

Using GAP

Alexander Hulpke

Department of Mathematics
The Ohio State University
231 West 18th Avenue
Columbus, OH 43210, USA

ahulpke@math.ohio-state.edu
<http://www.math.ohio-state.edu/~ahulpke>

1 Preface

This set of notes on GAP was prepared for a tutorial at ISSAC 2000 in St Andrews.

This tutorial is intended neither to give an introduction to computational group theory, nor to be a replacement for a manual. Instead, I want to give ideas what can be done with the system and how I would tackle certain problems. Much of this is based on questions colleagues came up with over the last years.

While working with the system might be a bit foreign at the beginning, usually there is no problem in finding out what commands such as:

```
1+2;  
Normalizer(group, subgroup);  
IsSolvableGroup(G);
```

do. This tutorial therefore is not intended as a hands-on tutorial you follow simultaneously (there is a part of the GAP manual devoted to this purpose). Instead I hope you will follow the mathematical concepts how things can be done and then can adapt this tutorial's examples for your own purposes.

If you have never used the system before, I hope you will get an idea what the system is capable of and might find starting points to translate the problems you face for the system. If you used the system, I hope this tutorial will elucidate some of the more quirky bits of the system.

Some of the examples might look as if I was throwing an overkill of technology at problems one could solve easily by hand. That's certainly true. On the other hand I want to show how to use the system, and not how clever a human can be. Besides, I guess most people value their personal time higher than a computer's and having it slaving away for a while is what it was built for.

Due to my own background in group theory the emphasis might be overly on groups. I hope, however, the examples and descriptions will show enough of the general "flavour" or "look-and-feel" of the system to be useful also if you work with other structures.

I would like to thank Inna Korchagina, David Pollack, and Jeff Riedl for helpful comments on a first version.

Columbus, July 2000

2 The system

2.1 Authors

Consisting of several 10^6 s lines of code, the system is the result of the work of many people.

An explicit authors list can be found in the manual and on the web page.

2.2 Getting Information and Help

<http://www-gap.dcs.st-and.ac.uk>
(and mirrors): System information. Download.

gap-trouble@dcs.st-and.ac.uk
General system help.

gap-forum@dcs.st-and.ac.uk
Discussion list

2.3 User interface

The user interface is text based (there is a graphical front end XGAP for Unix). The editing commands are Unix/EMACS-like. (In particular <TAB> offers command completion and ^P, ^N go back and forth in the history). If you have used a text-only version of Maple, say, you will feel at home soon. *?topic* calls the online help. *??sub* lists all help topics that contain the substring *sub*. *SetHelpViewer* permits to change the way the online help is displayed. For example:

```
gap> SetHelpViewer("netscape");
```

under UNIX.

2.4 Break Loop

If you interrupt a calculation with ^C or run into an error you enter the *break loop*, indicated by the *brk>* prompt.

At this point you can:

- Abandon the calculation with `quit`; or ^D.
In rare cases this might leave a corrupted data structure.
- Continue after an interruption with `return`;
- Investigate local variables.
- See the function call stack with `Where()`.

- Step up and down the call stack (and investigate variables there) with `DownEnv(n)`.

`DownEnv` takes *negative* values to step up.

If you cause an error in a break loop, you enter another (deeper) break loop.

2.5 Bugs and Problems

In the course of your calculations you might run into problems:

- There seems to be no way to do the calculation you want to.
- The calculation is too slow.
- GAP might run into an error (or worse: returns a wrong result)

While this tutorial tries to help you around problems of the first two kinds, there still could be performance problems we don't yet know about.

If you suspect the system to have any such problem, please *do not keep this knowledge to yourself and grumble quietly*. If we don't know about a problem we cannot fix it. We might not be able to remedy all performance problems you come up with, but we will certainly try (and the experience so far with this has been quite good).

The email address `gap-trouble@dcs.st-and.ac.uk` has been set up especially for this purpose.

(In most cases we will need not only a description of the problem, but it is extremely helpful to also have the input which caused the problem.)

2.6 View, Print and Display

There are three default ways how GAP shows objects to the user:

2.6.1 View

shows a short description of an object. All output that GAP provides without being asked for is created by `View`. A double semicolon at the end of a line will inhibit `View`.

```
gap> GL(3,2).1;
<an immutable 3x3 matrix over GF2>
```

2.6.2 Print

shows output that describes the object in sufficient detail that it could be used to recreate the object.

```
gap> Print(GL(3,2).1);
[[Z(2)^0,Z(2)^0,0*Z(2)], [0*Z(2),Z(2)^0,0*Z(2)],
 [0*Z(2),0*Z(2),Z(2)^0]]
```

`Print` is the bottom level function used to output Numbers or strings. Do not forget the `\n` line end character.

There also are `PrintTo` and `AppendTo` to write to files.

`Print/Read` so far provide the only possibility to save single objects. The file created by `Print` has to be edited to add a variable assignment and a semicolon.

2.6.3 Display

pretty-prints the most human-readable version.

```
gap> Display(GL(3,2).1);
1 1 .
. 1 .
. . 1
```

In case you want to install own methods: The corresponding single-argument operations are called `ViewObj`, `PrintObj`, `DisplayObj`.

2.7 Language

The GAP programming language is of the ALGOL (Pascal, C, Java, ...) family. It is interpreted but has time-critical routines in the (C-) kernel. There also is the possibility to compile the interpreted language into C for further optimization.

The standard way for argument lists is to have the "largest" arguments first.

By default, a lot of user operations test the input for validity (for example checking whether a prescribed map actually is a homomorphism). These tests are helpful when working interactively to spot careless errors, they can be a severe nuisance in an algorithm where the context ensures the input data is valid. In such cases there often is a second operation whose name is the same but for an appended NC (for "no check").

2.8 Basic objects

The basic objects are (essentially) rational and cyclotomic numbers, finite field elements, permutations, words in free structures, lists and records. Using the latter two iteratively, it is possible to create more complicate data structures.

GAP internally uses *positional objects* and *component objects* which are essentially like lists and records, however component access requires use of an `!.NamesOfComponents` will return the components stored for an component object. You can access these components, but in most cases such access is undocumented and will lead to code that might not run with future versions.

Finite field elements are given by *Zech logarithms* as power of a primitive element.

```
gap> AsList(GF(8));
[0*Z(2),Z(2)^0,Z(2^3),Z(2^3)^2,Z(2^3)^3,
 Z(2^3)^4,Z(2^3)^5,Z(2^3)^6]
```

Use `n*Z(p)^0` to convert an integer into a prime field and `Int(n)` to get the integer corresponding to a prime field element.

2.9 List functions

These functions are incredibly convenient and permit to code substantial code in just a single line. (They are not necessarily the fastest possible way of doing the task so use them with care in inner loops). For example to find all sublists in a list which contain exactly two prime numbers, one could use:

```
Filtered(l,i->Number(i,IsPrimeInt)=2);
```

All of these functions take as input a list and a one-argument function. The shorthand notation

```
x->myfunction(x+1)
for
function(x) return myfunction(x+1);end
is useful in this context.
```

2.10 Sorted lists

Finding elements in a list using `in` or `Position` works much faster if the list is sorted and binary search can be used. This however requires that a total order can be computed cheaply for the objects in the list, which is not the case for complicated objects, such as subgroups. (The filter `CanEasilySortElements` is useful here.) Therefore it can be worth to convert a list into sorted form (using `AsSortedList/Set` or `Sort`) before working with it. A list of mutable objects cannot remember that it is sorted (\rightarrow 4.12). A general setup for such problems is provided by dictionaries (\rightarrow 4.10).

2.11 Equal versus identical

Objects are identical, if they are indeed the same object in memory. However the “same” object might be stored several times or several objects might be considered to be equal (for example different words representing the same element of a finitely presented group). Avoiding to keep equal, nonidentical, objects can save a lot of memory. When working with (mutable) objects, however a bit of care has to be taken:

```
gap> a:=[1,2];;
gap> b:=[a,[1,2],a];
[ [ 1, 2 ], [ 1, 2 ], [ 1, 2 ] ]
gap> b[1][2];
2
gap> b[1][2]:=3;;b;
[ [ 1, 3 ], [ 1, 2 ], [ 1, 3 ] ]
gap> b[2][2]:=4;; b;
[ [ 1, 3 ], [ 1, 4 ], [ 1, 3 ] ]
gap> b[1]:=[1,2];; b;
[ [ 1, 2 ], [ 1, 4 ], [ 1, 3 ] ]
```

3 System Capabilities

3.1 Standard Questions

There are different levels of difficulty of computing tasks:

Level 0: Element Arithmetic

Level 1: $|G|$, $[G:U]$, $a \in U$,

Level 2: Composition series, Normalizer, Centralizer, Centre, Normal subgroups, factor groups, Complements, Homomorphisms

Level 3: Conjugacy classes, Subgroups, Character table, Automorphism group, group isomorphism

Level 4: Test conjectures/find counterexample, Enumerations of structures,

Except for finitely presented groups (where element equality is potentially undecidable) there exist algorithms for problems of level up to 3. However most research problems fall in level 4. Therefore I want to concentrate on how to phrase these problems in a way suitable for GAP.

3.2 How hard are questions

The different levels of difficulty roughly correspond to the time I’m willing to wait for a result. The following table gives some ideas of how large problems one can tackle on a “current” (Summer 2000) system.

Level	Perm	PcGroup
	Degree/Size	Composition Length
0	$10^7/10^7!$	300
1	$5 \cdot 10^5/10^{500}$	100
2	$10^5/10^{80}$	60
3	$10^4/10^{20}$	20

Matrix groups defer to permutation groups, the only extra problem there is to find a faithful permutation representation. Finitely presented groups again suffer from undecidability results.

It is usually better to work in matrix groups than fp groups, permutation groups than matrix groups and pc groups rather than permutation groups.

3.3 Creating Domains

Groups, rings, Algebras etc. are implemented in GAP as sets of elements, for which certain operations ($+$, $-$, $*$, $/$, \wedge , \dots) are defined. The manual (\rightarrow **Manual: Domains and their Elements**) calls such sets *domains*.

On the lowest level domains work like sets: The standard operations `in`, `IsSubset`, `Size`, `Intersection`, `Union` work as for sets (but often try to be more clever than to use element lists).

There are three common ways to define domains:

- Generators — The domain created is the closure of a given generating set under the admitted operations. Examples are matrices, permutations or endomorphisms.
- Relations — The domain is given as
 - Quotient of a universal object by factoring out relations. Examples are Finitely presented groups, algebras (structure constants are a special case)
 - Subset of a larger domain which leaves a given structure invariant. (So far this is rarely used, element tests in the classical matrix groups are the only situation that comes to mind.)
- Special objects that claim to be certain domains and use bespoke methods. GAP for example implements most prime fields and polynomial rings in this way.

3.3.1 Permutations

A permutation is written in cycle form

```
gap> g:=Group((1,2,3,4,5),(1,2,3));
```

Permutations act from the right, multiplication is accordingly.

3.3.2 Matrices

Matrices are lists of lists of field elements

```
gap> mat:=[[1,2],[3,4]]*Z(5)^0;
[ [ Z(5)^0, Z(5) ], [ Z(5)^3, Z(5)^2 ] ]
```

Matrices implemented as mutable, plain lists are problematic (→ 4.12). GAP therefore converts all matrix group generators into immutable compressed matrices if possible, and all matrix group elements are of this kind as well.

3.4 Finitely presented structures

The first step here is to define a suitable free structure. Relators (for groups and algebras) are elements of this free structure, relations (for semigroups) pairs of elements. The finitely presented structure then is obtained by taking the quotient of the free structure by the relators.

```
gap> f:=FreeGroup("a","b");
<free group on the generators [ a, b ]>
gap> x:=f.1;y:=f.2;
a
b
gap> rels:=[x^2,y^3,(x*y)^5];
[ a^2, b^3, a*b*a*b*a*b*a*b*a*b ]
gap> g:=f/rels;
<fp group on the generators [ a, b ]>
```

The resulting finitely presented structure has generators which are the images of the free generators. Note that the relators are formed from products of variables, the variable names displayed need not correspond to the identifier names. Though they view under the same name, the elements of the free group and the corresponding elements of the finitely presented group are not the same

```
gap> g.1;
a
gap> g.1^3=g.1;
true
gap> f.1^3=f.1;
false
```

You can use `UnderlyingElement` to obtain a representative in the free group. Vice versa `ElementOfFpGroup` (the first argument is the family of elements of the finitely presented group) creates an element of a finitely presented group represented by a given word.

```
gap> UnderlyingElement(g.1);
a
gap> UnderlyingElement(g.1)=f.1;
true
gap> ElementOfFpGroup(FamilyObj(One(g)),f.1*f.2);
a*b
gap> Order(last);
5
```

While two permutation groups created by the same permutations are equal, finitely presented groups given by the same presentation (even as quotient of the same free group by the same relators) are different and incompatible. When trying to multiply elements from different FpGroups, GAP will complain even though the elements seem to be of the same kind (→ 7.2).

At the moment calculations in finitely presented groups will cause the calculation of coset tables for subgroups and (if needed) a faithful permutation representation.

Words are stored in generator/exponent form. This makes operations which are letter-based such as `Subword`, and `Length` comparatively expensive. Try to use syllable indexing instead with `NrSyllables`, `GeneratorSyllable` and `ExponentSyllable`.

3.5 PC groups

A special case is polycyclic presentations (which exist for a finite group if and only if the group is solvable.) For these groups there is an algorithm, *collection*, to compute a normal form for an element. To make use of this, it is not sufficient to write down such a presentation, but the group must be stored in a special form as `PcGroup`. The easiest way to create such a group is via `IsomorphismPcGroup` (→ 3.10). Creating them directly is a bit technical.

You cannot `Print` a Pc group to a file and then read it in. The function `GapInputPcGroup` creates a special string that can be printed to a file to produce a program that creates a pc group with the same pc presentation. This output also shows how to create a `PcGroup` from scratch.

A calculation in a pc group will have to do a large number of exponent tuple calculations (→ 6.2). If the pcgs used is not the *family pcgs* this can cause a severe slowdown. It can be worth to change to an isomorphic pc group, whose family pcgs is better adapted to the calculations, for example by using `IsomorphismSpecialPcGroup`.

3.6 External input

The easiest way to communicate with an external program is to have GAP `PrintTo/AppendTo` input for another program to a file, call a program with `Exec` and let this program print GAP input (assignments to variables) to a file which then is read into GAP.

There also is a more elaborate way of communicating with other processes using streams and one can use this approach to read in arbitrary text files. In particular this removes the need for the external program to print GAP variable assignments (or the need for a translation script). In this case however your code might have to do the parsing itself.

3.7 Standard tasks

Computing things such as an element's `Order`, a group's `Size`, `Normalizer` or `Centralizer` do not require much explanation. Tests for properties start with `Is` such as `IsSolvableGroup`, `IsNilpotentGroup`, `IsPerfect`.

The performance of such operations of course might depend substantially on the chosen representation.

If the same name is used for different concepts for different algebraic structures (for example a Lie algebra is not simple if it is simple as a group), the name of the structure gets added. Similarly there is `KernelOfMultiplicativeGeneralMapping` – the mapping could be additive as well.

There are overlay functions that try to guess the right structure, so `IsSolvable` is recognized as well.

Some operations are related to homomorphisms: `Conversion of representations` (→ 3.10), `Factor groups` (via

NaturalHomomorphismByNormalSubgroup) and group actions are done by asking for the corresponding homomorphism.

3.8 Structured subsets

Conjugacy classes of elements or subgroups and cosets are stored as objects that keep a `Representative`. They usually provide methods for `Size` and `in`. One can use `AsList` or `AsSSortedList` to get an explicit element list.

If the set is an orbit under a group action, it is often stored in the form of an *external set* (\rightarrow **Manual: External Sets**). This means that besides `Representative`, it also supports the attributes `ActingDomain`, `FunctionAction`, and `StabilizerOfExternalSet` (a stabilizer of the representative).

3.9 Homomorphisms

Homomorphisms are one of the most important concepts in algebra. As GAP provided a setup that will support also relations (multivalued functions), the user interface might look a bit complicated at the first look, but most complications can simply be ignored. This is the gist about homomorphisms:

There are three ways to create homomorphisms:

- By assigning images for generators: `GroupHomomorphismByImages`, or — better (as no test for being a homomorphism is done) — `GroupHomomorphismByImagesNC`.

But if there is no algorithm to decompose into generators, evaluating images will cause GAP to list all group elements internally.

- Induced by action: `ActionHomomorphism`. The image permutations are computed by acting on a domain. (To compute preimages we decompose into permutation generators in the image.)

The kind of action is specified by a GAP function whose first argument is a point and the second argument a group element. By default `OnPoints` is used.

Unless you create the homomorphism to be surjective, the range will be the full symmetric group.

- By a function: `MappingByFunction`.

No test for being a homomorphism is done.

The basic operations for a homomorphism are:

- `Source` and `Range` (domain and codomain) – these are usually already stored.
- `ImagesSource` is the image of the homomorphism., `KernelOfMultiplicativeGeneralMapping` the kernel. You can use shorthand commands `Image` and `Kernel` if you like.
- `Image` is also used to compute the image of an element or a subgroup. In contrast to `ImageElm` and `ImagesSet` it will check whether the mapped element is in the source.
- `PreImage` will return the full preimage.
The preimage of a subgroup is always a subgroup, the preimage of an element a list of elements.

- `PreImagesRepresentative` returns *one* preimage of an element.
- `InverseGeneralMapping` creates the inverse ($\hat{\ }^{-1}$ is not recognized for mappings which are not automorphisms),
- `CompositionMapping` (or `*`, but note the different argument order) the composition of mappings.

Do not try to compute images or preimages of objects such as cosets or conjugacy classes – GAP will often take the preimage of them as sets of elements and return a large element list. Instead take the preimage of the `Representative` and create a class or coset anew in the original group.

3.10 Conversion

For reasons of efficiency (\rightarrow 3.2) or because of nonavailability of methods it can be helpful or necessary to convert from one representation to another. The standard way to change representations is to create a homomorphism onto an isomorphic domain in the new representation. This isomorphism can also be used to transfer results between the old and the new representation. The `Image` of the isomorphism is the new domain.

```
gap> g:=GL(3,3);
GL(3,3)
gap> iso:=IsomorphismPermGroup(g);
<action homomorphism>
gap> Image(iso);
Group([ (10,19)(11,20)(12,21)(13,22)(14,23)
(15,24)(16,25)(17,26)(18,27),
(2,7,10,20,18,5,25,21,15,14,17,8,16)
(3,4,19,12,23,9,13,11,26,27,24,6,22) ])
```

3.11 Computing in another representation

As long as one is only interested in the abstract group, the best strategy usually is to go to a “good” representation and do all calculations there. In some cases however one needs a result in a particular representation which is not optimal for calculations.

In such a situation one can calculate in the “nicer” representation and then pull the result back by taking pre-images under the isomorphism.

In particular one can take pre-images of attributes and store them as attributes in the original group, thus having the information automatically available.

```
gap> g:=Group((1,2,3,4),(1,2),(5,6,7));;
gap> iso:=IsomorphismPcGroup(g);;
gap> h:=Image(iso);;
gap> z:=Centre(h);;
gap> SetCentre(g,PreImage(iso,z));;
gap> cl:=ConjugacyClasses(h);;
gap> ncl:=[];
gap> for c in cl do
> nc:=ConjugacyClass(g,
>   PreImage(iso,Representative(c)));;
> SetSize(nc,Size(c));
> SetStabilizerOfExternalSet(nc,
>   PreImage(iso,StabilizerOfExternalSet(c)));;
> Add(ncl,nc);
```

```
> od;
gap> List(nc1,Size);
[ 1, 1, 6, 8, 3, 1, 6, 8, 3, 6, 6, 8, 3, 6, 6 ]
gap> SetConjugacyClasses(g,nc1);
```

A useful operation to know in this context is `SmallerDegreePermutationRepresentation` which tries to find a smaller degree faithful permutation representation for a permutation group.

3.12 NiceMonomorphisms

This situation happens all the time for matrix groups and groups of automorphisms. The best way implemented so far to compute with these groups is via a faithful permutation representation. The way this is done in GAP is via the concept of a *nice monomorphism*:

- There is a special filter `IsHandledByNiceMonomorphism`.
- This filter is implied by `IsMatrixGroup` and `IsFinite`, respectively by `IsGroupOfAutomorphisms`. (Note: `IsGroupOfAutomorphisms` is not set automatically if you create the group via `Group`.)
- There are high-ranking special methods (in `lib/grpnice.g?`) applicable under the condition `IsHandledByNiceMonomorphism`. These methods translate the input via the `NiceMonomorphism` of the group.
- `NiceMonomorphism` is an attribute. It must be a homomorphism that can be evaluated without much information about the group. Thus it is usually an action homomorphism.

3.13 Character Tables

The typical questions here are to compute a table, get the `CharacterDegrees` or to compute structure constants.

`CharacterTable` takes either a group (in this case the table will be computed) or a string (fetching the table from the library) as argument.

Unless you deliberately ask for `Irr` (the list of irreducible characters) the table might not be computed in full.

Characters are a special case of *class functions*. A class function is created from a list of values by `ClassFunction`, the list of values is stored as `ValuesOfClassFunction`. `ScalarProduct` computes the scalar product of class functions.

`InducedClassFunction` and `RestrictedClassFunction` compute inductions and restrictions, but one needs the fusion map of the classes. If such a map is not stored already in the table library, one can compute candidates with `PossibleClassFusions`, often the number of choices can be restricted with `RepresentativesFusions`:

```
gap> c:=CharacterTable("A7");;
gap> d:=CharacterTable("A6");;
gap> Irr(d)[2];
Character( CharacterTable( "A6" ),
  [ 5, 1, 2, -1, -1, 0, 0 ] )
gap> List(Irr(c),i->i[1]);
```

```
[ 1, 6, 10, 10, 14, 14, 15, 21, 35 ]
gap> fus:=PossibleClassFusions(d,c);
[[1,2,3,4,5,6,6],[1,2,4,3,5,6,6]]
gap> fus:=RepresentativesFusions(
  > AutomorphismsOfTable(d),
  > fus,AutomorphismsOfTable(c));
[ [ 1, 2, 3, 4, 5, 6, 6 ] ]
gap> StoreFusion(d,c,fus[1]);
```

```
gap> ind:=InducedClassFunction(Irr(d)[2],c);
Character( CharacterTable( "A7" ),
  [ 35, 3, 8, -1, -1, 0, 0, 0 ] )
gap> res:=RestrictedClassFunction(ind,d);
Character( CharacterTable( "A6" ),
  [ 35, 3, 8, -1, -1, 0, 0 ] )
gap> ScalarProduct(d,res,res);
12
gap> MatScalarProducts(d,Irr(d),[res]);
[ [ 1, 3, 0, 0, 0, 1, 1 ] ]
```

`Display` will print the table nicely. You might want to use `SizeScreen` and `LogTo` to produce an output of suitable line length for printing. GAP tries to use the ATLAS notation for algebraic irrationalities.

The classes of a character tables are indexed by numbers. If the table is computed from a group the class arrangement is likely to be a bit arbitrary (it corresponds to the attribute `ConjugacyClasses` of the table which is a list of classes of the group). For tables in the library it is the same as given in the ATLAS.

You can use `ClassNames` to get names for the conjugacy classes and (once `ClassNames` has been called) refer to class numbers via their name.

```
gap> ClassMultiplicationCoefficient(c,2,3,2);
12
gap> ClassNames(c);
["1a","2a","3a","3b","4a","5a","6a","7a","7b"]
gap> c.7a;
8
```

4 Harder Problems

A problem which one wants to investigate on the computer typically falls in one of the following three classes:

- Compute a property or associated value/object (size, character table, number of classes, p -class) for a given domain. These questions are usually easily asked to the system. In many cases there is already an operation for this defined (\rightarrow **8**) about own operations).

The only problem might be to get the right group in the first place (\rightarrow **5.1**)

- Find an element or substructure with certain properties. While there are GAP functions that can do this for moderate-sized problems, larger problems might require some care in phrasing the problem.
- Classify (or at least count) all objects with certain properties. This can be a major task, requiring a lot of preparation and task-specific programming.

4.1 General Strategies

- Do not ask for more than you need. Use the most specific command available.

For example to prove that groups are (non)isomorphic with no need for an explicit isomorphism compute invariants, use `IdGroup` and `RandomIsomorphismTest` for small solvable groups.

- It might be easier not to check the definition of a property, but (as a first step if they are not equivalent) deduced properties.

For example when looking for characteristic subgroups you need to check only normal subgroups.

- Reduce the search space.

4.2 Reducing the search space

The most naïve reduction is to reduce the group you work in. For example a problem that only involves p -elements can be solved in a Sylow subgroup and a better representation for this group may then be used.

A more natural approach is to use symmetries. This reduces the search space by the size of the acting symmetry group.

Since we work in a group itself, chances are good that this group (or its automorphism group or at least a subgroup) acts by conjugation on potential solutions. It is thus sufficient to search for the solution only up to conjugacy. In practice this means running only over class representatives (\rightarrow 4.7).

When using conjugacy under the automorphism group it can become necessary to change the representation to one, in which the automorphisms can be realized.

4.3 Homomorphism principle

Another, often even better and not mutually exclusive, way of reduction is to look at homomorphisms. If the problem is in some way invariant under homomorphisms, try to solve it first in a (smaller) homomorphic image and then to lift the result to the group. Using a series of normal subgroups (for example a `ChiefSeries`) this can be iterated. Use `NaturalHomomorphismByNormalSubgroup` to construct the factor groups.

Many of the algorithms for solvable groups work this way (\rightarrow 6.2).

4.4 All Elements

One type of questions asks for elements with certain properties. GAP can list all elements of a group, but this will not only clog up the memory (one can often avoid this sometimes by using an `Enumerator` instead), but searching for all elements is likely to be slow.

If conjugacy is unimportant, try to use only class representatives.

Elements of certain orders might reside in smaller subgroups (search for p -elements in Sylow subgroups).

If the group is very big, but you expect that certain elements exist, a random search might bring up the desired type of elements. (This can give a solution but of course cannot be used to disprove a conjecture.)

A few rules of thumb:

- How likely is a random search to find an element, respectively: How many elements of the group should be “good”? (For example random search might be a good strategy to find a pair of elements whose product is in a given class, it is not good for finding an element to conjugate one given subgroup to another.)
- If the group is big, do not use `Random` (which guarantees equal distribution but therefore might need to precompute a lot of information) but `PseudoRandom`.
- Elements of small order are hard to find directly. But they often have a root of high order which is easy to find.

4.5 All Subgroups

Everything said for elements holds for subgroups even more. There are operations to compute the subgroup lattice, but if the size of the group gets in the range of 10^6 , or if the group has large elementary abelian subfactors (vector spaces have *awfully* many subgroups) problems arise.

There are a few classes of subgroups which can be found comparatively easily:

- Cyclic subgroups (via `ConjugacyClasses`).
- `NormalSubgroups`.
- Derived, Lower Central etc. series.
- Sylow subgroups.
- Hall subgroups (so far only for solvable groups).
- Maximal subgroups (so far only for solvable groups).
- Complements to solvable normal subgroups.

`MaximalSubgroups` will return *all* subgroups. You are likely to want only `MaximalSubgroupClassReps`.

- Subgroups derived from these.

In particular, it can be worth to construct step-by-step by first constructing only subgroups of a subgroup, and then obtaining the desired subgroups as their `Normalizers` or similar. A particular advantage is that the extending step then can be done sequentially with no need to keep all subgroups simultaneously in memory.

The homomorphism principle might reduce the search to a subgroup (the preimage of the solution in the image).

At the moment, GAP uses two different methods to compute the subgroup lattice of a group. Both provide ways (which are corresponding to the algorithm employed and therefore differ) to compute only a subset of groups.

`SubgroupsSolvableGroup` works only for solvable groups and uses an approach based on the homomorphism principle. It can be adapted to compute only groups invariant under given automorphisms, or of at least a certain size.

`LatticeByCyclicExtension` works for all groups and uses the cyclic extension algorithm. It can be adapted to stop extending groups which don't fulfill certain properties which can be effective if only “small” subgroups are searched for.

If you *must* run a subgroup lattice calculation consider adapting either of the algorithms for your task – by adding extra checks it is usually possible to prune the construction tree much better than could be done with the predefined features.

4.6 Example 1: Testing a conjecture

A typical question solved by checking the data library is (P. CAMERON, group-pub-forum, 1999):

If P is a 2-group which is not elementary abelian, then some non-identity element of the centre of P is a square?

We want to check this for the groups given in the small groups library. For this we have to run through the groups of a given size (2-power) and check the property for each nonabelian group.

Checking whether a given element is a square (i.e. searching for the roots) can be a bit hard. (Roots must centralize the element, but this does not help here.) But we search only for a nontrivial central element which is a square. And of course we do not need to consider all elements, but only representatives up to conjugation.

In other words: Is there a class representative which has a nontrivial power in the centre. Of course we can take this power to be of the smallest possible order, that is 2.

```
Check:=function(size)
local i,g,r,reprs,prop,z;
for i in [1..NrSmallGroups(size)] do
g:=SmallGroup(size,i);
if not IsAbelian(g) then
# test
z:=Centre(g);
prop:=false; # so far we did not find
# a suitable element
reprs:=List(ConjugacyClasses(g),Representative);
for r in reprs do
if Order(r)>2 and r^(Order(r)/2) in z then
prop:=true;
fi;
od;
if not prop then return g; fi;
fi;
Print(".\c"); # progress report
od;
return true;
end;
```

Running for size 128, we find a counterexample:

```
gap> Check(128);
.....
<pc group of size 128 with 7 generators>
```

4.7 Example 2: Generating Set

Consider the question:

$U_3(3)$ cannot be generated by three involutions but by an involution and an element of order 6.

The naïve way is to look though all n -tuples of generators. Of course we need to do this only up to conjugacy. Such a search is provided by GQuotients from free product of the desired cyclic groups:

```
gap> g:=PSU(3,3);
gap> F:=FreeGroup(3);
gap> F:=F/[F.1^2,F.2^2,F.3^2];
gap> GQuotients(F,g);
[ ]
```

Similarly, 2, 3, 2, 4 and 2, 5 generation fails, but:

```
gap> F:=FreeGroup(2);;F:=F/[F.1^2,F.2^6];;
gap> GQuotients(F,g);
[[f1,f2] -> [(1,61)(2,90)(3,29)(4,49)... ]
[f1,f2] -> [(1,28)(2,14)(3,33)(5,84)... ]]
```

The images of the generators of F give the desired 2,6 generating set.

4.8 Details

But let's look at this example in slightly more detail:

By conjugating in the image group, we can assign the image of the first generator up to conjugacy. The image of the second generator then is determined only up to conjugacy with elements *centralizing* the first generator image. So the image of the second generator is determined up to $C_G(g_1)$ -conjugacy, where g_1 is the image of the first generator. We get those classes by conjugating class representatives g_2 with representatives of the double cosets $C_G(g_2) \backslash G / C_G(g_1)$.

We could code it in the following way:

```
class:=ConjugacyClasses(G);
for c1 in class do
img1:=Representative(c1);
for c2 in class do
dc:=DoubleCosets(G,Centralizer(c2),
Centralizer(c1));
dc:=List(dc,Representative);
# Even better: DoubleCosetRepsAndSizes
for rep in dc do
img2:=Representative(c2)^rep;
# now check whether img1,img2 is
# a suitable set of images
od;
od;
od;
```

There is a whole family of algorithms similar to GQuotients, which are used for IsomorphicSubgroups, for Isomorphism and AutomorphismGroup (of nonsolvable groups). They try to find images for a generating set. Their runtime therefore depends not only on the size of a image group but also the size of a generating set. For example they don't perform well for p -groups.

If the group gets very big, such an exhaustive search might take too long. In this case one might want to look only at random elements.

4.9 Orbit-Type algorithms

The main tool to work with the action of a group is via the orbit-stabilizer algorithm.

One computes images under the group generators until no new images arise. By keeping transversal elements as well, one can generate Schreier generator for the point stabilizer:

```
orb:=[pnt];
t:=[One(group)];
s:=TrivialSubgroup(group);
for p in orb do
for g in gens do
img:=p^g;
if not img in orb then
```

```

    Add(orb,img);
    Add(t,t[Position(orb,p)]*g);
else
    s:=ClosureGroup(s,
        t[Position(orb,p)]*g
        /t[Position(orb,img)]);
fi;
od;
od;

```

At the end, `orb` contains the full orbit and `s` is the stabilizer of `pnt`.

Instead of “~” another *action function* can be used.

As only the generators act, it is possible to act implicitly via a homomorphism, given only by the generator images. A typical case is the action of a group on an elementary abelian subfactor via matrices. For this it is not necessary to be able to compute the images of other elements.

The user interface for operations thus is quite general. Parameters are:

- the acting group,
- the domain,
- a (start) point,
- group generators and their acting images,
- an action function.

Most of the parameters are optional with automatic default values. To avoid implementing 2^n different almost equal operations, all user commands are actually functions (say `Stabilizer`) that will supply the default parameters and then call the actual operation (say `StabilizerOp`) with a full parameter set.

The only drawback is that this makes the calling stack displayed in the break loop a bit more complicated: Typically the operation that does the work will show up under the name `orbish`.

4.10 Object Lookup

To get good performance, it is crucial to get the test `img` in `orb` and the `Position` tests to perform quickly. Depending on the kind of objects involved, there are various possibilities:

- Linear search in a list.
- Binary search in a sorted list. (Requires `fast <` comparison.)
- Hashing. (Requires hash key function.)
- Perfect hashing. (If one can get the index in the domain fast.)

GAP takes care of all these using *dictionaries*. The basic operations for these are:

- `NewDictionary(obj,look[, actiondomain])` creates a new dictionary. `obj` is one “typical” object you will work with `actiondomain` a set that will contain all orbits.
- `AddDictionary(dict,obj[, value])` adds a new element (with position value).

- `KnowsDictionary(dict,obj)` checks whether `obj` is known.

- `LookupDictionary(dict,obj)` returns the corresponding (position) value.

4.11 Linear algebra

Linear algebra exists on two different levels in GAP. The top-level is that of abstract vector spaces. These are domains which admit operations such as `Intersection`, `in`, `IsSubset`, `Size`. Elements of this spaces are not necessarily row vectors, even though the methods internally use coefficient lists.

On this level a `Basis` is an object itself which permits the computation of `Coefficients`.

For use within algorithms the cost caused by this comfort might be too high and it is better to stay on the level of row vectors and matrices.

Vectors are represented in GAP as lists of field elements, matrices are lists of vectors. Addition and multiplication of vectors and matrices performs the usual products. A vector is automatically transposed if the product otherwise does not make sense.

There are however a couple of performance issues related to matrices and vectors that might be crucial to get satisfactory performance.

4.12 Mutability

The fact that vectors and matrices are built from lists can cause two types of problems. The first is with potentially unfriendly code:

```

gap> v1:=[1,2,3];v2:=[4,5,6];;l:=[v1,v2];
[ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
gap> IsSSortedList(l);
true
gap> v1[1]:=5;
5
gap> IsSSortedList(l);
false
gap> IsVector(v1);
true
gap> v1[1]:='a';
'a'
gap> IsVector(v1);
false

```

The assignments to `v1` might be sideeffects from code that gets called. So a list of vectors could not store that it is sorted, nor could an object have the type “vector”.

We get around this problem by declaring objects to be *immutable*, that is not admitting any change. By definition, this is inherited by all subobjects.

Lists of immutable objects may store that they are sorted, immutable vectors can remember their type.

- Vectors and Matrices should be immutable, unless you want to change them physically.
- Products of immutable matrices/vectors are immutable again.

- Use `ShallowCopy` for vectors (or `List(m,ShallowCopy)` for matrices) to get a modifiable copy. (You will need to do this surprisingly infrequent, as modification does not correspond to an operation in the matrix algebra.)

4.13 Fast matrix arithmetic

GAP will consider a list of lists (of the same lengths) of ring elements as a matrix and provides matrix multiplication for such objects. As the ring elements might be “library” objects, the multiplication routine thus has to go through the generic multiplication/addition routines.

If element arithmetic is very cheap (for example over finite fields where one can do table lookup) this can greatly decrease the performance. GAP therefore contains special representations for compact vectors and matrices over small (up to size 256) finite fields. Arithmetic with these compact objects is much faster than with generic lists. However access to entries via the sublist operator can be a bit slower.

As long as one works in the matrix algebra or in a vector space without changing objects, it is beneficial to use these compact representations.

A vector can be brought into compact form by `ConvertToVectorRep(vec,field)` which *changes* a vector *vec* in place. *field* is either a finite field or a field size.

A compact matrix consists of rows of immutable compact vectors over the same field.

A further subtlety with this is given by field extensions. The compact representation over GF(4) is different than the compact representation over GF(16), even if all matrix entries are in GF(4).

Depending on the input, you cannot assume that all rows of a matrix are in the right representation, but might need to convert (or even recreate) lines.

The function `ImmutableMatrix(field,matrix)` returns a mathematically equal, immutable, matrix which is written in compact form over *field* and is obtained by converting or rewriting as necessary.

4.14 Gauss-Type algorithms

There is a variety of algorithms available that compute triangularized and normal forms of matrices.

These operations still undergo performance engineering. In the next release there will be extra versions that work destructively (avoiding to copy the matrix first).

Almost all algorithms for linear algebra perform row or column operations, i.e. multiply a row/column by a factor or add a multiple of a row/column to another.

To access columns, the sublist operator can be useful: `mat{[1..dim]}[nr]` returns a column vector of the matrix.

There are operations `MultRowVector` and `AddRowVector` that will *replace* a row vector *v* by *av*, respectively *v+aw*.

4.15 Polynomials

GAP supports (the quotient field) of multivariate polynomial rings. (But there is no multivariate GCD so far.) There is basic arithmetic and univariate factorization.

I’m not aware of anyone who has seriously used the multivariate polynomial arithmetic for topics such as Gröbner

Bases. I would be most interested to hear about performance bottlenecks.

Polynomials are stored in terms of indeterminates (which are represented by numbers internally). You can create polynomials by arithmetic from monomials:

```
gap> x:=X(Rationals,"x");
x
gap> y:=X(Integers,"y");
y
gap> 4*x*y+3*x^7*y;
4*x*y+3*x^7*y
```

There is no correlation between variable names and printing names.

Polynomials live in a *characteristic*, not over a single ring. The polynomial $3x$ over the integers is equal to $3x$ over the rationals. When factorizing, the appropriate field must be given.

Alternatively, one can use the internal representation. For this one needs to get the rational functions family (in our case over the cyclotomics, represented by the number 1):

```
gap> rfam:=RationalFunctionsFamily(FamilyObj(1));
gap> PolynomialByExtRep(rfam,
[ [2,1,3,1], 4, [2,7,3,1], 3 ]);
4*x*y+3*x^7*y
```

The monomials must be sorted according to degree/lex order! In most cases you will not need to define a polynomial ring. Polynomials themselves are defined over a family, so there is no need to distinguish between $GF(3)$ and $GF(9)$, say.

5 Building Groups

5.1 Obtaining a group

The typical case when working with groups is not that one has generators or a presentation, but a more informal description. The first task therefore usually is to get the group(s) one wants.

5.2 Groups included with the system

If the group is “small”, chances are not too bad, that an isomorphic group is part of the data libraries that come with GAP. These include:

- Classical groups (S_n , $PGU(n,q)$, $O_n^+(q)$, ...)
- Groups of order up to 1000. (Isomorphism types.)
- Transitive groups up to degree 23. (Permutation isomorphism)
- Perfect groups of order up to 10^6 . (Isomorphism types.)
- Primitive groups up to degree 999. (Permutation isomorphism.)

A full list with author information can be found in the manual. The libraries provide selection functions to find all groups with certain properties. For example

```
AllPrimitiveGroups(SocleTypePrimitiveGroup,
rec(series:="A",parameter:=6,width:=2));
```

will find all primitive groups whose socle is isomorphic to $A_6 \times A_6$. See `IsomorphismTypeFiniteSimpleGroup` for a description of the names of the socles.

Try to get a small list of candidates and then check their properties to end up with one candidate.

For example, lets find $M_{12,2}$:

```
gap> l:=AllPrimitiveGroups(Size,190080,
> SocleTypePrimitiveGroup,
> rec(series:="Spor",parameter:="M(12)",width:=1));
[m12#144.2,m12#144n.1,m12#396.2,m12#495.2,
m12#495a.2,m12#880.1]
```

The names in the primitive groups library refer to numbering according to *cohorts*. A .n suffix is an index number and does not reflect the composition structure.

The selection functions treat some precomputed information specially and do not know about implications. Thus it can be worth to add superfluous conditions that act as a cheap filter. For example the (obvious) version

```
AllSmallGroups(Size,720,IsPerfectGroup,true);
```

is magnitudes slower than

```
AllSmallGroups(Size,720,IsSolvableGroup,false,
IsPerfectGroup,true);
```

For Almost/Quasi simple groups the lists of perfect groups or primitive groups are good hunting grounds, if they prove to be inefficient, the ATLAS web pages <http://www.mat.bham.ac.uk/atlas/> are a useful repository of generators.

5.3 Constructions

A more general approach is to try to build the group from smaller constituents, using generic product or extension theory.

A descriptive name such as $2.A_5.2$ or structure description may not correspond to one unique group. An explicit construction will end up constructing all possible candidates and you will have to decide which one you want.

Not all product constructions work for all types of groups. You will have most success with two groups in similar representations, preferably both permutation groups or both Pc groups (\rightarrow 3.10).

5.4 Semidirect Products

Semidirect products initially might look a bit scary as they are built from a lot of homomorphisms. These are required to tell how the complement acts on the normal subgroup. So the construction is not really hard, but one might have to be a bit stubborn.

Lets construct $5^2 : SL_2(5)$:

Of course one could immediately write down affine matrices for the group as well, but I want to show how to use the `SemidirectProduct` construction.

First we need the normal subgroup: $GF(5)^2$. This is an abelian group. We want it as a permutation group, because $SL_2(5)$ is not solvable:

```
gap> N:=AbelianGroup(IsPermGroup,[5,5]);
gap> bas:=IndependentGeneratorsOfAbelianGroup(N);
[ (1,2,3,4,5), (6,7,8,9,10) ]
```

Then we get $SL_2(5)$ – in the matrix form to compute the action and in permutation form for the semidirect product:

```
gap> S:=SL(2,5);
gap> iso:=IsomorphismPermGroup(S);
<action homomorphism>
gap> SPE:=Image(iso);;
```

Now for the action on the vector space. For each matrix generator we need to write down the corresponding automorphism. For this we have to do the matrix multiplication by hand.

```
gap> mats:=GeneratorsOfGroup(S);m1:=mats[1];
[[Z(5),0*Z(5)],[0*Z(5),Z(5)^3]];
gap> Product([1,2],i->bas[i]^Int(m1[1][i]));
(1,3,5,2,4)
```

This is the image of the first basis element under the first matrix. We need to map both generators.

```
gap> hom1:=GroupHomomorphismByImages(N, N, bas,
> List([1,2],
> b->Product([1,2],i->bas[i]^Int(m1[b][i]))
> ) );
[ (1,2,3,4,5), (6,7,8,9,10) ]
-> [ (1,3,5,2,4), (6,9,7,10,8) ]
```

Since we don't need a check, we can use the NC variant. And of course we need homomorphisms for all matrix generators:

```
gap> homs:=List(mats,
> m-> GroupHomomorphismByImagesNC(N, N, bas,
> List([1,2],b->Product([1,2],
> i->bas[i]^Int(m[b][i]))
> ) ) );
[[ (1,2,3,4,5), (6,7,8,9,10) ]
-> [ (1,3,5,2,4), (6,9,7,10,8) ],
[ (1,2,3,4,5), (6,7,8,9,10) ]
-> [ (1,5,4,3,2), (6,7,8,9,10), (1,5,4,3,2) ]]
```

GAP would find it out by itself, but it is a bit quicker if we tell it that these mappings are indeed bijections:

```
gap> for h in homs do SetIsBijective(h,true);od;
```

Now we form a group of automorphisms of N and a mapping from the permutation group into this group. The generating sets to map all correspond to the original matrices.

```
gap> au:=Group(homs);
<group with 2 generators>
gap> auiso:=GroupHomomorphismByImagesNC(SPE,au,
> List(mats,i->Image(iso,i)),
> homs);
[ (2,5,4,3)(6,11,16,21)(7,15,19,23)(...),
(2,16,9)(3,21,15)(4,6,17)(5,11,23)(...) ]
-> [[ (1,2,3,4,5), (6,7,8,9,10) ]
-> [ (1,3,5,2,4), (6,9,7,10,8) ],
[ (1,2,3,4,5), (6,7,8,9,10) ]
-> [ (1,5,4,3,2), (6,7,8,9,10), (1,5,4,3,2) ]]
```

Be sure not to forget the NC here – checks can be very expensive

Now we can form the product:

```

gap> SDP:=SemidirectProduct(SPE,auido,N);
<permutation group with 4 generators>
gap> DisplayCompositionSeries(SDP);
G (5 gens, size 3000)
 | A(5)
S (3 gens, size 50)
 | Z(2)
S (2 gens, size 25)
 | Z(5)
S (1 gens, size 5)
 | Z(5)
1 (0 gens, size 1)

```

The operations `Embedding` and `Projection` permit to get back the constituent groups from the product.

Exercise: Construct $(2^4).SL_2(4)$. You will need a $GF(2)$ -basis of $GF(4)$. Or use `BlowUpMat`. Construction of semidirect products will be a bit easier in release 4.3

5.5 Forming Extensions

More general forms of products – central products or amalgams – are not supported directly by functions. Instead we have to construct these step by step by hand. I'll show how to get groups of the form $3.(A_6 \times A_6).2$, an extension of a central product with an outer automorphism acting simultaneously on both A_6 -factors as induced by $GL(2,9)$ ($A_6 \cong PSL_2(9)$).

The first step is to get one $3.A_6$. We get it from the perfect groups library:

```

gap> NrPerfectGroups(3*360);
1
gap> g:=PerfectGroup(IsPermGroup,3*360,1);
A6 3^1

```

For $3.(A_6 \times A_6)$ we factor out the diagonal 3 in the direct product

```

g> d:=DirectProduct(g,g);
g> e1:=Embedding(d,1);
1st embedding into <permutation group
of size 1166400 with 6 generators>
g> e2:=Embedding(d,2);
2nd embedding into <permutation group
of size 1166400 with 6 generators>
g> zg:=Centre(g).1;;
(1,2,4)(3,5,8)(6,9,12)(7,10,15)(11,16,14)(13,17,18)
g> zg:=Image(e1,zg)*Image(e2,zg);
g> diag:=Subgroup(d,[zg]);
g> nhom:=NaturalHomomorphismByNormalSubgroup(d,
diag);
g> cp:=Image(nhom);
<permutation group of size 388800 with 6 generators>
g> Size(Centre(cp));
3

```

Now we know how the outer automorphism should act on both A_6 , but we don't know how to lift it to the full group. We therefore compute the automorphism group of the central product, and search for a suitable automorphism in there.

```

gap> aug:=AutomorphismGroup(cp);
<group with 10 generators>
gap> Size(aug);
2073600

```

This calculation takes a while.

We now get representatives for the outer automorphisms of order 2. We do so by computing class representatives in the outer automorphism group and taking preimages of the representatives of order 2.

```

g> autin:=InnerAutomorphismsAutomorphismGroup(aug);
<group of size 129600 with 6 generators>
g> authom:=NaturalHomomorphismByNormalSubgroupNC
> (aug,autin);;
g> out:=Image(authom);;
g> cl:=List(ConjugacyClasses(out),Representative);
[ <identity> of ..., f1, f2, f3, f4, f1*f3, f1*f4,
f2*f4, f3*f4, f1*f3*f4 ]
g> cl:=Filtered(cl,i->Order(i)=2);
[ f1, f2, f3, f4, f1*f4, f2*f4, f3*f4 ]
g> cl:=List(cl,
> i->PreImagesRepresentative(authom,i));;

```

Next, we need to filter those, which act in the right way. First we ensure that both normal subgroups $3.A_6$ are fixed.

```

gap> n:=NormalSubgroups(cp);
[ Group(), <perm. group of size 3 with 1 gens.>,
<perm. group of size 1080 with 5 gens.>,
<perm. group of size 1080 with 5 gens.>,
<perm. group of size 388800 with 6 gens.> ]
gap> cl:=Filtered(cl,i->Image(i,n[3])=n[3]);;
gap> Length(cl);
5

```

Now we check, which automorphisms are induced on the factors isomorphic A_6 . For this we construct a mapping on the first subfactor.

Since we want to compute with automorphisms, we compute the full automorphism group as well. This way GAP internally computes a faithful permutation representation that is used for all calculations (\rightarrow 3.12).

```

g> n1hom:=NaturalHomomorphismByNormalSubgroup
(n[3],n[2]);
<action homomorphism>
g> a6:=Image(n1hom);;
g> a6aut:=AutomorphismGroup(a6);
<group with 7 generators>
g> a6inn:=InnerAutomorphismsAutomorphismGroup
(a6aut);
<group with 5 generators>

```

In the next step, we restrict the outer automorphisms found before on the normal subgroup and compute the automorphism induced on the factor. The image of each group under its `NiceMonomorphism` is an isomorphic permutation group.

```

g> ind:=List(cl,i->
InducedAutomorphism(n1hom,
RestrictedMapping(i,n[3]));;
g> indgp:=List(ind,
i->ClosureSubgroupNC(a6inn,i));
[ <group>, <group>, <group>, <group>, <group> ]
g> indprm:=List(indgp,
i->Image(NiceMonomorphism(i,i));;

```

As the groups are small enough, we can use `IdGroup` to identify their type (otherwise we would have to do an isomorphism test). The third and fourth automorphisms are of PGL type.

```
gap> List(indprm,IdGroup);
[ [ 360, 118 ], [ 720, 765 ], [ 720, 764 ],
  [ 720, 764 ], [ 720, 763 ] ]
gap> IdGroup(PGL(2,9));
[ 720, 764 ]
gap> c12:=c1{[3,4]};;
```

You might ask why I did not call `IdGroup` on the automorphism groups in ‘indgp’. Indeed this *ought* to work – alas there was still an error lurking in the system, that will be fixed in the next release.

We now duplicate the calculation for the second normal subgroup: It turns out only the first automorphism in `c12` is of the right type.

```
g> n2hom:=NaturalHomomorphismByNormalSubgroup
      (n[4],n[2]);;
g> a6:=Image(n2hom);;
g> a6aut:=AutomorphismGroup(a6);;
g> a6inn:=InnerAutomorphismsAutomorphismGroup
      (a6aut);;
g> ind:=List(c12,
  > i->InducedAutomorphism(n2hom,
  > RestrictedMapping(i,n[4]));;
g> indgp:=List(ind,i->ClosureSubgroupNC(a6inn,i));
[ <group>, <group> ]
g> indprm:=List(indgp,
  i->Image(NiceMonomorphism(i),i));;
g> List(indprm,IdGroup);
[ [ 720, 764 ], [ 720, 763 ] ]
g> outo:=c12[1];;
```

We finally have the group and a suitable automorphism, its time to build the extension. However, there is no generic code for constructing extensions (there is some code for solvable groups), mainly because specifying the cocycle information could be tedious. We therefore use the theorem, that an extension $N.G$ can be embedded in the wreath product $N \wr G$ with G acting regularly.

```
gap> w:=WreathProduct(cp,Group((1,2)));
<permutation group of size 302330880000
  with 13 generators>
gap> e1:=Embedding(w,1);;
gap> e2:=Embedding(w,2);;
gap> e3:=Embedding(w,3);;
```

The base subgroup (isomorphic to N) of the embedded group consists of diagonals under the action of G .

```
gap> diag:=List(GeneratorsOfGroup(cp),
  > i->Image(e1,i)*Image(e2,Image(outo,i)));;
gap> u:=Subgroup(w,diag);;
gap> Size(u);
388800
```

We then find an element that maps these diagonal generators in the same way the automorphism does. This element normalizes the subgroup.

```
g> diaemb:=GroupHomomorphismByImages(cp,w,
  > GeneratorsOfGroup(cp),diag);;
g> diagim:=List(GeneratorsOfGroup(cp),
  i->Image(diaemb,Image(outo,i)));;
g> r:=RepresentativeOperation(w,
  > diag,diagim,OnTuples);;
g> Order(r);
30
g> u^r=u;
true
```

The resulting group however has an extra (centralizing) 3. We thus go to the factor group to eliminate the 3 there.

```
g> cu:=ClosureGroup(u,r);;
g> Index(cu,u);
6
g> Size(Centralizer(cu,u));
9
g> Size(Centralizer(w,u));
9
g> hom:=NaturalHomomorphismByNormalSubgroup(cu,u);;
g> gp:=PreImage(hom,SylowSubgroup(Image(hom),2));
<permutation group with 7 generators>
```

As 2 and 3 are coprime, there is no problem of isoclinism with this group. (Otherwise we would have a larger centralizer and choices about which subgroup of the factor to take.)

5.6 Blatant Advertisement

A further longish example of constructing groups can be found in the article of S. LINTON and mine: <http://www-gap.dcs.st-and.ac.uk/~ahulpke/paper/bathexample.html>

5.7 Automorphism Groups

One of the most important applications of group theory are automorphisms of structures. They can often be represented as permutation groups acting on a suitable set (for example the vertices of a graph).

Finding such automorphism groups is a much harder task. The share packages GRAPE and GUAVA provide interfaces to external programs that compute automorphism groups of graphs or codes.

There is a generic way to do this in GAP (`SubgroupProperty` finds the largest subgroup of S_n or another permutation group that fulfills a property). In most cases the performance of this will be very slow. To get a better performance one would have to adapt the generic backtrack search to make more use of the concrete problem given when pruning the search tree (→ **Manual: Extending GAP**).

6 Data structures and Algorithms

It would be easy to fill several lectures with descriptions of all the algorithms used in GAP and I won’t even try to attempt to do this here.

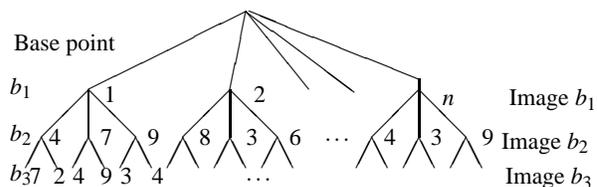
On the other hand a basic knowledge of the algorithms and data structures involved might be crucial for estimating which kinds of calculations are feasible and which are not.

6.1 Permutation Groups

The data structure used to represent a permutation group (given by generators) is called a *stabilizer chain*. The idea behind this is to take a sequence of points $[b_1, b_2, \dots, b_l]$ (a *base*) such that the only element of the group that fixes all points is the identity.

Unless the group is the full symmetric group, a base is (often much) shorter, than the permutation domain.

Each group element g then corresponds to a base image $[b_1^g, b_2^g, \dots, b_l^g]$. We can arrange these base images on the leaves of a tree, whose branches on different levels correspond to the images of the first, second, etc. base point.



Since there is a group acting, we only need to store the left branches each time.

The first step needed to compute with a permutation group is to compute such a stabilizer chain (and base). The hard part therein is to know that the full chain has been computed. If the group's size is known, one can speed up this process substantially by storing the size in the group using `SetSize`. It is also possible to permit a randomized calculation (which might end up with a size too small in rare cases). (→ **Manual: Permutation Groups**).

An element is contained in the group, if we can trace *its base images* through the tree and end up with the same element as the group. This process is called *sifting*.

When searching for elements to find centralizer, normalizer, conjugating elements, etc., we run through the tree in a backtrack algorithm. Much care is taken to prune branches, *but essentially these remain exponential algorithms*. The commands `SubgroupProperty` and `ElementProperty` offer a simple-minded interface for these operations, however they do not offer any clever pruning.

6.2 PCGS

The algorithms for Pc-Groups use a general data structure, called a *polycyclic generating system*. Such systems exist for every finite solvable group, but computing with them is most efficient for PcGroups where the internal data structures used for arithmetic already form a pcgs.

To get a pcgs, take a subnormal series of the group with cyclic factors. On each level, take a representative g_i , which generates the cyclic factor. The resulting sequence $[g_1, \dots, g_n]$ forms a pcgs.

Each group element can be written in unique form (iterated decomposition into generators for the cyclic factors)

$$g_1^{e_1} \cdot g_2^{e_2} \cdots g_n^{e_n}$$

with $0 \leq e_i < m_i$ where m_i is the order of the cyclic factor. We call $[e_1, e_2, \dots, e_n]$ the *exponent vector* of the element.

We can do the same for a solvable factor group, taking the g_i representatives from the group. In this case we call the generating sequence a *modulo pcgs*.

The basic operations of a Pcgs are:

a) element \rightarrow exponents.

b) exponents \rightarrow element.

In general, step a) requires element tests in subgroups of the subnormal series. Step b) requires multiplications of elements.

Fortunately, Pc groups use the same data structure for their arithmetic internally. The corresponding generating system is called the *family pcgs*. Both types of operations are *very* fast with respect to the family pcgs.

It can therefore be worth to change to another pc group for which the family pcgs is better adapted to the calculation intended. For example there is `IsomorphismSpecialPcGroup`.

Calculations using pcgs use the homomorphism principle and lift a result via a series of normal subgroups. Each new factor is elementary abelian and one can (try to) do linear algebra with exponent vectors there.

6.3 Induced PCGS

Subgroups are also represented via pcgs, these must be compatible (this means, that its elements have ascending depths) with the pcgs of the whole group. We call such a subgroup pcgs q , compatible with the pcgs p an *induced pcgs* and call p the *parent pcgs* of q .

If U is a subgroup, the command `InducedPcgs(p,U)` will return an induced pcgs. (If only a generating set is known, `InducedPcgsByGeneratorsNC` will do the task.)

Do *not* call `InducedPcgsWrtHomePcgs` or `InducedPcgsWrtSpecialPcgs` yourself. `InducedPcgs` will utilize these if applicable, but it could happen (in particular for non-pc groups) that the pcgs they refer to is different.

Induction of pcgs is transitive, but inducing from an induced pcgs will deteriorate the performance. If possible always induce with respect to the original pcgs (or its parent). A typical pc group algorithm thus computes once a pcgs for the whole group and then always induces with respect to this pcgs (or even the parent pcgs).

If you get a generator sequence for a subgroup, you can make a pcgs from it by calling `InducedPcgsByPcSequenceNC`.

If you don't need to compute exponents with respect to this sequence, it can be faster to keep it simply as a generating sequence and remember in the algorithm implicitly that it is a pcgs.

7 Types and method selection

Each object in GAP carries a *type* which stores

- i) its mathematical identity (say, ring-with-one consisting of matrices over a finite field),
- ii) how the object is represented (say, by a list of generators),
- iii) what is known so far about it (say: it is finite and its size is known).

All these informations are represented by bits in a long bit list (for example there is a bit (or set of bits) for "matrix group over finite field", a bit for: "univariate polynomial stored by coefficient list", a bit for: "object knows its size"). Some of

this information will be acquired over time and the type will change accordingly.

On the user level, each bit corresponds to a *filter*, one can call this filter like a function and get the corresponding bit value.

Formally, the filters that represent concepts i) are called *Categories*, concept ii) *Representations* and concept iii) *Properties* and *Attribute tester*.

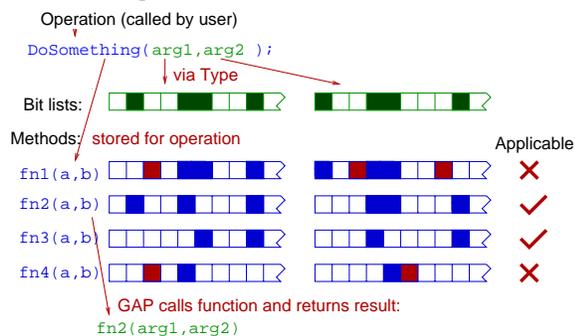
There also is a type iv): Filter that represent that certain basic calculations (for example equality test or computing the < total order can be done with reasonable amount of work.

Unless you want to implement new concepts or create own objects, you don't need to know about the formal difference between these concepts.

Most of the user "functions" are formally declared as *operations*, taking a certain input and promising certain output, but not specifying anything about how it is done. (For example: the `Centralizer` is the set of all elements in the group that commute with a given element.) Normally this is all one needs to know when using the system.

The library then installs *methods* for the operations which do the actual work. A method is an ordinary GAP function (that will do the computation), together with a collection of bit lists for all the argument.

Internally, whenever an operation is called, the system looks at the bit lists of all the arguments and compares these (logical and) with the bit lists of all the functions installed for the operation. If the required bits are a subset, the method is called *applicable*. The applicable method with the largest number of required filters then is executed.



7.1 Attributes

Attributes are one-argument operations that store the result once computed. For this, they use an extra filter, the *Attribute tester*. If the value is stored this filter is set. There is a high-ranking method that will just fetch the precomputed result. You can use the attribute tester `HasAttribute` in the method selection if you require attributes to be known.

Properties are attributes whose value can only be true or false. A known true value of a property can also be used in the method selection.

For attributes that are not properties to be stored, the representation of the object must be `IsAttributeStoringRep`.

7.2 Families

Families are the part of the type system which usually causes most confusion. The system would work without families, but checking compatibility of objects then would be harder. For

method installation you *can ignore families if you prefer, but you will then have to check compatibility yourself*.

There is an extra bit of information in the type, which describes how objects fit together. For example

- Elements of finitely presented groups for different presentations.
- Polynomials in different characteristic
- Homomorphisms from different permutation groups.

We want to check quickly, whether two objects fit together (say in `Centralizer` if a finitely presented group and an element belong to the same presentation). This cannot be done on the level of bit lists – these check each argument separately.

Instead the type carries an extra object we call a *family*. In the method selection, GAP can also check, whether families are compatible.

Different finitely presented groups for example have different families. So the method in the `Centralizer` example above does not have to check whether group and element belong to the same presentation – this is done by requiring family compatibility in the method selection.

The family of a group is different from the family of its elements. A collection of objects (list, group, ...) automatically gets a the *collections family*. Nothing needs to be declared for this.

7.3 Method installation

A typical method installation thus looks like this:

```
InstallMethod(
  DoSomething, operation
  "ident.
  string", for debugging
  fampred, A function that takes the
            argument's families and must return
            true for applicability. Just use
            true itself to ignore the feature
            Requirements for the arguments

  [req1,
  ...
  reqn],
  rank, ranking offset, normally 0
  func) function of n arguments
        to implement method
```

`InstallOtherMethod` works the same but does not require the number of arguments to correspond to the declaration.

7.4 Which method is used

When debugging (or just for curiosity) one might want to find out, which method is used for certain arguments. One can use `TraceMethods(Operation)` (respectively `UntraceMethods`) to print the identification string, whenever a method for a given operation is called. `ApplicableMethod(operation, argumentlist, printlevel)` will do the method selection "by hand" and return the actual method for given arguments. Its print level can be used to display why prior methods were not applicable. It is also possible to get "next best" methods.

```

gap> g:=Group((1,2,3,4),(1,2));;
gap> h:=DerivedSubgroup(g);
gap> me:=ApplicableMethod(\=, [g,h],2);
#I Searching Method for EQ with 2 arguments:
#I Total: 146 entries
#I 1: ‘EQ: 2 lists, second empty’,
    value: 1*SUM_FLAGS+13
...
#I 39: ‘EQ: for GF2 vectors’, value: 38
#I 40: ‘EQ: generic for groups’, value: 38
function( G, H ) ... end
gap> Print(me);
function ( G, H )
    if IsFinite( G ) then
    ...
end;

```

A few operations (Arithmetic, Position, Length) dispatch through the kernel first and ApplicableMethod might not return the right method.

If the operation is in fact a function, you will get an error message. Remember that some operations are actual functions and there is a corresponding operation XyzOp or XyzNC. For operations with a variable number of arguments the method-selected operation might only take two arguments and be called iteratively.

8 Adding to the system

This is a vast subject one could probably spend months on. There “Programming in GAP” manual gives much more detail than I can do here and thus might be the next place to look at.

8.1 Own methods

Installing own methods is not hard, once you know the syntax of Install(Other)Method. You need to know:

- The name (and parameter set) of the actual operation and
- The filter that specify the set of parameters for which the function should work.

For example a special method to compute the normalizer of the trivial group in a solvable permutation group would be installed as:

```

InstallMethod(NormalizerOp,
  "trivial group in solvable permgroup",
  true, # or: IsIdenticalObj
  [IsPermGroup and IsSolvableGroup,
   IsTrivial], 0
function(a,b)
  ...
end);

```

Normally there should be no reason to change the ranking (i.e. keep the value to 0) – the specification of the filters should have taken care of it. You can use TraceMethods to see that your new method gets called or ApplicableMethod to see its place in the list of methods.

If the method gets ranked too low, you might have forgotten to specify “obvious” filters for some arguments – for example IsFinite. If filters are properties, keep in mind that

an unknown property is false for purposes of the method selection.

To add own operations one can simply use NewOperation or DeclareOperation, the arguments are a string and a list of filters that specify the reach of the operation:

```
DeclareOperation("Blubber", [IsMyObject, IsFinite]);
```

8.2 Own objects

Lets finally look a bit at what is necessary to implement a new kind of elements. There is a do-it-all function ArithmeticElementCreator which might be the easiest way to get started. If you don’t want to use it, you first need to specify at least one type for your objects.

In general, own objects need at least one type. For this you need a family (just define

```
MyObjectsFamily:=NewFamily("MyObjectsFamily");
```

to get one, if you don’t need extra features.)

If you want to use category collections the family declaration will have to specify the category.

You also need at least one category (IsObject would do, but you are likely to use for example IsMultiplicativeElementWithInverse).

This category would specify that the object *potentially* might be multiplied or inverted, but it might fail for certain objects. for example *all* matrices are IsMultiplicativeElementWithInverse.

You then need a representation. Either use IsPositionalObjectRep for list-like objects (access to entries via *obj!*[3]) or IsComponentObjectRep (and IsAttributeStoringRep if you want to be able to store attributes) for (more complex) record-like objects (access to components via *obj!.component*).

Either as category or representation you will want to throw in one filter, that describes your objects (so you can method-select for them. The easiest way is this to have this filter imply all the filters you want. (So you only need to refer to this single filter later on.)

```

IsMyKindOfObject:=NewRepresentation(
  "IsMyKindOfObject",
  IsMultiplicativeElementWithInverse
  and IsPositionalObjectRep,
  []);

```

The extra empty list argument is a relic from an attempt to be very picky with declarations.

Finally you need a type:

```
IsMyType:=NewType(MyObjectsFamily,
  IsMyKindOfObject);
```

You then can create objects of your type via

```
newobj:=Objectify(IsMyType,
  [data,you,want,in,the,object]);
```

In this case newobj would be a positional object and newobj!. [3] would be the value of want.

Once this is done you can forget about everything but IsMyKindOfObject.

8.3 Required operations

For the generic functions to work, you must implement methods for:

Arithmetic Operations. You must implement `\+`, `*`, `AdditiveInverseOp` and `InverseOp` (if applicable). There are default methods which enable `\/, -` and `\^` in the obvious ways, but you can implement your own methods if there are better ways for the specific objects.

`PrintObj` to actually see the objects.

Equality via `\=`. Equal objects should be mathematically equal – they can be represented in completely different ways.

Total order within the family. If this is left out anything which requires sorting or sets will fail. Only `\<` needs to be implemented.

When testing, note that reading in the input file *again* will define *new* families, types, etc. under the same names. You might also have to create the objects anew (or move the declarations into a separate file).

Once these methods are implemented, it should be possible to form groups, rings etc. from the new objects and generic methods for the operations are available.