The Package
“SONATA”

(System of Nearrings and Their Applications)

Reference Manual

by

the SONATA–team
Institut f. Algebra, Stochastik und wissensbasierte mathematische Systeme
Univ. Linz, 4040 Linz, Austria

Contents

The pages are numbered chapterwise. This table of contents gives for each section the page number within the chapter.

The authors

SONATA was written at

Institut f. Algebra, Stochastik und wissensbasierte mathematische Systeme
Universität Linz

by the SONATA Team:

Erhard Aichinger
Franz Binder
Jürgen Ecker
Peter Mayr
Christof Nöbauer

Preface

When working on our master's and PhD-projects in nearring theory we hoped that we would gain more insight into the problems we worked on if we had significant examples at hand. For example, Erhard Aichinger wanted to find 1-affine complete groups; these are groups on which every unary compatible, i.e., congruence preserving, function can be interpolated by a polynomial function. In other words, a group G is 1-affine complete if the nearring $I(G)$ contains all functions in the nearring $C_0(G)$ of all zero-preserving compatible functions. After having written a straight-forward program to compute all polynomial functions on a group, which basically relied on computing all terms and checking whether the arising functions were equal, he found that in that way probably only the groups of order less than 10 could be treated, and therefore abandoned his hope to find help in computers for quite a while.

At about the same time, Christof Nöbauer was collecting a library of *all* small nearrings; and he decided to implement his library into the group-theory system GAP. Then Jürgen Ecker started to break rings into their subdirectly irreducible parts using GAP.

In May 1995, we realized that the problem of computing the number of polynomial functions on a group was actually an easy task if one used the power of computational group theory. The easy key observation is that it is easy to compute how big the group generated by some group elements is. Representing the functions on G as elements of $G^{|G|}$, it is easy to compute first the generators of the group of polynomial functions and then all polynomial functions as the closure of these generators. This observation, albeit strikingly easy, and of course not even original, made it possible to compute the number of polynomial functions on S_4, which is 22 265 110 462 464, in a few seconds. The same strategy also worked for other kinds of distributively generated nearrings, such as $A(G)$ or $E(G)$.

At this point, we decided to make a package of our functions and make them available to a wider community. Encouraged by the enthusiasm of Prof. Günter Pilz, and paid by the "Fonds zur Förderung wissenschaftlicher Forschung", we started to bring our functions into a common form, and to add many functions that we found useful, as e.g. the computation of Noetherian Quotients, which works especially fine for polynomial nearrings. We wanted to include the applications of nearring theory to design-theory, because especially in this field we thought that it could only be through the examination of examples that the contribution of nearrings to this field could be investigated. This part was then mainly carried forward by Peter Mayr.

In the beginning of 1997, with the conference at Stellenbosch coming near, we decided to make our programs available to nearringers six months later. Despite of the fact that SONATA does not contain many new sophisticated algorithms, and therefore hardly represents a big deal in computational nearring theory, it uses a lot of well-established sophisticated algorithms in group theory. We think that our real contribution is to take advantage of these algorithms for computing with nearrings. Nevertheless, computational nearring theory could be interesting: A typical problem arising in the computation of nearrings would be the following: Given a function on a group, how big is the nearring generated by it? And what if we take more than one function? And how can a given function in the nearring be represented by the generators? We recall that Sim's stabilizer chains give solutions to similar problems in the theory of permutation groups. We think that even an answer for one function, and on special groups, would be delightful.

Examples of nearrings are nice; but it would be even nicer to have a lot of interesting examples in a small booklet. At this point, Franz Binder joined us, and started to work on a nearring table containing all nearrings up to a certain order and giving meaningful information about each of them, such as for example the ideal lattice with commutators in the sense of universal algebra. It was when he started to work on this complete library that the people at the Maths Department, whose printers we constantly fed with new nearring information, started to give us rather strange looks, which we could not explain to ourselves but as signs of starting admiration for the beauty of nearrings.

When a beta version 4 of GAP came out, we found that SONATA should be written in this new version of GAP4. Jürgen Ecker's effort to translate all existing GAP3-code into GAP4 was rewarded by the observation that many things worked much better in GAP4 than they did before. Just before leaving to Stellenbosch, he invented the name SONATA for our programs that sometimes proceed *andante*, but at other times really rather *presto*.

At the nearring conference in Stellenbosch in July 1997, we showed some possibilities of our system to many people doing research in nearring theory. Their interest in our system showed us that our hope that nearringers would actually use SONATA was by no means unfounded.

On October 1st, 1997, we gave away the first version of SONATA. In January 2002, finally, we were able to submit SONATA for refereeing as a GAP4 share package.

We are eager to hear YOUR feedback in order to make SONATA nicer in all respects.

We, the SONATA team, want to say THANK YOU to all that have helped us in some way to realize this project:

- to Tim Boykett for helping us with his expertise in computers and how they could be used in algebra;
- to Roland Eggetsberger for his ideas about planar nearrings and designs;
- to Marcel Widi, who contributed some functions on semigroups which, however, somehow do not lie in the scope of the SONATA project;
- to Christof Nöbauer, who not only filled our hard disks with an evergrowing number of nearrings, but also worked a lot to keep our computers running;
- to Markus Hetzmannseder for administrating our computers and keeping our GAP installation alive and up-to-date for the last few years;
- to the staff at the algebra group at our department, and in particular to all our visitors, who have put lots of ideas into our heads: this includes Peter Fuchs, Gerhard Betsch, Jim Clay, Wen-Fong Ke, Carl Maxson, John Meldrum, Gary Birkenmeier;
- to the Austrian "Fonds zur Förderung der wissenschaftlichen Forschung"(FWF) for supporting SONATA via the grants P11486-TEC and P12911-INF.

Needless to say, this project would never been carried out without the encouragement and suggestions of Prof. Günter Pilz.

So, what you have now is a system that contains a library of all small nearrings and many functions to construct and analyze a lot of interesting big nearrings. Have fun !

Linz, 16.1.02, the SONATA Team*

* The SONATA Team consists of

Erhard Aichinger
Franz Binder
Jürgen Ecker
Peter Mayr
Christof Nöbauer

1 Supportive functions for groups

In order to support nearring calculations, a few functions for groups had to be added to the standard GAP group library functions.

The functions described here can be found in the source files `grpend.g?` and `grpsupp.g?`

1.1 Predefined groups

All groups of order 2 to 32 are predefined. They can be accessed by variables of the kind GTWo_n where o defines the order of the group and n the number of the group of order o as they appear in [TW80] . For example, `GTW16_6` defines the group of Thomas – Wood type 16/6, which is actually $D_4 \times C_2$.

Alternatively, these groups can be accessed via the function

▶ `TWGroup(` o, n `)`

with o and n as above. In addition, all these groups are stored in the list `GroupList`.

Conversely, for any group G of order at most 32,

▶ `IdTWGroup(` G `)`

returns a pair [o,n], meaning that G is isomorphic to the group o/n.

```
gap> G := GTW6_2;
6/2
gap> H := TWGroup( 4, 2 );
4/2
gap> D := DirectProduct( G, H );
Group([ (1,2), (1,2,3), (4,5), (6,7) ])
gap> IdTWGroup( D );
[ 24, 4 ]
gap> GroupList;
[ 2/1, 3/1, 4/1, 4/2, 5/1, 6/1, 6/2, 7/1, 8/1, 8/2, 8/3, 8/4, 8/5,
  9/1, 9/2, 10/1, 10/2, 11/1, 12/1, 12/2, 12/3, 12/4, 12/5, 13/1,
  14/1, 14/2, 15/1, 16/1, 16/2, 16/3, 16/4, 16/5, 16/6, 16/7, 16/8,
  16/9, 16/10, 16/11, 16/12, 16/13, 16/14, 17/1, 18/1, 18/2, 18/3,
  18/4, 18/5, 19/1, 20/1, 20/2, 20/3, 20/4, 20/5, 21/1, 21/2, 22/1,
  22/2, 23/1, 24/1, 24/2, 24/3, 24/4, 24/5, 24/6, 24/7, 24/8, 24/9,
  24/10, 24/11, 24/12, 24/13, 24/14, 24/15, 25/1, 25/2, 26/1, 26/2,
  27/1, 27/2, 27/3, 27/4, 27/5, 28/1, 28/2, 28/3, 28/4, 29/1, 30/1,
  30/2, 30/3, 30/4, 31/1, 32/1, 32/2, 32/3, 32/4, 32/5, 32/6, 32/7,
  32/8, 32/9, 32/10, 32/11, 32/12, 32/13, 32/14, 32/15, 32/16, 32/17,
  32/18, 32/19, 32/20, 32/21, 32/22, 32/23, 32/24, 32/25, 32/26,
  32/27, 32/28, 32/29, 32/30, 32/31, 32/32, 32/33, 32/34, 32/35,
  32/36, 32/37, 32/38, 32/39, 32/40, 32/41, 32/42, 32/43, 32/44,
  32/45, 32/46, 32/47, 32/48, 32/49, 32/50, 32/51 ]
```

1.2 Operation tables for groups

▶ `PrintTable(` *G* `)`

`PrintTable` prints the Cayley table of the group *G*.

```
gap> G := GTW4_2;
4/2
gap> PrintTable( G );
Let:
g0 := ()
g1 := (3,4)
g2 := (1,2)
g3 := (1,2)(3,4)

  *   g0 g1 g2 g3
 ------------------
  g0  g0 g1 g2 g3
  g1  g1 g0 g3 g2
  g2  g2 g3 g0 g1
  g3  g3 g2 g1 g0
```

Sometimes different symbols for the elements in the would make the table look nicer. For the group 4/2 ($\mathbb{Z}_2 \times \mathbb{Z}_2$) one could choose the canonical form as pairs of zeros and ones.

```
gap> G := GTW4_2;
4/2
gap> SetSymbols( G, ["(0,0)","(1,0)","(0,1)","(1,1)"] );
gap> PrintTable( G );
Let:
(0,0) := ()
(1,0) := (3,4)
(0,1) := (1,2)
(1,1) := (1,2)(3,4)

     *   (0,0)  (1,0)  (0,1)  (1,1)
  ---------------------------------
  (0,0)   (0,0)  (1,0)  (0,1)  (1,1)
  (1,0)   (1,0)  (0,0)  (1,1)  (0,1)
  (0,1)   (0,1)  (1,1)  (0,0)  (1,0)
  (1,1)   (1,1)  (0,1)  (1,0)  (0,0)
```

1.3 Group endomorphisms

▶ `Endomorphisms(` *G* `)`

`Endomorphisms` computes all the endomorphisms of the group G. This function is most essential for computing the nearrings on a group. The endomorphisms are returned as a list of group homomorphisms. So all functions for mappings and homomorphisms are applicable.

```
gap> G := TWGroup( 4, 2 );
4/2
gap> Endomorphisms( G );
[ [ (3,4), (1,2) ] -> [ (), () ], [ (3,4), (1,2) ] -> [ (3,4), () ],
  [ (3,4), (1,2) ] -> [ (1,2), () ],
  [ (3,4), (1,2) ] -> [ (1,2)(3,4), () ],
  [ (3,4), (1,2) ] -> [ (), (3,4) ],
  [ (3,4), (1,2) ] -> [ (), (1,2) ],
  [ (3,4), (1,2) ] -> [ (), (1,2)(3,4) ],
  [ (3,4), (1,2) ] -> [ (3,4), (3,4) ],
  [ (3,4), (1,2) ] -> [ (1,2), (1,2) ],
  [ (3,4), (1,2) ] -> [ (1,2)(3,4), (1,2)(3,4) ],
  [ (1,2), (3,4) ] -> [ (3,4), (1,2) ],
  [ (3,4), (1,2) ] -> [ (1,2)(3,4), (3,4) ],
  [ (3,4), (1,2) ] -> [ (3,4), (1,2) ],
  [ (3,4), (1,2) ] -> [ (1,2)(3,4), (1,2) ],
  [ (3,4), (1,2) ] -> [ (3,4), (1,2)(3,4) ],
  [ (3,4), (1,2) ] -> [ (1,2), (1,2)(3,4) ] ]
```

1.4 Group automorphisms

▶ `Automorphisms(` *G* `)`

`Automorphisms` computes all the automorphisms of the group G. The automorphisms are returned as a list of group homomorphisms. So all functions for mappings and homomorphisms are applicable.

```
gap> Automorphisms( GTW4_2 );
[ IdentityMapping( 4/2 ), [ (1,2), (3,4) ] -> [ (1,2)(3,4), (3,4) ],
  [ (1,2), (3,4) ] -> [ (3,4), (1,2) ],
  [ (3,4), (1,2) ] -> [ (1,2), (1,2)(3,4) ],
  [ (3,4), (1,2) ] -> [ (1,2)(3,4), (3,4) ],
  [ (3,4), (1,2) ] -> [ (1,2)(3,4), (1,2) ] ]
```

1.5 Inner automorphisms of a group

▶ `InnerAutomorphisms(` *G* `)`

`InnerAutomorphisms` computes all the inner automorphisms of the group *G*. The inner automorphisms are returned as a list of group homomorphisms. So all functions for mappings and homomorphisms are applicable.

```
gap> InnerAutomorphisms( AlternatingGroup( 4 ) );
[ ^(), ^(2,3,4), ^(2,4,3), ^(1,2)(3,4), ^(1,2,3), ^(1,2,4),
  ^(1,3,2), ^(1,3,4), ^(1,3)(2,4), ^(1,4,2), ^(1,4,3), ^(1,4)(2,3) ]
```

1.6 Isomorphic groups

▶ `IsIsomorphicGroup(` *G*, *H* `)`

`IsIsomorphicGroup` determines if the groups *G* and *H* are isomorphic. If they are isomorphic, an isomorphism between these two groups can be found with `IsomorphismGroups`.

```
gap> IsIsomorphicGroup( SymmetricGroup( 4 ), GTW24_12 );
true
```

1.7 Subgroups of a group

▶ `Subgroups(` *G* `)`

`Subgroups` returns a list of all subgroups of the group *G*, if there are only finitely many subgroups.

```
gap> Subgroups( TWGroup( 8, 4 ) );
[ Group(()), Group([ (1,3)(2,4) ]), Group([ (2,4) ]),
  Group([ (1,3) ]), Group([ (1,2)(3,4) ]), Group([ (1,4)(2,3) ]),
  Group([ (2,4), (1,3) ]), Group([ (1,2)(3,4), (1,3)(2,4) ]),
  Group([ (1,3)(2,4), (1,2,3,4) ]), Group([ (1,2,3,4), (2,4) ]) ]
```

1.8 Normal subgroups generated by a single element

▶ `OneGeneratedNormalSubgroups(` *G* `)`

`OneGeneratedSubgroups` returns a list of all normal subgroups of the group *G* which are generated by one element.

```
gap> OneGeneratedNormalSubgroups( GTW12_4 );
[ Group(()), Group([ (1,3)(2,4), (1,4)(2,3) ]),
  Group([ (2,3,4), (1,4,3) ]), Group([ (2,4,3), (1,3,4) ]) ]
```

1.9 Invariant subgroups

▶ `IsInvariantUnderMaps(` *G*, *U*, *maps* `)`

For a list of mappings, *maps* on the group G and a subgroup U of G, `IsInvariantUnderMaps` returns the truth value of "U is invariant under all mappings in *maps*". In the following example this function is used to compute all fully invariant subgroups of the dihedral group of order 12.

```
gap> D12 := DihedralGroup( 12 );
<pc group of size 12 with 3 generators>
gap> s := Subgroups( D12 );
[ Group([  ]), Group([ f1 ]), Group([ f1*f3^2 ]), Group([ f1*f3 ]),
  Group([ f2*f3 ]), Group([ f1*f2 ]), Group([ f1*f2*f3^2 ]),
  Group([ f1*f2*f3 ]), Group([ f3 ]), Group([ f1, f2*f3 ]),
  Group([ f1*f3^2, f2*f3 ]), Group([ f1*f3, f2*f3 ]),
  Group([ f3, f1 ]), Group([ f3, f2 ]), Group([ f3, f1*f2 ]),
  Group([ f3, f1, f2 ]) ]
gap> e := Endomorphisms( D12 );;
gap> f := Filtered( s, sg -> IsInvariantUnderMaps( D12, sg, e ) );
[ Group([  ]), Group([ f3 ]), Group([ f3, f1, f2 ]) ]
```

▶ `IsCharacteristic(` *G*, *U* `)`

A subgroup U of the group G is *characteristic* if it is invariant under all automorphisms on G. For a subgroup U of the group G, `IsCharacteristic` returns the truth value of "U is a characteristic subgroup of G". If the group U is defined as the subgroup of a group G then the function call

▶ `IsCharacteristicInParent(` *U* `)`

has the same result.

```
gap> IsCharacteristicInParent( Centre( GTW16_11 ) );
true
```

▶ `IsFullinvariant(` *G*, *U* `)`

A subgroup U of the group G is *fully invariant* if it is invariant under all endomorphisms on G. For a subgroup U of the group G, `IsFullinvariant` returns the truth value of "U is a fully invariant subgroup of G".

```
gap> G := GTW6_2;
6/2
gap> S := Subgroup( G, [(1,2)] );
Group([ (1,2) ])
gap> IsFullinvariant( G, S );
false
```

If the group U is defined as the subgroup of a group G then the function call

▶ `IsFullinvariantInParent(` *U* `)`

has the same result.

```
gap> IsFullinvariantInParent( Centre( GTW16_11 ) );
true
```

1.10 Coset representatives

▶ `RepresentativesModNormalSubgroup(` G, N `)`

If G is a group and N is a normal subgroup of G then the function `RepresentativesModNormalSubgroup` returns a set of representatives for the congruence classes modulo the normal subgroup N, i.e. a set of elements of G with exactly one element from each cogruence class modulo N.

```
gap> G := DihedralGroup( 16 );
<pc group of size 16 with 4 generators>
gap> C := Centre( G );
Group([ f4 ])
gap> RepresentativesModNormalSubgroup( G, C );
[ <identity> of ..., f1, f2, f3, f2*f3, f1*f2*f4, f1*f3*f4,
  f1*f2*f3*f4 ]
```

▶ `NontrivialRepresentativesModNormalSubgroup(` G, N `)`

This function behaves as `RepresentativesModNormalSubgroup` but it excludes the representative for the congruence class which contains the neutral element of the group.

```
gap> G := DihedralGroup( 16 );
<pc group of size 16 with 4 generators>
gap> C := Centre( G );
Group([ f4 ])
gap> NontrivialRepresentativesModNormalSubgroup( G, C );
[ f1, f2, f3, f2*f3, f1*f2*f4, f1*f3*f4, f1*f2*f3*f4 ]
```

1.11 Nilpotency class

▶ `NilpotencyClass(` G `)`

If G is a nilpotent group then the function `NilpotencyClass` returns the nilpotency class of G and `fail` otherwise.

```
gap> NilpotencyClass( SymmetricGroup( 7 ) );
fail
gap> NilpotencyClass( GTW32_47 );
3
```

1.12 Scott length

▶ `ScottLength(` G `)`

The function `ScottLength` retuns the Scott-length of the group G. For a definition of the Scott-length of a group and an idea for an algorithm for the general case see [Sco69]. In the case of a class 2 nilpotent finite group G a faster algorithm described in [Eck98] is used.

```
gap> ScottLength( GTW6_2 );
2
gap> ScottLength( GTW16_11 );
4
```

1.13 Other useful functions for groups

▶ `AsPermGroup(` *G* `)`

For a group *G*, `AsPermGroup` returns a permutation group that is isomorphic to *G*. In the case of a permutation group this is the group itself.

```
gap> D24 := DihedralGroup( 24 );
<pc group of size 24 with 4 generators>
gap> D24p := AsPermGroup( D24 );
<permutation group with 4 generators>
gap> IsomorphismGroups( D24, D24p );
[ f1, f2, f3, f4 ] ->
[ ( 1,19)( 2,21)( 3,20)( 4,24)( 5,23)( 6,22)( 7,13)( 8,15)( 9,14)
    (10,18)(11,17)(12,16), ( 1,12, 6, 9, 3,11, 5, 8, 2,10, 4, 7)
    (13,24,18,21,15,23,17,20,14,22,16,19),
  ( 1, 6, 3, 5, 2, 4)( 7,12, 9,11, 8,10)(13,18,15,17,14,16)
    (19,24,21,23,20,22), ( 1, 3, 2)( 4, 6, 5)( 7, 9, 8)(10,12,11)
    (13,15,14)(16,18,17)(19,21,20)(22,24,23) ]
gap> C12 := CyclicGroup( 12 );
<pc group of size 12 with 3 generators>
gap> AsPermGroup( C12 );
Group([ ( 1, 7, 4,10, 2, 8, 5,11, 3, 9, 6,12),
  ( 1, 4, 2, 5, 3, 6)( 7,10, 8,11, 9,12),
  ( 1, 2, 3)( 4, 5, 6)( 7, 8, 9)(10,11,12) ])
```

2 Nearrings

A *(left) nearring* is a nonempty set N together with two binary operations on N, $+$ and $\cdot$ s.t. $(N, +)$ is a group, $(N, \cdot)$ is a semigroup, and $\cdot$ is left distributive over $+$, i.e. $\forall n_1, n_2, n_3 \in N : n_1 \cdot (n_2 + n_3) = n_1 \cdot n_2 + n_1 \cdot n_3$.

For more information we suggest [Pil83], [Mel85], and [Cla92].

The functions described in this section can be found in the source files `nr.g?` and `nrconstr.g?`.

2.1 Defining a nearring multiplication

▸ `IsNearRingMultiplication(` *G*`,` *mul*`, [`*lcs*`] )`

The arguments of the function `IsNrMultiplication` are a group G, a GAP-function *mul* which has two arguments `x` and `y` which must both be elements of the group G and returns an element `z` of G s.t. *mul* defines a binary operation on G. As an optional third parameter `IsNrMultiplication` accepts a list of control strings *lcs*.

`IsNearRingMultiplication` returns `true` (`false`) if *mul* is (is not) a nearring multiplication on G i.e. it checks whether it is well-defined, associative and left distributive over the group operation of G. The list *lcs* may contain one or more of the strings `"closed"` , `"ass"` and `"rdistr"` in which case the according property is not tested. In this case it is assumed that the user has checked it. This feature should only be used in cases where it would take too long to check certain laws element by element and the user is absolutely sure about the correctness.

```
gap> G := TWGroup( 24, 6 );
24/6
gap> mul_l := function ( x, y ) return y; end;
function ( x, y ) ... end
gap> IsNearRingMultiplication( G, mul_l );
true
gap> mul_r := function ( x, y ) return x; end;
function ( x, y ) ... end
gap> IsNearRingMultiplication( G, mul_r );
#I  specified multiplication is not left distributive.
false
gap> IsNearRingMultiplication( G, mul_r, ["closed","ldistr"] );
true
```

▸ `NearRingMultiplicationByOperationTable(` *G*`,` *table*`,` *elmlist* `)`

The function `NearRingMultiplicationByOperationTable` returns the nearring multiplication on the group G which is defined by the multiplication table *table*. Rather than group elements the entries of *table* are the positions of the group element in the list *elmlist* (the first element in *elmlist* is 1, the second is 2, a.s.o.). Usually the neutral element of the group will be the first.

`IsNearRingMultiplication` can be used to check whether the resulting multiplication is indeed a nearring multiplication on G.

```
gap> G := CyclicGroup( 4 );
<pc group of size 4 with 2 generators>
gap> GeneratorsOfGroup( G );
[ f1, f2 ]
gap> a := last[1];
f1
gap> Order( a );
4
gap> # a generates G indeed
gap> elmlist := List( [0..3], x -> a^x );
[ <identity> of ..., f1, f2, f1*f2 ]
gap> # Let: 1 := identity of ..., 2 := f1, 3 := f2, 4 := f1*f2
gap> # Consider the following multiplication table on G:
gap> OT := [[1, 1, 1, 1],
> [1, 4, 3, 2],
> [1, 1, 1, 1],
> [1, 2, 3, 4]];;
    gap> mul := NearRingMultiplicationByOperationTable( G, OT, elmlist );
function ( x, y ) ... end
gap> IsNearRingMultiplication( G, mul );
true
```

2.2 Construction of nearrings

▶ `ExplicitMultiplicationNearRing(` *G*, *mul* `)`

The constructor function `ExplicitMultiplicationNearRing` returns the nearring defined by the group *G* and the nearring multiplication *mul*. (For a detailed explanation of *mul* see Section 2.1.)

`ExplicitMultiplicationNearRing` calls `IsNearRingMultiplication` in order to make sure that *mul* is really a nearring multiplication. If the nearring multiplication should not be checked,

▶ `ExplicitMultiplicationNearRingNC(` *G*, *mul* `)`

may be called.

```
gap> n := ExplicitMultiplicationNearRing( GTW18_3, mul_l );
ExplicitMultiplicationNearRing ( 18/3 , multiplication )
gap> n = ExplicitMultiplicationNearRingNC( GTW18_3, mul_l );
true
```

▶ `IsNearRing(` *obj* `)`

`IsNearRing` returns `true` if the object *obj* is a nearring and `false` otherwise.

```
gap> n := ExplicitMultiplicationNearRingNC( GTW18_3, mul_l );
ExplicitMultiplicationNearRing ( 18/3 , multiplication )
gap> IsNearRing( n );
true
gap> IsNearRing( GroupReduct( n ) );
false
```

▶ `IsExplicitMultiplicationNearRing(` *obj* `)`

`IsExplicitMultiplicationNearRing` returns `true`, if the object *obj* is a nearring defined by a group and a multiplication as with 2.2.

```
gap> IsExplicitMultiplicationNearRing( n );
true
```

2.3 Direct products of nearrings

▶ `DirectProductNearRing(` *nr1* `,` *nr2* `)`

Given two nearrings *nr1* and *nr2*, the function `DirectProductNearRing` constructs the direct product of these.

```
gap> n := ExplicitMultiplicationNearRingNC( GTW18_3, mul_l );
ExplicitMultiplicationNearRing ( 18/3 , multiplication )
gap> zero_mul := function ( x, y ) return (); end;
function ( x, y ) ... end
gap> z := ExplicitMultiplicationNearRingNC( GTW12_3, zero_mul );
ExplicitMultiplicationNearRing ( 12/3 , multiplication )
gap> d := DirectProductNearRing( n, z );
DirectProductNearRing( ExplicitMultiplicationNearRing ( 18/3 , multi\
plication ), ExplicitMultiplicationNearRing ( 12/3 , multiplication \
) )
gap> IsExplicitMultiplicationNearRing( d );
true
```

2.4 Operation tables for nearrings

▶ `PrintTable(` *nr* `)`

`PrintTable` prints the additive and multiplicative Cayley tables of the nearring *nr*. This function works the same way as for groups.

```
gap> n := ExplicitMultiplicationNearRingNC( CyclicGroup( 3 ), mul_l );
ExplicitMultiplicationNearRing ( <pc group of size 3 with
1 generators> , multiplication )
gap> SetSymbols( n, ["0","1","2"] );
gap> PrintTable( n );
Let:
0 := (<identity> of ...)
1 := (f1)
2 := (f1^2)

  +   0  1  2
  ------------
  0   0  1  2
  1   1  2  0
  2   2  0  1

  *   0  1  2
  ------------
  0   0  1  2
  1   0  1  2
  2   0  1  2
```

Optionally, `PrintTable` can be used in the form `PrintTable(` *nr*`,` *mode* `)`, where *mode* is a string. If the letter `e` is contained in this string, the definitions of the symbols used are printed, if the letter `a` is contained in the string, the addition table is printed, and if the letter `m` is contained in the string, the multiplication table of the nearring is printed. Every combination of these three letters in any order is possible.

2.5 Modified symbols for the operation tables

▶ `SetSymbols(` *nr*, *symblist* `)`
▶ `SetSymbolsSupervised(` *nr*, *symblist* `)`

The function `SetSymbols` and `SetSymbolsSupervised` allow you to define a list *symblist* of strings to be used when printing the operation tables of the nearring. `SetSymbols` simply sets the set of strings to the given value. `SetSymbolsSupervised` checks, if there are more symbols than the nearring has elements. In this case the superfluous strings are ignored. If there are less symbols than the nearring has elements, `SetSymbolsSupervised` "invents" unique names for the rest of the elements. In any case a warning is printed. If there are repetitions or holes in the list *symblist* an error is signaled.

▶ `Symbols(` *nr* `)`

allows you to look at the set of symbols, which are currently in use.

```
gap> n := LibraryNearRing( GTW3_1, 4 );
LibraryNearRing(3/1, 4)
gap> Symbols( n );
[ "n0", "n1", "n2" ]
gap> SetSymbolsSupervised( n,
        ["apple", "banana", "coconut", "donut", "potato"] );
Warning: too many symbols ...ignoring the last 2 symbols
gap> PrintTable( n, "m" );

         *   apple    banana   coconut
    ------------------------------------
    apple     apple    apple    apple
    banana    apple    banana   coconut
    coconut   apple    banana   coconut
```

2.6 Accessing nearring elements

The elements of a nearring are different from those of its group reduct. In order to make group elements and nearring elements distinguishable for the user, nearring elements are printed with an extra pair of parentheses. The two functions `AsGroupReductElement` and `AsNearRingElement` can be used to switch between these two representations.

▶ `AsNearRingElement(` *nr*, *grpelm* `)`

returns the representation as a nearring element of an element *grpelm* of the group reduct of the nearring *nr*.

▶ `AsGroupReductElement(` *nrelm* `)`

returns the representation as an element of the group reduct of the nearring of the nearring element *nrelm*.

```
gap> mul_l := function ( x, y ) return y; end;
function ( x, y ) ... end
gap> n := ExplicitMultiplicationNearRingNC( GTW6_2, mul_l );
ExplicitMultiplicationNearRing ( 6/2 , multiplication )
gap> AsList( n );
[ (()), ((2,3)), ((1,2)), ((1,2,3)), ((1,3,2)), ((1,3)) ]
gap> e := AsNearRingElement( n, (2,3) );
((2,3))
gap> e in n;
true
gap> f := AsNearRingElement( n, (1,3) );
((1,3))
gap> e + f;
((1,3,2))
gap> e * f;
((1,3))
gap> p := AsGroupReductElement( e );
(2,3)
gap> IsPerm( p );
true
gap> p + p;
Error no method found for operation SUM with 2 arguments at
Error( "no method found for operation ", NAME_FUNC( operation ),
 " with 2 arguments" );
Entering break read-eval-print loop, you can 'quit;' to quit to outer l\
oop,
or you can return to continue
brk>
```

2.7 Nearring elements

There are three different ways to ask for the elements of a nearring.

▶ `AsList(` *nr* `)`

The function `AsList` computes the elements of the nearring *nr*. It returns the elements as a list.

▶ `AsSortedList(` *nr* `)`

does essentially the same, but returns a set of elements.

▶ `Enumerator(` *nr* `)`

does essentially the same as `AsList`, but returns an enumerator for the elements of *nr*. An enumerator is an object that is capable of enumerating the elements the nearring one by one. This is especially important if the nearring is very big and not every element can be stored.

```
gap> n := LibraryNearRing( GTW6_2, 39 );
LibraryNearRing(6/2, 39)
gap> e := Enumerator( n );
<enumerator of near ring>
gap> e[1];
(())
gap> x := AsNearRingElement( n, (1,2,3) );
((1,2,3))
gap> Position( e, x );
2
gap> Length(e);
6
gap> l := AsList( n );
[ (()), ((2,3)), ((1,2)), ((1,2,3)), ((1,3,2)), ((1,3)) ]
gap> e[3] = l[3];
false
gap> AsSortedList( n );
[ (()), ((2,3)), ((1,2)), ((1,2,3)), ((1,3,2)), ((1,3)) ]
```

2.8 Random nearring elements

▶ `Random(` *nr* `)`

`Random` returns a random element of the nearring *nr*.

```
gap> n := LibraryNearRing( GTW6_2, 39 );
LibraryNearRing(6/2, 39)
gap> Random(n);
((1,3))
```

2.9 Nearring generators

▶ `GeneratorsOfNearRing(` *nr* `)`

The function `GeneratorsOfNearRing` returns a set of (not necessarily additive) generators of the nearring *nr*.

```
gap> n := ExplicitMultiplicationNearRingNC( GTW8_4, mul_l );
ExplicitMultiplicationNearRing ( 8/4 , multiplication )
gap> GeneratorsOfNearRing( n );
[ ((1,2,3,4)), ((2,4)) ]
```

2.10 Size of a nearring

▶ `Size(` *nr* `)`

`Size` returns the number of elements in the nearring *nr*.

```
gap> n := LibraryNearRingWithOne( GTW24_3, 8 );
LibraryNearRingWithOne(24/3, 8)
gap> Size(n);
24
```

2.11 The additive group of a nearring

▸ `GroupReduct(` *nr* `)`

The function `GroupReduct` returns the nearring *nr* as a (multiplicative) group.

```
gap> GroupReduct( LibraryNearRingWithOne( GTW24_3, 8 ) );
24/3
```

2.12 Nearring endomorphisms

▸ `Endomorphisms(` *nr* `)`

`Endomorphisms` computes all the endomorphisms of the nearring *nr*. The endomorphisms are returned as a list of transformations. In fact, the returned list contains those endomorphisms of the group reduct of *nr* which are also nearring endomorphisms.

```
gap> Endomorphisms ( LibraryNearRing( GTW12_4, 4 ) ) ;
[ [ (1,2,4), (2,3,4) ] -> [ (), () ],
  [ (1,2,4), (2,3,4) ] -> [ (1,2,4), (2,3,4) ] ]
gap> Length( Endomorphisms( GTW12_4 ) );
33
```

2.13 Nearring automorphisms

▸ `Automorphisms(` *nr* `)`

`Automorphisms` computes all the automorphisms of the nearring *nr*. The automorphisms are returned as a list of transformations. In fact, the returned list contains those automorphisms of the group reduct of *nr* which are also nearring automorphisms.

```
gap> Automorphisms( LibraryNearRing( GTW12_4, 4 ) );
[ IdentityMapping( 12/4 ) ]
```

2.14 Isomorphic nearrings

▸ `IsIsomorphicNearRing(` *nr1* `,` *nr2* `)`

The function `IsIsomorphicNearRing` returns `true` if the two nearrings *nr1* and *nr2* are isomorphic and `false` otherwise.

```
gap> IsIsomorphicNearRing( MapNearRing( GTW2_1 ),
> LibraryNearRingWithOne( GTW4_2, 5 ) );
true
```

2.15 Subnearrings

▸ `SubNearRings(` *nr* `)`

The function `SubNearRings` computes all subnearrings of the nearring *nr*. The function returns a list of nearrings representing the according subnearrings.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> SubNearRings( n );
[ ExplicitMultiplicationNearRing ( Group(()) , multiplication ),
  ExplicitMultiplicationNearRing ( Group([ (1,4)(2,3) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group([ (1,2)(3,4) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group([ (1,4)(2,3), (1,3)(2,4)
     ]) , multiplication ), ExplicitMultiplicationNearRing ( Group(
    [ (1,4)(2,3), (1,3)(2,4), (1,2)(3,4), (2,3,4) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group([ (2,3,4) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group([ (1,4,3) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group([ (1,2,4) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group([ (1,3,2) ]) , multiplication ) ]
```

2.16 Invariant subnearrings

▶ `InvariantSubNearRings(` *nr* `)`

A subnearring $(M, +, \cdot)$ of a nearring $(N, +, \cdot)$ is called an *invariant subnearring* if both, $M \cdot N$ and $N \cdot M$ are subsets of M.

The function `InvariantSubNearRings` computes all invariant subnearrings of the nearring *nr*. The function returns a list of nearrings representing the according invariant subnearrings.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> i := InvariantSubNearRings( n );
[ ExplicitMultiplicationNearRing ( Group(()) , multiplication ),
  ExplicitMultiplicationNearRing ( Group([ (1,2)(3,4) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group(
    [ (1,4)(2,3), (1,3)(2,4), (1,2)(3,4), (2,3,4) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group([ (2,3,4) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group([ (1,4,3) ]) , multiplication ) ]
```

2.17 Constructing subnearrings

▶ `SubNearRingBySubgroupNC(` *nr*, *S* `)`

For a subgroup S of the group reduct of the nearring which is closed under the multiplication of *nr* the function `SubNearRingBySubgroupNC` returns the subnearRing of *nr*, which is induced by this subgroup. The *nr*-invariance is not explicitely tested.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> S := Subgroup( GTW12_4, [ (1,2)(3,4) ] );
Group([ (1,2)(3,4) ])
gap> sn := SubNearRingBySubgroupNC( n, S );
ExplicitMultiplicationNearRing ( Group([ (1,2)(3,4) ]) , multiplication )
```

2.18 Intersection of nearrings

▶ `Intersection(` *listofnrs* `)`

computes the intersection of the nearrings in the list *listofnrs*. All of the nearrings in *listofnrs* must be subnearrings of a common supernearring.

```
gap> n := LibraryNearRingWithOne( GTW27_4, 5 );
LibraryNearRingWithOne(27/4, 5)
gap> si := Filtered( SubNearRings( n ), s -> Identity( n ) in s );
[ ExplicitMultiplicationNearRing ( Group(
    [ ( 1,23,14)( 2,13, 6)( 3,27,22)( 4,18, 9)( 5,20,12)( 7,16,26)
        ( 8,25,17)(10,21,19)(11,24,15) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group(
    [ ( 1,26,27)( 2,19,20)( 3,14,16)( 4,24,25)( 5, 6,21)( 7,22,23)
        ( 8, 9,11)(10,12,13)(15,17,18),
      ( 1,22,16)( 2,12,21)( 3,26,23)( 4,17,11)( 5,19,13)( 6,20,10)
        ( 7,14,27)( 8,24,18)( 9,25,15) ]) , multiplication ),
  ExplicitMultiplicationNearRing ( Group(
    [ ( 1, 6,15)( 2, 9, 7)( 3,13,25)( 4,14,10)( 5,18,27)( 8,23,20)
        (11,22,19)(12,24,16)(17,26,21),
      ( 1, 5,17)( 2, 8,22)( 3,12, 4)( 6,18,26)( 7,20,11)( 9,23,19)
        (10,25,16)(13,24,14)(15,27,21),
      ( 1, 2, 4)( 3, 6,11)( 5, 9,16)( 7,13,17)( 8,14,21)(10,18,22)
        (12,15,23)(19,24,26)(20,25,27) ]) , multiplication ) ]
gap> Intersection( si );
ExplicitMultiplicationNearRing ( Group(
[ ( 1,23,14)( 2,13, 6)( 3,27,22)( 4,18, 9)( 5,20,12)( 7,16,26)
    ( 8,25,17)(10,21,19)(11,24,15) ]) , multiplication )
gap> Size( last );
3
```

2.19 Identity of a nearring

▶ `Identity(` *nr* `)`

The function `Identity` returns the identity of the multiplicative semigroup of the nearring *nr* if it exists and `fail` otherwise.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> Identity( n );
fail
gap> n := LibraryNearRingWithOne( GTW24_4, 8 );
LibraryNearRingWithOne(24/4, 8)
gap> Identity( n );
((1,2,3,4,5,6)(7,8))
```

▶ `IsNearRingWithOne(` *nr* `)`

The function `IsNearRingWithOne` returns `true` if the nearring was constructed as a nearring with one and `false` otherwise. To decide whether a nearring has an identity use `Identity(`*nr*`)=true`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsNearRingWithOne( n );
false
gap> n := LibraryNearRingWithOne( GTW24_4, 8 );
LibraryNearRingWithOne(24/4, 8)
gap> Identity( n );
((1,2,3,4,5,6)(7,8))
gap> IsNearRingWithOne( n );
true
```

2.20 Distributivity in a nearring

▶ `Distributors(` *nr* `)`

An element x of a nearring $(N, +, \cdot)$ is called a **distributor** if $x = (n_1 + n_2) \cdot n_3 - (n_1 \cdot n_3 + n_2 \cdot n_3)$ for some elements n_1, n_2, n_3 of N.

The function `Distributors` returns a list containing the distributors of the nearring *nr*.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsNearRingWithOne( n );
false
gap> Distributors( n );
[ (()), ((2,3,4)), ((2,4,3)), ((1,2)(3,4)), ((1,2,3)), ((1,2,4)),
  ((1,3,2)), ((1,3,4)), ((1,3)(2,4)), ((1,4,2)), ((1,4,3)),
  ((1,4)(2,3)) ]
```

▶ `DistributiveElements(` *nr* `)`

An element d of a left nearring $(N, +, \cdot)$ is called a *distributive element* if it is also right distributive over all elements, i.e. $\forall n_1, n_2 \in N : (n_1 + n_2) \cdot d = n_1 \cdot d + n_2 \cdot d$.

The function `DistributiveElements` returns a list containing the distributive elements of the nearring *nr*.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> DistributiveElements( n );
[ (()) ]
```

▶ `IsDistributiveNearRing(` *nr* `)`

A left nearring N is called *distributive nearring* if its multiplication is also right distributive.

The function `IsDistributiveNearRing` simply checks if all elements are distributive and returns the according boolean value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsDistributiveNearRing( n );
false
```

2.21 Elements of a nearring with special properties

▶ `ZeroSymmetricElements(` *nr* `)`

Let $(N, +, \cdot)$ be a left nearring and denote by 0 the neutral element of $(N, +)$. An element n of N is called a *zero-symmetric element* if $0 \cdot n = 0$.

Remark: note that in a *left* nearring $n \cdot 0 = 0$ is true for all elements n.

The function `ZeroSymmetricElements` returns a list containing the zero-symmetric elements of the nearring nr.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> ZeroSymmetricElements( n );
[ (()), ((2,3,4)), ((2,4,3)), ((1,2)(3,4)), ((1,2,3)), ((1,2,4)),
  ((1,3,2)), ((1,3,4)), ((1,3)(2,4)), ((1,4,2)), ((1,4,3)),
  ((1,4)(2,3)) ]
```

▶ `IdempotentElements(` *nr* `)`

The function `IdempotentElements` returns a list containing the idempotent elements of the multiplicative semigroup of the nearring nr.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IdempotentElements( n );
[ (()), ((1,4)(2,3)) ]
```

▶ `NilpotentElements(` *nr* `)`

Let $(N, +, \cdot)$ be a nearring with zero 0. An element n of N is called *nilpotent* if there is a positive integer k such that $n^k = 0$.

The function `NilpotentElements` returns a list of sublists of length 2 where the first entry is a nilpotent element n and the second entry is the smallest k such that $n^k = 0$.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> NilpotentElements( n );
[ [ (()), 1 ], [ ((2,3,4)), 2 ], [ ((2,4,3)), 2 ],
  [ ((1,2)(3,4)), 2 ], [ ((1,2,3)), 2 ], [ ((1,2,4)), 2 ],
  [ ((1,3,2)), 2 ], [ ((1,3,4)), 2 ], [ ((1,4,2)), 2 ],
  [ ((1,4,3)), 2 ] ]
```

▶ `QuasiregularElements(` *nr* `)`

Let $(N, +, \cdot)$ be a left nearring. For an element $z \in N$, denote the right ideal generated by the set $\{n - z \cdot n | n \in N\}$ by L_z. An element z of N is called *quasiregular* if $z \in L_z$.

The function `QuasiregularElements` returns a list of all quasiregular elements of a nearring nr.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> QuasiregularElements( n );
[ (()), ((2,3,4)), ((2,4,3)), ((1,2)(3,4)), ((1,2,3)), ((1,2,4)),
  ((1,3,2)), ((1,3,4)), ((1,3)(2,4)), ((1,4,2)), ((1,4,3)) ]
```

▶ `RegularElements(` *nr* `)`

Let $(N, +, \cdot)$ be a nearring. An element n of N is called *regular* if there is an element x such that $n \cdot x \cdot n = n$.

The function `RegularElements` returns a list of all regular elements of a nearring nr.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> RegularElements( n );
[ (()), ((1,3)(2,4)), ((1,4)(2,3)) ]
```

2.22 Special properties of a nearring

▶ `IsAbelianNearRing(` *nr* `)`

A nearring is called *abelian* if its group reduct is abelian.

The function `IsAbelianNearRing` returns the according boolean value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsAbelianNearRing( n );
false
```

▶ `IsAbstractAffineNearRing(` *nr* `)`

A left nearring N is called *abstract affine* if its group reduct is abelian and its zero-symmetric elements are exactly its distributive elements.

The function `IsAbstractAffineNearRing` returns the according boolean value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsAbstractAffineNearRing( n );
false
```

▶ `IsBooleanNearRing(` *nr* `)`

A left nearring N is called *boolean* if all its elements are idempotent with respect to multiplication.

The function `IsBooleanNearRing` simply checks if all elements are idempotent and returns the according boolean value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsBooleanNearRing( n );
false
```

▶ `IsNilNearRing(` *nr* `)`

A nearring N is called *nil* if all its elements are nilpotent.

The function `IsNilNearRing` checks if all elements are nilpotent and returns the according boolean value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsNilNearRing( n );
false
```

▶ `IsNilpotentNearRing(` *nr* `)`

A nearring N is called *nilpotent* if there is a positive integer k, s.t. $N^k = \{0\}$.

The function `IsNilpotentNearRing` tests if the nearring *nr* is nilpotent and returns the according boolean value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsNilpotentNearRing( n );
false
```

▶ `IsNilpotentFreeNearRing(` *nr* `)`

A nearring N is called *nilpotent free* if its only nilpotent element is 0.

The function `IsNilpotentFreeNearRing` checks if 0 is the only nilpotent and returns the according boolean value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsNilpotentFreeNearRing( n );
false
```

▶ `IsCommutative(` *nr* `)`

A nearring $(N, +, \cdot)$ is called *commutative* if its multiplicative semigroup is commutative.

The function `IsCommutative` returns the according value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsCommutative( n );
false
```

▶ `IsDgNearRing(` *nr* `)`

A nearring $(N, +, \cdot)$ is called *distributively generated (d.g.)* if $(N, +)$ is generated additively by the distributive elements of the nearring.

The function `IsDgNearRing` returns the according value `true` or `false` for a nearring *nr*.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsDgNearRing( n );
false
```

▶ `IsIntegralNearRing(` *nr* `)`

A nearring $(N, +, \cdot)$ with zero element 0 is called *integral* if it has no zero divisors, i.e. the condition $\forall n_1, n_2 : n_1 \cdot n_2 = 0 \Rightarrow n_1 = 0 \vee n_2 = 0$ holds.

The function `IsIntegralNearRing` returns the according value `true` or `false` for a nearring *nr*.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsIntegralNearRing( n );
false
```

▶ `IsPrimeNearRing(` *nr* `)`

A nearring $(N, +, \cdot)$ with zero element 0 is called *prime* if the ideal $\{0\}$ is a prime ideal.

The function `IsPrimeNearRing` checks if *nr* is a prime nearring by using the condition *for all non-zero ideals* $I, J : I \cdot J \neq \{0\}$ and returns the according value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsPrimeNearRing( n );
true
```

▸ `IsQuasiregularNearRing(` *nr* `)`

A nearring N is called *quasiregular* if all its elements are quasiregular.

The function `IsQuasiregularNearRing` simply checks if all elements of the nearring *nr* are quasiregular and returns the according boolean value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsQuasiregularNearRing( n );
false
```

▸ `IsRegularNearRing(` *nr* `)`

A nearring N is called *regular* if all its elements are regular.

The function `IsRegularNearRing` simply checks if all elements of the nearring *nr* are regular and returns the according boolean value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsRegularNearRing( n );
false
```

▸ `IsNearField(` *nr* `)`

Let $(N, +, \cdot)$ be a nearring with zero 0 and denote by N^* the set $N - \{0\}$. N is a *nearfield* if $(N, +, \cdot)$ has an identity and $(N^*, \cdot)$ is a group.

The function `IsNearField` tests if *nr* has an identity and if every non-zero element has a multiplicative inverse and returns the according value `true` or `false`.

```
gap> n := LibraryNearRing( GTW12_4, 8 );
LibraryNearRing(12/4, 8)
gap> IsNearField( n );
false
```

▸ `IsPlanarNearRing(` *nr* `)`

Let $(N, +, \cdot)$ be a left nearring. For $a, b \in N$ we define $a \equiv b$ iff $a \cdot n = b \cdot n$ for all $n \in N$. If $a \equiv b$, then a and b are called *equivalent multipliers*. A nearring N is called *planar* if $|N/_{\equiv}| \geq 3$ and if for any two non-equivalent multipliers a and b in N, for any $c \in N$, the equation $a \cdot x = b \cdot x + c$ has a unique solution.

The function `IsPlanarNearRing` returns the according value `true` or `false` for a nearring *nr*.

```
gap> n := LibraryNearRing( GTW9_2, 90 );
LibraryNearRing(9/2, 90)
gap> IsPlanarNearRing( n );
true
```

▸ `IsWdNearRing(` *nr* `)`

A left nearring $(N, +, \cdot)$ is called *weakly divisible* if $\forall a, b \in N \exists x \in N : a \cdot x = b$ or $b \cdot x = a$.

The function `IsWdNearRing` returns the according value *true* or *false* for the nearring *nr*.

```
gap> nr := LibraryNearRing( GTW9_1, 185 );
LibraryNearRing(9/1, 185)
gap> IsWdNearRing( nr );
true
```

3 The nearring library

The nearring library contains all nearrings up to order 15 and all nearrings with identity up to order 31. All nearrings in the library are nearrings constructed via `ExplicitMultiplicationNearRingNC`, so all functions for these nearrings are applicable to `LibraryNearRing`s.

3.1 Extracting nearrings from the library

▶ `LibraryNearRing(` *G*`,` *num* `)`

`LibraryNearRing` retrieves a nearring from the nearrings library files. G must be a group of order ≤ 15. *num* must be an integer which indicates the number of the class of nearrings on the specified group.

(*Remark:* due to the large number of nearrings on D_{12}, make sure that you have enough main memory - say at least 32 MB - available if you want to get a library nearring on D_{12}).

If G is given as a `TWGroup`, then a nearring is returned whose group reduct is *equal to* G. Otherwise the result is a nearring whose group reduct is *isomorphic to* G, and a warning is issued.

The number of nearrings definable on a certain group G can be accessed via

▶ `NumberLibraryNearRings(` *G* `)`

▶ `AllLibraryNearRings(` *G* `)`

returns a list of all nearrings (in the library) that have the group G as group reduct.

```
gap> l := AllLibraryNearRings( GTW3_1 );
[ LibraryNearRing(3/1, 1), LibraryNearRing(3/1, 2),
  LibraryNearRing(3/1, 3), LibraryNearRing(3/1, 4),
  LibraryNearRing(3/1, 5) ]
gap> Filtered( l, IsNearField );
[ LibraryNearRing(3/1, 3) ]
gap> NumberLibraryNearRings( GTW14_2 );
1821
gap> LN14_2_1234 := LibraryNearRing( GTW14_2, 1234 );
LibraryNearRing(14/2, 1234)
```

▶ `LibraryNearRingWithOne(` *G*`,` *num* `)`

`LibraryNearRingWithOne` retrieves a nearring from the nearrings library files. G must be one of the predefined groups of order ≤ 31. *num* must be an integer which indicates the number of the class of nearrings with identity on the specified group.

The number of nearrings with identity definable on a certain group G can be accessed via

▶ `NumberLibraryNearRingsWithOne(` *G* `)`

▶ `AllLibraryNearRingsWithOne(` *G* `)`

returns a list of all nearrings with identity (in the library) that have the group G as group reduct.

```
gap> NumberLibraryNearRingsWithOne( GTW24_6 );
0
gap> NumberLibraryNearRingsWithOne( GTW24_4 );
10
gap> LNwI24_4_8 := LibraryNearRingWithOne( GTW24_4, 8 );
LibraryNearRingWithOne(24/4, 8)
gap> AllLibraryNearRingsWithOne( GTW24_6 );
[  ]
```

3.2 Identifying nearrings

▶ `IdLibraryNearRing(` *nr* `)`

The function `IdLibraryNearRing` returns a pair $[G, n]$ such that the nearring *nr* is isomorphic to the nth library nearring on the group G.

```
gap> p := PolynomialNearRing( GTW4_2 );
PolynomialNearRing( 4/2 )
gap> IdLibraryNearRing( p );
[ 8/3, 833 ]
gap> n := LibraryNearRing( GTW3_1, 4 );
LibraryNearRing(3/1, 4)
gap> d := DirectProductNearRing( n, n );
DirectProductNearRing( LibraryNearRing(3/1, 4), LibraryNearRing(3/1, 4)\
 )
gap> IdLibraryNearRing( d );
[ 9/2, 220 ]
```

▶ `IdLibraryNearRingWithOne(` *nr* `)`

The function `IdLibraryNearRingWithOne` returns a pair $[G, n]$ such that the nearring *nr* is isomorphic to the nth library nearring with identity on the group G. This function can only be applied to nearrings which have an identity.

```
gap> l := LibraryNearRingWithOne( GTW12_3, 1 );
LibraryNearRingWithOne(12/3, 1)
gap> IdLibraryNearRing( l ); #this command requires time and memory!!!
[ 12/3, 37984 ]
gap> IdLibraryNearRingWithOne( l );
[ 12/3, 1 ]
```

3.3 IsLibraryNearRing

▶ `IsLibraryNearRing(` *nr* `)`

The function `IsLibraryNearRing` returns `true` if the nearring *nr* has been read from the nearring library and `false` otherwise.

```
gap> IsLibraryNearRing( LNwI24_4_8 );
true
```

3.4 Accessing the information about a nearring stored in the library

▶ `LibraryNearRingInfo(` *group*, *list*, *string* `)`

This function provides information about the specified library nearrings in a way similar to how nearrings are presented in the appendix of [Pil??].The parameter *group* specifies a predefined group; valid names are exactly those which are also valid for the function `LibraryNearrings` (cf. Section 3.1).

The parameter *list* must be a non-empty list of integers defining the classes of nearrings to be printed. Naturally, these integers must all fit into the ranges described in Section 3.1 for the according groups.

The third parameter *string* is optional. *string* must be a string consisting of one or more (or all) of the following characters: `l`, `m`, `i`, `v`, `s`, `e`, `a`. Per default, (i.e. if this parameter is not specified), the output is minimal. According to each specified character, the following is added:

`a`
: list the nearring automorphisms.

`c`
: print the meaning of the letters used in the output.

`e`
: list the nearring endomorphisms.

`g`
: list the endomorphisms of the group reduct.

`i`
: list the ideals.

`l`
: list the left ideals.

`m`
: print the multiplication tables.

`r`
: list the right ideals.

`s`
: list the subnearrings.

`v`
: list the invariant subnearrings.

Examples:

`LibraryNearRingInfo( GTW3_1, [ 3 ], "lmivsea" )` will print all available information about the third class of nearrings on the group Z_3.

`LibraryNearRingInfo( GTW4_1, [ 1..12 ] )` will provide a minimal output for all classes of nearrings on Z_4.

`LibraryNearRingInfo( GTW6_2, [ 5, 18, 21 ], "mi" )` will print the minimal information plus the multiplication tables plus the ideals for the classes 5, 18, and 21 of nearrings on the group S_3.

4 Arbitrary functions on groups: EndoMappings

An *endomapping* is a mapping with equal source and range, say G, where G is a group. An endomapping on G then acts on G by *transforming* each element of G into (precisely one) element of G. Endomappings are special cases of Mappings.

Endomappings are created by the constructor functions `EndoMappingByPositionList`, `EndoMappingByFunction`, `IdentityEndoMapping`, `ConstantEndoMapping`, and are represented as mappings. The functions described in this section can be found in the file `grptfms.g?`.

4.1 Defining endo mappings

▶ `EndoMappingByPositionList ( ` G`, ` *list* ` )`

The constructor function `EndoMappingByPositionList` returns the the endomapping that maps the i-th element of the group (in the ordering given by AsSortedList) to the i-th element of list.

```
gap> G := GTW4_2;
4/2
gap> t1 := EndoMappingByPositionList ( G, [1, 2, 4, 4] );
<mapping: 4/2 -> 4/2 >
```

▶ `EndoMappingByFunction( ` G`, ` *fun* ` )`

The constructor function `EndoMappingByFunction` returns the function *fun* that maps elements of the group G into G as an endomapping.

```
gap> t2 := EndoMappingByFunction ( GTW8_2, g -> g^-1 );
<mapping: 8/2 -> 8/2 >
gap> IsGroupHomomorphism ( t2 );
true
gap> t3 := EndoMappingByFunction ( GTW6_2, g -> g^-1 );
<mapping: 6/2 -> 6/2 >
gap> IsGroupHomomorphism ( t3 );
false
```

`EndoMappings` and `GroupGeneralMappings` are different kinds of objects in GAP: `GroupGeneralMappings` model homomorphisms between two different groups, whereas `EndoMappings` model nonlinear functions on one group. However, `GroupGeneralMappings` can be transformed into `Endomappings` if they have equal source and range.

▶ `AsEndoMapping( ` *map* ` )`

The constructor function `AsEndoMapping` returns the mapping *map* as an endomapping.

```
gap> G1 := Group ((1,2,3), (1, 2));
Group([ (1,2,3), (1,2) ])
gap> G2 := Group ((2,3,4), (2, 3));
Group([ (2,3,4), (2,3) ])
gap> f1 := IsomorphismGroups ( G1, G2 );
[ (1,2,3), (1,2) ] -> [ (2,3,4), (2,3) ]
gap> f2 := IsomorphismGroups ( G2, G1 );
[ (2,3,4), (2,3) ] -> [ (1,2,3), (1,2) ]
gap> AsEndoMapping ( CompositionMapping ( f1, f2 ) );
<mapping: Group( [ (2,3,4), (2,3) ] ) -> Group( [ (2,3,4), (2,3)
] ) >
```

`EndoMappings` and `GroupGeneralMappings` are two completely different kinds of objects in GAP, but they can be transformed into one another.

▸ `AsGroupGeneralMappingByImages(` *endomap* `)`

`AsGroupGeneralMappingByImages` returns the `GroupGeneralMappingByImages` that acts on the group the same way as the endomapping *endomap*. It only makes sense to use this function for endomappings that are group endomorphisms.

```
gap> m := IdentityEndoMapping ( GTW6_2 );
<mapping: 6/2 -> 6/2 >
gap> AsGroupGeneralMappingByImages ( m );
[ (1,2), (1,2,3) ] -> [ (1,2), (1,2,3) ]
```

▸ `IsEndoMapping(` *obj* `)`

`IsEndoMapping` returns `true` if the object *obj* is an endomapping and `false` otherwise.

```
gap> IsEndoMapping ( InnerAutomorphisms ( GTW6_2 ) [3] );
true
```

▸ `IdentityEndoMapping(` G `)`

`IdentitEndoMapping` is the counterpart to the GAP standard library function `IdentityMapping`. It returns the identity transformation on the group G.

```
gap> AsList ( UnderlyingRelation ( IdentityEndoMapping ( Group ((1,2,3,4)) ) ) );
[ Tuple( [ (), () ] ), Tuple( [ (1,2,3,4), (1,2,3,4) ] ),
  Tuple( [ (1,3)(2,4), (1,3)(2,4) ] ), Tuple( [ (1,4,3,2), (1,4,3,2) ] )
 ]
```

▸ `ConstantEndoMapping(` G, g `)`

`ConstantEndoMapping` returns the endomapping on the group G which maps everything to the group element g of G.

```
gap> C3 := CyclicGroup (3);
<pc group of size 3 with 1 generators>
gap> m := ConstantEndoMapping (C3, AsSortedList (C3) [2]);
MappingByFunction( <pc group of size 3 with
1 generators>, <pc group of size 3 with
1 generators>, function( x ) ... end )
gap> List (AsList (C3), x -> Image (m, x));
[ f1, f1, f1 ]
```

4.2 Properties of endo mappings

▶ `IsIdentityEndoMapping(` *endomap* `)`

`IsIdentityEndoMapping` returns `true` if *endomap* is the identity function on a group.

```
gap> IsIdentityEndoMapping (EndoMappingByFunction (
> AlternatingGroup ( [1..5] ), x -> x^31));
true
```

▶ `IsConstantEndoMapping(` *endomap* `)`

`IsConstantEndoMapping` returns `true` if the endomapping *endomap* is constant and `false` otherwise.

```
gap> C3 := CyclicGroup ( 3 );
<pc group of size 3 with 1 generators>
gap> IsConstantEndoMapping ( EndoMappingByFunction ( C3,  x -> x^3 ));
true
```

▶ `IsDistributiveEndoMapping(` *endomap* `)`

A mapping t on an (additively written) group G is called *distributive* if for all elements x and y in G: $t(x + y) = t(x) + t(y)$. The function `IsDistributiveEndoMapping` returns the according boolean value `true` or `false`.

```
gap> G := Group ( (1,2,3), (1,2) );
Group([ (1,2,3), (1,2) ])
gap> IsDistributiveEndoMapping ( EndoMappingByFunction ( G, x -> x^3));
false
gap> IsDistributiveEndoMapping ( EndoMappingByFunction ( G, x -> x^7));
true
```

4.3 Operations for endo mappings

While the composition operator `*` is applicable to mappings and transformations, the operation `+` (pointwise addition of the images) can only be applied to transformations.

The product operator `*` returns the transformation which is obtained from the transformations *t1* and *t2* by composition of *t1* and *t2* (i.e. performing *t2 after t1*).

```
gap> t1 := ConstantEndoMapping ( GTW2_1, ());
MappingByFunction( 2/1, 2/1, function( x ) ... end )
gap> t2 := ConstantEndoMapping (GTW2_1, (1, 2));
MappingByFunction( 2/1, 2/1, function( x ) ... end )
gap> List ( AsList ( GTW2_1 ), x -> Image ( t1 * t2, x ));
[ (1,2), (1,2) ]
```

The add operator `+` returns the endomapping which is obtained from the endomappings *t1* and *t2* by pointwise addition of *t1* and *t2*. (Note that in this context addition means that for every place x in the source of *t1* and *t2*, GAP performs the operation `p * q`, where `p` is the image of *t1* at *x* and `q` is the image of *t2* at *x*.)

The subtract operator `-` returns the endomapping which is obtained from the endomappings *t1* and *t2* by pointwise subtraction of *t1* and *t2*. (Note that in this context subtraction means performing the GAP operation `p * q^(-1)`, where `p` is the image of *t1* at a place *x* and `q` is the image of *t2* at *x*.)

```
gap> G := SymmetricGroup ( 3 );
Sym( [ 1 .. 3 ] )
gap> invertingOnG := EndoMappingByFunction ( G, x -> x^-1 );
<mapping: SymmetricGroup( [ 1 .. 3 ] ) -> SymmetricGroup(
[ 1 .. 3 ] ) >
gap> identityOnG := IdentityEndoMapping (G);
<mapping: SymmetricGroup( [ 1 .. 3 ] ) -> SymmetricGroup(
[ 1 .. 3 ] ) >
gap> AsSortedList ( G );
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
gap> List ( AsSortedList (G),
>              x -> Image ( identityOnG * invertingOnG, x ));
[ (), (2,3), (1,2), (1,3,2), (1,2,3), (1,3) ]
gap> List ( AsSortedList (G),
>              x -> Image ( identityOnG + invertingOnG, x ));
[ (), (), (), (), (), () ]
gap> IsIdentityEndoMapping ( - invertingOnG );
true
gap> - invertingOnG = identityOnG;
true
```

4.4 Nicer ways to print a mapping

▶ `GraphOfMapping(` *mapping* `)`

`GraphOfMapping` returns the set of all pairs (x,m(x)), where x lies in the source of the mapping. In particular, it returns List (Source (m), x -¿ [x, Image (m, x)]);

```
gap> G := SymmetricGroup ( 3 );
Sym( [ 1 .. 3 ] )
gap> m := ConstantEndoMapping (G, (1,2,3)) + IdentityEndoMapping( G );
MappingByFunction( Sym( [ 1 .. 3 ] ), Sym( [ 1 .. 3 ] ), function( g ) ... end )
gap> PrintArray( GraphOfMapping( m ) );
[ [       (),  (1,2,3) ],
  [    (2,3),    (1,3) ],
  [    (1,2),    (2,3) ],
  [  (1,2,3),  (1,3,2) ],
  [  (1,3,2),       () ],
  [    (1,3),    (1,2) ] ]
```

▶ `PrintAsTerm(` *mapping* `)`

If *mapping* is a polynomial function on its source then `PrintAsTerm` prints a polynomial that induces the mapping *mapping*.

```
gap> G := SymmetricGroup ( 3 );
Sym( [ 1 .. 3 ] )
gap> p := Random( PolynomialNearRing( G ) );
<mapping: SymmetricGroup( [ 1 .. 3 ] ) -> SymmetricGroup( [ 1 .. 3 ] ) >
gap> PrintAsTerm( p );
g1 - x - 2 * g1 - g2 - x - g1 - g2 + g1 - x - 2 * g1 -
g2 - x - g1 - g2 - 3 * x + g1
gap> GeneratorsOfGroup( g );
[ (1,2,3), (1,2) ]
```

The expressions `g1` and `g2` stand for the first and secong generator of the group G respectively. The result is not necessarily a polynomial of minimal length.

5 Transformation nearrings

In the previous chapter we introduced mappings on groups, and we called them *endomappings*. We also introduced the operation of pointwise addition `+` for endomappings. Now we are able to use these mappings together with pointwise addition `+` and composition `*` to construct left nearrings. These nearrings satisfy the distributive law $x * (y + z) = x * y + x * z$.

A *transformation nearring* is a set of mappings on a group G that is closed under pointwise addition of mappings, under forming the additive inverse and under functional composition. For more information we suggest [Pil83], [Mel85], and [Cla92],

The algorithms used can be found in [AEN00] and [Aic00].

The elements of a transformation nearring are given as endomappings on the group G (cf. Chapter "Functions on groups that are not necessarily homomorphisms: EndoMappings").

5.1 Constructing transformation nearrings

▶ `TransformationNearRingByGenerators(` *G* `,` *endomaplist* `)`

For a (possibly empty) list *endomaplist* of endomappings on a group G, the constructor function `TransformationNearRingByGenerators` returns the nearring generated by these mappings. All of them must be endomappings on the group G.

```
gap> g := AlternatingGroup ( 4 );
Alt( [ 1 .. 4 ] )
gap> AsSortedList ( g );
[ (), (2,3,4), (2,4,3), (1,2)(3,4), (1,2,3), (1,2,4), (1,3,2),
  (1,3,4), (1,3)(2,4), (1,4,2), (1,4,3), (1,4)(2,3) ]
gap> t := EndoMappingByPositionList ( g, [1,3,4,5,2,1,1,1,1,1,1,1] );
<mapping: AlternatingGroup( [ 1 .. 4 ] ) -> AlternatingGroup(
[ 1 .. 4 ] ) >
gap> m := TransformationNearRingByGenerators ( g, [t] );
TransformationNearRingByGenerators(
[ <mapping: AlternatingGroup( [ 1 .. 4 ] ) -> AlternatingGroup(
    [ 1 .. 4 ] ) > ])
gap> Size (m); # may take a few moments
20736
gap> IsCommutative ( m );
false
```

▶ `TransformationNearRingByAdditiveGenerators(` *G* `,` *endomaplist* `)`

If a transformation nearring is known to be additively generated by a set of endomappings on a group (as for example the distributively generated nearrings $E(G)$, $A(G)$ and $I(G)$), the function `TransformationNearRingByAdditiveGenerators` allows to construct this nearring. The only difference between `TransformationNearRingByGenerators` and `TransformationNearRingByAdditiveGenerators` is that `TransformationNearRingByAdditiveGenerators` is much faster.

```
gap> endos := Endomorphisms ( GTW6_2 );
[ [ (2,3), (1,3,2) ] -> [ (), () ],
  [ (2,3), (1,3,2) ] -> [ (2,3), () ],
  [ (2,3), (1,3,2) ] -> [ (1,2), () ],
  [ (2,3), (1,3,2) ] -> [ (1,3), () ],
  [ (1,2), (1,2,3) ] -> [ (2,3), (1,2,3) ],
  [ (2,3), (1,3,2) ] -> [ (1,2), (1,2,3) ],
  [ (2,3), (1,3,2) ] -> [ (1,3), (1,2,3) ],
  [ (2,3), (1,3,2) ] -> [ (2,3), (1,3,2) ],
  [ (2,3), (1,3,2) ] -> [ (1,2), (1,3,2) ],
  [ (2,3), (1,3,2) ] -> [ (2,3), (1,2,3) ] ]
gap> Endo := TransformationNearRingByAdditiveGenerators ( GTW6_2, endos );
< transformation nearring with 10 generators >
gap> Size ( Endo );
54
```

5.2 Nearrings of transformations

▶ `MapNearRing(` *G* `)`

▶ `TransformationNearRing(` *G* `)`

`MapNearRing` and `TransformationNearRing` both return the nearring of all mappings on G.

```
gap> m := MapNearRing ( GTW32_12 );
TransformationNearRing(32/12)
gap> Size ( m );
1461501637330902918203684832716283019655932542976
gap> NearRingIdeals ( m );
[ < nearring ideal >, < nearring ideal > ]
```

▶ `IsFullTransformationNearRing(` *tfmnr* `)`

The function `IsFullTransformationNearRing` returns `true` if the transformation nearring *tfmnr* is the nearring of all mappings over the group.

```
gap> g := CyclicGroup ( 4 );
<pc group of size 4 with 2 generators>
gap> m := MapNearRing ( g );
TransformationNearRing(<pc group of size 4 with 2 generators>)
gap> gens := Filtered ( AsList ( m ),
>       f -> IsFullTransformationNearRing (
>               TransformationNearRingByGenerators ( g, [ f ] )));;
gap> Length(gens);
12
```

▶ `PolynomialNearRing(` *G* `)`

`PolynomialNearRing` returns the nearring of all polynomial functions on G.

```
gap> P := PolynomialNearRing ( GTW16_6 );
PolynomialNearRing( 16/6 )
gap> Size ( P );
256
```

▶ `EndomorphismNearRing(` *G* `)`

`EndomorphismNearRing` returns the nearring generated by all endomorphisms on G.

```
gap> ES4 := EndomorphismNearRing ( SymmetricGroup ( 4 ) );
EndomorphismNearRing( Sym( [ 1 .. 4 ] ) )
gap> Size ( ES4 );
927712935936
```

▶ `AutomorphismNearRing(` *G* `)`

`AutomorphismNearRing` returns the nearring generated by all automorphisms on G.

```
gap> A := AutomorphismNearRing ( DihedralGroup ( 8 ) );
AutomorphismNearRing( <pc group of size 8 with 3 generators> )
gap> Length(NearRingRightIdeals ( A ));
28
gap> Size (A);
32
```

▶ `InnerAutomorphismNearRing(` *G* `)`

`InnerAutomorphismNearRing` returns the nearring generated by all inner automorphisms on G.

```
gap> I := InnerAutomorphismNearRing ( AlternatingGroup ( 4 ) );
InnerAutomorphismNearRing( Alt( [ 1 .. 4 ] ) )
gap> Size ( I );
3072
gap> m := Enumerator( I )[1000];
<mapping: AlternatingGroup( [ 1 .. 4 ] ) -> AlternatingGroup(
[ 1 .. 4 ] ) >
gap> graph := List ( AsList ( AlternatingGroup ( 4 ) ),
> x -> [x, Image (m, x)] );
[ [ (), () ], [ (2,3,4), (1,2)(3,4) ], [ (2,4,3), (1,4)(2,3) ],
  [ (1,2)(3,4), (1,2)(3,4) ], [ (1,2,3), (1,4)(2,3) ],
  [ (1,2,4), (1,3)(2,4) ], [ (1,3,2), () ], [ (1,3,4), (1,4)(2,3) ],
  [ (1,3)(2,4), (1,3)(2,4) ], [ (1,4,2), (1,4)(2,3) ],
  [ (1,4,3), (1,4)(2,3) ], [ (1,4)(2,3), (1,4)(2,3) ] ]
```

▶ `CompatibleFunctionNearRing(` *G* `)`

`CompatibleFunctionNearRing` returns the nearring of all compatible functions on the group G. A function $m : G \rightarrow G$ is compatible iff for every normal subgroup N of G and all $g, h \in G$ if g and h are in the same coset of N then their images under m are in the same coset of G.

▶ `ZeroSymmetricCompatibleFunctionNearRing(` *G* `)`

`ZeroSymmetricCompatibleFunctionNearRing` returns the nearring of all zerosymmetric compatible functions on the group G. This function is also called by `CompatibleFunctionNearRing`.

▶ `IsCompatibleEndoMapping(` *m* `)`

`IsCompatibleEndoMapping` returns `true` iff m is a compatible function on its source.

▶ `Is1AffineComplete(` *G* `)`

A group G is called 1-affine complete, iff every compatible function on G is polynomial. `Is1AffineComplete` returns `true` iff G is 1-affine complete.

▶ `CentralizerNearRing(` *G*`,` *endos* `)`

`CentralizerNearRing` returns the nearring of all functions $m : G \rightarrow G$ such that for all endomorphisms e in *endos* the equality $m \circ e = e \circ m$ holds.

```
gap> autos := Automorphisms ( GTW8_4 );
[ IdentityMapping( 8/4 ), ^(2,4),
  [ (1,2,3,4), (2,4) ] -> [ (1,4,3,2), (1,2)(3,4) ],
  [ (1,2,3,4), (2,4) ] -> [ (1,2,3,4), (1,2)(3,4) ], ^(1,4)(2,3),
  ^(1,2,3,4), [ (2,4), (1,2,3,4) ] -> [ (1,4)(2,3), (1,2,3,4) ],
  [ (1,2,3,4), (2,4) ] -> [ (1,4,3,2), (1,4)(2,3) ] ]
gap> C := CentralizerNearRing ( GTW8_4, autos );
CentralizerNearRing( 8/4, ... )
gap> C0 := ZeroSymmetricPart ( C );
< transformation nearring with 4 generators >
gap> Size ( C0 );
32
gap> Is := NearRingIdeals ( C0 );
[ < nearring ideal >, < nearring ideal >, < nearring ideal >,
  < nearring ideal >, < nearring ideal >, < nearring ideal >,
  < nearring ideal >, < nearring ideal >, < nearring ideal >,
  < nearring ideal >, < nearring ideal >, < nearring ideal >,
  < nearring ideal > ]
gap> List (Is, Size);
[ 1, 2, 4, 2, 4, 8, 4, 8, 16, 8, 16, 16, 32 ]
```

▶ `RestrictedEndomorphismNearRing(` G, U `)`

`RestrictedEndomorphismNearRing` returns the nearring generated by all endomorphisms e on G with $e(G) \subseteq U$.

```
gap> G := GTW16_8;
16/8
gap> U := First ( NormalSubgroups ( G ),
>               x -> Size (x) = 2 );
Group([ ( 1, 5)( 2,10)( 3,11)( 4,12)( 6,15)( 7,16)( 8, 9)(13,14) ])
gap> HGU := RestrictedEndomorphismNearRing (G, U);
RestrictedEndomorphismNearRing( 16/8, Group(
[ ( 1, 5)( 2,10)( 3,11)( 4,12)( 6,15)( 7,16)( 8, 9)(13,14) ]) )
gap> Size (HGU);
8
gap> IsDistributiveNearRing ( HGU );
true
gap> Filtered ( AsList ( HGU),
>        x -> x = x * x );
[ <mapping: 16/8 -> 16/8 > ]
```

▶ `LocalInterpolationNearRing(` *tfmnr*, m `)`

`LocalInterpolationNearRing` returns the nearring of all mappings on G that can be interpolated at any set of m places by a mapping in *tfmnr*, where G is the domain and codomain of the elements in *tfmnr*.

```
gap> P := PolynomialNearRing ( GTW8_5 );
PolynomialNearRing( 8/5 )
gap> L := LocalInterpolationNearRing ( P, 2 );
LocalInterpolationNearRing( PolynomialNearRing( 8/5 ), 2 )
gap> Size ( L ) / Size ( P );
16
```

5.3 The group a transformation nearring acts on

▶ `Gamma(` *tfmnr* `)`

The function `Gamma` returns the group on which the mappings of the nearring *tfmnr* act.

```
gap> Gamma ( PolynomialNearRing ( CyclicGroup ( 25 ) ) );
<pc group of size 25 with 2 generators>
gap> IsCyclic (last);
true
```

5.4 Transformation nearrings and other nearrings

▶ `AsTransformationNearRing(` *nr* `)`

Provided that *nr* is not already a transformation nearring, `AsTransformationNearRing` returns a transformation nearring that is isomorphic to the nearring *nr*.

```
gap> L := LibraryNearRing (GTW8_3, 12);
LibraryNearRing(8/3, 12)
gap> Lt := AsTransformationNearRing ( L );
< transformation nearring with 3 generators >
gap> Gamma ( Lt );
8/3 x C_2
```

▶ `AsExplicitMultiplicationNearRing(` *nr* `)`

Provided that *nr* is not already an explicit multiplication nearring (i. e. a transformation nearring), `AsExplicitMultiplicationNearRing` returns an explicit multiplication nearring that is isomorphic to the nearring *nr*.

```
gap> P := PolynomialNearRing ( GTW4_2 );
PolynomialNearRing( 4/2 )
gap> n := AsExplicitMultiplicationNearRing ( P );
ExplicitMultiplicationNearRing ( Group(
[ ( 1, 2)( 5, 6)( 9,10)(13,14), ( 3, 4)( 7, 8)(11,12)(15,16),
  ( 7, 8)( 9,10)(13,14)(15,16) ]) , multiplication )
```

5.5 Noetherian quotients for transformation nearrings

▶ `NoetherianQuotient(` *tfmnr*, *target*, *source* `)`

`NoetherianQuotient` returns the set of all mappings t in *tfmnr* with $t($`source`$) \subseteq$ `target`.

```
gap> G := SymmetricGroup ( 4 );
Sym( [ 1 .. 4 ] )
gap> V := First ( NormalSubgroups ( G ), x -> Size ( x ) = 4 );
Group([ (1,4)(2,3), (1,2)(3,4) ])
gap> P := InnerAutomorphismNearRing ( G );
InnerAutomorphismNearRing( Sym( [ 1 .. 4 ] ) )
gap> N := NoetherianQuotient ( P, V, G );
NoetherianQuotient( Group([ (1,4)(2,3), (1,3)(2,4) ]) ,Sym(
[ 1 .. 4 ] ) )
gap> Size ( P ) / Size ( N );
54
```

▸ `CongruenceNoetherianQuotient(` *P*, *A*, *B*, *C* `)`

`CongruenceNoetherianQuotient` returns the ideal of all those mappings in *P* that map every element of the group Gamma(P) into *C*, and maps two elements that are congruent modulo *B* into elements that are congruent modulo *A*. Input conditions: (1) *P* is the nearring of polynomial functions on a group G, (2) *A* is a normal subgroup of G, (3) *B* is a normal subgroup of G, (4) *C* is a normal subgroup of G, (5) [C,B] is less or equal to A.

```
gap> G := GTW8_4;
8/4
gap> P := PolynomialNearRing (G);
PolynomialNearRing( 8/4 )
gap> A := TrivialSubgroup (G);
Group(())
gap> B := DerivedSubgroup (G);
Group([ (1,3)(2,4) ])
gap> C := G;
8/4
gap> I := CongruenceNoetherianQuotient (P, A, B, C);
< nearring ideal >
gap> Size (P/I);
2
```

▸ `CongruenceNoetherianQuotientForInnerAutomorphismNearRings (`*I*, *A*, *B*, *C* `)`

`CongruenceNoetherianQuotientForInnerAutomorphismNearRings` returns the ideal of all those mappings in *I* that map every element of the group Gamma(I) into *C*, and maps two elements that are congruent modulo *B* into elements that are congruent modulo *A*. Input conditions: (1) *P* is the nearring of polynomial functions on a group G, (2) *A* is a normal subgroup of G, (3) *B* is a normal subgroup of G, (4) *C* is a normal subgroup of G, (5) [C,B] is less or equal to A.

```
gap> G := GTW8_4;
8/4
gap> I := InnerAutomorphismNearRing (G);
InnerAutomorphismNearRing( 8/4 )
gap> A := TrivialSubgroup (G);
Group(())
gap> B := DerivedSubgroup (G);
Group([ (1,3)(2,4) ])
gap> C := G;
8/4
gap> j := CongruenceNoetherianQuotientForInnerAutomorphismNearRings (I,A,B,C);
< nearring ideal >
gap> Size (I/j);
2
```

5.6 Zerosymmetric mappings

▶ `ZeroSymmetricPart(` *tfmnr* `)`

`ZeroSymmetricPart` returns the nearring of all mappings t in *tfmnr* with $t(0) = 0$.

```
gap> g := GTW8_4;
8/4
gap> P := PolynomialNearRing ( g );
PolynomialNearRing( 8/4 )
gap> Zp := ZeroSymmetricPart ( P );
< transformation nearring with 4 generators >
gap> InnerAutomorphismNearRing ( g ) = Zp;
true
```

6 Nearring ideals

For an introduction to nearring ideals we suggest [Pil83], [Mel85], and [Cla92].

Ideals of nearrings can either be left, right or twosided ideals. However, all of them are called ideals. Mathematicians tend to use the expression ideal also for subgroups of the group reduct of the nearring. GAP does not allow that.

Left, right or twosided ideals in GAP form their own category `IsNRI`. Whenever a left, right or twosided ideal is constructed it lies in this category. The objects in this category are what GAP considers as ideals. We will refer to them as `NRI`s.

All the functions in this chapter can be applied to all types of nearrings.

The functions described in this chapter can be found in the source files `nrid.g?`, `idlatt.g?` and `nrconstr.g?`.

6.1 Construction of nearring ideals

There are several ways to construct ideals in nearrings. `NearRingLeftIdealByGenerators`, `NearRingRightIdealByGenerators` and `NearRingIdealByGenerators` can be used to construct (left / right) ideals generated by a subset of the nearring. `NearRingLeftIdealBySubgroupNC`, `NearRingRightIdealBySubgroupNC` and `NearRingIdealBySubgroupNC` construct (left / right) ideals from a subgroup of the group reduct of the nearring which is an ideal. Finally `NearRingLeftIdeals`, `NearRingRightIdeals` and `NearRingIdeals` compute lists of all (left / right) ideals of a nearring.

▸ `NearRingIdealByGenerators(` *nr*, *gens* `)`

The function `NearRingIdealByGenerators` takes as arguments a nearring *nr* and a list *gens* of arbitrarily many elements of *nr*. It returns the smallest ideal of *nr* containing all elements of *gens*.

▸ `NearRingLeftIdealByGenerators(` *nr*, *gens* `)`

The function `NearRingLeftIdealByGenerators` takes as arguments a nearring *nr* and a list *gens* of arbitrarily many elements of *nr*. It returns the smallest left ideal of *nr* containing all elements of *gens*.

▸ `NearRingRightIdealByGenerators(` *nr*, *gens* `)`

The function `NearRingRightIdealByGenerators` takes as arguments a nearring *nr* and a list *gens* of arbitrarily many elements of *nr*. It returns the smallest right ideal of *nr* containing all elements of *gens*.

```
gap> n := LibraryNearRing( GTW8_4, 12 );
LibraryNearRing(8/4, 12)
gap> e := AsNearRingElement( n, (1,3)(2,4) );
((1,3)(2,4))
gap> r := NearRingRightIdealByGenerators( n, [e] );
< nearring right ideal >
gap> l := NearRingLeftIdealByGenerators( n, [e] );
< nearring left ideal >
gap> i := NearRingIdealByGenerators( n, [e] );
< nearring ideal >
gap> r = i;
true
gap> l = i;
false
gap> l = r;
false
```

▶ `NearRingIdealBySubgroupNC(` *nr*, *S* `)`

From a nearring *nr* and a subgroup *S* of the group reduct of *nr*, `NearRingIdealBySubgroupNC` constructs a (GAP–) ideal of *nr*. It is assumed (and hence not checked) that *S* is an ideal of *nr*. See Section 6.2 for information how to check this.

▶ `NearRingLeftIdealBySubgroupNC(` *nr*, *S* `)`

From a nearring *nr* and a subgroup *S* of the group reduct of *nr*, `NearRingLeftIdealBySubgroupNC` constructs a (GAP–) left ideal of *nr*. It is assumed (and hence not checked) that *S* is a left ideal of *nr*. See Section 6.2 for information how to check this.

▶ `NearRingRightIdealBySubgroupNC(` *nr*, *S* `)`

From a nearring *nr* and a subgroup *S* of the group reduct of *nr*, `NearRingRightIdealBySubgroupNC` constructs a (GAP–) right ideal of *nr*. It is assumed (and hence not checked) that *S* is a right ideal of *nr*. See Section 6.2 for information how to check this.

```
gap> a := GroupReduct( n );
8/4
gap> nsgps := NormalSubgroups( a );
[ Group(()), Group([ (1,3)(2,4) ]),
  Group([ (1,3)(2,4), (1,2)(3,4) ]), Group([ (1,3)(2,4), (2,4) ]),
  Group([ (1,2,3,4), (1,3)(2,4) ]), 8/4 ]
gap> l := Filtered( nsgps,
> s -> IsSubgroupNearRingRightIdeal( n, s ) );
[ Group(()), Group([ (1,3)(2,4), (2,4) ]), 8/4 ]
gap> l := List( l,
> s -> NearRingRightIdealBySubgroupNC( n, s ) );
[ < nearring right ideal >, < nearring right ideal >,
  < nearring right ideal > ]
```

▶ `NearRingIdeals(` *nr* `)`

`NearRingIdeals` computes all ideals of the nearring *nr*. The return value is a list of ideals of *nr*

For one-sided ideals the functions

▶ `NearRingLeftIdeals(` *nr* `)`

and

▶ `NearRingRightIdeals(` *nr* `)`

can be used.

```
gap> NearRingIdeals( n );
[ < nearring ideal >, < nearring ideal >, < nearring ideal > ]
gap> NearRingRightIdeals( n );
[ < nearring left ideal >, < nearring left ideal >,
  < nearring left ideal > ]
gap> NearRingLeftIdeals( n );
[ < nearring right ideal >, < nearring right ideal >,
  < nearring right ideal >, < nearring right ideal > ]
```

6.2 Testing for ideal properties

▶ `IsNRI(` *obj* `)`

`IsNRI` returns `true` if the object *obj* is a left ideal, a right ideal or an ideal of a nearring. (Such an object may be considered as a (one or twosided) GAP – nearring ideal.)

▶ `IsNearRingLeftIdeal(` *I* `)`

The function `IsNearRingLeftIdeal` can be applied to any `NRI`. It returns `true` if *I* is a left ideal in its parent nearring.

▶ `IsNearRingRightIdeal(` *I* `)`

The function `IsNearRingRightIdeal` can be applied to any `NRI`. It returns `true` if *I* is a right ideal in its parent nearring.

▶ `IsNearRingIdeal(` *I* `)`

The function `IsNearRingIdeal` can be applied to any `NRI`. It returns `true` if *I* is an ideal in its parent nearring.

```
gap> n := LibraryNearRing( GTW6_2, 39 );
LibraryNearRing(6/2, 39)
gap> e := Enumerator(n)[3];
((1,3,2))
gap> l := NearRingLeftIdealByGenerators( n, [e] );
< nearring left ideal >
gap> IsNRI( l );
true
gap> IsNearRingLeftIdeal( l );
true
gap> IsNearRingRightIdeal( l );
true
gap> l;
< nearring ideal >
```

▶ `IsSubgroupNearRingLeftIdeal(` *nr*, *S* `)`

Let $(N, +, \cdot)$ be a nearring. A subgroup S of the group $(N, +)$ is a *left ideal* of N if for all a, b in N and s in S: $a \cdot (b + s) - a \cdot b$ in S. `IsSubgroupNearRingLeftIdeal` takes as arguments a nearring *nr* and a subgroup S of the group reduct of *nr* and returns `true` if S is a nearring ideal of *nr* and `false` otherwise.

Note, that if `IsSubgroupNearRingLeftIdeal` returns `true` this means that S is a left ideal only in the mathematical sense, not in GAP–sense (it is a group, not a left ideal). You can use `NearRingLeftIdealBySubgroupNC` (see Section 6.1) to construct the corresponding left ideal.

▶ `IsSubgroupNearRingRightIdeal(` *nr*, *S* `)`

Let $(N, +, \cdot)$ be a nearring. A subgroup S of the group $(N, +)$ is a *right ideal* of N if $S \cdot N \subseteq S$. `IsSubgroupNearRingRightIdeal` takes as arguments a nearring *nr* and a subgroup S of the group reduct of *nr* and returns `true` if S is a right ideal of *nr* and `false` otherwise.

Note, that if `IsSubgroupNearRingRightIdeal` returns `true` this means that S is a right ideal only in the mathematical sense, not in GAP–sense (it is a group, not a right ideal). You can use `NearRingRightIdealBySubgroupNC` (see Section 6.1) to construct the corresponding right ideal.

```
gap> n := LibraryNearRing( GTW6_2, 39 );
LibraryNearRing(6/2, 39)
gap> s := Subgroups( GroupReduct( n ) );
[ Group(()), Group([ (2,3) ]), Group([ (1,3) ]), Group([ (1,2) ]),
  Group([ (1,3,2) ]), Group([ (1,2,3), (1,2) ]) ]
gap> List( s, sg -> IsSubgroupNearRingLeftIdeal( n, sg ) );
[ true, false, false, false, true, true ]
gap> List( s, sg -> IsSubgroupNearRingRightIdeal( n, sg ) );
[ true, false, false, false, true, true ]
```

6.3 Special ideal properties

▶ `IsPrimeNearRingIdeal(` *I* `)`

An ideal I of a nearring N is *prime* if for any two ideals J and K of N whenever $J \cdot K$ is contained in I then at least one of them is contained in I. `IsPrimeNearRingIdeal` returns `true` if I is a prime ideal in its parent nearring and `false` otherwise.

```
gap> n := LibraryNearRingWithOne( GTW27_2, 5 );
LibraryNearRingWithOne(27/2, 5)
gap> Filtered( NearRingIdeals( n ), IsPrimeNearRingIdeal );
[ < nearring ideal of size 9 >, < nearring ideal of size 27 > ]
```

▶ `IsMaximalNearRingIdeal(` *I* `)`

A proper ideal I of a nearring N is *maximal* if there is no proper ideal containing I properly. `IsMaximalNearRingIdeal(` *I* `) returns ‘true` if I is a maximal ideal in its parent nearring and `false` otherwise.

```
gap> n := LibraryNearRingWithOne( GTW27_2, 5 );
LibraryNearRingWithOne(27/2, 5)
gap> Filtered( NearRingIdeals( n ), IsMaximalNearRingIdeal );
[ < nearring ideal > ]
```

6.4 Generators of nearring ideals

▶ `GeneratorsOfNearRingIdeal(` *I* `)`

For an `NRI` I the function `GeneratorsOfNearRingIdeal` returns a set of elements of the parent nearring of I that generates I as an ideal.

▶ `GeneratorsOfNearRingLeftIdeal(` *I* `)`

For an `NRI` I the function `GeneratorsOfNearRingLeftIdeal` returns a set of elements of the parent nearring of I that generates I as a left ideal.

▶ `GeneratorsOfNearRingRightIdeal(` *I* `)`

For an `NRI` *I* the function `GeneratorsOfNearRingRightIdeal` returns a set of elements of the parent nearring of *I* that generates *I* as a right ideal.

6.5 Near-ring ideal elements

▶ `AsList(` *I* `)`

The function `AsList` computes the elements of the (left / right) ideal *I*. It returns the elements as a list.

▶ `AsSortedList(` *I* `)`

does essentially the same, but returns a set of elements.

▶ `Enumerator(` *I* `)`

does essentially the same as `AsList`, but returns an enumerator for the elements of *nr*.

```
gap> n := LibraryNearRing( GTW8_2, 2 );
LibraryNearRing(8/2, 2)
gap> li := NearRingLeftIdeals( n );
[ < nearring left ideal >, < nearring left ideal >,
  < nearring left ideal >, < nearring left ideal >,
  < nearring left ideal >, < nearring left ideal > ]
gap> l := li[3];
< nearring left ideal >
gap> e := Enumerator( l );;
gap> e[2];
((1,2)(3,6,5,4))
gap> AsList( e ); AsList( l );
[ (()), ((1,2)(3,6,5,4)), ((3,5)(4,6)), ((1,2)(3,4,5,6)) ]
[ (()), ((1,2)(3,6,5,4)), ((3,5)(4,6)), ((1,2)(3,4,5,6)) ]
```

6.6 Random ideal elements

▶ `Random(` *I* `)`

`Random` returns a random element of the (left / right) ideal *I*.

```
gap> Random( l );
((3,5)(4,6))
```

6.7 Membership of an ideal

For a (left / right) ideal *I* of a nearring *N* and an element *n* of *N*

▶ *n* `in` *I*

tests whether *n* is an element of *I*.

```
gap> Random( n ) in l;
true
gap> Random( n ) in l;
false
```

6.8 Size of ideals

▶ `Size(` *I* `)`

`Size` returns the number of elements of the (left / right) ideal *I*.

6.9 Group reducts of ideals

▶ `GroupReduct(` *I* `)`

`GroupReduct` returns the group reduct of the (left / right) ideal *I*.

6.10 Comparision of ideals

▶ *I* `=` *J*

If *I* and *J* are (left / right) ideals of the same nearring and consist of the same elements, then `true` is returned. Otherwise the answer is `false`.

6.11 Operations with ideals

The most important operations for nearring (left / right) ideals are *meet* and *join* in the lattice. GAP offers the functions `Intersection`, `ClosureNearRingLeftIdeal`, `ClosureNearRingRightIdeal` and `ClosureNearRingLeftIdeal` for this purpose.

▶ `Intersection(` *ideallist* `)`

computes the intersection of the (left / right) ideals in the list *ideallist*. All of the (left / right) ideals in *ideallist* must be (left / right) ideals of the same nearring.

▶ `Intersection(` *I1* `, ... ,` *In* `)`

computes the intersection of the (left / right) ideals *I1*, ..., *In*.

In both cases the result is again a (left / right) ideal.

▶ `ClosureNearRingLeftIdeal(` *L1* `,` *L2* `)`

The function `ClosureNearRingLeftIdeal` computes the left ideal *L1* + *L2* of the nearRing *N* if both *L1* and *L2* are (left) ideals of *N*.

▶ `ClosureNearRingRightIdeal(` *R1* `,` *R2* `)`

The function `ClosureNearRingRightIdeal` computes the right ideal *L1* + *L2* of the nearring *N* if both *R1* and *R2* are (right) ideals of *N*.

▶ `ClosureNearRingIdeal(` *I1* `,` *I2* `)`

The function `ClosureNearRingIdeal` computes the ideal *L1* + *L2* of the nearring *N* if both *I1* and *I2* are ideals of *N*.

6.12 Commutators

▶ `NearRingCommutator(` *I*, *J* `)`

The function `NearRingCommutator` returns the commutator of the two ideals *I* and *J* of a common nearring.

```
gap> l := LibraryNearRing( GTW6_2, 3 );
LibraryNearRing(6/2, 3)
gap> i := NearRingIdeals( l );
[ < nearring ideal >, < nearring ideal > ]
gap> List( i, Size );
[ 1, 6 ]
gap> NearRingCommutator( i[2], i[2] );
< nearring ideal of size 6 >
```

The function `PrintNearRingCommutatorsTable` prints a complete overview over the action of the commutator operator on a group.

```
gap> l := LibraryNearRing( GTW8_4, 13 );
LibraryNearRing(8/4, 13)
gap> NearRingIdeals( l );
[ < nearring ideal >, < nearring ideal >, < nearring ideal > ]
gap> PrintNearRingCommutatorsTable( l );
[ 1, 1, 1 ]
[ 1, 1, 2 ]
[ 1, 2, 2 ]
```

6.13 Simple nearrings

▶ `IsSimpleNearRing(` *nr* `)`

The function `IsSimpleNearRing` returns `true` if the nearring *nr* has no proper (two-sided) ideals.

```
gap> NumberLibraryNearRings( GTW4_2 );
23
gap> Filtered( AllLibraryNearRings( GTW4_2 ), IsSimpleNearRing );
[ LibraryNearRing(4/2, 3), LibraryNearRing(4/2, 16),
  LibraryNearRing(4/2, 17) ]
```

6.14 Factor nearrings

▶ `FactorNearRing(` *nr*, *I* `)`

For a nearring *nr* and an ideal *I* of the nearring *nr* the function `FactorNearRing` returns the factor nearring of *nr* modulo the ideal *I*. Alternatively,

▶ *nr* `/` *I*

can be used and has the same effect.

The result is always an `ExplicitMultiplicationNearRing`, so all functions for such nearrings can be applied to the factor nearring.

```
gap> n := LibraryNearRing( GTW8_2, 2 );
LibraryNearRing(8/2, 2)
gap> e := AsNearRingElement( n, (1,2) );
((1,2))
gap> e in n;
true
gap> i := NearRingRightIdealByGenerators( n, [e] );
< nearring right ideal >
gap> Size(i);
4
gap> IsNearRingLeftIdeal( i );
true
gap> i;
< nearring ideal of size 4 >
gap> f := n/i;
FactorNearRing( LibraryNearRing(8/2, 2), < nearring ideal of size 4 > )
gap> IdLibraryNearRing(f);
[ 2/1, 1 ]
```

7 Graphic ideal lattices (X-GAP only)

If you run SONATA under XGAP it is possible to study ideal lattices of nearrings graphically.

▶ `GraphicIdealLattice(` *nr*, *string* `)`

The function `GraphicIdealLattice` computes one- and two-sided nearring ideals and uses XGAP's graphic capabilities to draw the ideal lattice of the nearring *nr*. The string *string* determines, which ideals are shown:

If *string* contains the letter `l`, left ideals are shown, if *string* contains the letter `r`, right ideals are shown, and if *string* contains the letter `i`, two-sided ideals are shown. Any combination of these letters is possible.

Right ideals of the nearring are represented by squares, left ideals by diamonds and two-sided ideals by circles. It is possible, that two-sided ideals are shown as right or left ideals, if the two-sided ideal property has not yet been tested.

Left clicking on an ideal allows one to select the ideal and determine more information about the ideal. Choosing `Ideal type` from the `Ideals`-menu determines, whether the selected ideal is two-sided (in which case its shape changes to a circle immediately). On a right click on an ideal a window opens showing more information about the ideal, such as its size and the isomorphism class of the factor nearring by the ideal (if it is two-sided). The information is computed and displayed as soon as you click on the corresponding entry in the window. Finally clicking on `Export ideal to GAP` makes the ideal the last output of XGAP's main window, where you may then go on working with it.

It is also possible to select several ideals at the same time pressing the Ctrl-key while left clicking on the ideals. When several ideals are selected it is possible to compute their sums and intersection from the corresponding `Ideals`-menu entries. The result of the computation is then indicated by green color.

8 N-groups

In SONATA every N-group is a group, the only difference is, that there is a nearring that acts on the group. And since in SONATA all nearrings are left distributive, they act on the elements of an N-group from the right. *Note*, that the elements of an N-group are added via `*`, not `+`.

The functions described in this section can be found in the source files `ngroups.g?`.

8.1 Construction of N-groups

There is a natural way to construct an N-group. It is to take a group, a nearring and define an action of the nearring on the group. The function `NGroup` allows one to do this. The special case, where (the group reduct of) a nearring is viewed as an N-group over the nearring itself, can be constructed easily via `NGroupByNearRingMultiplication`.

▶ `NGroup(` *G*`,` *nr*`,` *action* `)`

The function `NGroup` has three arguments. *G* must be a group, *nr* the nearring that acts on the group and *action* a binary operation from the direct product of *G* and *nr* into *G*. It returns the N-group.

```
gap> G := GTW4_2;
4/2
gap> n := MapNearRing( G );
TransformationNearRing(4/2)
gap> action := function ( g, f )
> return Image( f, g );
> end;
function ( g, f ) ... end
gap> gamma := NGroup( G, n, action );
< N-group of TransformationNearRing(4/2) >
gap> IsNGroup( gamma );
true
gap> NearRingActingOnNGroup( gamma );
TransformationNearRing(4/2)
gap> ActionOfNearRingOnNGroup( gamma );
function ( g, f ) ... end
gap> Print( ActionOfNearRingOnNGroup( gamma ) );
function ( g, f )
    return Image( f, g );
```

▶ `NGroupByNearRingMultiplication(` *nr* `)`

For every (left) nearring $(N, +, \cdot)$ the group $(N, +)$ is an N-group over N with respect to nearring multiplication from the right as the action. The function `NGroupByNearRingMultiplication` returns this N-group of the nearring *nr*.

```
gap> n := LibraryNearRing( GTW8_2, 3 );
LibraryNearRing(8/2, 3)
gap> NGroupByNearRingMultiplication( n ) = GTW8_2;
true
```

▸ `NGroupByApplication(` *tfmnr* `)`

For a nearring T of transformations on a group G, G is an N-group of T with the application of functions as the action. The function `NGroupByApplication` returns this `N-group` of the nearring *tfmnr*.

Another way to construct an N-Group is to take a nearring N, a right ideal R and let N act on the factor N/R in the canonical way. This is accomplished by

▸ `NGroupByRightIdealFactor(` *nr*, *R* `)`

The function `NGroupByRightIdealFactor` has two arguments, a nearring *nr* and a right ideal *R*. It returns the N-group *nr/R*.

```
gap> N := LibraryNearRing( GTW4_2, 11 );
LibraryNearRing(4/2, 11)
gap> R := NearRingRightIdeals( N )[ 3 ];
< nearring right ideal >
gap> ng := NGroupByRightIdealFactor( N, R );
< N-group of LibraryNearRing(4/2, 11) >
gap> PrintTable( ng );
Let:
(0,0) := (())
(1,0) := ((3,4))
(0,1) := ((1,2))
(1,1) := ((1,2)(3,4))
---------------------------------------------------------------------
g0 := <identity> of ...
g1 := f1

N = LibraryNearRing(4/2, 11) acts on
G = Group( [ f1 ] )
from the right by the following action:

          g0  g1
  ---------------
  (0,0)   g0  g0
  (1,0)   g0  g0
  (0,1)   g0  g1
  (1,1)   g0  g1
```

8.2 Operation tables of N-groups

▸ `PrintTable(` *G* `)`

The function `PrintTable` prints out the operation table of the action of a nearring on its N-group G

```
gap> n := LibraryNearRing( TWGroup( 8, 2 ), 3 );
LibraryNearRing(8/2, 3)
gap> gamma := NGroupByNearRingMultiplication( n );
< N-group of LibraryNearRing(8/2, 3) >
gap> PrintTable( gamma );
Let:
n0 := (())
n1 := ((3,4,5,6))
n2 := ((3,5)(4,6))
n3 := ((3,6,5,4))
n4 := ((1,2))
n5 := ((1,2)(3,4,5,6))
n6 := ((1,2)(3,5)(4,6))
n7 := ((1,2)(3,6,5,4))
--------------------------------------------------------------------
g0 := ()
g1 := (3,4,5,6)
g2 := (3,5)(4,6)
g3 := (3,6,5,4)
g4 := (1,2)
g5 := (1,2)(3,4,5,6)
g6 := (1,2)(3,5)(4,6)
g7 := (1,2)(3,6,5,4)

N = LibraryNearRing(8/2, 3) acts on
G = Group( [ (1,2), (3,4,5,6) ] )
from the right by the following action:

       g0  g1  g2  g3  g4  g5  g6  g7
  ------------------------------------
  n0   g0  g0  g0  g0  g0  g0  g0  g0
  n1   g0  g0  g0  g0  g0  g0  g0  g2
  n2   g0  g0  g0  g0  g0  g0  g0  g0
  n3   g0  g0  g0  g0  g0  g0  g0  g2
  n4   g0  g0  g0  g0  g0  g0  g0  g0
  n5   g0  g0  g0  g0  g0  g0  g0  g2
  n6   g0  g0  g0  g0  g0  g0  g0  g0
  n7   g0  g0  g0  g0  g0  g0  g0  g2
```

8.3 Functions for N-groups

▶ `IsNGroup(` *G* `)`

For any group *G* the function `IsNGroup` tests whether there is a nearring defined that acts on *G*.

▶ `NearRingActingOnNGroup(` *G* `)`

The function `NearRingActingOnNGroup` returns the nearring that acts on the N-group *G*.

```
gap> n := LibraryNearRing( TWGroup( 8, 2 ), 3 );
LibraryNearRing(8/2, 3)
gap> gamma := NGroupByNearRingMultiplication( n );
< N-group of LibraryNearRing(8/2, 3) >
gap> NearRingActingOnNGroup( gamma );
LibraryNearRing(8/2, 3)
```

▶ `ActionOfNearRingOnNGroup(` *G* `)`

The function `ActionOfNearRingOnNGroup` returns the action of the nearring that acts on the N-group G as a 2-ary GAP-function.

```
gap> n := LibraryNearRing( TWGroup( 8, 2 ), 3 );
LibraryNearRing(8/2, 3)
gap> gamma := NGroupByNearRingMultiplication( n );
< N-group of LibraryNearRing(8/2, 3) >
gap> ActionOfNearRingOnNGroup( gamma );
function ( g, n ) ... end
```

8.4 N-subgroups

▶ `NSubgroup(` *G*`,` *gens* `)`

The function `NSubgroup` returns the N-subgroup of the N-group G generated by *gens*.

▶ `NSubgroups(` *G* `)`

The function `NSubgroups` returns a list of all N-subgroups of the N-group G.

▶ `IsNSubgroup(` *G*`,` *S* `)`

The function `IsNSubgroup` returns `true` iff S is an N-subgroup of G.

8.5 N0-subgroups

▶ `N0Subgroups(` *G* `)`

The function `N0Subgroups` returns a list of all N_0-subgroups of the N-group G.

```
gap> n := LibraryNearRing(GTW12_3,20465);
LibraryNearRing(12/3, 20465)
gap> ng := NGroupByNearRingMultiplication( n );
< N-group of LibraryNearRing(12/3, 20465) >
gap> Length( N0Subgroups( ng ) );
9
```

8.6 Ideals of N-groups

▶ `NIdeal(` *G*`,` *gens* `)`

The function `NIdeal` returns the N-ideal of the N-group G generated by *gens*.

▶ `NIdeals(` *G* `)`

The function `NGroupIdeals` returns a list of all ideals of the N-group G.

```
gap> n:=LibraryNearRing(GTW12_3,20465);
LibraryNearRing(12/3, 20465)
gap> ng := NGroupByNearRingMultiplication( n );
< N-group of LibraryNearRing(12/3, 20465) >
gap> NIdeals( ng );
[ < N-group of LibraryNearRing(12/3, 20465) >,
  < N-group of LibraryNearRing(12/3, 20465) >,
  < N-group of LibraryNearRing(12/3, 20465) > ]
```

▶ `IsNIdeal(` *G*, *I* `)`

The function `IsNIdeal` returns `true` iff *I* is an N-ideal of the N-group *G*.

▶ `IsSimpleNGroup(` *G* `)`

The function `IsSimpleNGroup` returns `true` if *G* is a simple N-group and `false` otherwise.

▶ `IsN0SimpleNGroup(` *G* `)`

The function `IsN0SimpleNGroup` returns `true` if the N-group *G* is N_0-simple and `false` otherwise.

8.7 Special properties of N-groups

▶ `IsCompatible(` *G* `)`

The function `IsCompatible` returns `true` if the N-group *G* is compatible and `false` otherwise.

▶ `IsTame(` *G* `)`

The function `IsTame` returns `true` if *G* is a tame N-group and `false` otherwise.

▶ `Is2Tame(` *G* `)`

The function `Is2Tame` returns `true` if the N-group *G* is 2-tame and `false` otherwise.

▶ `Is3Tame(` *G* `)`

The function `Is3Tame` returns `true` if the N-group *G* is 3-tame and `false` otherwise.

▶ `IsMonogenic(` *G* `)`

The function `IsMonogenic` returns `true` if the N-group *G* is monogenic and `false` otherwise.

▶ `IsStronglyMonogenic(` *G* `)`

The function `IsStronglyMonogenic` returns `true` if the N-group *G* is strongly monogenic and `false` otherwise.

▶ `TypeOfNGroup(` *G* `)`

The function `TypeOfNGroup` returns the type of a monogenic N-group *G*. If *N* is not monogenic or not of type 0, 1 or 2 it returns `fail`.

```
gap> n:=LibraryNearRing(GTW12_3,20465);
LibraryNearRing(12/3, 20465)
gap> ng := NGroupByNearRingMultiplication( n );
< N-group of LibraryNearRing(12/3, 20465) >
gap> TypeOfNGroup( ng );
fail
```

8.8 Noetherian quotients

▶ `NoetherianQuotient(` *nr*, *ngrp*, *target*, *source* `)`

It is assumed that *source* and *target* are subsets of the *nr*-group *ngrp*. The function `NoetherianQuotient` computes the set of all elements *f* of *nr* such that *source*f* is a subset of *target*. If *target* is an *nr*-ideal of *ngrp*, the Noetherian quotient is returned as a near ring ideal, if *target* is an *nr*-subgroup of *ngrp*, a left ideal of *nr* is returned. Otherwise the result is a subset of *nr*.

In the following example we let a nearring act on its group reduct and compute the noetherian quotient $(I, I)_N$ for an ideal I of N.

```
gap> N := LibraryNearRing( GTW12_3, 100 );
LibraryNearRing(12/3, 100)
gap> I := NearRingIdeals( N );
[ < nearring ideal >, < nearring ideal >, < nearring ideal > ]
gap> List(I,Size);
[ 1, 6, 12 ]
gap> NN := NGroupByNearRingMultiplication( N );
< N-group of LibraryNearRing(12/3, 100) >
gap> NoetherianQuotient( N, NN, GroupReduct(I[2]), GroupReduct(I[2]) );
< nearring ideal >
gap> Size(last);
12
```

8.9 Nearring radicals

▶ `NuRadical(` *nr*, *nu* `)`

The function `NuRadical` has two arguments, a nearring *nr* and a number *nu* which must be one of `0`, `1/2`, `1` and `2`. It returns the ν-radical for $\nu = 0, 1/2, 1, 2$ respectively.

▶ `NuRadicals(` *nr* `)`

the function `NuRadicals` returns a record with the components `J_0`, `J1_2`, `J1` and `J2` with the corresponding radicals.

```
gap> f := LibraryNearRing( GTW8_4, 3 );
LibraryNearRing(8/4, 3)
gap> NuRadicals( f );
rec( J2 := < nearring ideal >, J1 := < nearring ideal >,
  J1_2 := < nearring right ideal >, J0 := < nearring ideal > )
gap> NuRadical( f, 1/2 );
< nearring right ideal >
gap> Size( NuRadical( f, 0 ) );
8
gap> AsList( NuRadical( f, 1 ) );
[ (()), ((2,4)), ((1,2)(3,4)), ((1,2,3,4)), ((1,3)), ((1,3)(2,4)),
  ((1,4,3,2)), ((1,4)(2,3)) ]
gap> NuRadical( f, 1/2 ) = NuRadical( f, 2 );
true
```

9 Fixed-point-free automorphism groups

The functions described in this chapter are purely group-theoretic and are meant to provide solvable fixed-point-free automorphism groups acting on abelian groups (Frobenius groups with abelian Frobenius kernel and solvable Frobenius complement) for the construction of centralizer nearrings, planar nearrings, designs, and so on.

The classification of fixed-point-free automorphism groups in types I - IV follows Zassenhaus' papers and [Wol67]. The fixed-point-free automorphism groups acting on abelian groups are constructed from fixed-point-free representations as described in [May00].

9.1 Fixed-point-free automorphism groups and Frobenius groups

▸ `IsFpfAutomorphismGroup(` *phi*, *G* `)`

An automorphism group Φ of a group G acts fixed-point-free (fpf) on G if Φ has more than 1 element and no automorphism in Φ except the identity mapping has a fixed point besides the group identity of G.

Φ is fpf on G, iff the semidirect product of G and Φ, with Φ acting naturally on G, is a Frobenius group.

The function `IsFpfAutomorphismGroup` returns the according value `true` or `false` for a group of automorphisms *phi* on the group G.

```
gap> C9 := CyclicGroup( 9 );
<pc group of size 9 with 2 generators>
gap> a := GroupHomomorphismByFunction( C9, C9, x -> x^-1 );;
gap> phi := Group( a );;
gap> Size( phi );
2
gap> IsFpfAutomorphismGroup( phi, C9 );
true
```

▸ `FpfAutomorphismGroupsMaxSize(` *G* `)`

`FpfAutomorphismGroupsMaxSize` returns a list with integers *kmax* and *dmax* where *kmax* is an upper bound for the size of an fpf automorphism group on the group G; for example, the order of G is congruent to 1 modulo *kmax* and *kmax* is odd for nonabelian groups G. The order of any fpf automorphism group *phi* on G divides *kmax*.

Let *phi* be a metacyclic fpf automorphism group acting on G. Then *phi* has a cyclic normal subgroup whose index in *phi* divides *dmax*. Thus, if *dmax* is 1, then G admits cyclic fpf automorphism groups only.

```
gap> G := ElementaryAbelianGroup( 49 );;
gap> FpfAutomorphismGroupsMaxSize( G );
[ 48, 2 ]
gap> C15 := CyclicGroup( 15 );;
gap> FpfAutomorphismGroupsMaxSize( C15 );
[ 2, 1 ]
```

▸ `FrobeniusGroup(` *phi*, *G* `)`

`FrobeniusGroup` constructs the semidirect product of G with the fpf automorphism group *phi* of G with the multiplication $(a, g) * (b, h) = (ab, g^a h)$ by using the function `SemidirectProduct`. Thus a Frobenius group with Frobenius kernel G and Frobenius complement *phi* where the action of *phi* on G is the natural action of automorphisms on the group is returned.

The unique Frobenius group with kernel $G = (Z_3)^2 \times (Z_5)^2$ and quaternion complement is constructed as follows:

```
gap> aux := FpfAutomorphismGroupsMetacyclic( [3,3,5,5], 4, -1 );
[ [ [ [ f1, f2, f3, f4 ] -> [ f1^2, f2^3, f3*f4, f3*f4^2 ],
          [ f1, f2, f3, f4 ] -> [ f2^4, f1, f4^2, f3 ] ] ],
  <pc group of size 225 with 4 generators> ]
gap> phi := Group( aux[1][1] );
<group with 2 generators>
gap> G := aux[2];
<pc group of size 225 with 4 generators>
gap> FrobeniusGroup( phi, G );
<pc group of size 1800 with 7 generators>
```

9.2 Fixed-point-free representations

▶ `IsFpfRepresentation(` *matrices*`,` *F* `)`

Let π be a representation of the group Φ over the finite field F. If for all $\varphi \in \Phi$ except for the identity the matrix $\pi(\varphi)$ does not have 1 as an eigenvalue, then π is said to be fpf.

Let π be an fpf representation of Φ over F with degree d. Then π is faithful, the order of Φ and the characteristic of F are coprime and π is a sum of irreducible faithful fpf F-representations. The matrix group $\pi(\Phi)$ acts fpf on the vectorspace F^d.

For a list of $d \times d$ matrices, *matrices*, over the field F, the function `IsFpfRepresentation` returns `true` if the group generated by *matrices* acts fpf on the d-dimensional vectorspace over F, and `false` otherwise.

```
gap> F := GF(5);;
gap> A := [[2,0],[0,3]]*One(F);
[ [ Z(5), 0*Z(5) ], [ 0*Z(5), Z(5)^3 ] ]
gap> IsFpfRepresentation( [A], F );
true
```

▶ `DegreeOfIrredFpfRepCyclic(` *p*`,` *m* `)`

returns the degree of the irreducible fpf representations of the cyclic group of order m over GF(p), where m and p are coprime.

Note, that all irreducible fpf representations of the cyclic group of order m over GF(p) have the same degree, the multiplicative order of p modulo m, `OrderMod( p, m )`.

```
gap> DegreeOfIrredFpfRepCyclic( 5, 9 );
6
```

▶ `DegreeOfIrredFpfRepMetacyclic(` *p*`,` *m*`,` *r* `)`

returns the degree of the irreducible fpf representations of the metacyclic group Φ determined by parameters m and r over GF(p). If the parameters are not feasible, then an error is returned. See `FpfRepresentationsMetacyclic` for a presentation of this group.

All irreducible fpf representations of the metacyclic group over GF(p) have the same degree, namely the size of multiplicative group generated by p and r modulo m.

We determine the degree of the irreducible fpf representation of the quaternion group over GF(5):

```
gap> DegreeOfIrredFpfRepMetacyclic( 5, 4, -1 );
2
```

▶ `DegreeOfIrredFpfRep2(` *p*`,` *m*`,` *r*`,` *k* `)`

returns the degree of the irreducible fpf representations of the type-II-group Φ determined by parameters m, r, and k over GF(p). If the parameters are not feasible or if the parameters describe the presentation of a metacyclic group, then an error is returned. See `FpfRepresentations2` for a presentation of this group.

All irreducible fpf representations of Φ over GF(p) have the same degree, namely the size of the multiplicative group generated by p, r, and k modulo m.

We determine the degree of the irreducible fpf representation of the smallest, not metacyclic type-2-group (order 120) over the field GF(7):

```
gap> DegreeOfIrredFpfRep2( 7, 30, 11, -1 );
8
```

▶ `DegreeOfIrredFpfRep3(` *p*`,` *m*`,` *r* `)`

returns the degree of the irreducible fpf representations of the type-III-group Φ determined by parameters m and r over GF(p). If the parameters are not feasible, then an error is returned. See `FpfRepresentations3` for a presentation of this group.

All irreducible fpf representations of this group over GF(p) have the same degree.

We determine the degree of the irreducible fpf representation of SL(2,3) over GF(5):

```
gap> DegreeOfIrredFpfRep3( 5, 3, 1 );
2
```

▶ `DegreeOfIrredFpfRep4(` *p*`,` *m*`,` *r*`,` *k* `)`

returns the degree of the irreducible fpf representations of the type-IV-group Φ determined by parameters m, r, and k over GF(p). If the parameters are not feasible, then an error is returned. See `FpfRepresentations4` for a presentation of this group.

All irreducible fpf representations of Φ over GF(p) have the same degree.

We determine the degree of the irreducible fpf representation of the smallest type-4-group, the binary octahedral group of order 48, over $GF(5)$:

```
gap> DegreeOfIrredFpfRep4( 5, 3, 1, -1 );
4
```

▶ `FpfRepresentationsCyclic(` *p*`,` *m* `)`

Let a generate a cyclic group of order m. For p and m coprime `FpfRepresentationsCyclic` returns a list of matrices $\{A^i|i$ in *indexlist*$\}$ over GF(p) as well as the list *indexlist*. For all i in *indexlist* the representation $a \mapsto A^i$ is irreducible and fpf. The A^i with i in *indexlist* describe all irreducible fpf representations up to equivalence; each irreducible fpf representation is equivalent to one $a \mapsto A^i$ and no two representations $a \mapsto A^i$, $a \mapsto A^j$ with $i \neq j$ and i, j in *indexlist* are equivalent.

Note, that every faithful irreducible representation of a cyclic group is fpf. The number of nonequivalent faithful irreducible representations over GF(p) is given as $\phi(m)/d$, where the degree d is given as the multiplicative order of p modulo m and $\phi(m)$ denotes the number of residues coprime to m.

We determine the irreducible matrix representations of the cyclic group of size 8 over $GF(5)$:

```
gap> aux := FpfRepresentationsCyclic( 5, 8 );
[ [ [ [ Z(5)^3, Z(5)^2 ], [ Z(5), Z(5) ] ],
      [ [ Z(5)^2, Z(5) ], [ Z(5)^0, Z(5)^0 ] ] ], [ 1, 7 ] ]
gap> mats := aux[1];
[ [ [ Z(5)^3, Z(5)^2 ], [ Z(5), Z(5) ] ],
  [ [ Z(5)^2, Z(5) ], [ Z(5)^0, Z(5)^0 ] ] ]
gap> indexlist := aux[2];
[ 1, 7 ]
```

▶ `FpfRepresentationsMetacyclic(` *p*`,` *m*`,` *r* `)`

Let Φ be a metacyclic group (i.e., Φ has a cyclic normal subgroup with cyclic factor) admitting an fpf representation. Then Φ fulfills one of the following two presentations, I or II. Both presentations are determined by integers m and r satisfying certain conditions:

Type I. Presentation of an fpf metacyclic group Φ with all Sylow subgroups cyclic. Let m and r satisfy the following conditions:

(a) m and r are coprime.

(b) Let n be the multiplicative order of r modulo m. Then each prime divisor of n divides m.

(c) Let m' be maximal such that m' divides m and m' is coprime to n. Then $r = 1 \bmod (m/m')$.

Type II. Presentation of an fpf metacyclic group Φ with generalized quaternion 2-Sylow subgroup. Let m and r satisfy the following conditions:

(a) m and r are coprime.

(b) Let n be the multiplicative order of r modulo m. Then n is 2 times an odd number and each prime divisor of n divides m.

(c) Let s be maximal such that 2^s divides m. Then $2^s \geq 4$ and $r = -1 \bmod 2^s$.

(d) Let m' be maximal such that m' divides $m/2$ and m' is coprime to $n/2$. Then $r = 1 \bmod (m/m')$.

Then the group Φ with 2 generators a, b satisfying the relations $a^m = 1, b^n = a^{m'}, b^{-1}ab = a^r$ is metacyclic and fpf and has size mn.

A group satisfying presentation I is of type I in the notation of Zassenhaus, Wolf; presentation II gives a type-II-group.

Let m, r be as above, and let p coprime to m. Additionally, we require that m does not divide r-1. (Otherwise, $\Phi = \langle a, b | a^m = 1, b^n = a^{m'}, b^{-1}ab = a^r \rangle$ would be cyclic.) Then `FpfRepresentationsMetacyclic` returns a list of matrices $\{(A^i, B_i) | i$ in *indexlist*$\}$ over GF(p) as well as the list *indexlist*. The GF(p)-representations determined by $a \mapsto A^i$ and $b \mapsto B_i$ are all irreducible and fpf representations of $\Phi = \langle a, b | a^m = 1, b^n = a^{m'}, b^{-1}ab = a^r \rangle$ up to equivalence.

We determine the irreducible matrix representation of the quaternion group (parameters $m = 4, r = -1$) over $GF(7)$:

```
gap> aux := FpfRepresentationsMetacyclic( 7, 4, -1 );
[ [ [ [ [ Z(7)^2, Z(7) ], [ Z(7), Z(7)^5 ] ],
          [ [ 0*Z(7), Z(7)^3 ], [ Z(7)^0, 0*Z(7) ] ] ] ], [ 1 ] ]
gap> mats := aux[1];
[ [ [ [ Z(7)^2, Z(7) ], [ Z(7), Z(7)^5 ] ],
      [ [ 0*Z(7), Z(7)^3 ], [ Z(7)^0, 0*Z(7) ] ] ] ]
```

▶ `FpfRepresentations2(` *p*`,` *m*`,` *r*`,` *k* `)`

The presentation of a type-II-group which is not metacyclic is determined by integers m, r, k satisfying the following conditions:

(a) m and r are coprime, m and k are coprime.

(b) Let n be the multiplicative order of r modulo m. Then n is 2 times an odd number and each prime divisor of n divides m.

(c) Let m' be maximal such that m' divides m and m' is coprime to n. Then $r = 1 \bmod (m/m')$.

(d) Let 2^{s-1} be maximal such that 2^{s-1} divides m. Define $l = -1 \bmod 2^{s-1}$ and $l = 1 \bmod (nm/(2^{s-1}m'))$. Then $k = l \bmod (m/m')$.

(e) The multiplicative order of k modulo m equals 2 and $k \neq r^{(n/2)} \bmod m$.

Then the group Φ with generators a, b, q satisfying the relations $a^m = 1, b^n = a^{m'}, b^{-1}ab = a^r$ and furthermore $q^{-1}aq = a^k, q^{-1}bq = b^l$ is fpf of type II and has size $2mn$.

a, b generate a metacyclic group with all Sylow subgroups cyclic (see conditions (a), (b), (c)) of index 2 in Φ.

For m, r, k as above and p coprime to m `FpfRepresentations2` returns a list of matrices $\{(A_i, B_i, Q_i)|i$ in *indexlist*$\}$ over GF(p) as well as the list *indexlist*. The GF(p)-representations determined by $a \mapsto A_i$, $b \mapsto B_i$ and $q \mapsto Q_i$ are all irreducible, fpf representations of Φ upto equivalence.

We determine the irreducible matrix representations of the smallest type-II-group which is not metacyclic (parameters m = 30, r = 11, k = -1, size 120) over the field GF(11) and obtain 2 nonequivalent fpf representations, each of degree 4:

```
gap> DegreeOfIrredFpfRep2( 11, 30, 11, -1 );
4
gap> aux := FpfRepresentations2( 11, 30, 11, -1 );
[ [ [ <block matrix of dimensions (2*2)x(2*2)>,
          <block matrix of dimensions (2*2)x(2*2)>,
          <block matrix of dimensions (2*2)x(2*2)> ],
      [ <block matrix of dimensions (2*2)x(2*2)>,
          <block matrix of dimensions (2*2)x(2*2)>,
          <block matrix of dimensions (2*2)x(2*2)> ] ], [ 1, 13 ] ]
```

▶ `FpfRepresentations3(` *p*`,` *m*`,` *r* `)`

A group Φ admitting an fpf representation is said to be of type III if Φ is the semidirect product of the quaternion group and a metacyclic fpf group H of odd size, with the quaternion group normal and H permuting the 3 subgroups of order 4.

The presentation of a type-III-group is determined by integers m and r, describing the metacyclic group H and its action on the normal quaternion subgroup. The following conditions have to be satisfied for m, r:

(a) 3 divides m; m is odd; m and r are coprime.

(b) Let n be the multiplicative order of r modulo m. Then each prime divisor of n divides m.

(c) Let m' be maximal such that m' divides m and m' is coprime to n. Then $r = 1 \bmod (m/m')$.

Let p, q with relations $p^4 = 1, q^2 = p^2, q^{-1}pq = p^{-1}$ generate the quaternion group. Let a, b generate a metacyclic group determined by m and r (See `FpfRepresentationsMetacyclic`).

If 3 divides n, then let a commute with p, q and let $b^{-1}pb = q, b^{-1}qb = pq$.

If 3 does not divide n, then let b commute with p, q and let $a^{-1}pa = q, a^{-1}qa = pq$

Then the group Φ with generators p, q, a, b is of type III and has size $8mn$.

For $r \neq 1 \bmod m$, `FpfRepresentations3` returns a list of matrices $\{(P, Q, A_i, B_i)|i$ in *indexlist*$\}$ over GF(p) as well as the list *indexlist*.

For $r = 1$ mod m, the group H is cyclic and `FpfRepresentations3` returns $\{(P, Q, A_i)|i$ in *indexlist*$\}$ over GF(p) and *indexlist*.

The GF(p)-representations determined by $p \mapsto P, q \mapsto Q$ and $a \mapsto A_i, b \mapsto B_i$ are all irreducible, fpf representations of Φ upto equivalence.

We determine the irreducible matrix representation of the smallest type-III-group, namely SL(2,3), (parameters m = 3, r = 1, size 24) over the field GF(5):

```
gap> aux := FpfRepresentations3( 5, 3, 1 );
[ [ [ [ [ Z(5), 0*Z(5) ], [ 0*Z(5), Z(5)^3 ] ],
          [ [ 0*Z(5), Z(5)^2 ], [ Z(5)^0, 0*Z(5) ] ],
          [ [ Z(5)^3, Z(5)^0 ], [ Z(5), Z(5)^0 ] ] ] ], [ 1 ] ]
```

▶ `FpfRepresentations4(` *p*`,` *m*`,` *r*`,` *k* `)`

A group $\Phi = \langle p, q, a, b, z\rangle$ admitting an fpf representation is said to be of type IV, if it has a normal subgroup $H = \langle p, q, a, b\rangle$ of type III and index 2.

The presentation of a type-IV-group is determined by integers m, r, k satisfying the following conditions:

(a) Let s be maximal such that 3^s divides m. Then $s \geq 1$; m is odd; m and r are coprime.

(b) Let n be the multiplicative order of r modulo m. Then 3 does not divide n; each prime divisor of n divides m.

(c) Let m' be maximal such that m' divides m and m' is coprime to n. Then $r = 1$ mod (m/m').

(d) $k = -1$ mod 3^s, $k = 1$ mod (m/m') and $k^2 = 1$ modulo m.

Let p, q, a, b generate a type-III-group determined by m, r with relations as given in Section `FpfRepresentations3`. Additionally, let $z^2 = p^2, z^{-1}pz = qp, z^{-1}qz = q^{-1}$ and $z^{-1}az = a^k, z^{-1}bz = b$.

Then the group Φ with generators p, q, a, b and z is of type IV and has size $16mn$.

For $r \neq 1$ mod m, `FpfRepresentations4` returns a list of matrices $\{(P, Q, A_i, B_i, Z_i)|i$ in *indexlist*$\}$ over GF(p) as well as the list *indexlist*.

For $r = 1$ mod m, the function `FpfRepresentations4` returns $\{(P, Q, A_i, Z_i)|i$ in *indexlist*$\}$ over GF(p) and *indexlist*.

The GF(p)-representations determined by $p \mapsto P, q \mapsto Q$ and $a \mapsto A_i, b \mapsto B_i, z \mapsto Z_i$ are all irreducible, fpf representations of Φ upto equivalence.

We determine the 2 nonequivalent irreducible matrix representations of the smallest type-IV-group (binary octahedral group, m = 3, r = 1, k = -1, size 48) over the field GF(7):

```
gap> aux := FpfRepresentations4( 7, 3, 1, -1 );
[ [ [ [ [ Z(7)^2, Z(7) ], [ Z(7), Z(7)^5 ] ],
          [ [ 0*Z(7), Z(7)^3 ], [ Z(7)^0, 0*Z(7) ] ],
          [ [ Z(7)^2, 0*Z(7) ], [ Z(7)^0, Z(7)^4 ] ],
          [ [ Z(7)^5, Z(7) ], [ Z(7), Z(7)^2 ] ] ],
      [ [ [ Z(7)^2, Z(7) ], [ Z(7), Z(7)^5 ] ],
          [ [ 0*Z(7), Z(7)^3 ], [ Z(7)^0, 0*Z(7) ] ],
          [ [ Z(7)^2, 0*Z(7) ], [ Z(7)^0, Z(7)^4 ] ],
          [ [ Z(7)^2, Z(7)^4 ], [ Z(7)^4, Z(7)^5 ] ] ] ],
  [ [ 1, 1 ], [ -1, 1 ] ] ]
```

9.3 Fixed-point-free automorphism groups

▶ `FpfAutomorphismGroupsCyclic(` *ints*`,` *m* `)`

If `AbelianGroup(`*ints*`)` admits a cyclic fpf automorphism group of size m, then `FpfAutomorphismGroupsCyclic` determines one representative for each conjugacy class of such fpf automorphism groups. Conjugacy is determined within the whole automorphism group of `AbelianGroup(`*ints*`)`.

ints has to be a list of prime power integers and is sorted in the function, according to the order $p^i \leq q^j \Leftrightarrow p < q$ or ($p = q$ and $j < i$).

`AbelianGroup(`*ints*`)` admits a cyclic fpf automorphism group of size m iff the multiplicity of each prime power p^i in *ints* is divisible by `DegreeOfIrredFpfRepCyclic( p, m )`.

A list of generators of the nonconjugate fpf automorphism groups is returned together with the group `AbelianGroup(`*ints*`)`, on which the automorphisms act. Here *ints* is sorted with the order above.

The generators, *as*, of the 2 nonconjugate cyclic fpf automorphism groups of order 4 on $Z_{25} \times Z_5$ are computed as follows:

```
gap> aux := FpfAutomorphismGroupsCyclic( [25,5], 4 );
[ [ [ f1, f3 ] -> [ f1^2*f2, f3^2 ],
      [ f1, f3 ] -> [ f1^2*f2, f3^3 ] ],
  <pc group of size 125 with 2 generators> ]
gap> as := aux[1];
[ [ f1, f3 ] -> [ f1^2*f2, f3^2 ], [ f1, f3 ] -> [ f1^2*f2, f3^3 ] ]
gap> G := aux[2];
<pc group of size 125 with 2 generators>
```

▶ `FpfAutomorphismGroupsMetacyclic(` *ints*`,` *m*`,` *r* `)`

If `AbelianGroup(`*ints*`)` admits a metacyclic fpf automorphism group determined by parameters m and r that is not cyclic (see `FpfRepresentationsMetacyclic` for a presentation), then `FpfAutomorphismGroupsMetacyclic` determines one representative for each conjugacy class of such fpf automorphism groups. Conjugacy is determined within the whole automorphism group of `AbelianGroup(`*ints*`)`.

ints has to be a list of prime power integers and is sorted in the function, according to the order $p^i \leq q^j \Leftrightarrow p < q$ or ($p = q$ and $i \geq j$).

Moreover, the multiplicity of each prime power p^i in *ints* has to be divisible by `DegreeOfIrredFpfRepMetacyclic( p, m, r )`, which is a multiple of the multiplicative order of r modulo m.

A list of pairs of generators (a, b satisfying $b^{-1}ab = a^r, a^m = 1$ and $b^n = a^{m'}$) of the nonconjugate fpf automorphism groups is returned together with the group `AbelianGroup(`*ints*`)`, on which the automorphisms act. Here *ints* is sorted with the order above.

For $G = (Z_3)^2 \times (Z_5)^2$ the quaternion fpf automorphism group of size 8 (parameters $m = 4, r = -1$) is computed as follows:

```
gap> aux := FpfAutomorphismGroupsMetacyclic( [3,3,5,5], 4, -1 );
[ [ [ [ f1, f2, f3, f4 ] -> [ f1^2, f2^3, f3*f4, f3*f4^2 ],
          [ f1, f2, f3, f4 ] -> [ f2^4, f1, f4^2, f3 ] ] ],
  <pc group of size 225 with 4 generators> ]
gap> fs := aux[1];
[ [ [ f1, f2, f3, f4 ] -> [ f1^2, f2^3, f3*f4, f3*f4^2 ],
      [ f1, f2, f3, f4 ] -> [ f2^4, f1, f4^2, f3 ] ] ]
gap> phi := Group( fs[1] );
<group with 2 generators>
gap> G := aux[2];
<pc group of size 225 with 4 generators>
```

On $G = (Z_7)^2 \times (Z_{17})^2$ there are 2 nonconjugate fpf automorphism groups isomorphic to the generalized quaternion group of size 16 (parameters $m = 8, r = -1$):

```
gap> aux := FpfAutomorphismGroupsMetacyclic( [7,7,17,17], 8, -1 );;
gap> fs := aux[1];
[ [ [ f1, f2, f3, f4 ] -> [ f1^9, f2^2, f3^4*f4^2, f3*f4^6 ],
      [ f1, f2, f3, f4 ] -> [ f2^16, f1, f3^4*f4^5, f3^5*f4^3 ] ],
  [ [ f1, f2, f3, f4 ] -> [ f1^9, f2^2, f3^3*f4^5, f3^6*f4 ],
      [ f1, f2, f3, f4 ] -> [ f2^16, f1, f3^3*f4^4, f3*f4^4 ] ] ]
gap> phis := List( fs, Group );
[ <group with 2 generators>, <group with 2 generators> ]
gap> G := aux[2];
<pc group of size 14161 with 4 generators>
```

▶ `FpfAutomorphismGroups2(` *ints*, *m*, *r*, *k* `)`

If `AbelianGroup(`*ints*`)` admits an fpf automorphism group of type II, determined by parameters m, r, k that is not metacyclic (see `FpfRepresentations2` for a presentation), then `FpfAutomorphismGroups2` determines one representative for each conjugacy class of such fpf automorphism groups. Conjugacy is determined within the whole automorphism group of `AbelianGroup(`*ints*`)`.

ints has to be a list of prime power integers and is sorted in the function, according to the order $p^i \leq q^j \Leftrightarrow p < q$ or $(p = q$ and $i \geq j)$.

Note, that the degree of an irreducible fpf representation of a type-II-group which is not metacyclic is divisible by 4 and that the multiplicity of each prime power p^i in *ints* has to be divisible by `DegreeOfIrredFpfRep2( p, m, r, k )`.

A list of triples of generators $(a, b, z$ satisfying $b^{-1}ab = a^r$, $a^m = 1$ and $z^{-1}az = a^k)$ of the nonconjugate fpf automorphism groups is returned together with the group `AbelianGroup(`*ints*`)`, on which the automorphisms act. Here *ints* is sorted with the order above.

Upto conjugacy there is only one fpf automorphism group of type II with parameters $m = 30, r = 11, k = -1$, size 120 on the elementary abelian group of size 11^4:

```
gap> aux := FpfAutomorphismGroups2( [11,11,11,11], 30, 11, -1 );
[ [ [ [ f1, f2, f3, f4 ] -> [ f1^5*f2^4, f1^3*f2^10, f3^2*f4^8,
            f3^6*f4 ],
          [ f1, f2, f3, f4 ] -> [ f1^3*f2^10, f1^10*f2^8, f3^8*f4,
            f3*f4^3 ],
          [ f1, f2, f3, f4 ] -> [ f3^10, f4^10, f1, f2 ] ] ],
  <pc group of size 14641 with 4 generators> ]
gap> phi := Group( aux[1][1] );
<group with 3 generators>
gap> G := aux[2];
<pc group of size 14641 with 4 generators>
```

▶ `FpfAutomorphismGroups3(` *ints*, *m*, *r* `)`

If `AbelianGroup(`*ints*`)` admits an fpf automorphism group of type III determined by parameters m and r (see `FpfRepresentations3` for a presentation), then `FpfAutomorphismGroups3` determines one representative for each conjugacy class of such fpf automorphism groups. Conjugacy is determined within the whole automorphism group of `AbelianGroup(`*ints*`)`.

ints has to be a list of prime power integers and is sorted in the function, according to the order $p^i \leq q^j \Leftrightarrow p < q$ or $(p = q$ and $i \geq j)$.

Moreover, the multiplicity of each prime power p^i in *ints* has to be divisible by `DegreeOfIrredFpfRep3( p, m, r )`, which is a multiple of $2n$ where n is the multiplicative order of r modulo m.

A list of tuples of generators, *[p,q,a,b]*, $(p, q$ generating the quaternion group, a, b satisfying $b^{-1}ab = a^r, a^m = 1$ and $b^n = a^{m'})$ of the nonconjugate fpf automorphism groups is returned together with the group `AbelianGroup(`*ints*`)`, on which the automorphisms act. Here *ints* is sorted with the order above.

For $G = (Z_5)^2$ the fpf automorphism type-III-group isomorphic to SL(2,3) is computed as follows (parameters $m = 3, r = 1$):

```
gap> aux := FpfAutomorphismGroups3( [5,5], 3, 1 );
[ [ [ [ f1, f2 ] -> [ f1^2, f2^3 ], [ f1, f2 ] -> [ f2^4, f1 ],
          [ f1, f2 ] -> [ f1^3*f2, f1^2*f2 ] ] ],
  <pc group of size 25 with 2 generators> ]
gap> phi := Group( aux[1][1] );
<group with 3 generators>
gap> G := aux[2];
<pc group of size 25 with 2 generators>
```

▸ `FpfAutomorphismGroups4(` *ints*`,` *m*`,` *r*`,` *k* `)`

If `AbelianGroup(`*ints*`)` admits an fpf automorphism group of type IV determined by parameters m, r, k (see `FpfRepresentations4` for a presentation), then `FpfAutomorphismGroups4` determines one representative for each conjugacy class of such fpf automorphism groups. Conjugacy is determined within the whole automorphism group of `AbelianGroup(`*ints*`)`.

ints has to be a list of prime power integers and is sorted in the function, according to the order $p^i \le q^j \Leftrightarrow p < q$ or $(p = q$ and $i \ge j)$.

Moreover, the multiplicity of each prime power p^i in *ints* has to be divisible by `DegreeOfIrredFpfRep4( p, m, r )`, which is a multiple of $2n$ where n is the multiplicative order of r modulo m.

A list of tuples of generators, *[p,q,a,b,z]*, of the nonconjugate fpf automorphism groups is returned together with the group `AbelianGroup(`*ints*`)`, on which the automorphisms act. Here *ints* is sorted with the order above. If $r = 1$ mod m, then a list of tuples, *[p,q,a,z]*, is returned instead.

For $G = (Z_7)^2$ the fpf automorphism type-IV-group isomorphic the binary octahedral group of size 48 (parameters $m = 3, r = 1, k = -1$) is computed as follows:

```
gap> aux := FpfAutomorphismGroups4( [7,7], 3, 1, -1 );
[ [ [ [ f1, f2 ] -> [ f1^2*f2^3, f1^3*f2^5 ],
          [ f1, f2 ] -> [ f2^6, f1 ], [ f1, f2 ] -> [ f1^2, f1*f2^4 ],
          [ f1, f2 ] -> [ f1^5*f2^3, f1^3*f2^2 ] ] ],
  <pc group of size 49 with 2 generators> ]
gap> phi := Group( aux[1][1] );
<group with 4 generators>
gap> G := aux[2];
<pc group of size 49 with 2 generators>
```

10 Nearfields, planar nearrings and weakly divisible nearrings

A *nearfield* is a nearring with 1 where each nonzero element has a multiplicative inverse. The (additive) group reduct of a finite nearfield is necessarily elementary abelian. For an exposition of nearfields we refer to [Wäh87].

Let $(N, +, \cdot)$ be a left nearring. For $a, b \in N$ we define $a \equiv b$ iff $a \cdot n = b \cdot n$ for all $n \in N$. If $a \equiv b$, then a and b are called *equivalent multipliers*. A nearring N is called *planar* if $|N/_{\equiv}| \geq 3$ and if for any two non-equivalent multipliers a and b in N, for any $c \in N$, the equation $a \cdot x = b \cdot x + c$ has a unique solution. See [Cla92] for basic results on planar nearrings.

All finite nearfields are planar nearrings.

A left nearring $(N, +, \cdot)$ is called *weakly divisible* if $\forall a, b \in N \exists x \in N : a \cdot x = b$ or $b \cdot x = a$.

All finite integral planar nearrings are weakly divisible.

10.1 Dickson numbers

▶ IsPairOfDicksonNumbers(q, n)

A pair of Dickson numbers (q, n) consists of a prime power integer q and a natural number n such that for $p = 4$ or p prime, $p|n$ implies $p|q - 1$.

```
gap> IsPairOfDicksonNumbers( 5, 4 );
true
```

10.2 Dickson nearfields

▶ DicksonNearFields(q, n)

All finite nearfields with 7 exceptions can be obtained via socalled coupling maps from finite fields. These nearfields are called Dickson nearfields.

The multiplication map of such a Dickson nearfield is given by a pair of Dickson numbers (q, n) in the following way:

Let $F = GF(q^n)$ and w be a primitive element of F. Let H be the subgroup of $(F \setminus \{0\}, \cdot)$ generated by w^n. Then $\{w^{(q^i-1)/(q-1)} | 0 \leq i \leq n - 1\}$ is a set of coset representatives of H in $F \setminus \{0\}$. For $f \in Hw^{(q^i-1)/(q-1)}$ and $x \in F$ define $f * x = f \cdot x^{q^i}$ and $0 * x = 0$. Then $*$ is a nearfield multiplication on the additive group $(F, +)$.

Note that a Dickson nearfield is not uniquely determined by (q, n), since w can be chosen arbitrarily. Different choices of w may yield isomorphic nearfields.

DicksonNearFields returns a list of the non-isomorphic Dickson nearfields determined by the pair of Dickson numbers (q, n)

```
gap> DicksonNearFields( 5, 4 );
[ ExplicitMultiplicationNearRing ( <pc group of size 625 with
    4 generators> , multiplication ),
  ExplicitMultiplicationNearRing ( <pc group of size 625 with
    4 generators> , multiplication ) ]
```

▶ NumberOfDicksonNearFields(*q*, *n*)

NumberOfDicksonNearFields returns the number of non-isomorphic Dickson nearfields which can be obtained from a pair of Dickson numbers (q, n). This number is given by $\Phi(n)/k$. Here $\Phi(n)$ denotes the number of relatively prime residues modulo n and k is the multiplicative order of p modulo n where p is the prime divisor of q.

```
gap> NumberOfDicksonNearFields( 5, 4 );
2
```

10.3 Exceptional nearfields

▶ ExceptionalNearFields(*q*)

There are 7 finite nearfields which cannot be obtained from finite fields via a Dickson process. They are of size p^2 for $p = 5, 7, 11, 11, 23, 29, 59$. (There exist 2 exceptional nearfields of size 121.)

ExceptionalNearFields returns the list of exceptional nearfields for a given size q.

```
gap> ExceptionalNearFields( 25 );
[ ExplicitMultiplicationNearRing ( <pc group of size 25 with
    2 generators> , multiplication ) ]
```

▶ AllExceptionalNearFields()

There are 7 finite nearfields which cannot be obtained from finite fields via a Dickson process. They are of size p^2 for $p = 5, 7, 11, 11, 23, 29, 59$. (There exist 2 exceptional nearfields of size 121.)

AllExceptionalNearFields without argument returns the list of exceptional nearfields.

```
gap> AllExceptionalNearFields();
[ ExplicitMultiplicationNearRing ( <pc group of size 25 with
    2 generators> , multiplication ),
  ExplicitMultiplicationNearRing ( <pc group of size 49 with
    2 generators> , multiplication ),
  ExplicitMultiplicationNearRing ( <pc group of size 121 with
    2 generators> , multiplication ),
  ExplicitMultiplicationNearRing ( <pc group of size 121 with
    2 generators> , multiplication ),
  ExplicitMultiplicationNearRing ( <pc group of size 529 with
    2 generators> , multiplication ),
  ExplicitMultiplicationNearRing ( <pc group of size 841 with
    2 generators> , multiplication ),
  ExplicitMultiplicationNearRing ( <pc group of size 3481 with
    2 generators> , multiplication ) ]
```

10.4 Planar nearrings

▶ `PlanarNearRing(` *G*, *phi*, *reps* `)`

A finite *Ferrero pair* is a pair of groups (N,Φ) where Φ is a fixed-point-free automorphism group of $(N,+)$.

Starting with a Ferrero pair (N,Φ) we can construct a planar nearring in the following way, [Cla92]: Select representatives, say $e_1,\ldots,e_t$, for some or all of the non-trivial orbits of N under Φ. Let $C=\Phi(e_1)\cup\ldots\cup\Phi(e_t)$. For each $x\in N$ we define $a*x=0$ for $a\in N\setminus C$, and $a*x=\phi_a(x)$ for $a\in\Phi(e_i)\subset C$ and $\phi_a(e_i)=a$. Then $(N,+,*)$ is a (left) planar nearring.

Every finite planar nearring can be constructed from some Ferrero pair together with a set of orbit representatives in this way.

`PlanarNearRing` returns the planar nearring on the group *G* determined by the fixed-point-free automorphism group *phi* and the list of chosen orbit representatives *reps*.

```
gap> C7 := CyclicGroup( 7 );;
gap> i := GroupHomomorphismByFunction( C7, C7, x -> x^-1 );;
gap> phi := Group( i );;
gap> orbs := Orbits( phi, C7 );
[ [ <identity> of ... ], [ f1, f1^6 ], [ f1^2, f1^5 ],
  [ f1^3, f1^4 ] ]
gap> # choose reps from the orbits
gap> reps := [orbs[2][1], orbs[3][2]];
[ f1, f1^5 ]
gap> n := PlanarNearRing( C7, phi, reps );
ExplicitMultiplicationNearRing ( <pc group of size 7 with
1 generators> , multiplication )
```

▶ `OrbitRepresentativesForPlanarNearRing(` *G*, *phi*, *i* `)`

Let (N,Φ) be a Ferrero pair, and let $E=\{e_1,\ldots,e_s\}$ and $F=\{f_1,\ldots,f_t\}$ be two sets of non-zero orbit representatives. The nearring obtained from N,Φ,E by the Ferrero construction (see `PlanarNearRing`) is isomorphic to the nearring obtained from N,Φ,F iff there exists an automorphism α of $(N,+)$ that normalizes Φ such that $\{\alpha(e_1),\ldots,\alpha(e_s)\}=\{f_1,\ldots,f_t\}$.

The function `OrbitRepresentativesForPlanarNearRing` returns precisely one set of representatives of cardinality *i* for each isomorphism class of planar nearrings which can be generated from the Ferrero pair (*G*, *phi*).

```
gap> C7 := CyclicGroup( 7 );;
gap> i := GroupHomomorphismByFunction( C7, C7, x -> x^-1 );;
gap> phi := Group( i );;
gap> reps := OrbitRepresentativesForPlanarNearRing( C7, phi, 2 );
[ [ f1, f1^2 ], [ f1, f1^5 ] ]
gap> n1 := PlanarNearRing( C7, phi, reps[1] );;
gap> n2 := PlanarNearRing( C7, phi, reps[2] );;
gap> IsIsomorphicNearRing( n1, n2 );
false
```

10.5 Weakly divisible nearrings

▶ `WdNearRing(` *G*, *psi*, *phi*, *reps* `)`

Every finite (left) weakly divisible nearring $(N, +, \cdot)$ can be constructed in the following way:

(1) Let ψ be an endomorphism of the group $(N, +)$ such that Ker ψ = Image ψ^{r-1} for some integer $r, r > 0$. (Let $\psi^0 :=$ id.)

(2) Let Φ be an automorphism group of $(N, +)$ such that $\psi\Phi \subseteq \Phi\psi$ and Φ acts fixed-point-free on $N\setminus$ Image ψ. (That is, for each $\varphi \in \Phi$ there exists $\varphi' \in \Phi$ such that $\psi\varphi = \varphi'\psi$ and for all $n \in N\setminus$ Image ψ the equality $n^{\varphi} = n$ implies $\varphi =$ id. Note that our functions operate from the right just like GAP-mappings do.)

(3) Let $E \subseteq N$ be a complete set of orbit representatives for Φ on $N\setminus$ Image ψ, such that for all $e_1, e_2 \in E$, for all $\varphi \in \Phi$ and for all $1 \leq i \leq r-1$ the equality $e_1^{\varphi\psi^i} = e_2^{\psi^i}$ implies $\varphi\psi^i = \psi^i$.

Then for all $n \in N, n \neq 0$, there are $i \geq 0, \varphi \in \Phi$ and $e \in E$ such that $n = e^{\varphi\psi^i}$; furthermore, for fixed n, the endomorphism $\varphi\psi^i$ is independent of the choice of e and φ in the representation of n.

For all $x \in N, e \in E, \varphi \in \Phi$ and $i \geq 0$ define $0 \cdot x := 0$ and

$$e^{\varphi\psi^i} \cdot x := x^{\varphi\psi^i}$$

Then $(N, +, \cdot)$ is a zerosymmetric (left) wd nearring.

`WdNearRing` returns the wd nearring on the group G as defined above by the nilpotent endomorphism *psi*, the automorphism group *phi* and a list of orbit representatives *reps* where the arguments fulfill the conditions (1) to (3).

```
gap> C9 := CyclicGroup( 9 );;
gap> psi := GroupHomomorphismByFunction( C9, C9, x -> x^3 );;
gap> Image( psi );
Group([ f2, <identity> of ... ])
gap> Image( psi ) = Kernel( psi );
true
gap> a := GroupHomomorphismByFunction( C9, C9, x -> x^4 );;
gap> phi := Group( a );;
gap> Size( phi );
3
gap> orbs := Orbits( phi, C9 );
[ [ <identity> of ... ], [ f1, f1*f2, f1*f2^2 ], [ f2 ],
  [ f1^2, f1^2*f2^2, f1^2*f2 ], [ f2^2 ] ]
gap> # choose reps from the orbits outside of Image( psi )
gap> reps := [orbs[2][1], orbs[4][1]];
[ f1, f1^2 ]
gap> n := WdNearRing( C9, psi, phi, reps );
ExplicitMultiplicationNearRing ( <pc group of size 9 with
2 generators> , multiplication )
```

11 Designs

Although the functions described in this section were initially meant to investigate designs generated from nearrings, they can also be applied to other incidence structures. In principal a design is represented as a set of points and a set of blocks, a subset of the powerset of the points, with containment as incidence relation.

11.1 Constructing a design

▸ `DesignFromPointsAndBlocks(` *points*`,` *blocks* `)`

`DesignFromPointsAndBlocks` returns the design with the set of points *points* and the set of blocks *blocks*, a subset of the powerset of *points*.

```
gap> points := [1..7];;
gap> blocks := [[1,2,3],[1,4,5],[1,6,7],[2,4,7],[2,5,6],[3,5,7],
>                                                          [3,4,6]];;
gap> D := DesignFromPointsAndBlocks( points, blocks );
<an incidence structure with 7 points and 7 blocks>
```

▸ `DesignFromIncidenceMat( M )` ;

`DesignFromIncidenceMat` returns the design with incidence matrix M, The rows of M are labelled by the set of points 1 to v, the columns represent the blocks. If the (i,j) entry of the matrix M is 1, then the point i is incident with the j-th block, i.e. the j-th block consists of those points i for which the entry (i,j) of M is 1. All other entries have to be 0.

```
gap> M := [[1,0,1,1],
>          [1,1,0,0],
>          [1,1,1,0]];;
gap> DesignFromIncidenceMat( M );
<an incidence structure with 3 points and 4 blocks>
```

▸ `DesignFromPlanarNearRing(` *N*`,` *type* `)`

`DesignFromPlanarNearRing` returns a design obtained from the planar nearring N following the constructions of James R. Clay [Cla92].

If *type* = `"*"`, `DesignFromPlanarNearRing` returns the design $(N, B^*, \in)$ in the notation of J. R. Clay with the elements of N as set of points and $\{N^* \cdot a + b | a, b \in N, a \neq 0\}$ as set of blocks. Here N^* is the set of elements $x \in N$ satisfying $x \cdot N = N$.

If *type* = `" "` (blank), `DesignFromPlanarNearRing` returns the design $(N, B, \in)$ in the notation of J. R. Clay with the elements of N as set of points and $\{N \cdot a + b | a, b \in N, a \neq 0\}$ as set of blocks.

```
gap> n := LibraryNearRing( GTW9_2, 90 );
LibraryNearRing(9/2, 90)
gap> IsPlanarNearRing( n );
true
gap> D1 := DesignFromPlanarNearRing( n, "*" );
<a 2 - ( 9, 4, 3 ) nearring generated design>
gap> D2 := DesignFromPlanarNearRing( n, " " );
<a 2 - ( 9, 5, 5 ) nearring generated design>
```

▶ `DesignFromFerreroPair(` *G*, *phi*, *type* `)`

`DesignFromFerreroPair` returns a design obtained from the group G, and a group of fixed-point-free automorphisms *phi* acting on G following the constructions of James R. Clay [Cla92].

If *type* = `"*"`, `DesignFromFerreroPair` returns the design $(G, B^*, \in)$ in the notation of J. R. Clay with the elements of G as set of points and the nonzero orbits of G under *phi* and their translates by group-elements as set of blocks.

If *type* = `" "` (blank), `DesignFromFerreroPair` returns the design $(G, B, \in)$ in the notation of J. R. Clay with the elements of G as set of points and the nonzero orbits of G under *phi* joined with the zero of G and their translates by group-elements as set of blocks.

```
gap> aux := FpfAutomorphismGroupsCyclic( [3,3], 4 );
[ [ [ f1, f2 ] -> [ f1*f2, f1*f2^2 ] ],
  <pc group of size 9 with 2 generators> ]
gap> f := aux[1][1];
[ f1, f2 ] -> [ f1*f2, f1*f2^2 ]
gap> phi := Group( f );
<group with 1 generators>
gap> G := aux[2];
<pc group of size 9 with 2 generators>
gap> D3 := DesignFromFerreroPair( G, phi, "*" );
<a 2 - ( 9, 4, 3 ) nearring generated design>
gap> # D3 is actually isomorphic to D1
```

▶ `DesignFromWdNearRing(` *N* `)`

`DesignFromWdNearring` returns a design obtained from the weakly divisible nearring N with cyclic additive group of prime power order. Following the constructions of A. Benini, F. Morini and S. Pellegrini, we take the elements of N as set of points and $\{N \cdot a + b | a \in C, b \in N\}$ as set of blocks. Here C is the set of elements $a \in N$ such that $a \cdot N = N$.

```
gap> n := LibraryNearRing( GTW9_1, 202 );
LibraryNearRing(9/1, 202)
gap> IsWdNearRing( n );
true
gap> DesignFromWdNearRing( n );
<a 1 - ( 9, 5, 10 ) nearring generated design>
```

11.2 Properties of a design

▶ `PointsOfDesign(` *D* `)`

`PointsOfDesign` returns the actual list of points of the design D, not their positions, no matter how the points of the design D may be represented. To get the representation of those points, whose positions in the list of all points are given by the list *pointnrs*, one can use `PointsOfDesign(` *D* `){`*pointnrs*`}`.

```
gap> D1;
<a 2 - ( 9, 4, 3 ) nearring generated design>
gap> PointsOfDesign( D1 );
[ (()), ((4,5,6)), ((4,6,5)), ((1,2,3)), ((1,2,3)(4,5,6)),
  ((1,2,3)(4,6,5)), ((1,3,2)), ((1,3,2)(4,5,6)), ((1,3,2)(4,6,5)) ]
gap> PointsOfDesign( D1 ){[2,4]};
[ ((4,5,6)), ((1,2,3)) ]
gap> # returns the points in position 2 and 4
```

▶ `BlocksOfDesign(` *D* `)`

`BlocksOfDesign` returns the actual list of blocks of the design D, not their positions. Blocks are represented as lists of points. A point is incident with a block if the point is an element of the block. To get the representation of those blocks, whose positions in the list of all blocks are given by the list *blocknrs*, one can use `BlocksOfDesign(` *D* `){`*blocknrs*`}`.

```
gap> Length( BlocksOfDesign( D1 ) );
18
gap> BlocksOfDesign( D1 ){[3]};
[ [ ((4,6,5)), (()), ((1,2,3)(4,5,6)), ((1,3,2)(4,5,6)) ] ]
gap> # returns the block in position 3 as a list of points
```

▶ `DesignParameter(` *D* `)`

`DesignParameter` returns the set of paramaters t, v, b, r, k, λ of the design D. Here v is the size of the set of points `PointsOfDesign`, b is the size of the set of blocks `PointsOfDesign`, every point is incident with precisely r blocks, every block is incident with precisely k points, every t distinct points are together incident with precisely λ blocks.

```
gap> DesignParameter( D1 );
[ 2, 9, 18, 8, 4, 3 ]
gap> # t = 2, v = 9, b = 18, r = 8, k = 4, lambda = 3
```

▶ `IncidenceMat(` *D* `)`

`IncidenceMat` returns the incidence matrix of the design D, where the rows are labelled by the positions of the points in `PointsOfDesign`, the columns are labelled by the positions of the blocks in `BlocksOfDesign`. If the point in position i is incident with the block in position j, then the (i, j) entry of the matrix `IncidenceMat` is 1, else it is 0.

```
gap> M1 := IncidenceMat( D1 );
[ [ 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1 ],
  [ 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1 ],
  [ 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0 ],
  [ 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1 ],
  [ 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1 ],
  [ 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0 ],
  [ 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0 ],
  [ 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0 ],
  [ 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0 ] ]
```

▶ `PrintIncidenceMat(` *D* `)`

`PrintIncidenceMat` prints only the entries of the incidence matrix `IncidenceMat` of the design without commas. If the point in position i is incident with the block in position j, then there is 1 in the i-th row, j-th column, else there is '.', a dot.

```
gap> PrintIncidenceMat( D1 );
..1.1.1..1.11..1.1
1...1..11..1.11..1
1.1....1.11..1.11.
1..1.1..1.1.1..1.1
.11..11...1..11..1
.1.11.1.1....1.11.
1..1.11..1.1..1.1.
.11..1.11..11...1.
.1.11..1.11.1.1...
```

▶ `BlockIntersectionNumbers(` *D* `)`
▶ `BlockIntersectionNumbersK(` *D*`,` *blocknr* `)`

In the first form `BlockIntersectionNumbers` returns the list of cardinalities of the intersection of each block with all other blocks of the design *D*. In the second form `BlockIntersectionNumbers` returns the list of cardinalities of the intersection of the block in position *blocknr* with all other blocks of the design *D*.

```
gap> BlockIntersectionNumbers( D1, 2 );
[ 0, 4, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, 2, 2, 1, 2, 1 ]
gap> # the second has empty intersection with the first block
gap> # and intersects all others in at most 2 points
```

▶ `IsCircularDesign(` *D* `)`

`IsCircularDesign` returns `true` if the design *D* is circular and `false` otherwise. The design *D* has to be the result of `DesignFromPlanarNearRing` or `DesignFromFerreroPair`, since `IsCircularDesign` assumes the particular structure of such a nearring-generated design.

A design *D* is *circular* if every two distinct blocks intersect in at most two points.

`IsCircularDesign` calls the function `BlockIntersectionNumbers`.

```
gap> IsCircularDesign( D1 );
true
```

11.3 Working with the points and blocks of a design

▶ `IsPointIncidentBlock(` *D*`,` *pointnr*`,` *blocknr* `)`

`IsPointIncidentBlock` returns `true` if the point whose position in the list `PointsOfDesign(` *D* `)` is given by *pointnr* is incident with the block whose position in the list `BlocksOfDesign(` *D* `)` is given by *blocknr*, that is, the point is contained in the block as an element in a set.

```
gap> IsPointIncidentBlock( D1, 3, 1 );
true
gap> # point 3 is incident with block 1
gap> IsPointIncidentBlock( D1, 3, 2 );
false
```

▶ `PointsIncidentBlocks(` *D*`,` *blocknrs* `)`

`PointsIncidentBlocks` returns a list of positions of those points of the design *D* which are incident with the blocks, whose positions are given in the list *blocknrs*.

```
gap> PointsIncidentBlocks( D1, [1, 4] );
[ 4, 7 ]
gap> # block 1 and block 4 are together incident with
gap> # points 4 and 7
```

▶ `BlocksIncidentPoints(` *D*`,` *pointnrs* `)`

`BlocksIncidentPoints` returns a list of positions of the blocks of the design *D* which are incident with those points, whose positions are given in the list *pointnrs*.

```
gap> BlocksIncidentPoints( D1, [2, 7] );
[ 1, 12, 15 ]
gap> # point 2 and point 7 are together incident with
gap> # blocks 1, 12, 15
gap> BlocksOfDesign( D1 ){last};
[ [ ((4,5,6)), ((4,6,5)), ((1,2,3)), ((1,3,2)) ],
  [ ((1,3,2)), ((1,3,2)(4,5,6)), (()), ((4,5,6)) ],
  [ ((1,3,2)(4,6,5)), ((1,3,2)), ((4,5,6)), ((1,2,3)(4,5,6)) ] ]
gap> # the actual point sets of blocks 1, 12, and 15
gap> BlocksIncidentPoints( D1, [2, 3, 7] );
[ 1 ]
gap> # points 2, 3, 7 are together incident with block 1
gap> PointsIncidentBlocks( D1, [1] );
[ 2, 3, 4, 7 ]
gap> # block 1 is incident with points 2, 3, 4, 7
```

A Bibliography

[AEN00] E. Aichinger, J. Ecker, and C. Nöbauer. The use of computers in near-ring theory. In *Nearrings And Nearfields (Stellenbosch, 1997)*, pages 35–41. Kluwer Academic Publishing, Dordrecht, the Netherlands, 2000.

[Aic00] E. Aichinger. F. Binder, J. Ecker, P. Mayr, and C. Nöbauer. Algorithms for near rings of non-linear transformations. In *Proceedings of the ISSAC 2000*, pages 23–29, St. Andrews, Scotland, 2000.

[Cla92] James R. Clay. *Nearrings. Geneses and Applications.* Oxford University Press, Oxford, New York, Tokyo, 1992.

[Eck98] Jürgen Ecker. On the number of polynomial functions on nilpotent groups of class 2. In *Contributions to general algebra, 10 (Klagenfurt, 1997)*, pages 133–137. Heyn, Klagenfurt, 1998.

[May00] P. Mayr. Fixed-point-free representations over fields of prime characteristic. Reports of the Mathematical Institutes 554, Johannes Kepler University Linz, 2000.

[Mel85] J. D. P. Meldrum. *Near-rings and their links with groups.* Pitman (Advanced Publishing Program), Boston, Mass., 1985.

[Pil83] Günter Pilz. *Near-Rings. The Theory and its Applications*, volume 23 of *North-Holland Mathematics Studies.* North-Holland Publishing Company, Amsterdam, New York, Oxford, revised edition, 1983.

[Sco69] S. D. Scott. The arithmetic of polynomial maps over a group and the structure of certain permutational polynomial groups. I. *Monatsh. Math.*, 73:250–267, 1969.

[TW80] A. D. Thomas and G. V. Wood. *Group tables.* Shiva Publishing Ltd., Nantwich, 1980.

[Wäh87] Heinz Wähling. *Theorie der Fastkörper.* Thales Verlag, Essen, 1987.

[Wol67] Joseph Albert Wolf. *Spaces of Constant Curvature.* McGraw-Hill, New York, 1967.