joachim Neubüser, Max Neunhöffer, Werner Nickel, Alice Niemeyer, Dima Pasechnik, Götz Pfeiffer, Udo Polis, Ferenc Rákóczi, Sarah Rees, Edmund Robertson, Ute Schiffer, Martin Schönert, Ákos Seress, Andrew Solomon, Heiko Theißen, Rob Wainwright, Alex Wegner, Chris Wensley and Charles Wright.

The following list gives the authors, indicated by `A`, who designed the code in the first place as well as the current maintainers, indicated by `M` of the various modules of which GAP is composed.

Since the process of modularization was started only recently, there might be omissions both in scope and in contributors. The compilers of the manual apologize for any such errors and promise to rectify them in future editions.

Kernel
: Frank Celler (A), Steve Linton (AM), Frank Lübeck (AM), Werner Nickel (AM), Martin Schönert (A)

Automorphism groups of finite pc groups
: Bettina Eick (AM)

Binary Relations
: Robert Morse (AM), Andrew Solomon (A)

Classes in nonsolvable groups
: Alexander Hulpke (AM)

Classical Groups
: Thomas Breuer (AM), Frank Celler (A), Stefan Kohl (AM), Frank Lübeck (AM), Heiko Theißen (A)

Congruences of magmas, semigroups and monoids
: Robert Morse (AM), Andrew Solomon (A)

Cosets and Double Cosets
Alexander Hulpke (AM)

Cyclotomics
Thomas Breuer (AM)

Dixon-Schneider Algorithm
Alexander Hulpke (AM)

Documentation Utilities
Frank Celler (A), Heiko Theißen (A), Alexander Hulpke (A), Willem de Graaf (A), Steve Linton (A), Werner Nickel (A), Greg Gamble (AM)

Factor groups
Alexander Hulpke (AM)

Finitely presented groups
Volkmar Felsch (AM), Alexander Hulpke (AM), Martin Schoenert (A)

Finitely presented monoids and semigroups
Isabel Araújo (AM), Derek Holt (A), Alexander Hulpke (A), Götz Pfeiffer (A), Andrew Solomon (AM)

Group actions
Heiko Theißen (A) and Alexander Hulpke (AM)

Homomorphism search
Alexander Hulpke (AM)

Homomorphisms for finitely presented groups
Alexander Hulpke (AM)

Intersection of subgroups of finite pc groups
Frank Celler (A), Bettina Eick (AM)

Irreducible Modules over finite fields for finite pc groups
Bettina Eick (AM)

Isomorphism testing with random methods
Hans Ulrich Besche (AM), Bettina Eick (AM)

Multiplier and Schur cover
Werner Nickel (AM), Alexander Hulpke (AM)

One-Cohomology and Complements
Frank Celler (A) and Alexander Hulpke (AM)

Partition Backtrack algorithm
Heiko Theißen (A), Alexander Hulpke (M)

Permutation group composition series
Ákos Seress (AM)

Permutation group homomorphisms
Ákos Seress (AM), Heiko Theißen (A), Alexander Hulpke (M)

Permutation Group Pcgs
Heiko Theißen (A), Alexander Hulpke (M)

Primitive groups library
Heiko Theißen (A), Alexander Hulpke (M)

Properties and attributes of finite pc groups
Frank Celler (A), Bettina Eick (AM)

Random Schreier-Sims
Ákos Seress (AM)

Rational Functions
Frank Celler (A) and Alexander Hulpke (AM)

Semigroup relations
Isabel Araujo (A), Robert F. Morse (AM), Andrew Solomon (A)

Special Pcgs for finite pc groups
Bettina Eick (AM)

Stabilizer Chains
Ákos Seress (AM), Heiko Theißen (A), Alexander Hulpke (M)

Strings and Characters
Martin Schönert (A), Frank Celler (A), Thomas Breuer (A), Frank Lübeck (AM)

Subgroup lattice
Martin Schönert (A), Alexander Hulpke (AM)

Subgroup lattice for solvable groups
Alexander Hulpke (AM)

Subgroup presentations
Volkmar Felsch (AM)

The Help System
Frank Celler (A), Frank Lübeck (AM)

Tietze transformations
Volkmar Felsch (AM)

Transformation semigroups
Isabel Araujo (A), Robert Arthur (A), Robert F. Morse (AM), Andrew Solomon (A)

Transitive groups library
Alexander Hulpke (AM)

Two-cohomology and extensions of finite pc groups
Bettina Eick (AM)

Lie algebras
Thomas Breuer (A), Craig Struble (A), Juergen Wisliceny (A), Willem A. de Graaf (AM)

GAP for MacOS
Burkhard Höfling (AM)

Contents

Copyright Notice

1 About the New Features Manual

This is a supplementary volume to the four main parts of the GAP documentation: the GAP **Reference Manual**, which describes all the main features of GAP for users, the GAP **Tutorial**, a beginner's introduction to GAP, **Programming in** GAP and **Extending** GAP, which provide information for those who want to write their own GAP extensions.

This manual, **New Features for Developers**, describes certain features of GAP, which meet the following conditions:

- They are **new**. Usually they were introduced at the last major release of GAP
- They are likely to be of more interest to GAP programmers and package developers than to interactive users
- We wish to retain the freedom to make some changes in them at the time of the next release

We would encourage users to employ these features in their own GAP programs or packages, but ask them to let us know that they are doing so. We will then invite feedback from them, and, as we approach the next release, discuss with them any changes to the features that might be desirable for inclusion in the next release. Unless substantial problems are found, we would normally expect to move the documentation into the reference manual at that time, and regard the documented behaviour as fixed from that time onwards.

2 Dictionaries and General Hash Tables (preliminary)

People and computers spend a large amount of time with searching. Dictionaries are an abstract data structure which facilitates searching for certain objects. An important way of implementing dictionaries is via hash tables.

The functions and operations described in this chapter have been added very recently and are still undergoing development. It is conceivable that names of variants of the functionality might change in future versions. If you plan to use these functions in your own code, please contact us.

2.1 Dictionaries

1 ▶ `IsDictionary(` *obj* `)` C

A dictionary is a growable collection of objects that permits to add objects (with associated values) and to check whether an object is already known.

2 ▶ `IsLookupDictionary(` *obj* `)` C

A **lookup dictionary** is a dictionary, which permits not only to check whether an object is contained, but also to retrieve associated values, using the operation `LookupDictionary`.

3 ▶ `KnowsDictionary(` *dict*`,` *key* `)` O

checks, whether *key* is known to the dictionary *dict*, and returns `true` or `false` accordingly. *key* **must** be an object of the kind for which the dictionary was specified, otherwise the results are unpredictable.

4 ▶ `LookupDictionary(` *dict*`,` *key* `)` O

looks up *key* in the lookup dictionary *dict* and returns the associated value. If *key* is not known to the dictionary, `fail` is returned.

There are several ways how dictionaries are implemented: As lists, as sorted lists, as hash tables or via binary lists. A user however will just have to call `NewDictionary` and obtain a "suitable" dictionary for the kind of objects she wants to create. It is possible however to create hash tables (see 2.3) and dictionaries using binary lists (see 2.1.6).

5 ▶ `NewDictionary(` *obj*`,` *look*`[,` *objcoll*`] )` F

creates a new dictionary for objects such as *obj*. If *objcoll* is given the dictionary will be for objects only from this collection, knowing this can improve the performance. If *objcoll* is given, *obj* may be replaced by `false`, i.e. no sample object is needed.

The function tries to find the right kind of dictionary for the basic dictionary functions to be quick. If *look* is `true`, the dictionary will be a lookup dictionary, otherwise it is an ordinary dictionary.

The use of two objects, *obj* and *objcoll* to parametrize the objects a dictionary is able to store might look confusing. However there are situations where either of them might be needed:

The first situation is that of objects, for which no formal "collection object" has been defined. A typical example here might be subspaces of a vector space. GAP does not formally define a "Grassmannian" or anything else to represent the multitude of all subspaces. So it is only possible to give the dictionary a "sample object".

The other situation is that of an object which might represent quite varied domains. The permutation $(1, 10^6)$ might be the nontrivial element of a cyclic group of order 2, it might be a representative of S_{10^6}. In the first situation the best approach might be just to have two entries for the two possible objects, in the second situation a much more elaborate approach might be needed.

An algorithm that creates a dictionary will usually know a priori, from what domain all the objects will be, giving this domain permits to use a more efficient dictionary.

This is particularly true for vectors. From a single vector one cannot decide whether a calculation will take place over the smallest field containing all its entries or over a larger field.

As there are situations where the approach via binary lists is explicitly desired, such dictionaries can be created deliberately.

6 ► `DictionaryByPosition(` *list*`,` *lookup* `)` F

creates a new (lookup) dictionary which uses `PositionCanonical` in *list* for indexing. The dictionary will have an entry *dict*`!.blist` which is a bit list corresponding to *list* indicating the known If *look* is `true`, the dictionary will be a lookup dictionary, otherwise it is an ordinary dictionary.

2.2 General Hash Tables

This chapter describes hash tables for general objects. We hash by keys and also store a value. Keys cannot be removed from the table, but the corresponding value can be changed. Fast access to last hash index allows you to efficiently store more than one array of values – this facility should be used with care.

This code works for any kind of object, provided you have a DenseIntKey or KeyIntSparse method to convert the key into a positive integer. These methods should ideally be implemented efficiently in the core.

Note that, for efficiency, it is currently impossible to create a hash table with non-positive integers.

2.3 General hash table definitions and operations

1 ► `IsHash(` *obj* `)` C

The category of hash tables for arbitrary objects (provided an `IntKey` function is defined).

2 ► `PrintHashWithNames(` *hash*`,` *keyName*`,` *valueName* `)` O

Print a hash table with the given names for the keys and values.

3 ► `GetHashEntry(` *hash*`,` *key* `)` O

If the key is in hash, return the corresponding value. Otherwise return fail. Note that it is not a good idea to use fail as a value.

4 ► `AddHashEntry(` *hash*`,` *key*`,` *value* `)` O

Add the key and value to the hash table.

5 ► `RandomHashKey(` *hash* `)` O

Return a random Key from the hash table (Random returns a random value).

6 ► `HashKeyEnumerator(` *hash* `)` O

Enumerates the keys of the hash table (Enumerator enumerates values).

2.4 Hash keys

The crucial step of hashing is to transform key objects into integers such that equal objects produce the same integer.

1 ► `TableHasIntKeyFun(` *hash* `)` P

If this filter is set, the hash table has an `IntKey` function in its component *hash*`!.intKeyFun`.

The actual function used will vary very much on the type of objects. However GAP provides already key functions for some commonly encountered objects.

2 ► `DenseIntKey(` *objcoll*, *obj* `)` O

returns a function that can be used as hash key function for objects such as *obj* in the collection *objcoll*. *objcoll* typically will be a large domain. If the domain is not available, it can be given as `false` in which case the hash key function will be determined only based on *obj*. (For a further discussion of these two arguments see `NewDictionary`, section 2.1.5).

The function returned by `DenseIntKey` is guaranteed to give different values for different objects. If no suitable hash key function has been predefined, `fail` is returned.

3 ► `SparseIntKey(` *objcoll*, *obj* `)` O

returns a function that can be used as hash key function for objects such as *obj* in the collection *objcoll*. In contrast to `DenseIntKey`, the function returned may return the same key value for different objects. If no suitable hash key function has been predefined, `fail` is returned.

2.5 Dense hash tables

Dense hash tables are used for hashing dense sets without collisions, in particular integers. Stores keys as an unordered list and values as an array with holes. The position of a value is given by the attribute `IntKeyFun` or the function returned by `DenseIntKey`, and so KeyIntDense must be one-to-one.

1 ► `DenseHashTable( )` F

Construct an empty dense hash table. This is the only correct way to construct such a table.

2.6 Sparse hash tables

Sparse hash tables are used for hashing sparse sets. Stores keys as an array with fail denoting an empty position, stores values as an array with holes. Uses `HashFunct` applied to the `IntKeyFun` (respectively the result of calling `SparseIntKey`) of the key. DefaultHashLength is the default starting hash table length; the table is doubled when it becomes half full.

1 ► `SparseHashTable( [`*intkeyfun*`] )` F

Construct an empty sparse hash table. This is the only correct way to construct such a table. If the argument *intkeyfun* is given, this function will be used to obtain numbers for the keys passed to it.

2 ► `GetHashEntryIndex(` *hash*, *key* `)` F

If the key is in hash, return its index in the hash array.

3 ► `DoubleHashArraySize(` *hash* `)` F

Double the size of the hash array and rehash all the entries. This will also happen automatically when the hash array is half full.

In sparse hash tables, the integer obtained from the hash key is then transformed to an index position, this transformation is done using the hash function `HashFunct`:

4 ► `HashFunct(` *key*, *i*, *size* `)` F

This will be a good double hashing function for any reasonable KeyInt (see Cormen, Leiserson and Rivest, Introduction to Algorithms, 1e, p. 235).

2.7 Fast access to last hash index

These functions allow you to use the index of last hash access or modification. Note that this is global across all hash tables. If you want to have two hash tables with identical layouts, the following works: GetHashEntry(hashTable1, object); GetHashEntryAtLastIndex(hashTable2); These functions should be used with extreme care, as they bypass most of the inbuilt error checking for hash tables.

1 ► `GetHashEntryAtLastIndex(` *hash* `)` O

Returns the value of the last hash entry accessed.

2 ► `SetHashEntryAtLastIndex(` *hash*`,` *newValue* `)` O

Resets the value of the last hash entry accessed.

3 ► `SetHashEntry(` *hash*`,` *key*`,` *value* `)` O

Resets the value corresponding to *key*.

3 Quotient groups by homomorphisms (preliminary)

Given a group homomorphism, the cosets of its kernel correspond to elements in the image. Our hom coset representation stores the homomorphism and the element in the source group. The image is an attribute which is computed as necessary. Two cosets are equal if their images are the same. Where ever practical a coset is identified with its image. For example, if the homomorphism maps into a permutation group, the cosets are considered to be permutations. Since cosets can be multiplied, we can use them to form a quotient group. Any computation in this quotient group will be "shadowed" in the source group.

The functions and operations described in this chapter have been added very recently and are still undergoing development. It is conceivable that names of variants of the functionality might change in future versions. If you plan to use these functions in your own code, please contact us.

1 ► `IsHomCoset(` *obj* `)` C

`IsHomCoset` has one category for each kind of image (and corresponding representations).

2 ► `IsHomCosetToPerm(` *obj* `)` C

3 ► `IsHomCosetToPermRep(` *obj* `)` R

4 ► `IsHomCosetToMatrix(` *obj* `)` C

gdc - We need `HomCosetToMatrix` to be in same family as `Matrix`, so that GAP allows vector $*$ for `HomCosetToMatrix` and other algorithms that take elements of the `HomCosetToMatrix`. Unfortunately, I don't know how to set the family correctly for compatibility.

5 ► `IsHomCosetToMatrixRep(` *obj* `)` R

6 ► `IsHomCosetToFp(` *obj* `)` C

7 ► `IsHomCosetToFpRep(` *obj* `)` R

8 ► `IsHomCosetToTuple(` *obj* `)` C

9 ► `IsHomCosetToTupleRep(` *obj* `)` R

10 ► `IsHomCosetToAdditiveElt(` *obj* `)` C

Here the image is an ADDITIVE group of matrices.

11 ► `IsHomCosetToAdditiveEltRep(` *obj* `)` R

12 ► `IsHomCosetToObjectRep(` *obj* `)` R

The generic representation.

It also has one property for each kind of source.

13 ► `IsHomCosetOfPerm(` *obj* `)` P

14 ► `IsHomCosetOfMatrix(` *obj* `)` P

15 ► `IsHomCosetOfFp(` *obj* `)` P

16 ► `IsHomCosetOfTuple(` *obj* `)` P

17 ► `IsHomCosetOfAdditiveElt(` *obj* `)` P

3.1 Creating hom cosets and quotient groups

1 ► `HomCoset(` *hom*`,` *elt* `)` F

Creates a hom coset. It is better to use one of the `QuotientGroupBy...` functions.

2 ► `HomCosetWithImage(` *hom*`,` *srcElt*`,` *imgElt* `)` F

Creates a hom coset with given homomorphism *hom*, source element *srcElt* and image element *imgElt*. It is better to use one of the `QuotientGroupBy...` functions.

3 ► `QuotientGroupHom(` *hom* `)` A

The quotient group associated with the homomorphism *hom*. It is better to use one of the `Quotient-GroupBy...` functions.

4 ► `QuotientGroupByHomomorphism(` *hom* `)` F

The quotient group associated with the homomorphism *hom*.

5 ► `QuotientGroupByImages(` *srcGroup*`,` *rangeGroup*`,` *srcGens*`,` *imgGens* `)` F

creates a quotient group from the homomorphism which takes maps *srcGens*`[`*i*`]` in *srcGroup* to *imgGens*`[`*i*`]` in *rangeGroup*.

6 ► `QuotientGroupByImagesNC(` *srcGroup*`,` *rangeGroup*`,` *srcGens*`,` *imgGens* `)` F

Same as `QuotientGroupByImages` (see 3.1.5) but without checking that the homomorphism makes sense.

3.2 Operations on hom cosets

1 ► `Homomorphism(` *hcoset* `)` O
► `Homomorphism(` *Q* `)` O

The homomorphism of a hom coset *hcoset*, respectively a hom quotient group *Q*.

2 ► `SourceElt(` *hcoset* `)` O

The source element of a hom coset *hcoset*.

3 ► `ImageElt(` *hcoset* `)` A

The image element of a hom coset *hcoset*.

4 ► `CanonicalElt(` *hcoset* `)` A

A canonical element of a hom coset *hcoset*. Note that SourceElt may be different for non-identical equal cosets. `CanonicalElt` gives the same element for different representation of a coset. This will compute a chain for the range group if one does not already exist.

5 ► `Source( `*Q*` )` A

Source group of a hom quotient group Q.

6 ► `Range( `*Q*` )` A

Range group of a hom quotient group Q.

7 ► `ImagesSource( `*Q*` )` A

Image group of a hom quotient group Q.

4 Transversals of subgroups (preliminary)

This chapter describes the category of transversals of subgroups. This category has the following representations: `TransvBySchreierTree`, `TransvByHomomorphism`, `TransvByDirProd`, `TransvByTrivSubgrp`, `TransvBySiftFunct`.

The functions and operations described in this chapter have been added very recently and are still undergoing development. It is conceivable that names of variants of the functionality might change in future versions. If you plan to use these functions in your own code, please contact us.

4.1 General operations on transversals

Every kind of transversal has the following common operations/attributes: `Size`, `Enumerator`, `Iterator`, `Random`, `TransversalElt`, `SiftOneLevel`.

1 ▶ `TransversalElt(` *ss*, *elt* `)` O

for a transversal *ss* and group element *elt*, returns the representative of the coset containing the element *elt*. The representative is unique, i.e. `TransversalElt` will return the same thing for different elements of the same coset.

2 ▶ `SiftOneLevel(` *ss*, *g* `)` O

For a transversal *ss* and group element *g*, the following relationship with `TransversalElt` (see 4.1.1) defines `SiftOneLevel`:

`SiftOneLevel(`*ss*, *g*`) = ` *g* ` * TransversalElt(`*ss*, *g*`)`

For some kinds of transversal `TransversalElt` is more efficient, for others `SiftOneLevel` is.

4.2 Transversals by Schreier tree

For transversals of stabiliser subgroups, we store a Schreier tree to allow us to find transversal elements.

Note: `SiftOneLevel` is more efficient that `TransversalElt`.

Transversals can be extended as more generators are found for the stabiliser. Orbit generators are generators for the original group, stored separately so we can add extra generators to form a shallower tree. Orbits are stored as hash tables.

1 ▶ `SchreierTransversal(` *basePoint*, *Action*, *strongGens* `)` F

creates a transversal by Schreier tree for the subgroup stabilising the point *basePoint* (an object, typically an integer or vector) inside the group generated by *strongGens* (a list of strong generators for the group). This is the only correct way to create a transversal by Schreier tree.

2 ▶ `OrbitGenerators(` *ss* `)` O

The elements used to compute the orbit *ss*. These will be generators for the larger group, however there will often be redundancies to keep the Schreier tree shallow.

3 ▶ `OrbitGeneratorsInv(` *ss* `)` O

Inverses of the orbit generators of the orbit *ss*.

4 ▶ `BasePointOfSchreierTransversal(` *ss* `)` O

The base point of transversal by Schreier tree *ss*, i.e. the point stabilised.

5 ▶ `One(` *ss* `)` A

The identity of group *ss*.

6 ▶ `ExtendSchreierTransversal(` *st*, *newGens* `)` F
▶ `ExtendSchreierTransversal(` *st*, *newGens*, *newGensInv* `)` F

Extend a transversal by Schreier tree *st* with new generators *newGens*.

7 ▶ `ExtendSchreierTransversalShortCube(` *ss*, *newGens* `)` F
▶ `ExtendSchreierTransversalShortCube(` *ss*, *newGens*, *newGensInv* `)` F

gdc - Ideally, `ExtendSchreierTransversal` should be a field of the Schreier tree, chosen by `Schreier-Transversal()`.

gdc - This is the new function with the cube control tree.

EXPERIMENTAL IDEA: IT WOULD NEED TO BE TUNED. NOT CURRENTLY COMPETITIVE WITH METHOD BELOW.

8 ▶ `ExtendSchreierTransversalShortTree(` *ss*, *newGens* `)` F
▶ `ExtendSchreierTransversalShortTree(` *ss*, *newGens*, *newGensInv* `)` F

gdc - This is the original function with the traditional control tree

BASED ON: [CF94] “A Random Base Change Algorithm for Permutation Groups”, G. Cooperman and L. Finkelstein, J. of Symbolic Computation 17, 1994, pp. 513–528

9 ▶ `CompleteSchreierTransversal(` *ss* `)` F

Complete the transversal. In order to ensure that the Schreier tree does not become too deep, the `Extend...` functions do not complete the transversal. Rather they extend it by depth one.

10 ▶ `PreferredGenerators(` *ss* `)` A

returns the preferred generators of the transversal by Schreier tree *ss*. The preferred generators are always used first when computing the Schreier tree.

11 ▶ `SchreierTreeDepth(` *ss* `)` F

The depth of Schreier tree *ss*.

4.3 Transversals by homomorphic images

For the transversal of the kernel of a homomorphism, a quotient group for the kernel of a homomorphism is stored. Transversal elements are computed by finding a chain for the image group and doing shadowed stripping.

Note: `TransversalElt` is more efficient that `SiftOneLevel`.

1 ▶ `HomTransversal(` *h* `)` F

creates a hom transversal for the homomorphism *h*.

2 ▶ `Homomorphism(` *homtr* `)` O

The homomorphism of hom transversal *homtr*.

3 ▶ `QuotientGroup(` *homtr* `)` A

The quotient group of hom transversal *homtr*.

4 ▶ `ImageGroup(` *homtr* `)` O

The image group of hom transversal *homtr*.

4.4 Transversals by direct products

Stores projection and injection for a direct product. The chain subgroup is the kernel of the projection.

1 ▶ `Projection(` *dpt* `)` O

The projection of the direct product transversal *dpt*.

2 ▶ `Injection(` *dpt* `)` O

The injection of a direct product transversal *dpt*.

4.5 Transversals by Trivial subgroups

For use when our group is small enough to enumerate.

1 ▶ `TransversalByTrivial(` G `)` F

returns a transversal by trivial subgroup for the group G.

4.6 Transversals by sift functions

Given a group, subgroup, and sift function from group to subgroup that is constant on cosets, this defines a transversal. One typically prefers a normalized sift function that is the the identity map on subgroups. For situations when there is a non-group theoretic method for computing the transversal element, e.g. using row reduction for the stabiliser of an invariant subspace.

Note: `SiftOneLevel` is more efficient than `TransversalElt`.

1 ▶ `TransversalBySiftFunction(` *supergroup*, *subgroup*, *sift* `)` F

returns a transversal by sift function.

5 Chains of subgroups (preliminary)

The functions and operations described in this chapter have been added very recently and are still undergoing development. It is conceivable that names of variants of the functionality might change in future versions. If you plan to use these functions in your own code, please contact us.

Data structures for storing general group chains. Note that this does **not** replace `StabChain`. The group attribute `ChainSubgroup(`*G*`)` stores the next group down in the chain (i.e. the structure is recursive). `Chain-Subgroup(`*G*`)` should have an attribute `Transversal` which describes a transversal of `ChainSubgroup(`*G*`)` in *G*, as in `gptransv.[gd,gi]`.

The command `ChainSubgroup` will use the default method for computing chains – currently this is random Schreier-Sims, unless the group is nilpotent. **Warning:** This algorithm is Monte-Carlo. `ChainSubgroup` is mutable, since it may start as the trivial subgroup, and then grow as elements are sifted in, and some stick. This allows us to do, if we want, things like:

```
SetChainSubgroup(G, ClosureGroup(ChainSubgroup(G), siftee) );
```

Whether this code is used instead of previous methods is determined by 4 variables which control the behaviour of the filter `IsChainTypeGroup`. See the file `gap.../lib/grpchain.gd` for details.

1 ► `IsChainTypeGroup(` *G* `)` P

returns `true` if the group *G* is "chain type", i.e. it is the kind of group where computations are best done with chains.

2 ► `ChainSubgroup(` *G* `)` AM

Computes the chain, if necessary, and returns the next subgroup in the chain. The current default is to use the random Schreier-Sims algorithm, unless the group is known to be nilpotent, in which case `MakeHomChain` is used.

3 ► `Transversal(` *G* `)` A

The transversal of the group *G* in the previous subgroup of the chain.

4 ► `IsInChain(` *G* `)` O

A group *G* is in a chain if it has either a `ChainSubgroup` or a `Transversal`.

5 ► `GeneratingSetIsComplete(` *G* `)` P

returns `true` if the generating set of the group *G* is complete. For example, for a stabiliser subgroup this is true if our strong generators have been verified.

6 ► `SiftOneLevel(` *G*, *g* `)` O

Sift *g* though one level of the chain.

7 ► `Sift(` *G*, *g* `)` O

Sift *g* through the entire chain.

8 ▶ `SizeOfChainOfGroup(` *G* `)` F

Uses the chain to compute the size of a group. Unlike `Size(`*G*`)`, this does not set the `Size` attribute, which is useful if the chain is not known to be complete.

9 ▶ `TransversalOfChainSubgroup(` *G* `)` F

Returns the transversal of the next group in the chain, inside *G*.

10 ▶ `ChainStatistics(` *G* `)` F

Returns a record containing useful statistics about the chain of *G*.

11 ▶ `HasChainHomomorphicImage(` *G* `)` F

Does *G* have a chain subgroup derived from a homomorphic image? This will be `false` for stabiliser, trivial, and sift function chain subgroups. It will be true for homomorphism and direct product chain subgroups.

12 ▶ `ChainHomomorphicImage(` *G* `)` F

Returns the chain homomorphic image, or `fail` if no such image exists.

5.1 Stabiliser chain subgroups

1 ▶ `BaseOfGroup(` *G* `)` A

If the group *G* has a chain consisting entirely of stabiliser subgroups, then this command returns the base as a list. This command does not compute a base, however.

2 ▶ `ExtendedGroup(` *G*, *g* `)` O

Add a new Schreier generator for *G*.

3 ▶ `StrongGens(` *G* `)` F

Returns a list of generating sets for each level of the chain.

4 ▶ `ChainSubgroupByStabiliser(` *G*, *basePoint*, *Action* `)` F

Form a chain subgroup by stabilising *basePoint* under the given action. The subgroup will start with no generators, and will have a transversal by Schreier tree.

5 ▶ `OrbitGeneratorsOfGroup(` *G* `)` A

Generators used to compute the orbit of *G*. Used by `baseim.[gd,gi]`.

6 ▶ `RandomSchreierSims(` *G* `)` F

The random Schreier-Sims algorithm.

7 ▶ `ChangedBaseGroup(` *G* `)` F

We assume we have a chain for *G*, which gives a complete BSGS. We are given a new base *newBase* and wish to find strong generators for it. Options are the same as for random Schreier-Sims. Note that this function does not modify *G*, but returns a new group, isomorphic to *G* with the specified base.

5.2 Hom coset chain subgroups

1 ► `ChainSubgroupByHomomorphism(` *hom* `)` F

Form a chain subgroup by the kernel of *hom*. The subgroup will start with no generators, and will have a *hom* transversal.

2 ► `ChainSubgroupByProjectionFunction(` *G*`,` *kernelSubgp*`,` *imgSubgp*`,` *projFnc* `)` F

When the homomorphism of a quotient group is a projection, then there is an internal semidirect product, for which `TransversalElt()` has a direct implementation as the projection. *hom* will be the projection, and *elt* `-> ImageElm(`*hom*`,` *elt*`)` is the map.

3 ► `QuotientGroupByChainHomomorphicImage(` *quo*`[,` *quo2*`] )` F

This function deals with quotient groups of quotient groups in a chain.

4 ► `ChainSubgroupQuotient(` *G* `)` A

The quotient by the chain subgroup.

5 ► `MakeHomChain(` *G* `)` O

Computes a chain of subgroups for the group G which are kernels of homomorphisms. Currently only implemented for nilpotent groups. We use the algorithm of E. Luks, Computing in Solvable Matrix Groups, FOCS/STOC.

5.3 Direct product chain subgroups

1 ► `ChainSubgroupByDirectProduct(` *proj*`,` *inj* `)` F

Form a chain subgroup by internal direct product.

2 ► `ChainSubgroupByPSubgroupOfAbelian(` *G*`,` *p* `)` F

G is an abelian group, p a prime involved in G. Form a direct sum chain where the subgroup is the p-prime part of G.

5.4 Trivial chain subgroups and sift function chain subgroups

1 ► `ChainSubgroupByTrivialSubgroup(` *G* `)` F

Form a chain subgroup by enumerating the group.

2 ► `ChainSubgroupBySiftFunction(` *G*`,` *subgroup*`,` *siftFnc* `)` F

Form a chain subgroup using a sift function.

Bibliography

[CF94] Gene Cooperman and Larry Finkelstein. A random base change algorithm for permutation groups. *J. Symbolic Comput.*, 17(6):513–528, 1994.

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".