recog

A collection of group recognition methods

1.3.2

15 April 2018

Max Neunhöffer

Ákos Seress

Nurullah Ankaralioglu

Peter Brooksbank

Frank Celler

Stephen Howe

Maska Law

Steve Linton

Gunter Malle

Alice Niemeyer

Eamonn O'Brien

Colva M. Roney-Dougal

Max Neunhöffer
Email: max@9hoeffer.de
Homepage: http://www-groups.mcs.st-and.ac.uk/~neunhoef
Address: Gustav-Freytag-Straße 40
50354 Hürth
Germany

Nurullah Ankaralioglu
Email: ankarali@atauni.edu.tr

Peter Brooksbank
Email: pbrooksb@bucknell.edu
Homepage: http://www.facstaff.bucknell.edu/pbrooksb/
Address: Peter A. Brooksbank
Mathematics Department
Bucknell University
Lewisburg, PA 17837
USA

Frank Celler
Email: frank@celler.de
Homepage: http://www.celler.de/

Stephen Howe

Address: Unknown

Maska Law
Email: maska@maths.uwa.edu.au
Address: Maska Law
University of Western Australia
School of Mathematics and Statistics
35 Stirling Highway
Crawley 6009
Western Australia

Steve Linton
Email: sal@cs.st-andrews.ac.uk
Homepage: http://www-circa.mcs.st-and.ac.uk/~sal/
Address: School of Computer Science
Jack Cole Building
North Haugh
St Andrews, Fife KY16 9SX
Scotland, UK

Gunter Malle

Email: malle@mathematik.uni-kl.de
Homepage: http://www.mathematik.uni-kl.de/~malle/

Alice Niemeyer
Email: alice@maths.uwa.edu.au
Homepage: http://www.maths.uwa.edu.au/~alice/
Address: Alice C. Niemeyer
University of Western Australia
School of Mathematics and Statistics
35 Stirling Highway
Crawley 6009
Western Australia

Eamonn O'Brien
Email: obrien@math.auckland.ac.nz
Homepage: http://www.math.auckland.ac.nz/~obrien/

Colva M. Roney-Dougal
Email: colva@mcs.st-and.ac.uk
Homepage: http://www-groups.mcs.st-and.ac.uk/~colva
Address: School of Mathematics and Statistics
Mathematical Institute
North Haugh
St Andrews, Fife KY16 9SS
Scotland, UK

Copyright

Contents

Chapter 1

Introduction

1.1 Philosophy

This package is about group recognition. It provides a generic framework to implement methods of group recognition, regardless of what computational representation is used. This means, that the code in this package is useful at least for permutation groups, matrix groups and projective groups. The setup is described in [NS06].

The framework allows to build composition trees and handles the builtup and usage of these trees in a generic way. It also contains a method selection (described in Chapter 4) that allows install recognition methods in a convenient way and that automatically tries to try the different available methods in a sensible order.

1.2 Overview over this manual

Chapter 2 describes the installation of this package.

Chapter 3 describes the generic, recursive procedure used for group recognition throughout this package. At the heart of this procedure is the definition of “FindHomomorphism” methods, which is also described in that chapter. For the choice of the right method for finding a homomorphism (or an isomorphism) we use another generic procedure, the “method selection” which is not to be confused with the GAP method selection.

Our own method selection is described in detail in Chapter 4, because it is interesting in its own right and might be useful in other circumstances.

Chapter 6 describes the avalable “FindHomomorphism” methods.

Chapter 5 explains what one can do with a completed recognition tree.

Finally, Chapter 7 shows instructive examples for the usage of this package.

1.3 Feedback

For bug reports, feature requests and suggestions, please use our issue tracker.

Chapter 2

Installation of the recog-Package

To install this package just extract the package's archive file to the GAP `pkg` directory.

By default the recog package is not automatically loaded by GAP when it is installed. You must load the package with `LoadPackage("recog");` before its functions become available.

Please, send us an e-mail if you have any questions, remarks, suggestions, etc. concerning this package. Also, we would like to hear about applications of this package.

Chapter 3

Group Recognition

This chapter describes a generic framework for group recognition. The basic problem is, we want to solve the constructive membership problem: given any $g \in G$, $G = \langle X \rangle$, write a straight line program (SLP) from X to g, for $g \notin G$ (in the situation that G is naturally embedded into some bigger group), the algorithm should fail. This is usually done by constructing some nice generators (and then writing an SLP from the nice generators to g and concatenating with an SLP from X to the nice generators). Often, for efficiency reasons, we will just store the nice generators and then only be interested in the SLP from those to g. The framework presented here deals with exactly this process.

The generic framework was designed having three situations in mind: permutation groups, matrix groups and projective groups. Although the methods used are quite different for those cases, there is a common pattern in the procedure of recognition. Namely, first we have to find a homomorphism, solve the constructive membership problem recursively in image and kernel, then put it together. The recursion ends in groups where we can solve the constructive membership problem directly. The general framework reflects this idea and separates it from the rest of the recognition methods.

Solution of the constructive membership problem comes in two stages: first a "recognition phase" and then a "verification phase". The recognition phase usually consists of randomised algorithms with certain error or failure probabilities. The result is some kind of "recognition information" that will describe the group already very well, but which is not yet proven to be correct. However, one can already write arbitrary elements in the group as product of the given generators. In the verification phase a presentation of the group is calculated, thereby proving that the group generated by the given generators is in fact isomorphic to the group described by the recognition information. In many cases the verification phase will be much more expensive than the recognition phase.

In the following sections, we describe the generic framework. We begin with a technical description of the recursive procedure and describe then the way methods to find homomorphism have to be implemented. Finally, we have four sections in which we collect conventions for the recognition of different types of groups.

No actual recognition methods are implemented in this package. See the recog package for an implementation and description of them.

3.1 The recursive procedure

As explained at the beginning of this section, the heart of the recognition procedure is a function called `RecogniseGeneric` (3.1.1) which gets a GAP group object and returns a so-called "recognition info record" (see Subsection 3.2 for details). Success or failure will be indicated by this record being in the filter `IsReady` (3.2.4) or not.

To know how to find homomorphisms the function gets as another argument a database of methods (see Section 3.3 for a description of the setup for methods for finding homomorphisms and Section 4.1 in Chapter 4 for details about method databases). This database will be different according to the type of group in question.

To describe the algorithm executed by `RecogniseGeneric` (3.1.1) we first summarise it in steps:

1. Create a new, empty recognition info record.

2. Use the database of `FindHomomorphism` methods and the method selection procedure described in Chapter 4 to try to find a homomorphism onto a smaller group or an isomorphism onto another known group. Terminate with failure if this does not work.

3. If an isomorphism is found or a method somehow else recognises the group in question, such that we can write elements as straight line programs in the generators from now on, then make the recognition info record a leaf of the recognition tree and return success.

4. Otherwise the function sets up all the data for the homomorphism and calls itself with the image of the homomorphism. Note that this might use another database of recognition methods because the homomorphism might change the representation of the group.

5. After successful recognition of the factor group the procedure has to recognise the kernel of the homomorphism. The first step for this is to find generators. If they are not already known from the `FindHomomorphism` method, they are created by producing random elements in the group, mapping them through the homomorphism, writing them as a straight line program in the images of the generators and applying this straight line program to the original generators. The quotient of the random element and the result of the straight line program lies in the kernel of the homomorphism. After creating 20 (FIXME: is 20 correct?) random generators of the kernel we assume for the moment that they generate the kernel.

6. The function `RecogniseGeneric` (3.1.1) can now call itself for the kernel. After successful recognition of the kernel all the data for the node is completed and success is returned.

7. The function `RecogniseGeneric` (3.1.1) now acquires preimages of the nice generators behind the homomorphism and appends the nice generators of the kernel. This list of generators is now the list of nice generators for the current node.

Note that with the collected data one can write arbitrary elements of the group as a straight line program in the generators as follows:

1. Map the element through the homomorphism.

2. Write the element in the factor group as a product of the nice generators in the factor group.

3. Apply the resulting straight line program to the preimages of those nice generators and calculate the quotient, which will now lie in the kernel.

4. Write the kernel element as a straight line program in the kernel generators.

5. Assemble both straight line programs to one bigger straight line program (which is now in terms of our own nice generators) and return it.

If this procedure fails in the fourth step, this indicates that our random generators for the kernel did not yet generate the full kernel and makes further recognition steps necessary. This will not happen after a successful verification phase.

The latter procedure to write elements as straight line programs in the generators is implemented in the function `SLPforElementGeneric` (3.3.2) which will be called automatically if one calls the `SLPforElement` (3.2.14) function of the resulting recognition info record (see `slpforelement` (3.2.13)).

It is now high time to give you the calling details of the main recursive recognition function:

3.1.1 RecogniseGeneric

▷ `RecogniseGeneric(`*H, methoddb, depth[, knowledge]*`)` (function)
▷ `RecognizeGeneric(`*H, methoddb, depth[, knowledge]*`)` (function)

Returns: `fail` for failure or a recognition info record.

H must be a GAP group object, *methoddb* must be a method database in the sense of Section 4.1 containing `FindHomomorphism` methods in the sense of Section 3.3. *depth* is an integer which measures the depth in the recognition tree. It will be increased by one for each step we go into the tree. The top level has depth 0. *knowledge* is an optional record the components of which are copied into the new recognition info record which is created for the group *H*. Especially the component `hints` can contain a list of additional find homomorphism methods (described by records as in Section 4.1) which is prepended to the method database in *methoddb* before the recognition starts. This feature is intended to give hints about prior knowledge about which find homomorphism method might succeed.

The function performs the algorithm described above and returns either `fail` in case of failure or a recognition info record in case of success. For the content and definition of recognition info records see Section 3.2.

The user will usually not call this function directly, but will use the following convenience functions:

3.1.2 RecognisePermGroup

▷ `RecognisePermGroup(`*H*`)` (function)
▷ `RecognizePermGroup(`*H*`)` (function)

Returns: `fail` for failure or a recognition info record.

H must be a GAP permutation group object. This function calls `RecogniseGeneric` (3.1.1) with the method database used for permutation groups, which is stored in the global variable `FindHomDbPerm` (3.1.6), and no prior knowledge.

3.1.3 RecogniseMatrixGroup

▷ `RecogniseMatrixGroup(`*H*`)` (function)
▷ `RecognizeMatrixGroup(`*H*`)` (function)

Returns: `fail` for failure or a recognition info record.

H must be a GAP matrix group object. This function calls `RecogniseGeneric` (3.1.1) with the method database used for matrix groups, which is stored in the global variable `FindHomDbMatrix` (3.1.9), and no prior knowledge.

3.1.4 RecogniseProjectiveGroup

▷ `RecogniseProjectiveGroup(H)` (function)
▷ `RecognizeProjectiveGroup(H)` (function)

Returns: `fail` for failure or a recognition info record.

H must be a GAP matrix group object. Since as of now no actual projective groups are implemented in the GAP library we use matrix groups instead. The recognition will however view the group as the projective group, i.e. the matrix group modulo its scalar matrices. This function calls `RecogniseGeneric` (3.1.1) with the method database used for projective groups, which is stored in the global variable `FindHomDbProjective` (3.1.12), and no prior knowledge.

3.1.5 RecogniseGroup

▷ `RecogniseGroup(H)` (function)
▷ `RecognizeGroup(H)` (function)

Returns: `fail` for failure or a recognition info record.

H must be a GAP group object. This function automatically dispatches to one of the two previous functions `RecognisePermGroup` (3.1.2), or `RecogniseMatrixGroup` (3.1.3), according to the type of the group *H*. Note that since currently there is no implementation of projective groups in the GAP library, one cannot recognise a matrix group *H* as a projective group using this function.

3.1.6 FindHomDbPerm

▷ `FindHomDbPerm` (global variable)

This list contains the methods for finding homomorphisms for permutation group recognition that are stored in the record `FindHomMethodsPerm` (3.1.7). As described in Section 4.1 each method is described by a record. The list is always sorted with respect to decreasing ranks. The order in this list tells in which order the methods should be applied. Use `AddMethod` (4.1.1) to add methods to this database.

3.1.7 FindHomMethodsPerm

▷ `FindHomMethodsPerm` (global variable)

In this global record the functions that are methods for finding homomorphisms for permutation group recognition are stored. We collect them all in this record such that we do not use up too many global variable names.

3.1.8 SLPforElementFuncsPerm

▷ `SLPforElementFuncsPerm` (global variable)

This global record holds the functions that are methods for writing group elements as straight line programs (SLPs) in terms of the generators after successful permutation group recognition. We collect them all in this record such that we do not use up too many global variable names.

3.1.9 FindHomDbMatrix

▷ `FindHomDbMatrix` (global variable)

This list contains the methods for finding homomorphisms for matrix group recognition that are stored in the record `FindHomMethodsMatrix` (3.1.10). As described in Section 4.1 each method is described by a record. The list is always sorted with respect to decreasing ranks. The order in this list tells in which order the methods should be applied. Use `AddMethod` (4.1.1) to add methods to this database.

3.1.10 FindHomMethodsMatrix

▷ `FindHomMethodsMatrix` (global variable)

In this global record the functions that are methods for finding homomorphisms for matrix group recognition are stored. We collect them all in this record such that we do not use up too many global variable names.

3.1.11 SLPforElementFuncsMatrix

▷ `SLPforElementFuncsMatrix` (global variable)

This global record holds the functions that are methods for writing group elements as straight line programs (SLPs) in terms of the generators after successful matrix group recognition. We collect them all in this record such that we do not use up too many global variable names.

3.1.12 FindHomDbProjective

▷ `FindHomDbProjective` (global variable)

This list contains the methods for finding homomorphisms for projective group recognition that are stored in the record `FindHomMethodsProjective` (3.1.13). As described in Section 4.1 each method is described by a record. The list is always sorted with respect to decreasing ranks. The order in this list tells in which order the methods should be applied. Use `AddMethod` (4.1.1) to add methods to this database.

3.1.13 FindHomMethodsProjective

▷ `FindHomMethodsProjective` (global variable)

In this global record the functions that are methods for finding homomorphisms for projective group recognition are stored. We collect them all in this record such that we do not use up too many global variable names.

3.1.14 SLPforElementFuncsProjective

▷ `SLPforElementFuncsProjective` (global variable)

This global record holds the functions that are methods for writing group elements as straight line programs (SLPs) in terms of the generators after successful projective group recognition. We collect them all in this record such that we do not use up too many global variable names.

3.1.15 TryFindHomMethod

▷ `TryFindHomMethod(`*`H, method, projective`*`)` (function)

Returns: `fail` or `false` or a recognition info record.

Use this function to try to run a given find homomorphism method *`method`* on a group *`H`*. Indicate by the boolean *`projective`* whether or not the method works in projective mode. For permutation groups, set this to `false`. The result is either `fail` or `false` if the method fails or a recognition info record `ri`. If the method created a leaf then `ri` will be a leaf, otherwise it will have the attribute `Homom` (3.2.6) set, but no factor or kernel have been created or recognised yet. You can use for example the methods in `FindHomMethodsPerm` (3.1.7) or `FindHomMethodsMatrix` (3.1.10) or `FindHomMethodsProjective` (3.1.13) as the *`method`* argument.

3.2 Recognition info records

A recognition info record is a GAP positional object. It is a member of the family

3.2.1 RecognitionInfoFamily

▷ `RecognitionInfoFamily` (family)

and is in the category

3.2.2 IsRecognitionInfo

▷ `IsRecognitionInfo` (Category)

and is `IsAttributeStoringRep` (**Reference: IsAttributeStoringRep**), such that we can define attributes for it, the values of which are stored once they are known. A recognition info record always represents a whole binary tree of such records, see the attributes `RIFac` (3.2.9) and `RIKer` (3.2.10) below.

The following filters are defined for recognition info records:

3.2.3 IsLeaf

▷ `IsLeaf` (Flag)

This flag indicates, whether or not a recognition info record represents a leaf in the recognition tree. If it is not set, one finds at least one of the attributes `RIFac` (3.2.9) and `RIKer` (3.2.10) set for the corresponding node. This flag is normally reset and has to be set by a find homomorphism method to indicate a leaf.

3.2.4 IsReady

▷ `IsReady` (Flag)

This flag indicates during the recognition procedure, whether a node in the recognition tree is already completed or not. It is mainly set for debugging purposes during the recognition. However, if the recognition fails somewhere in a leaf, this flag is not set and all nodes above will also not have this flag set. In this way one can see whether the recognition failed and where the problem was.

The following attributes are defined for recognition info records:

3.2.5 Grp

▷ `Grp(`*ri*`)` (attribute)

The value of this attribute is the group that is to be recognised by this recognition info record *ri*. This attribute is always present during recognition and after completion. Note that the generators of the group object stored here always have a memory attached to them, such that elements that are generated from them remember, how they were acquired.

3.2.6 Homom

▷ `Homom(`*ri*`)` (attribute)

The value of this attribute is the homomorphism that was found from the group described by the recognition info record *ri* as a GAP object. It is set by a find homomorphism method that succeeded to find a homomorphism (or isomorphism). It does not have to be set in leaf nodes of the recognition tree.

3.2.7 NiceGens

▷ `NiceGens(`*ri*`)` (attribute)

The value of this attribute must be set for all nodes and contains the nice generators. The `SLPforElement` (3.2.14) function of the node will write its straight line program in terms of these nice generators. For leaf nodes, the find homomorphism method is responsible to set the value of `NiceGens`. By default, the original generators of the group at this node are taken. For a homomorphism (or isomorphism), the `NiceGens` will be the concatenation of preimages of the `NiceGens` of the factor group (see `pregensfac` (3.2.8)) and the `NiceGens` of the kernel. A find homomorphism method does not have to set `NiceGens` if it finds a homomorphism. Note however, that such a find homomorphism method has to ensure somehow, that preimages of the `NiceGens` of the factor group can be acquired. See `calcnicegens` (3.2.17), `CalcNiceGens` (3.2.20) and `slptonice` (3.2.21) for instructions.

3.2.8 pregensfac

▷ `pregensfac(`*ri*`)` (attribute)

The value of this attribute is only set for homomorphism nodes. In that case it contains preimages of the nice generators in the factor group. This attribute is set automatically by the generic recursive recognition function using the mechanism described with the attribute `calcnicegens` (3.2.17) below. A find homomorphism does not have to touch this attribute.

3.2.9 RIFac

▷ `RIFac(`*`ri`*`)` (attribute)

The value of this attribute is the recognition info record of the image of the homomorphism that was found from the group described by the recognition info record *`ri`*. It is set by the generic recursive procedure after a find homomorphism method has succeeded to find a homomorphism (or isomorphism). It does not have to be set in leaf nodes of the recognition tree. This attribute value provides the link to the “factor” subtree of the recognition tree.

3.2.10 RIKer

▷ `RIKer(`*`ri`*`)` (attribute)

The value of this attribute is the recognition info record of the kernel of the homomorphism that was found from the group described by the recognition info record *`ri`*. It is set by the generic recursive procedure after a find homomorphism method has succeeded to find a homomorphism (or isomorphism). It does not have to be set in leaf nodes of the recognition tree or if the homomorphism is known to be an isomorphism. In the latter case the value of the attribute is set to `fail`. This attribute value provides the link to the “kernel” subtree of the recognition tree.

3.2.11 RIParent

▷ `RIParent(`*`ri`*`)` (attribute)

The value of this attribute is the recognition info record of the parent of this node in the recognition tree. The top node does not have this attribute set.

3.2.12 fhmethsel

▷ `fhmethsel(`*`ri`*`)` (attribute)

The value of this attribute is the record returned by the method selection (see Section 4.2) after it ran to find a homomorphism (or isomorphism). It is there to be able to see which methods were tried until the recognition of the node was completed.

3.2.13 slpforelement

▷ `slpforelement(`*`ri`*`)` (attribute)

After the recognition phase is completed for the node *`ri`*, we are by definition able to write arbitrary elements in the group described by this node as a straight line program (SLP) in terms of the nice generators stored in `NiceGens` (3.2.7). This attribute value is a function taking the node *`ri`* and a group element as its arguments and returning the above mentioned straight line program. For the case that a find homomorphism method succeeds in finding a homomorphism, the generic recursive function sets this attribute to the function `SLPforElementGeneric` (3.3.2) which does the job for the generic homomorphism situation. In all other cases the successful find homomorphism method has to set this attribute to a function doing the job. The find homomorphism method is free to store additional data in the recognition info record or the group object such that the `SLPforElement` (3.2.14) function can work.

3.2.14 SLPforElement

▷ `SLPforElement(`*`ri, x`*`)` (function)

Returns: a straight line program expressing *`x`* in the nice generators.

This is a wrapper function which extracts the value of the attribute `slpforelement` (3.2.13) and calls that function with the arguments *`ri`* and *`x`*.

3.2.15 StdPresentation

▷ `StdPresentation(`*`ri`*`)` (attribute)

After the verification phase, the presentation is stored here. Details have still to be decided upon.

3.2.16 methodsforfactor

▷ `methodsforfactor(`*`ri`*`)` (attribute)

This attribute is initialised at the beginning of the recursive recognition function with the database of find homomorphism methods that was used to recognise the group corresponding to the recognition info record *`ri`*. If the found homomorphism changes the representation of the group (going for example from a matrix group to a permutation group), the find homomorphism method can report this by exchanging the database of find homomorphism methods to be used in the recognition of the image of the homomorphism by setting the value of this attribute to something different. It lies in the responsibility of the find homomorphism method to do so, if the representation changes through the homomorphism.

The following two attributes are concerned with the relation between the original generators and the nice generators for a node. They are used to transport this information from a successful find homomorphism method up to the recursive recognition function:

3.2.17 calcnicegens

▷ `calcnicegens(`*`ri`*`)` (attribute)

To make the recursion work, we have to acquire preimages of the nice generators in factor groups under the homomorphism found. But we want to keep the information, how the nice generators were found, locally at the node where they were found. This attribute solves this problem of acquiring preimages in the following way: Its value must be a function, taking the recognition info record *`ri`* as first argument, and a list *`origgens`* of preimages of the original generators of the current node, and has to return corresponding preimages of the nice generators. Usually this task can be done by storing a straight line program writing the nice generators in terms of the original generators and executing this with inputs *origgens*. Therefore the default value of this attribute is the function `CalcNiceGensGeneric` (3.2.18) described below.

3.2.18 CalcNiceGensGeneric

▷ `CalcNiceGensGeneric(`*ri, origgens*`)` (function)

Returns: a list of preimages of the nice generators

This is the default function for leaf nodes for the attribute `calcnicegens` (3.2.17) described above. It does the following: If the value of the attribute `slptonice` (3.2.21) is set, then it must be a straight line program expressing the nice generators in terms of the original generators of this node. In that case, this straight line program is executed with *origgens* as inputs and the result is returned. Otherwise, *origgens* is returned as is. Therefore a leaf node just has to do nothing if the nice generators are equal to the original generators, or can simply store the right straight line program into the attribute `slptonice` (3.2.21) to fulfill its duties.

3.2.19 CalcNiceGensHomNode

▷ `CalcNiceGensHomNode(`*ri, origgens*`)` (function)

Returns: a list of preimages of the nice generators

This is the default function for homomorphism node for the attribute `calcnicegens` (3.2.17). It just delegates to factor and kernel of the homomorphism, as the nice generators of a homomorphism (or isomorphism) node are just the concatenation of the nice generators of the factor and the kernel. A find homomorphism method finding a homomorphism or isomorphism does not have to do anything with respect to nice generators.

3.2.20 CalcNiceGens

▷ `CalcNiceGens(`*ri, origgens*`)` (function)

Returns: a list of preimages of the nice generators

This is a wrapper function which extracts the value of the attribute `calcnicegens` (3.2.17) and calls that function with the arguments *ri* and *origgens*.

3.2.21 slptonice

▷ `slptonice(`*ri*`)` (attribute)

As described above, the value, if set, must be a straight line program expressing the nice generators at this node in terms of the original generators. This is for leaf nodes, that choose to use the default function `CalcNiceGensGeneric` (3.2.18) installed in the `calcnicegens` (3.2.17) attribute.

The following three attributes are concerned with the administration of the kernel of a found homomorphism. Find homomorphism methods use them to report to the main recursive recognition function their knowledge about the kernel:

3.2.22 gensN

▷ `gensN(`*ri*`)` (attribute)

The value of this mutable attribute is a list of generators of the kernel of the homomorphism found at the node *ri*. It is initialised as an empty list when the recursive recognition function starts. Successful find homomorphism methods may append generators of the kernel to this list if they happen to stumble on them. After successful recognition of the image of the homomorphism the main recursive recognition function will try to create a few more generators of the kernel and append them to the list which is the value of the attribute `gensN`. The exact behaviour depends on the value of the attribute `findgensNmeth` (3.2.23) below. The list of generators after that step is used to recognise the kernel. Note that the generators in `gensN` have a memory attached to them, how they were obtained in terms of the original generators of the current node.

3.2.23 findgensNmeth

▷ `findgensNmeth(`*ri*`)` (attribute)

This attribute decides about how generators of the kernel of a found homomorphism are produced. Its value has to be a record with at least two components bound. The first is `method` which holds a function taking at least one argument *ri* and possibly more, and does not return anything. The second is `args` which holds a list of arguments for the above mentioned function. The real list of arguments is derived by prepending the recognition info record to the list of arguments in `args`. That is, the following code is used to call the method:

```
    methgensN := findmethgensN(ri);
    CallFuncList(methgensN(ri).method,Concatenation([ri],methgensN.args));
```

The record is initialised upon creation of the recognition info record to calling `FindKernelRandom` (3.2.24) with one argument of 20 (FIXME: is 20 correct?) (in addition to the first argument *ri*). See below for a choice of possible find kernel methods.

3.2.24 FindKernelRandom

▷ `FindKernelRandom(`*ri*, *n*`)` (function)

Returns: nothing

n random elements are generated, mapped through the homomorphism, written as a straight line program in the generators. Then the straight line program is executed with the original generators thereby producing elements in the same coset. The quotients are then elements of the kernel. The kernel elements created are stored in the attribute `gensN` (3.2.22).

3.2.25 FindKernelDoNothing

▷ `FindKernelDoNothing(`*ri*, *n*`)` (function)

Returns: nothing

Does nothing. This function is intended to be set as method for producing kernel elements if the kernel is known to be trivial or if one knows, that the attribute `gensN` (3.2.22) already contains a complete set of generators for the kernel.

3.2.26 FindKernelFastNormalClosure

▷ `FindKernelFastNormalClosure(ri, nr)` (function)

Returns: a probable generating set for the normal closure

This function takes the group G in the `Grp` (3.2.5) attribute in *ri* and the list of generators *gens* of the kernel in `gensN` (3.2.22) and the positive integer *nr*. This function computes a probable generating set of the normal closure in *G* of the group generated by the generators in *gens*. The integer *nr* indicates how hard it should try.

3.2.27 gensNslp

▷ `gensNslp(ri)` (attribute)

The recursive recognition function calculates a straight line program that computes the generators of the kernel stored in `gensN` (3.2.22) in terms of the generators of the group recognised by *ri*. This straight line program is stored in the value of this mutable attribute. It is used by the generic function `SLPforElementGeneric` (3.3.2).

3.2.28 immediateverification

▷ `immediateverification(ri)` (attribute)

Sometimes a find homomorphism has information that it will be difficult to create generators for the kernel, for example if it is known that the kernel will need lots of generators. In that case this attribute with the default boolean value `false` can be set to `true`. In that case, the generic recursive recognition function will perform an immediate verification phase after the kernel has been recognised. This is done as follows: A few random elements are created, mapped through the homomorphism and written as an SLP in the nice generators there. Then this SLP is executed with preimages of those nice generators. The quotient lies then in the kernel and is written as an SLP in terms of the nice generators of the would be kernel. If this is not possible, then probably the creation of kernel generators was not complete and a few more kernel elements are produced and recognition in the kernel starts all over again. This is for example done in case of the “Imprimitive” method which maps onto the action on a block system. In that case, the kernel often needs lots of generators.

The following attributes are used to give a successful find homomorphism method further possibilities to transport knowledge about the group recognised by the current recognition info record to the factor or kernel of the found homomorphism:

3.2.29 forkernel

▷ `forkernel(ri)` (attribute)

This attribute is initialised to a record with only the component `hints` bound to an empty list at the beginning of the recursive recognition function. Find homomorphism methods can put acquired knowledge about the group to be recognised (like for example an invariant subspace of a matrix group) into this record. When a homomorphism is found and recognition goes on in its kernel, the value of this attribute is taken as initialisation data for the newly created recognition info record for the kernel. Thus, information is transported down to the recognition process for the kernel. The component `hints` is special insofar as it has to contain records describing find homomorphism methods which might be particularly successful. They are prepended to the find homomorphism method database such that they are called before any other methods. This is a means to give hints to the recognition procedure in the kernel, because often during the finding of a homomorphism knowledge is acquired which might help the recognition of the kernel.

3.2.30 forfactor

▷ `forfactor(`*ri*`)` (attribute)

This attribute is initialised to a record with only the component `hints` bound to an empty list at the beginning of the recursive recognition function. Find homomorphism methods can put acquired knowledge about the group to be recognised (like for example an invariant subspace of a matrix group) into this record. When a homomorphism is found and recognition goes on in its image, the value of this attribute is taken as initialisation data for the newly created recognition info record for the factor. Thus, information is transported down to the recognition process for the factor. The component `hints` is special insofar as it has to contain records describing find homomorphism methods which might be particularly successful. They are prepended to the find homomorphism method database such that they are called before any other methods. This is a means to give hints to the recognition procedure in the factor, because often during the finding of a homomorphism knowledge is acquired which might help the recognition of the factor.

3.2.31 isone

▷ `isone(`*ri*`)` (attribute)

This attribute returns a function that tests, whether or not an element of the group is equal to the identity or not. Usually this is just the operation `IsOne` (**Reference: IsOne**) but for projective groups it is a special function returning `true` for scalar matrices. In generic code, one should always use the result of this attribute to compare an element to the identity such that the code works also for projective groups. Find homomorphism methods usually do not have to set this attribute.

3.2.32 isequal

▷ `isequal(`*ri*`)` (attribute)

This attribute returns a function that compares two elements of the group being recognised. Usually this is just the operation `EQ` (**Reference: equality of records**) but for projective groups it is a special function checking for equality up to a scalar factor. In generic code, one should always use the result of this attribute to compare two elements such that the code works also for projective groups. Find homomorphism methods usually do not have to set this attribute.

3.2.33 Other components of recognition info records

In this subsection we describe a few more components of recognition info records that can be queried or set by find homomorphism methods. Not all of these components are bound in all cases. See the individual descriptions about the conventions. Remember to use the `!.` notation to access these components of a recognition info record.

`leavegensNuntouched`

If this component is bound to `true` by a find homomorphism method or a find kernel generators method, the generic mechanism to remove duplicates and identities in the generator for the kernel is not used. This is important if your methods rely on the generating set of the kernel being exactly as it was when found.

3.3 Methods to find homomorphisms

A “find homomorphism method” has the objective to, given a group G, either find a homomorphism from G onto a group, or to find an isomorphism, or to solve the constructive membership problem directly for G, or to fail.

In case a homomorphism is found, it has to report that homomorphism back to the calling recursive recognition function together with as much information about the kernel as possible.

If a find homomorphism method determines that the node is a leaf in the recognition tree (by solving the constructive membership problem directly), then it has to ensure, that arbitrary elements can be written in terms of the nice generators of G. It does so by returning a function together with possible extra data, that can perform this job.

Of course, the find homomorphism method also has to report, how the nice generators were acquired in terms of the original generators.

If the find homomorphism method fails, it has to report, whether it has failed forever or if it possibly makes sense to try to call this method again later.

Find homomorphism methods have to fit into the framework for method selection described in Chapter 4. We now begin to describe the technical details of how a find homomorphism method has to look like and what it has to do and what it is not allowed to do. We first explain the calling convention by means of a hypothetical function:

3.3.1 FindHomomorphism

▷ `FindHomomorphism(`*`ri`*`, `*`G`*`)` (function)

Returns: One of the values `Success`, `NeverApplicable`, `TemporaryFailure`, or `NotEnoughInformation`.

Find homomorphism methods take two arguments *`ri`* and *`G`*, of which *`ri`* is a recognition info record and *`G`* is a GAP group object. The return value is one of the four possible values in the framework for method selection described in Chapter 4 indicating success, failure, or (temporary) non-applicability. The above mentioned additional information in case of success are all returned by changing the recognition info record *`ri`*. For the conventions about what a find homomorphism method has to do and return see below.

A failed or not applicable find homomorphism method does not have to report or do anything in the recognition info record *`ri`*. However, it can collect information and store it either in the group object or in the recognition info record. Note that for example it might be that a failed find homomorphism method acquires additional information that allows another find homomorphism method to become applicable.

A not applicable find homomorphism method should find out so relatively quickly, because otherwise the whole process might be slowed down, because a find homomorphism method repeatedly ponders about its applicability. Usually no big calculations should be triggered just to decide applicability.

A successful find homomorphism method has the following duties:

for leaves:
First it has to report whether the current node is a leaf or not in the recognition tree. That is, in case a leaf was found the method has to do `SetFilterObj(ri,IsLeaf);` thereby setting the `IsLeaf` (3.2.3) flag.

A method finding a homomorphism which is not an isomorphism indicates so by not touching the flags. *FIXME: What does that mean? Which flags? The IsLeaf filter? But then this sounds as if isomorphisms require settings some flag.. but which?!? perhaps remove that sentence?*

for leaves: `SLPforElement` (3.2.14) function
If a find homomorphism method has produced a leaf in the recognition tree, then it has to set the attribute `slpforelement` (3.2.13) to a function like `SLPforElementGeneric` (3.3.2) that can write an arbitrary element in *G* as a straight line program in the nice generators of *G*. The method may store additional data into the recognition info record for this to work. It does not have to set any other value in *ri*.

for leaves: information about nice generators
If a find homomorphism method has produced a leaf in the recognition tree, then it has to report what are the nice generators of the group described by the leaf. To this end, it has three possibilities: Firstly to do nothing, which means, that the original generators are the nice generators. Secondly to store a straight line program expressing the nice generators in terms of the original generators into the attribute `slptonice` (3.2.21). In that case, the generic frame work takes care of the rest. The third possibility is to store a function into the value of the attribute `calcnicegens` (3.2.17) which can calculate preimages of the nice generators in terms of preimages of the original generators. See the function `CalcNiceGensGeneric` (3.2.18) for an example of such a function.

for non-leaves: the homomorphism itself
If a find homomorphism method has found a homomorphism, it has to store it as a GAP homomorphism object from *G* to the image group in the attribute `Homom` (3.2.6). Note that if your homomorphism changes the representation (for example going from matrix groups to permutation groups), you will have to set the attribute `methodsforfactor` (3.2.16) accordingly.

for non-leaves: kernel generators
If a find homomorphism method has found a homomorphism, it has to provide information about already known generators of the kernel. This is done firstly by appending known generators of the kernel to the attribute value of `gensN` (3.2.22) and secondly by leaving or changing the attribute `findgensNmeth` (3.2.23) to a record describing the method that should be used (for details see `findgensNmeth` (3.2.23)). If one does not change the default value, the recursive recognition function will generate 20 (FIXME: is 20 correct?) random elements in *G* and produce random generators of the kernel by dividing by a preimage of an image under the homomorphism. Note that generators in `gensN` (3.2.22) have to have a memory attached to them that stores, how they were acquired from the generators of *G*.

additional information
A find homomorphism method may store any data into the attributes `forkernel` (3.2.29) and `forfactor` (3.2.30), which both are records. Components in these record that are bound during

the recognition will be copied into the recognition info record of the kernel and factor respectively of a found homomorphism upon creation and thus are available to all find homomorphism methods called for the kernel and factor. This feature might be interesting to transport information that is relevant for the recognition of the kernel or factor and was acquired during the recognition of `G` itself.

A special role is played by the component `hints` in both of the above records, which can hold a list of records describing find homomorphism methods that shall be tried first when recognising the kernel or factor.

In addition, a find homomorphism method might set the attribute `immediateverification` (3.2.28) to true, if it considers the problem of finding kernel generators particularly difficult.

To explain the calling conventions for `SLPforElement` (3.2.14) functions and for the sake of completeness we present now the function `SLPforElementGeneric` (3.3.2) which is used for the case of a “homomorphism node”:

3.3.2 SLPforElementGeneric

▷ `SLPforElementGeneric(ri, x)` (function)

Returns: a GAP straight line program

This function takes as arguments a recognition info record `ri` and a group element `x`. It returns a GAP straight line program that expresses the element `x` in terms of the nice generators of the group G recognised by `ri`.

This generic function here does exactly this job for the generic situation that we found a homomorphism from G to some other group say H with kernel N. It first maps `x` via the homomorphism to H and uses the recognition information there to write it as a straight line program in terms of the nice generators of H. Then it applies this straight line program to the preimages of those nice generators (see `pregensfac` (3.2.8)) thereby finding an element y of G with $x \cdot y^{-1}$ lying in the kernel N.

Then the function writes this element as a straight line program in the nice generators of N again using the recursively acquired recognition info about N. In the end a concatenated straight line program for x is built, which is in terms of the nice generators of the current node.

3.4 Conventions for the recognition of permutation groups

No conventions so far.

3.5 Conventions for the recognition of matrix groups

We are considering only the case of matrix groups over finite fields.

No conventions so far.

3.6 Conventions for the recognition of projective groups

We are considering only the case of projective groups over finite fields.

No conventions so far.

Chapter 4

Method Selection

The setup described in this chapter is intended for situations, in which lots of different methods are available to fulfill a certain task, but in which it is not possible in the beginning to decide, which one to use. Therefore this setup regulates, rather than just which method to choose, in which order the various methods are tried. The methods themselves return whether they were successful and, if not, whether it is sensible to try them again at a later stage.

The design is intentionally kept as simple as possible and at the same time as versatile as possible, thereby providing a useful framework for many situations as described above.

Note the differences to the GAP method selection, which is designed with the idea in mind that it will be quite clear in most situations, which one is "the best" method for a given set of input data, and that we do not want to try different things. On the other hand, the GAP method selection is quite complicated, which is to some extend necessary to make sure, that lots of different information about the objects in question can be used to really find the best method.

Our setup here in particular has to fulfill the requirement, that in the end, with lots of methods installed, one still has to be able to have an overview and to "prove", that the whole system always does the right thing.

4.1 What are methods?

A method is just a GAP function together with an agreement about what arguments it takes and what result it returns. The agreement about the arguments of course has to be made for every situation in which this generic method selection code is used, and the user is completely free there. A method can (and has to) return one of the following four values:

`Success`
: means that the method was successful and no more methods have to be tried.

`NeverApplicable`
: means that the method was not successful and that there is no point to call the method again in this situation whatsoever.

`TemporaryFailure`
: means that the method temporarily failed, that it however could be sensible to call it again in this situation at a later stage. This value is typical for a Las Vegas algorithm using randomised methods, which has failed, but which may succeed when called again.

`NotEnoughInformation`
: means that the method for some reason refused to do its work. However, it is possible that it will become applicable later such that it makes sense to call it again, may when more information is available.

For administration in the method selection, a method is described by a record with the following components bound:

`method`
: holds the function itself.

`rank`
: holds an integer used to sort the various methods. Higher numbers mean that the method is tried earlier. The numbering scheme is left to the user.

`stamp`
: holds a string value that uniquely describes the method. This is used for bookkeeping and to keep track of what has to be tried how often.

`comment`
: a string valued comment. This field is optional and can be left out.

The different methods for a certain task are collected in so-called "method databases". A *method database* is just a list of records, each describing a method in the format described above. Usually, the ranks will be descending, but that is not necessary.

There is one convenience function to put a new method into a method database:

4.1.1 AddMethod

▷ `AddMethod(`*`db, meth, rank, stamp[, comment]`*`)` (function)

Returns: nothing

`db` must be a method database (list of records, see above) with non-ascending rank values. *`meth`* is the method function, *`rank`* the rank and *`stamp`* a string valued stamp. The optional argument *`comment`* can be a string comment. The record describing the method is created and inserted at the correct position in the method database. Nothing is returned.

4.2 How methods are called

Whenever the method selection shall be used, one calls the following function:

4.2.1 CallMethods

▷ `CallMethods(`*`db, limit[, furtherargs]`*`)` (function)

Returns: a record `ms` describing this method selection procedure.

The argument *`db`* must be a method database in the sense of Section 4.1. *`limit`* must be a non-negative integer. *`furtherargs`* stands for an arbitrary number of additional arguments, which are handed down to the called methods. Of course they must fulfill the conventions defined for the methods in the database *`db`*.

The function first creates a "method selection" record keeping track of the things that happened during the method trying procedure, which is also used during this procedure. Then it calls methods with the algorithm described below and in the end returns the method selection record in its final state.

The method selection record has the following components:

`inapplicableMethods`
: a record, in which for every method that returned `NeverApplicable` the value 1 is bound to the component with name the stamp of the method.

`failedMethods`
: a record, in which for every time a method returned `TemporaryFailure` the value bound to the component with name the stamp of the method is increased by 1 (not being bound means zero).

`successMethod`
: the stamp of the method that succeeded, if one did. This component is only bound after successful completion.

`result`
: a boolean value which is either `Success` or `TemporaryFailure` depending on whether a successful method was found or the procedure gave up respectively. This component is only bound after completion of the method selection procedure.

`tolerance`
: the number of times all methods failed until one succeeded. See below.

The algorithm used by `CallMethods` (4.2.1) is extremely simple: It sets a counter `tolerance` to zero. The main loop starts at the beginning of the method database and runs through the methods in turn. Provided a method did not yet return `NeverApplicable` and did not yet return `TemporaryFailure` more than `tolerance` times before, it is tried. According to the value returned by the method, the following happens:

`NeverApplicable`
: this is marked in the method selection record and the main loop starts again at the beginning of the method database.

`TemporaryFailure`
: this is counted in the method selection record and the main loop starts again at the beginning of the method database.

`NotEnoughInformation`
: the main loop goes to the next method in the method database.

`true`
: this is marked in the method selection record and the procedure returns successfully.

If the main loop reaches the end of the method database without calling a method (because all methods have already failed or are not applicable), then the counter `tolerance` is increased by one and everything starts all over again. This is repeated until `tolerance` is greater than the `limit` which is the second argument of `CallMethods` (4.2.1). The last value of the `tolerance` counter is returned in the component `tolerance` of the method selection record.

Note that the main loop starts again at the beginning of the method database after each failed method call! However, this does not lead to an infinite loop, because the failure is recorded in the method selection record such that the method is skipped until the `tolerance` increases. Once the `tolerance` has been increased methods having returned `TemporaryFailure` will be called again. The idea behind this approach is that even failed methods can collect additional information about the arguments changing them accordingly. This might give methods that come earlier and were not applicable up to now the opportunity to begin working. Therefore one can install very good methods that depend on some already known knowledge which will only be acquired during the method selection procedure by other methods, with a high rank.

Chapter 5

After successful recognition

This chapter explains, what one can do with recognition info records after a successful recognition (and possibly verification).

Of course, one can inspect the whole tree of recognition info records just by looking at the stored attribute values. Moreover, constructive membership tests can be performed using the function SLPforElement (3.2.14), thereby writing an arbitrary element in terms of the nice generators, which are stored in the attribute NiceGens (3.2.7). If fail is returned, then the element in question does not lie in the recognised group or the recognition made an error.

Here is an example of a successful recognition tree:

Example
```
gap> g := DirectProduct(SymmetricGroup(12),SymmetricGroup(5));
Group([ (1,2,3,4,5,6,7,8,9,10,11,12), (1,2), (13,14,15,16,17), (13,14) ])
gap> ri := RecogniseGroup(g);
#I  Finished rank 90 method "NonTransitive": success.
#I  Going to the factor (depth=0, try=1).
#I  Finished rank 95 method "VeryFewPoints": success.
#I  Back from factor (depth=0).
#I  Calculating preimages of nice generators.
#I  Creating 20 random generators for kernel.
....................
#I  Going to the kernel (depth=0).
#I  Finished rank 80 method "Giant": success.
#I  Back from kernel (depth=0).
<recoginfo NonTransitive
 F:<recoginfo VeryFewPoints Size=120>
 K:<recoginfo Giant Size=479001600>>
```

One sees that the recursive process runs, first it finds that the permutation action is not transitive, a homomorphism is found by mapping onto the action on one of the orbits. The image is recognised to permute only a few points. The kernel is recognised to be a full symmetric group in its natural action on at least 10 points (recognised as “Giant”).

After this, we can write arbitrary group elements in the group g in terms of the nice generators:

Example
```

gap> x := PseudoRandom(g);
(1,12)(2,5,9,11,10,3,4)(7,8)(13,14,16,15,17)
gap> slp := SLPforElement(ri,x);
```

```
<straight line program>
gap> ResultOfStraightLineProgram(slp,NiceGens(ri));
(1,12)(2,5,9,11,10,3,4)(7,8)(13,14,16,15,17)
```

Note that this example only works by using also the recog package which contains the necessary recognition methods.

5.1 Functions and methods for recognition info records

If you need an element explicitly written in terms of the original generators, you can use the following function:

5.1.1 SLPforNiceGens

▷ `SLPforNiceGens(`*`ri`*`)` (function)

Returns: an SLP expressing the nice generators in the original ones

This function assembles a possibly quite large straight line program expressing the nice generators in terms of the original ones by using the locally stored information in the recognition tree recursively.

You can concatenate straight line programs in the nice generators with the result of this function to explicitly write an element in terms of the original generators.

5.1.2 \in

▷ `\in(`*`x, ri`*`)` (method)

Returns: `true` or `false`

This method tests, whether the element *x* lies in the group recognised by the recognition info record *ri*. Note that this is only a convenience method, in fact `SLPforElement` (3.2.14) is used and the resulting straight line program is thrown away.

5.1.3 Size

▷ `Size(`*`ri`*`)` (method)

Returns: the size of the recognised group

This method calculates the size of the recognised group by multiplying the size of the factor and the kernel recursively. It is assumed that leaf nodes know already or can calculate the size of their group.

5.1.4 DisplayCompositionFactors

▷ `DisplayCompositionFactors(`*`ri`*`)` (function)

Returns: nothing

This function displays a composition series by using the recursive recognition tree. It only works, if the usual operation `CompositionSeries` (**Reference: CompositionSeries**) works for all leaves. THIS DOES CURRENTLY NOT WORK FOR PROJECTIVE GROUPS AND THUS FOR MATRIX GROUPS!

Chapter 6

Methods for recognition

6.1 Methods for permutation groups

The following table gives an overview over the installed methods and their rank (higher rank means higher priority, the method is tried earlier, see Chapter 4).

300	`TrivialPermGroup`	just go through generators and compare to the identity	6.1.1
100	`ThrowAwayFixedPoints`	try to find a huge amount of (possible internal) fixed points	6.1.2
99	`FewGensAbelian`	if very few generators, check IsAbelian and if yes, do KnownNilpotent	6.2.4
97	`Pcgs`	use a Pcgs to calculate a StabChain	6.1.3
95	`VeryFewPoints`	calculate a stabchain if we act on very few points	6.1.4
90	`NonTransitive`	try to find non-transitivity and restrict to orbit	6.1.5
80	`Giant`	tries to find Sn and An	6.1.6
70	`Imprimitive`	for a imprimitive permutation group, restricts to block system	6.1.7
60	`SnkSetswrSr`	tries to find jellyfish	6.1.8
55	`StabilizerChain`	for a permutation group using a stabilizer chain (genss)	6.3.12
50	`StabChain`	for a permutation group using a stabilizer chain	6.1.9

Table: Permutation group find homomorphism methods

6.1.1 `TrivialPermGroup`

This method is successful if and only if all generators of the permutation group G are equal to the identity. Otherwise it returns `false` indicating that it will never succeed. This method is only installed to handle the trivial case such that we do not have to take this case into account in the other methods.

6.1.2 `ThrowAwayFixedPoints`

This method defines a homomorphism of a permutation group G to the action on the moved points of G if G does not have too many moved points. In the current setup, the homomorphism is defined if the number k of moved points is at most $1/3$ of the largest moved point of G, or k is at most half of the number of points on which G is stored internally by GAP. The method returns `false` if it does not define a homomorphism indicating that it will never succeed.

6.1.3 `Pcgs`

This is the GAP library function to compute a stabiliser chain for a solvable permutation group. If the method is successful then the calling node becomes a leaf node in the recursive scheme. If the input group is not solvable then the method returns `false`.

6.1.4 `VeryFewPoints`

If a permutation group acts only on a few points (the current limit is at most 10 points) then a stabiliser chain is computed by the randomized GAP library function for that purpose. If the method is successful then the calling node becomes a leaf node in the recursive scheme. If the input group acts on more than 10 points then the method returns `false`.

6.1.5 `Nontransitive`

If a permutation group `G` acts nontransitively then this method computes a homomorphism to the action of `G` on the orbit of the largest moved point. If `G` is transitive then the method returns `false`.

6.1.6 `Giant`

The method tries to determine whether the input group `G` is a giant (that is, A_n or S_n in its natural action on n points). The output is either a data structure D containing nice generators for `G` and a procedure to write an SLP for arbitrary elements of `G` from the nice generators; or `false` if `G` is not transitive; or `fail`, in the case that no evidence was found that `G` is a giant, or evidence was found, but the construction of D was unsuccessful. If the method constructs D then the calling node becomes a leaf.

6.1.7 `Imprimitive`

If the input group is not known to be transitive then this method returns `NotEnoughInformation`. If the input group is known to be transitive and primitive then the method returns `false`; otherwise, the method tries to compute a nontrivial block system. If successful then a homomorphism to the action on the blocks is defined; otherwise, the method returns `false`. If the method is successful then it also gives a hint for the children of the node by determining whether the kernel of the action on the block system is solvable. If the answer is yes then the default value 20 for the number of random generators in the kernel construction is increased by the number of blocks.

6.1.8 `SnkSetswrSr`

This method tries to determine whether the input group `G` is acting primitively on N points, and is isomorphic to a large subgroup of $H \wr S_r$ where H is S_n acting on k-sets and $N = \binom{n}{k}^r$ and $kr > 1$. "Large" means that `G` contains a subgroup isomorphic to A_n^r. If `G` is imprimitive then the output is `false`. If `G` is primitive then the output is either a homomorphism into the natural imprimitive action of `G` on nr points with r blocks of size n, or `fail`.

6.1.9 `StabChain`

This is the randomized GAP library function for computing a stabiliser chain. The method selection process ensures that this function is called only with small-base inputs, where the method works efficiently.

6.2 Methods for matrix groups

The following table gives an overview over the installed methods and their rank (higher rank means higher priority, the method is tried earlier, see Chapter 4). Note that there are not that many methods for matrix groups since the system can switch to projective groups by dividing out the subgroup of scalar matrices. The bulk of the recognition methods are then installed es methods for projective groups.

3100	`TrivialMatrixGroup`	check whether all generators are equal to the identity matrix	6.2.1
1175	`KnownStabilizerChain`	use an already known stabilizer chain for this group	6.2.2
1100	`DiagonalMatrices`	check whether all generators are multiples of the identity	6.2.3
1050	`FewGensAbelian`	if very few generators, check IsAbelian and if yes, do KnownNilpotent	6.2.4
1000	`ReducibleIso`	use the MeatAxe to find invariant subspaces	6.2.5
900	`GoProjective`	divide out scalars and recognise projectively	6.2.8

Table: Matrix group find homomorphism methods

6.2.1 `TrivialMatrixGroup`

This method is successful if and only if all generators of a matrix group `G` are equal to the identity. Otherwise, it returns `false`.

6.2.2 `KnownStabilizerChain`

TODO. use an already known stabilizer chain for this group

6.2.3 `DiagonalMatrices`

This method is successful if and only if all generators of a matrix group `G` are diagonal matrices. Otherwise, it returns `false`.

6.2.4 `FewGensAbelian`

TODO. if very few generators, check IsAbelian and if yes, do KnownNilpotent

6.2.5 `ReducibleIso`

This method determines whether a matrix group `G` acts irreducibly. If yes, then it returns `false`. If `G` acts reducibly then a composition series of the underlying module is computed and a base change is performed to write `G` in a block lower triangular form. Also, the method passes a hint to the image group that it is in block lower triangular form, so the image immediately can make recursive calls for the actions on the diagonal blocks, and then to the lower p-part. For the image the method `BlockLowerTriangular` (see 6.2.6) is used.

Note that this method is implemented in a way such that it can also be used as a method for a projective group `G`. In that case the recognition info record has the `!.projective` component bound to `true` and this information is passed down to image and kernel.

6.2.6 `BlockLowerTriangular`

This method is only called when a hint was passed down from the method `ReducibleIso` (see 6.2.5). In that case, it knows that a base change to block lower triangular form has been performed. The method can then immediately find a homomorphism by mapping to the diagonal blocks. It sets up this homomorphism and gives hints to image and kernel. For the image, the method `BlockDiagonal` (see 6.2.7) is used and for the kernel, the method `LowerLeftPGroup` (see 6.2.10) is used.

Note that this method is implemented in a way such that it can also be used as a method for a projective group `G`. In that case the recognition info record has the `!.projective` component bound to `true` and this information is passed down to image and kernel.

6.2.7 `BlockDiagonal`

This method is only called when a hint was passed down from the method `BlockLowerTriangular` (see 6.2.6). In that case, it knows that the group is in block diagonal form. The method is used both in the matrix- and the projective case.

The method immediately delegates to projective methods handling all the diagonal blocks projectively. This is done by giving a hint to the factor to use the method `BlocksModScalars` (see 6.3.4) is given. The method for the kernel then has to deal with only scalar blocks, either projectively or with scalars, which is again done by giving a hint to either use `BlockScalar` (see 6.2.9) or `BlockScalarProj` (see 6.3.13) respectively.

Note that this method is implemented in a way such that it can also be used as a method for a projective group `G`. In that case the recognition info record has the `!.projective` component bound to `true` and this information is passed down to image and kernel.

6.2.8 `GoProjective`

This method defines a homomorphism from a matrix group `G` into the projective group `G` modulo scalar matrices. In fact, since projective groups in GAP are represented as matrix groups, the homomorphism is the identity mapping and the only difference is that in the image the projective group methods can be applied. The bulk of the work in matrix recognition is done in the projective group setting.

6.2.9 `BlockScalar`

This method is only called by a hint. Alongside with the hint it gets a block decomposition respected by the matrix group `G` to be recognised and the promise that all diagonal blocks of all group elements will only be scalar matrices. This method recursively builds a balanced tree and does scalar recognition in each leaf.

6.2.10 `LowerLeftPGroup`

This method is only called by a hint from `BlockLowerTriangular` as the kernel of the homomorphism mapping to the diagonal blocks. The method uses the fact the this kernel is a p-group where p is the characteristic of the underlying field. It exploits this fact and uses this special structure to find nice generators and a method to express group elements in terms of these.

6.3 Methods for projective groups

The following table gives an overview over the installed methods and their rank (higher rank means higher priority, the method is tried earlier, see Chapter 4). Note that the recognition for matrix group switches to projective recognition rather soon in the recognition process such that most recognition methods in fact are installed as methods for projective groups.

3000	`TrivialProjectiveGroup`	check if all generators are scalar multiples of the identity matrix	6.3.1
1300	`ProjDeterminant`	find homomorphism to non-zero scalars mod d-th powers	6.3.2
1250	`FewGensAbelian`	if very few generators, check IsAbelian and if yes, do KnownNilpotent	6.2.4
1200	`ReducibleIso`	use MeatAxe to find a composition series, do base change	6.2.5
1100	`NotAbsolutelyIrred`	write over a bigger field with smaller degree	6.3.5
1050	`ClassicalNatural`	check whether it is a classical group in its natural representation	??
1000	`Subfield`	write over a smaller field with same degree	6.3.6
900	`C3C5`	compute a normal subgroup of derived and resolve C3 and C5	??
850	`C6`	find either an (imprimitive) action or a symplectic one	6.3.9
840	`D247`	play games to find a normal subgroup	??
820	`SporadicsByOrders`	generate a few random elements and compute the proj. orders	??
810	`AltSymBBByDegree`	try BB recognition for dim+1 and/or dim+2 if sensible	??
800	`Tensor`	find a tensor decomposition	6.3.1
700	`FindElmOfEvenNormal`	find D2, D4 or D7 by finding an element of an even normal subgroup	??
600	`LowIndex`	find an (imprimitive) action on subspaces	6.3.8
550	`ComputeSimpleSocle`	compute simple socle of almost simple group	??
500	`ThreeLargeElOrders`	look at three large element orders	??
400	`LieTypeNonConstr`	do non-constructive recognition of Lie type groups	??
100	`StabilizerChain`	last resort: compute a stabilizer chain (projectively)	6.3.1

Table: Projective group find homomorphism methods

6.3.1 `TrivialProjectiveGroup`

This method is successful if and only if all generators of a projective group `G` are equal to the identity (that is, in the matrix representation of `G`, all matrices are scalars). Otherwise, it returns `false`.

6.3.2 `ProjDeterminant`

The method defines a homomorphism from a projective group `G`$\leq PGL(d,q)$ to the cyclic group $GF(q)^*/D$, where D is the set of dth powers in $GF(q)^*$. The image of a group element $g \in$ `G` is the determinant of a matrix representative of g, modulo D.

6.3.3 `ReducibleIso`

This method is the same as the matrix group method with the same name (see 6.2.5), which is able to take into account the projective mode.

6.3.4 `BlocksModScalars`

This method is only called when hinted from above. In this method it is understood that G should *neither* be recognised as a matrix group *nor* as a projective group. Rather, it treats all diagonal blocks modulo scalars which means that two matrices are considered to be equal, if they differ only by a scalar factor in *corresponding* diagonal blocks, and this scalar can be different for each diagonal block. This means that the kernel of the homomorphism mapping to a node which is recognised using this method will have only scalar matrices in all diagonal blocks.

This method does the balanced tree approach mapping to subsets of the diagonal blocks and finally using projective recognition to recognise single diagonal block groups.

6.3.5 `NotAbsolutelyIrred`

If an irreducible projective group `G` acts absolutely irreducibly then this method returns `false`. If `G` is not absolutely irreducible then a homomorphism into a smaller dimensional representation over an extension field is defined. A hint is handed down to the image that no test for absolute irreducibility has to be done any more. Another hint is handed down to the kernel indicating that the only possible kernel elements can be elements in the centraliser of `G` in $PGL(d,q)$ that come from scalar matrices in the extension field.

6.3.6 `Subfield`

When this method runs it knows that the projective group `G` acts absolutely irreducibly. It then tries to realise this group over a smaller field. The algorithm used is the one using the “standard basis approach” known from isomorphism testing of absolutely irreducible modules. It finds a base change to write the projective group over the smallest field possible. Since the group is projective, it may choose to multiply generators with arbitrary scalars to write them over a smaller field.

However, sometimes the correct scalar can not be guessed. Therefore, if the first approach does not work, the method computes the derived subgroup. If the group can be written over a smaller field, then taking commutators loses the scalars preventing a direct base change to work. Therefore, if the derived subgroup still acts irreducibly, the standard basis approach can find the right base change that could also do the job for the whole group. If it acts reducibly, the method `Derived` (see 6.3.7) which is run next already has the computed derived subgroup and can try different things to find a reduction.

6.3.7 `Derived`

This method computes the derived subgroup, if this has not yet been done by other methods. It then uses the MeatAxe to decide whether the derived subgroup acts irreducibly or not. If it acts reducibly, then we can apply Clifford theory to the natural module. The natural module restricted to the derived subgroup is a direct sum of simple modules. If all the summands are isomorphic, we immediately get either an action of `G` on blocks or a tensor decomposition. Otherwise, we get an action of `G` on the isotypic components. Either way, we find a reduction.

If the derived group acts irreducibly, we return `false` in the current implementation.

6.3.8 `LowIndex`

This method is designed for the handling of the Aschbacher class C2 (stabiliser of a decomposition of the underlying vector space), but may succeed on other types of input as well. Given `G` $\leq PGL(d,q)$, the output is either the permutation action of `G` on a short orbit of subspaces or `fail`. In the current setup, "short orbit" is defined to have length at most $4d$.

6.3.9 `C6`

This method is designed for the handling of the Aschbacher class C6 (normaliser of an extraspecial group). If the input `G`$\leq PGL(d,q)$ does not satisfy $d = r^n$ and $r|q-1$ for some prime r and integer n then the method returns `false`. Otherwise, it returns either a homomorphism of `G` into $Sp(2n,r)$, or a homomorphism into the C2 permutation action of `G` on a decomposition of $GF(q)^d$, or `fail`.

6.3.10 `Tensor`

This method currently tries to find one tensor factor by powering up commutators of random elements to elements of prime order. This seems to work quite well provided that the two tensor factors are not "linked" too much such that there exist enough elements that act with different orders on both tensor factors.

This method and its description needs some improvement.

6.3.11 `TwoLargeElOrders`

In the case when the input group `G`$\leq PGL(d,p^e)$ is suspected to be simple but not alternating, this method takes the two largest element orders from a sample of pseudorandom elements of `G`. From these element orders, it tries to determine whether `G` is of Lie type or sporadic, and the characteristic of `G` if it is of Lie type. In the case when `G` is of Lie type of characteristic different from p or `G` is sporadic, the method also provides a short list of the possible isomorphism types of `G`.

6.3.12 `StabilizerChain`

This method computes a stabiliser chain and a base and strong generating set using projective actions. This is a last resort method since for bigger examples no short orbits can be found in the natural action. The strong generators are the nice generator in this case and expressing group elements in terms of the nice generators ist just sifting along the stabiliser chain.

6.3.13 `BlockScalarProj`

This method is only called by a hint. Alongside with the hint it gets a block decomposition respected by the matrix group `G` to be recognised and the promise that all diagonal blocks of all group elements will only be scalar matrices. This method simply norms the last diagonal block in all generators by multiplying with a scalar and then delegates to `BlockScalar` (see 6.2.9) and matrix group mode to do the recognition.

Chapter 7

Examples

TODO

References

[NS06] Max Neunhöffer and Ákos Seress. A data structure for a uniform approach to computations with finite groups. In *ISSAC 2006*, pages 254–261. ACM, New York, 2006. 4

Index