

# GAP 4 Package recog

## Group Recognition Methods

1.1

July 2011

**Max Neunhöffer**  
**Ákos Seress**  
**Nurullah Ankaralioglu**  
**Peter Brooksbank**  
**Frank Celler**  
**Stephen Howe**  
**Maska Law**  
**Stephen Linton**  
**Gunter Malle**  
**Eamonn O'Brien**  
**Colva Roney-Dougal**

**Max Neunhöffer** — Email: [neunhoef@mcs.st-and.ac.uk](mailto:neunhoef@mcs.st-and.ac.uk)

— Homepage: <http://www-groups.mcs.st-and.ac.uk/~neunhoef/>

— Address: School of Mathematics and Statistics, Mathematical Institute, North Haugh, St Andrews, Fife, KY16 9SS, Scotland, UK

**Ákos Seress** — Email: [akos@math.ohio-state.edu](mailto:akos@math.ohio-state.edu)

— Homepage: <http://www.math.ohio-state.edu/people/display/display.php?ID=302>

— Address: Department of Mathematics, The Ohio State University,  
231 W 18th Avenue, Columbus, OH 43210 USA

## **Copyright**

© 2005-2009 by Max Neunhöffer and Ákos Seress

This package may be distributed under the terms and conditions of the GNU Public License Version 3 or (at your option) any later version.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Philosophy . . . . .	5
1.2	Overview over this manual . . . . .	5
<b>2</b>	<b>Installation of the recog-Package</b>	<b>6</b>
<b>3</b>	<b>Methods for recognition</b>	<b>7</b>
3.1	Methods for permutation groups . . . . .	7
3.1.1	TrivialPermGroup . . . . .	7
3.1.2	ThrowAwayFixedPoints . . . . .	7
3.1.3	Pcgs . . . . .	7
3.1.4	VeryFewPoints . . . . .	8
3.1.5	Nontransitive . . . . .	8
3.1.6	Giant . . . . .	8
3.1.7	Imprimitive . . . . .	8
3.1.8	SnkSetswrSr . . . . .	8
3.1.9	StabChain . . . . .	8
3.2	Methods for matrix groups . . . . .	9
3.2.1	TrivialMatrixGroup . . . . .	9
3.2.2	DiagonalMatrices . . . . .	9
3.2.3	ReducibleIso . . . . .	9
3.2.4	BlockLowerTriangular . . . . .	9
3.2.5	BlockDiagonal . . . . .	10
3.2.6	GoProjective . . . . .	10
3.2.7	BlockScalar . . . . .	10
3.2.8	LowerLeftPGroup . . . . .	10
3.3	Methods for projective groups . . . . .	10
3.3.1	TrivialProjectiveGroup . . . . .	11
3.3.2	ProjDeterminant . . . . .	11
3.3.3	ReducibleIso . . . . .	11
3.3.4	BlocksModScalars . . . . .	11
3.3.5	NotAbsolutelyIrred . . . . .	11
3.3.6	Subfield . . . . .	12
3.3.7	Derived . . . . .	12
3.3.8	LowIndex . . . . .	12
3.3.9	C6 . . . . .	12

3.3.10	Tensor	12
3.3.11	TwoLargeElOrders	13
3.3.12	StabilizerChain	13
3.3.13	BlockScalarProj	13
3.4	Methods for black box groups	13
<b>4</b>	<b>Examples</b>	<b>14</b>

# Chapter 1

## Introduction

### 1.1 Philosophy

This package is about group recognition. To be written further.

### 1.2 Overview over this manual

Chapter 2 describes the installation of this package. Chapter (**recogbase: Recognition info records**) 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 (**recogbase: Method Selection**), because it is interesting in its own right and might be useful in other circumstances.

*More text on other chapters to be written.*

Finally, Chapter 4 shows instructive examples for the usage of this package.

## 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.

Note that the `recogbase` package is needed by this package.

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.

Max Neunhöffer and Ákos Seress

## Chapter 3

# Methods for recognition

### 3.1 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 (**recogbase: Method Selection**)).

300	TrivialPermGroup	3.1.1
100	ThrowAwayFixedPoints	3.1.2
97	Pcgs	3.1.3
95	VeryFewPoints	3.1.4
90	NonTransitive	3.1.5
80	Giant	3.1.6
70	Imprimitive	3.1.7
60	SnkSetswrSr	3.1.8
50	StabChain	3.1.9

**Table:** Permutation group find homomorphism methods

#### 3.1.1 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.

#### 3.1.2 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.

#### 3.1.3 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`.

### 3.1.4 `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`.

### 3.1.5 `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`.

### 3.1.6 `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.

### 3.1.7 `Imprimitive`

If the input group is not known to be transitive then this method returns `NotApplicable`. 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.

### 3.1.8 `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`.

### 3.1.9 `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.

## 3.2 Methods for matrix groups

THIS CHAPTER IS CURRENTLY A BIT OUT OF DATE!

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 (**recogbase: Method Selection**)). 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 as methods for projective groups.

3100	TrivialMatrixGroup	3.2.1
1100	DiagonalMatrices	3.2.2
1000	ReducibleIso	3.2.3
900	GoProjective	3.2.6

**Table:** Matrix group find homomorphism methods

### 3.2.1 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`.

### 3.2.2 DiagonalMatrices

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

### 3.2.3 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 3.2.4) 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.

### 3.2.4 BlockLowerTriangular

This method is only called when a hint was passed down from the method `ReducibleIso` (see 3.2.3). 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 3.2.5) is used and for the kernel, the method `LowerLeftPGroup` (see 3.2.8) 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.

### 3.2.5 BlockDiagonal

This method is only called when a hint was passed down from the method `BlockLowerTriangular` (see 3.2.4). 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 3.3.4) 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 3.2.7) or `BlockScalarProj` (see 3.3.13) 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`.

### 3.2.6 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.

### 3.2.7 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.

### 3.2.8 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 that 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.

## 3.3 Methods for projective groups

THIS CHAPTER IS CURRENTLY A BIT OUT OF DATE!

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 (**recogbase: Method Selection**)). 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	3.3.1
1300	ProjDeterminant	3.3.2
1200	ReducibleIso	3.3.3
1100	NotAbsolutelyIrred	3.3.5
1000	Subfield	3.3.6
900	Derived	3.3.7
800	LowIndex	3.3.8
700	C6	3.3.9
600	Tensor	3.3.10
500	TwoLargeElOrders	3.3.11
100	StabilizerChain	3.3.12

**Table:** Projective group find homomorphism methods

### 3.3.1 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`.

### 3.3.2 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  $d$ th powers in  $GF(q)^*$ . The image of a group element  $g \in G$  is the determinant of a matrix representative of  $g$ , modulo  $D$ .

### 3.3.3 ReducibleIso

This method is the same as the matrix group method with the same name (see 3.2.3), which is able to take into account the projective mode.

### 3.3.4 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.

### 3.3.5 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.

### 3.3.6 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 3.3.7) which is run next already has the computed derived subgroup and can try different things to find a reduction.

### 3.3.7 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.

### 3.3.8 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  $4d$ .

### 3.3.9 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`.

### 3.3.10 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.

### 3.3.11 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  $G$ .

### 3.3.12 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 is just sifting along the stabiliser chain.

### 3.3.13 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 3.2.7) and matrix group mode to do the recognition.

## 3.4 Methods for black box groups

## **Chapter 4**

### **Examples**

Here comes text.

# References

# Index

recog, [6](#), [7](#)