RDS

A GAP4 Package for Relative Difference Sets

Version 0.9beta21

by

Marc Röder

Fachbereich Mathematik, TU Kaiserslautern

roeder@mathematik.uni-kl.de
marc_roeder@web.de

November 2006

Contents

1 About this package

The RDS package is meant to help with complete searches for relative difference sets in non-abelian groups. Of course, it also works for abelian groups, but no special features are implemented for this case. In particular, there is no support for multipliers.

Furthermore, the focus is on difference sets defining projective planes. So RDS contains some methods for analyzing projective planes but has no support for other designs. Nonetheless other designs may be constructed if they can be described in terms of difference sets.

RDS has no undocumented functions. While this is generally regarded as a feature, it leads to a quite long manual and a lot of documentation not needed for everyday work. So each chapter has a short introduction helping you to distinguish the “interesting” features from the “uninteresting” ones.

For a quick overview, see chapter 2.

The package is entirely written in GAP. So it should run on every computer that runs GAP.

1.1 Installation

Just copy the archive to the directory where the other packages live . For example, `gap/pkg/` in your home directory. Or use the `pkg/` directory in one of the paths from the GAPvariable `GAP_ROOT`.

Then call GAPand type

```
gap> LoadPackage("rds");
----------------------------------------------------------------
Loading  RDS 0.9beta5
by Marc Roeder
For help, type: ?RDS
----------------------------------------------------------------
true
```

For a test, see the example in chapter 2.

2 A quick start

This chapter shows a quick example of how to use RDS. Some of the functions used here make choices which might not be optimal. So if you plan to do more involved computations, you should also see the other chapters to learn about the concepts behind these high-level functions.

Here we will construct relative difference sets of Dembowski-Piper type "b" and order 9. We will take the elementary abelian group as an example. The general idea is as follows: Find a "nice" normal subgroup U and generate relative difference sets coset by coset. The normal subgroup has to be chosen such that we know how many elements to choose from each coset modulo U.

The calculations here are very easy, a more demanding example can be found in chapter 5.

2.1 First Step: Integers instead of group elements

Difference sets are represented by lists of integers. Every difference set is assumed to contain 1. This is assumed implicitly. So the lists representing difference sets **must not** contain 1 (a partial difference set of length n is hence represented by a list of length $n-1$). If a partial difference set contains 1, many functions will produce errors.

To find Difference sets in a group, say G, begin with generating the group (and forbidden subgroup) and defining the parameters. Like this:

```
gap> LoadPackage("rds");
----------------------------------------------------------------
Loading  RDS 0.9beta5
by Marc Roeder
For help, type: ?RDS
----------------------------------------------------------------
true
gap> k:=9;;lambda:=1;;groupOrder:=81;;
gap> forbiddenGroupOrder:=9;;
gap> G:=ElementaryAbelianGroup(groupOrder);
<pc group of size 81 with 4 generators>
gap> Gdata:=PermutationRepForDiffsetCalculations(G);;
gap> N:=Group(GeneratorsOfGroup(G){[1,2]});
<pc group with 2 generators>
gap> Size(N)=forbiddenGroupOrder;   #just a test...
true
```

Once we have calculated $Gdata$, this will be used very often to represent the group G as it contains much more information.

2.2 Signatures: An important tool

The "signature" of a subset $S \subseteq G$ of a group relative to a normal subgroup U is the multiset of numbers of elements S contains from each coset modulo U. Possible values of these numbers can be calculated a priori for relative difference sets.

```
gap> sigdat:=SignatureData(Gdata,N,k,lambda,10^5);;
```

The argument 10^5 depends on your degree of impatience. Larger numbers take more time in this step, but give better results for later reduction steps.

Now we will look for a "nice" normal subgroup. A normal subgroup is "nice", if it has only few signatures and the number of different entries in each signature is low. If you have different choices here do some experiments, to see what works. Let's see what we have:

```
gap> NormalSgsHavingAtMostNSigs(sigdat,1,[1..7]);
[ rec( sigs := [ [ 3, 3, 3 ] ], subgroup := Group([ f1, f2, f3 ]) ),
  rec( sigs := [ [ 3, 3, 3 ] ], subgroup := Group([ f1, f2, f4 ]) ),
  rec( sigs := [ [ 3, 3, 3 ] ], subgroup := Group([ f1, f2, f3*f4 ]) ),
  rec( sigs := [ [ 3, 3, 3 ] ], subgroup := Group([ f1, f2, f3*f4^2 ]) ) ]
```

The second parameter of `NormalSgsHavingAtMostNSigs` is the maximal number of signatures the subgroup may have. The second parameter gives the desired lengths of the signatures (the index of the normal subgroup).

So in this example we have no real choice. Let's take the first group for U. The signature means that we have to get 3 elements from each coset modulo U. So we generate startsets of length 2 in the trivial coset U (representing partial relative difference sets of length 3). The function `StartsetsInCoset` generates startsets in U by generating an initial set of startsets and then raising the length of each startset by 1. Then a reduction using signatures and automorphism is performed. This is done until all startsets have the desired length or no startset remains (in which case there is no relative difference set). For the reduction, a suitable set of automorphisms must be chosen. This is done by the function `SuitableAutomorphismsForReduction`:

```
gap> U:=last[1].subgroup;
Group([ f1, f2, f3 ])
gap> auts:=SuitableAutomorphismsForReduction(Gdata,U);
[ <permutation group of size 303264 with 8 generators> ]
gap> startsets:=StartsetsInCoset([],U,N,2,auts,sigdat,Gdata,lambda);
#I  Size 18
#I  1/ 0 @ 0:00:00.328
#I  Size 8
#I  1/ 0 @ 0:00:00.180
[ [ 4, 22 ] ]
```

For larger examples, this takes a wile. Taking 10^6 (or even more) for the generation of *sigdat* can save some time here. A few remarks about the parameters of `StartsetsInCoset`. The first parameter `[]` is the set of startsets which we start with (as we just started, this is empty). The second parameter is the coset we use to generate startsets and third parameter is the forbidden subgroup. The fourth parameter is the length of the startsets we want to generate (remember that 1 is assumed to be in every startset without being listed. So we want startsets of size 3 represented by lists of length 2. Hence the 2 in this place). Instead of *auts* a suitable list of groups of automorphisms of G in permutation representation may be inserted. These are used for the reduction of startsets. For large groups *auts[1]* it is a good idea to add some subgroups of *auts[1]* to the list (ascending in order) *auts*, as the reduction is done using the first group in the list and then reducing the already reduced list again using the next group.

2.3 Change of coset vs. brute force

Now we have startsets of length 2 in U and there are two possibilities:

(1) Find 3 more elements from another coset like this:

```
gap> cosets:=RightCosets(G,U);
[ RightCoset(Group( [ f1, f2, f3 ] ),<identity> of ...),
  RightCoset(Group( [ f1, f2, f3 ] ),f4),
  RightCoset(Group( [ f1, f2, f3 ] ),f4^2) ]
gap> startsets:=StartsetsInCoset(startsets,cosets[2],N,5,auts,sigdat,Gdata,lambda);
#I  Size 27
#I  1/ 0 @ 0:00:00.632
#I  Size 11
#I  1/ 0 @ 0:00:00.260
#I  Size 12
#I  2/ 0 @ 0:00:00.340
[ [ 4, 22, 5, 48, 59 ], [ 4, 22, 5, 59, 61 ] ]
```

And 3 more from the last one (of course, we could also change to force, but it seems to work this way. . .).

```
gap> startsets:=StartsetsInCoset(startsets,cosets[3],N,8,auts,sigdat,Gdata,lambda);
#I  Size 9
#I  1/ 0 @ 0:00:00.300
#I  Size 1
#I  1/ 1 @ 0:00:00.024
#I  Size 1
#I  1/ 1 @ 0:00:00.028
[ [ 4, 22, 5, 48, 59, 29, 72, 78 ] ]
```

So we found one difference set of order 9 in the elementary abelian group of order 81. To get the difference set containing 1 explicitly and as a subset of G, say

```
gap> PermList2GroupList(Concatenation(startsets[1],[1]),Gdata);
[ f3, f1*f3^2, f4, f2*f3^2*f4, f1*f2^2*f3*f4, f2*f4^2, f1^2*f3^2*f4^2,
f1^2*f2^2*f3*f4^2, <identity> of ... ]
```

(2) Do a brute force search. Here we have to convert the forbidden group N into a list of integers Np. And we have to raise the length of the startsets by one before we can start. This is due to the ordering we chose (which is not necessarily compatible with the cosets modulo U).

```
gap> Np:=GroupList2PermList(Set(N),Gdata);
[ 1, 2, 3, 6, 7, 10, 16, 19, 32 ]
gap> startsets:=ExtendedStartsetsNoSort(startsets,[1..groupOrder],Np,8,Gdata,lambda);;
gap> Size(startsets);
54
gap> foundsets:=[];;
gap> for set in startsets
>  do
>   Append(foundsets,AllDiffsets(set,[1..groupOrder],k-1,Np,Gdata,lambda));
> od;
gap> Size(foundsets);
162
```

Now *foundsets* contains 162 relative $(9, 9, 9, 1)$-difference sets (represented by lists of length 8).

3 General concepts

In this chapter, we first give a definition of relative difference sets and outline a part of the theory. Then we have a quick look at the way difference sets are represented in RDS.

After that, some basic methods for the generation of difference sets are explained.

If you already read chapter 2 and want to know what `StartsetsInCoset` really does, you may want to read this chapter (and the following one, of course). The main high-level function in this chapter is `ExtendedStartsets`.

3.1 Introduction

Let G be a finite group and $N \subseteq G$. The set $R \subseteq G$ with $|R| = k$ is called a "relative difference set of order $k - \lambda$ relative to the forbidden set N" if the following properties hold:

(a) The multiset $\{a \cdot b^{-1} \colon a, b \in R\}$ contains every nontrivial ($\neq 1$) element of $G - N$ exactly λ times.

(b) $\{a \cdot b^{-1} \colon a, b \in R\}$ does not contain any element of N.

Relative difference sets with $N = 1$ are called (ordinary) difference sets. As a special case, difference sets with $N = 1$ and $\lambda = 1$ correspond to projective planes of order $k - 1$. Here the blocks are the translates of R and the points are the elements of G.

In group ring notation a relative difference set satisfies

$$RR^{-1} = k + \lambda(G - N)\cdot$$

The set $D \subseteq G$ is called **partial relative difference set** with forbidden set N, if

$$DD^{-1} = \kappa + \sum_{g \in G - N} v_g g$$

holds for some $1 \leq \kappa \leq k$ and $0 \leq v_g \leq \lambda$ for all $g \in G - N$. If D is a relative difference set then ,obviously, D is also a partial relative difference set.

Two relative difference sets $D, D' \subseteq G$ are called **strongly equivalent** if they have the same forbidden set $N \subseteq G$ and if there is $g \in G$ and an automorphism α of G such that $D'g^{-1} = D^{\alpha}$. The same term is applied to partial relative difference sets.

Let $D \subseteq G$ be a difference set, then the incidence structure with points G and blocks $\{Dg \mid g \in G\}$ is called the **development** of D. In short: $\mathrm{dev}D$. Obviously, G acts on $\mathrm{dev}D$ by multiplication from the right.

If D is a difference set, then D^{-1} is also a difference set. And $\mathrm{dev}D^{-1}$ is the dual of $\mathrm{dev}D$. So a group admitting an operation some structure defined by a difference set does also admit an operation on the dual structure. We may therefore change the notion of equivalence and take ϕ to be an automorphism or an anti-automorphism. Forbidden sets are closed under inversion, so this gives a "weak" sort of strong equivalence.

3.2 How partial difference sets are represented

Let G be a group. We define an enumeration $\{g_1, \ldots, g_n\} = G$ and represent $D \subseteq G$ as a list of integers (where ,of course, i represents g_i for all $1 \leq i \leq n$). So the automorphism group of G is represented as a permutation group of degree n. One of the operations performed most often by methods in RDS is the calculation of quotients in G. So we calculate a look-up table for this.

This pre-calculation is done by the operation `PermutationRepForDiffsetCalcuations`. So before you start generating difference set, call this function and work with the data structure returned by it.

For an exhaustive search, the ordering of G is very important. To avoid generating duplicate partial difference sets, we would like to represent partial difference sets by **sets**, i.e. ordered lists. But in fact, RDS does **not** assume that partial difference sets are sets. The operations `ExtendedStartSets` and `AllDiffsets` assume that the last element of partial difference set is its maximum. But they don't test it. So if you start from scratch, these methods generate difference sets which are really sets. Whereas the `NoSort` versions disregard the ordering of G and will produce duplicates.

The reason for this seemingly strange behaviour is the following: Assume that we have a normal subgroup $U \leq G$ and know that every difference set $D \subseteq G$ contains exactly n_i elements from the i^{th} coset modulo U. Then it is natural to generate difference sets by first searching all partial difference sets of length n_1 containing entirely of elements of the first coset modulo U and then proceed with the other cosets.

This method of difference set generation is normally not compatible with the ordering of G. This is why partial difference sets are not required to be **sets**. See chapter 5 for an example.

3.3 Basic functions for startset generation

Defining an enumeration of the a group G, every relative difference set may be represented by a list of integers. Indexing G in this way has the advantage of the automorphism group of G being a permutation group. As relative difference sets are normally calculated in small groups, it is possible to store a complete multiplication table of the group in terms of the enumeration.

If not stated otherwise, partial difference sets are always considered to be lists of integers. Note that it is not required for a partial difference set to be a set.

1 ► `PermutationRepForDiffsetCalculations(` *group* `)` O
► `PermutationRepForDiffsetCalculations(` *group*`,` *autgrp* `)` O

For a group *group*, `PermutationRepForDiffsetCalculations(`*group*`)` returns a record containing:

1. the group *.G*=*group*.
2. the sorted list *.Glist*=`Set(`*group*`)`,
3. the automorphism group *.A* of *group*,
4. the group *.Aac*, which is the permutation action of A on the indices of *.Glist*,
5. *.Ahom*=`ActionHomomorphism(`*.A*`,`*.Glist*`)`,
6. the group *.Ai* of anti-automorphisms of *.group* acting on the indices of *Glist*,
7. the multiplication table *.diffTable* of *.group* in a special form.

.diffTable is a matrix of integers defined such that *.difftable*`[i][j]` is the position of *Glist*`[i](`*Glist*`[j])^-1)` in *Glist* with *Glist*`[1]=One(`*group*`)`.

`PermutationRepForDiffsetCalculations` runs into an error if `Set(`*group*`)[1]` is not equal to `One(`*group*`)`.

If *autgrp* is given, `PermutationRepForDiffsetCalculations` will not calculate the automorphism group of *group* but will take *autgrp* instead without any test.

If 'Set(*group*)[1]' is not equal to `One(`*group*`)`, then `PermutationRepForDiffsetCalculations` returns an error message stating "Unable to generate *Glist*". In this case, calculating a representation helps:

```
gap> G:=DirectProduct(SL(2,3),CyclicGroup(3));
<group of size 72 with 3 generators>
gap> data:=PermutationRepForDiffsetCalculations(G);
Error, Unable to generate <Glist>
 called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> phi:=ActionHomomorphism(G,Set(G),OnRight);
<action homomorphism>
gap> Gnew:=ImagesSource(phi);
<permutation group with 3 generators>
gap> data:=PermutationRepForDiffsetCalculations(Gnew);
```

A partial difference set may be converted from a list of group elements to a list of integers using

2 ► `GroupList2PermList(` *list*, *dat* `)` O

where *dat* is a record containing *.diffTable* as returned by `PermutationRepForDiffsetCalculations`. The inverse operation is performed by

3 ► `PermList2GroupList(` *list*, *dat* `)` O

```
gap>  G:=DihedralGroup(6);
<pc group of size 6 with 2 generators>
gap> N:=NormalSubgroups(G)[2];
Group([ f2 ])
gap> dat:=PermutationRepForDiffsetCalculations(G);
rec( G := <pc group of size 6 with 2 generators>,
  Glist := [ <identity> of ..., f1, f2, f1*f2, f2^2, f1*f2^2 ],
  A := <group of size 6 with 2 generators>,
  Aac := Group([ (3,5)(4,6), (2,4,6) ]),
  Ahom := <action homomorphism>,
  Ai := Group([ (3,5), (3,5)(4,6), (2,4,6) ]),
  diffTable := [ [ 1, 2, 5, 4, 3, 6 ], [ 2, 1, 6, 3, 4, 5 ],
      [ 3, 6, 1, 2, 5, 4 ], [ 4, 5, 2, 1, 6, 3 ],
      [ 5, 4, 3, 6, 1, 2 ], [ 6, 3, 4, 5, 2, 1 ] ] )
gap> Nperm:=GroupList2PermList(Set(N),dat);
[ 1, 3, 5 ]
```

In the following functions the record *dat* has to contain a matrix *.diffTable* as returned by `Permutation-RepForDiffsetCalculations`.

4 ► `NewPresentables(` *list*, *newel*, *table* `)` O
► `NewPresentables(` *list*, *newel*, *dat* `)` O
► `NewPresentables(` *list*, *newlist*, *dat* `)` O
► `NewPresentables(` *list*, *newlist*, *table* `)` O

`NewPresentables(` *list*,*newel*,*dat* `)` takes a record *dat* as returned by `PermutationRepForDiffsetCal-culations(`*group*`)`. For `NewPresentables(` *list*,*newel*,*table* `)`, *table* has to be the multiplication table in the form of `NewPresentables(` *list*,*newel*,*dat.diffTable*`)`

The method returns the unordered list of quotients $d_1 newel^{-1}$ with $d_1 \in list \cup \{1\}$ (in permutation representation).

When used with a list *newlist*, a list of quotients $d_1 d_2^{-1}$ with $d_1 \in list \cup \{1\}$ and $d_2 \in newlist$ is returned.

5 ▶ `AllPresentables(` *list*`,` *table* `)` O
▶ `AllPresentables(` *list*`,` *dat* `)` O

Let *list* be a list of integers representing elements of a group defined by *dat* (or *table*). `AllPresentables(`*list*`,`*table*`)` returns an unordered list of quotients ab^{-1} for all group elements a, b represented by integers in *list*. If $1 \in list$, an error is issued. The multiplication table *table* has to be of the form as returned by `PermutationRepForDiffsetCalculations`. And *dat* is a record as calculated by `PermutationRepForDiffsetCalculations`.

```
gap> G:=CyclicGroup(7);;dat:=PermutationRepForDiffsetCalculations(G);;
gap> AllPresentables([2,3],dat);
[ 2, 3, 7, 2, 7, 6 ]
gap> AllPresentables([1,2,3],dat);
Error...
```

6 ▶ `RemainingCompletions(` *diffset*`,` *completions*`[,` *forbidden*`],` *dat*`[,` *lambda*`] )` O
▶ `RemainingCompletionsNoSort(` *diffset*`,` *completions*`[,` *forbidden*`],` *table*`[,` *lambda*`] )` O

For a partial difference set *diffset*, `RemainingCompletions(`*diffset*`,`*completions*`,`*dat*`)` returns a subset of the **set** *completions*, such that each of its elements may be added to *diffset* without it loosing the property to be a partial difference set. Only elements greater than the last element of *diffset* are returned.

For partial **relative** difference sets, *forbidden* is the forbidden set.

`RemainingCompletionsNoSort` does also return elements from *completions* which are smaller than *diffset*`[Size(`*diffset*`)]`.

```
gap> G:=CyclicGroup(7);
<pc group of size 7 with 1 generators>
gap> dat:=PermutationRepForDiffsetCalculations(G);;
gap> RemainingCompletionsNoSort([4],[1..7],dat);
[ 2, 3 ]
gap> RemainingCompletionsNoSort([4],[1..7],dat,2);
[ 2, 3, 6, 7 ]
gap> RemainingCompletions([4],[1..7],dat);
[  ]
gap> RemainingCompletions([4],[1..7],dat,2);
[ 6, 7 ]
```

7 ▶ `ExtendedStartsets(` *startsets*`,` *completions*`, [`*forbiddenset*`][,` *aim*`],` *Gdata*`[,` *lambda*`] )` O
▶ `ExtendedStartsetsNoSort(` *startsets*`,` *completions*`, [`*forbiddenset*`][,` *aim*`],` *Gdata*`[,` *lambda*`] )` O

For a set of partial (relative) difference sets *startsets*, the set of all extensions by one element from *completions* is returned. Here an “extension” of a partial diffence set S is a list which has one element more than S and contains S.

Here *completions* is a set of elements wich may be appended to the lists in *startsets* to generate new partial difference sets. For relative difference sets, the forbidden set *forbiddenset* must be given. And the integer *aim* gives the desired total length, i.e. the number of elements of *completions* that have to be added to each startset plus its length. Note that the elements of *startset* are always extended by **one** element (if they can be extended). *aim* does only tell how many elements from *completions* you want to add. A partial difference set is only be extended, if there are enough admissible elements in *completions*, so if for some $S \in startsets$, we have less than $aim - \texttt{Size}(S)$ elements in *completions* which can be added to S, no extension of S is returned.

If *lambda* is not passed as a parameter, it is assumed to be 1.

Note that `ExtendedStartsets` does use `RemainingCompletions` while `ExtendedStartsetsNoSort` uses `RemainingCompletionsNoSort`. Note that the partial difference sets generated with `ExtendedStartsetsNoSort` are **not** sets (i.e. not sorted). This may result in doing work twice. But it can also be useful, especially when generating difference sets "coset by coset".

```
gap> G:=CyclicGroup(7);;dat:=PermutationRepForDiffsetCalculations(G);;
gap> startsets:=[[2],[4],[6]];;
gap> ExtendedStartsets(startsets,[1..7],dat);
[ [ 2, 4 ], [ 2, 6 ] ]
gap> ExtendedStartsets(startsets,[1..7],3,dat);
[ [ 2, 4 ] ]
gap> ExtendedStartsets(startsets,[1..7],dat,2);
[ [ 2, 3 ], [ 2, 4 ], [ 2, 5 ], [ 2, 6 ], [ 4, 6 ], [ 4, 7 ], [ 6, 7 ] ]
gap> ExtendedStartsetsNoSort(startsets,[1..7],dat);
[ [ 2, 4 ], [ 2, 6 ], [ 4, 2 ], [ 4, 3 ], [ 6, 2 ], [ 6, 5 ] ]
```

3.4 A brute force method

The following method can be used to find (partial) difference sets by brute force.

1 ► `AllDiffsets(` *diffset*, *completions*, *aim*, *forbidden*, *Gdata*, *lambda* `)` O

Let *diffset* be partial relative difference set and *completions* a list of possible completions and *forbidden* the forbidden set. Then `AllDiffsets` returns a list of (partial) difference sets which contain diffset. *Gdata* is the record as always and *lambda* is the parameter of the relative difference set. *forbidden* and *completions* have to be lists of integers.

4 Invariants for Difference Sets

This chapter contains an important tool for the generation of difference sets. It is called the "coset signature" and is an invariant for equivalence of partial relative difference sets. For large λ, there is an invariant calculated by `MultiplicityInvariantLargeLambda`. This invariant can be used complementary to the coset signature and is explained in section 6.1.

Most of the methods explained here are not commonly used. If you do not want to know how coset signatures work in detail, you can safely skip a large part of this and go straight to the explanation of `SignatureDataForNormalSubgroups` and `ReducedStartsets`.

The last section (4.2) of this chapter has some functions which allow the user to use coset signatures with even less effort. But be aware that these functions make choices for you that you probably do not want if you do very involved calculations. In particular, the coset signatures are not stored globally and hence cannot be reused. For a demonstration of these easy-to-use functions, see chapter 2

4.1 The Coset Signature

Let $R \subseteq G$ be a (partial) relative difference set (for definition see 3.1) with forbidden set $N \subseteq G$. Let $U \leq G$ be a normal subgroup and $C = \{g_1, \dots, g_{|G:U|}\}$ be a system of representatives of G/U.

The intersection number of R with Ug_i is defined as $v_i = |R \cap Ug_i|$. For every normal subgroup $U \leq G$ the multiset $\{|R \cap Ug_i| \colon g_i \in C\}$ is called "coset signature of R (relative to U)".

Let $D \subseteq G$ be a relative difference set and $\{v_1, \dots, v_{|G:U|}\}$ its coset signature. Then the following equations hold (see [Bru55],[Röd06]):

$$\begin{aligned} \sum v_i &= k \\ \sum v_i^2 &= \lambda(|U| - |U \cap N|) + k \\ \sum_j v_j v_{ij} &= \lambda(|U| - |g_i U \cap N|) \quad \text{for } g_i \notin U \end{aligned}$$

where $v_{ij} = |D \cap g_i g_j U|$. If the forbidden set N is a subgroup of G we have $|g_i U \cap N|$ is either 0 or equal to $|U \cap N|$.

Given a group G, the forbidden set $N \subseteq G$ and some normal subgroup $U \leq G$, the right sides of this equations are known. So we may ask for tuples $(v_1, \dots, v_{|G:U|})$ solving this system of equations. Of course, we index the v_i with the elements of G/U, so the last equation poses conditions to the ordering of the tuple $(v_1, \dots, v_{|G:U|})$.

So we call any multiset $\{v_1, \dots, v_{|G:U|}\}$ solving the above equations an "admissible signature" for U.

1 ► `CosetSignatureOfSet(` *set*, *cosets* `)` F

`CosetSignatureOfSet(` *set*, *cosets*`)` returns the **ordered list** of intersection numbers of *set*. That is, the size of the intersection of *set* with each Element of *cosets*.

Note that it is not tested, if *cosets* is really a list of cosets. `CosetSignatureOfSet(` *set*, *cosets*`)` works for any List *set* and any list of lists *cosets*. So be careful!

```
gap> G:=SymmetricGroup(5);;
gap> A:=AlternatingGroup(5);;
gap> CosetSignatureOfSet([(1,2),(1,5),(1,2,3)],RightCosets(G,A));
[ 1, 2 ]
gap> CosetSignatureOfSet([(1,2),(1,5),(1,2,3)],[A]);
[ 1 ]
gap> CosetSignatureOfSet([(1,2),(1,5),(1,2,3)],[[(1,2),(1,2,3)],[(3,2,1)]]);
[ 0, 2 ]
```

2 ► `CosetSignatures(` *Gsize*, *Usize*, *diffsetorder* `)` O
► `CosetSignatures(` *Gsize*, *Nsize*, *Usize*, *Intersectsizes*, *k*, *lambda* `)` O

`CosetSignatures(` *Gsize*,*Usize*,*diffsetorder*`)` returns all *Gsize*/*Usize* tuples such that the sum of the squares of each tuple equals *Usize*+*diffsetorder*. And the sum of each tuple equals *diffsetorder*+1.

These are necessary conditions for signatures of difference sets and normal subgroups of order *Usize* in groups of order *Gsize* (see 4.1).

`CosetSignatures(` *Gsize*,*Nsize*,*Usize*,*Intersectsizes*,*k*,*lambda*`)` Calculates all multiset meeting some conditions for signatures of relative difference sets and normal subgroups of order *Usize* in groups of order *Gsize* (see 4.1). Here *Nsize* is the size of the forbidden group, *Intersectsizes* is a list of integers determining the size of the intersection of the forbidden set and the normal Subgroup of order *Usize*. The pararmeters *k* and *lambda* are the usual ones for designs. `CosetSignatures` returns a list containing one pair for each entry *i* of *Intersectsizes*. The first entry of this pair is [*Gsize*, *Nsize*, *Usize*, *i*, *k*, *lambda*] and the second one is a list of admissible signatures with these parameters.

```
gap> CosetSignatures(256,16,64,[1,4,8,16],17,1);
[ [ [ 256, 16, 64, 1, 17, 1 ], [  ] ],
  [ [ 256, 16, 64, 4, 17, 1 ], [ [ 3, 4, 4, 6 ] ] ],
  [ [ 256, 16, 64, 8, 17, 1 ], [ [ 4, 4, 4, 5 ] ] ],
  [ [ 256, 16, 64, 16, 17, 1 ], [  ] ] ]
#And for an ordinary difference set of order 16.
gap> CosetSignatures(273,1,39,[1],17,1);
[ [ [ 273, 1, 39, 1, 17, 1 ],
  [ [ 0, 1, 2, 3, 3, 4, 4 ], [ 0, 2, 2, 2, 3, 3, 5 ],
    [ 1, 1, 1, 2, 4, 4, 4 ], [ 1, 1, 1, 3, 3, 3, 5 ],
    [ 1, 1, 2, 2, 2, 4, 5 ] ] ] ]
```

3 ► `TestSignatureLargeIndex(` *sig*, *group*, *Normalsg*`[,` *factorgrp*`] )` O

this does only work for ordinary difference sets, not for relative difference sets in general

`TestSignatureLargeIndex(`*sig*,*group*,*Normalsg*`[,`*factorgrp*`])` tests if *sig* meets some necessary conditions of 4.1 to be a signature for a difference set in *group* for the normal subgroup *Normalsg*. *factorgrp* is the factorgroup *group*/*Normalsg*. The returned value is *true* or *false* resp.

4 ► `TestSignatureCyclicFactorGroup(` *sig*, *Nsize* `)` O

This does only work for ordinary difference sets, not for relative difference sets in general

`TestSignatureCyclicFactorGroup(`*sig*,*Nsize*`)` test if *sig* meets meets some necessary conditions of 4.1 to be a signature for a difference set in some group, which has a normal subgroup of size *Nsize* such that the factor group is cyclic. The returned value is *true* or *false* resp.

5 ► `TestedSignatures(` *sigs*, *group*, *normalsg*`[,` *maxtest*`][,` *moretest*`] )` O

this does only work for ordinary difference sets, not for relative difference sets in general

Let *sigs* be a list of possible signatures as returned from `CosetSignatures`. Let *normalsg* be a subgroup of *group*. For each signature in *sigs*, the necessary conditions described in 4.1 are tested to decide if the signature can be a signature of a difference set in *group* for for the normal subgroup *normalsg*.

As this involves computation for all permutations of the signature, this can be very costly. The argument *maxtest* determines how many permutations are admissible. If *maxtest*=0, all signatures are tested, regardless of how much work is necessary for this. If a signature has too many permutations, it is returned without test. Even though it is not wise, *maxtest*=0 is the default option. If `InfoLevel(InfoRDS)` is at least 2, information about skipped signatures is echoed.

If the boolean value *moretest* is *false* and all signatures in *sigs* but the last one are found to be not admissible, the last one is returned without test. This saves the time to test the last signature, but if chances are that there is no difference set in *group*, this may also give away a chance to find out early (every difference set has signatures, so no admissible signature means that no difference set can exist). Default is *true*.

`TestedSignatures` calls `TestSignatureCyclicFactorGroup` or `TestSignatureLargeIndex` and returns a sublist of *sigs*.

```
G:=SmallGroup(273,2);
gap> N:=First(NormalSubgroups(G),g->Order(g)=39);
Group([ f1, f3 ])
gap> sigs:=CosetSignatures(273,1,39,[1],17,1);
[ [ [ 273, 1, 39, 1, 17, 1 ],
    [ [ 0, 1, 2, 3, 3, 4, 4 ], [ 0, 2, 2, 2, 3, 3, 5 ],
      [ 1, 1, 1, 2, 4, 4, 4 ], [ 1, 1, 1, 3, 3, 3, 5 ],
      [ 1, 1, 2, 2, 2, 4, 5 ] ] ] ]
gap> TestedSignatures(sigs[1][2],G,N);
[ [ 1, 1, 1, 2, 4, 4, 4 ], [ 1, 1, 1, 3, 3, 3, 5 ] ]
```

6 ► `TestedSignaturesRelative(` *sigs*, *fgdata*, `[,` *maxtest*`][,` *moretest*`] )` O

`TestedSignaturesRelative` takes a list *sigs* of lists of integers and returns a those which may be signatures of relative difference sets with forbidden set.

fgdata is a record as returned by `RDSFactorGroupData(`*U*,*N*,*lambda*,*Gdata*`)` If *maxtest* is set, a signature *s* is only tested if `NrPermutationsList(s)` is less than *maxtest* if *maxtest* is set to 0, all signatures are tested this is the default. If *moretest* is tue, a signature is tested even if it is the only one left. This means we do not assume that there must be an admissable signature at all. The default for *moretest* is *true*.

7 ► `SigInvariant(` *prd* , *data* `)` O

Given a partial relative difference set *prd* and a list of records with entries *cosets* and *sigs*. Here *cosets* is a full list of cosets and *sigs* is a list of signatures that may occur for relative difference sets.

For each record *rec* in *data*, the intersection numbers of *prd* with the cosets of *rec.cosets* are computed stored in a set *sig*. If none of the signatures in *rec.sigs* is pointwise greater or equal *sig*, `SigInvariant(` *prd*,*data*`)` `returns` fail'. Otherwise *sig* is added to a list of signatures that is returned.

Note the returned invariant is that of $prd \cup \{1\}$. The output from `SignatureDataForNormalSubgroups` can be used as *data*.

8 ► `RDSFactorGroupData(` *U*, *N*, *lambda*, *Gdata* `)` O

takes the subgroup *U* of *G*, the forbidden set *N* as a subgroup or subset of *G* and the record of data *Gdata* as returned by `PermutationRepForDiffsetCalculations(`*G*`)` and returns a record containing

- .fg the factor group modulo *U*
- .fglist the factor group as a strictly ordered list
- .cosets the cosets modulo *U* as lists of integers

.lambda the parameter *lambda* as passed to the function

.Usize the size of U

.fgaut the automorphism group of *.fg*

.Nfg the image of N in *.fg*

.fgintersect a list of pairs such that the i^{th} entry is the pair consisting of *.fg[i]* and the size of the intersection of *.fg* with *.Nfg* as cosets modulo U.

.intersectshort ist just the second component of *.fgintersect*.

9 ▸ `MatchingFGDataNonGrp(` *fgdatalist*`,` *fgmatchdata* `)` O

Let *fgdatalist* be a list of records and *fgmatchdata* a record with components *.fg*, *.Nfg* and *.fgintersect* as returned by `RDSFactorGroupData`. Then `MatchingFGDataNonGrp` returns the entry of *fgdatalist* that defines the same admissible signatures as *fgmatchdata*. If no such entry exists, `fail` is returned.

The forbidden set N is not assumed to be a group.

10 ▸ `MatchingFGData(` *fgdatalist*`,` *fgmatchdata* `)` O

Let *fgdatalist* be a list of records and *fgmatchdata* a record with components *.fg*, *.Nfg*, *.fgintersect* and *.fgaut* as returned by `RDSFactorGroupData`. Then `MatchingFGDataNonGrp` returns the entry of *fgdatalist* that defines the same admissible signatures as *fgmatchdata*. If no such entry exists, `fail` is returned.

Here the forbidden set N has to be a group.

11 ▸ `SignatureDataForNormalSubgroups(` *Normals*`,` *globalSigData*`,` *forbiddenSet*`,` *Gdata*`,` *parameters* `)` O

Let *Gdata* be a record as returned by `PermutationRepForDiffsetCalculations`. Let *Normals* be a list of normal subgroups of *Gdata.G*, and *forbiddenSet* the forbidden set (as set of group elements or group).

parameters must be a list of length 4 of the form *[k,lambda,maxtest,moretest]* with k the length of the relative difference set to be constructed and *lambda* the parameter as always. *maxtest* and *moretest* are passed to `TestedSignaturesRelative` and must be set.

`SignatureDataForNormalSubgroups` returns a list containing one record for each group U in *Normals*. This record contains:

1. the subgroup U named *.subgroup*
2. the signatures *.sigs* for U
3. the cosets *.cosets* modulo U as lists of integers

Moreover, the list *globalSigData* is used to store global information which can be reused with other groups. The i^{th} entry of *globalSigData* is a list of records that contains all known information about subgroups of order i. Each of these records has the following components:

1. *.cspara* the parameters for `CosetSignatures`
2. *.sigs* the output of `CosetSignatures` when the input is *.cspara*
3. *.fgsigs* a list of records containing data about factor groups with parameters *.cspara*:

3.1. *.fg* the factor group

3.2. *.fgaut* the automorphism group of *.fg*

3.3. *.Nfg* the image of the forbidden set N under the natural epimorphism to *.fg*

3.4. *.fgintersect* the pairs $[g, |g \cap N|]$ for all g in *.fg*. Here N is the forbidden set.

3.5. *.sigs* the known admissible signatures (this is a subset of the set in number 2. of course)

The list *globalSigData* can be used if different groups are studied. If a group has a normal subgroup with parameters (in the sense of *.cspara*) listed in *globalSigData*, the signatures from a previous calculation may be used. Of course, the factor groups have to be checked first. This check is done with `MatchingFGData` or `MatchingFGDataNonGrp`.

So the second run of `SignatureDataForNormalSubgroups` with the same parameters and different *Gdata* and *Normals* will normally be much faster, as the signatures are already stored in *globalSigData*. Note that *maxtest* and *moretest* are not stored. So a second run with larger *maxtest* will not result in a recalculation of signatures.

12 ► `ReducedStartsets(` *startsets*`,` *autlist*`,` *csdata*`,` *Gdata* `)` O
► `ReducedStartsets(` *startsets*`,` *autlist*`,` *func*`,` *Gdata* `)` O

Let *startsets* be a set of partial relative difference sets, *autlist* a list of permutation groups and *Gdata* record returned by `PermutationRepForDiffsetCalculations`. Then `ReducedStartsets` partitions the list *startsets* according to the values of the function *func* and performs a test for equivalence on the elements of the partition. The list returned is a sublist of *startsets* of pairwise non-equivalent partial relative difference sets if *func* is an invariant for partial relative difference sets. All elements for which *func* returns `fail` are discarded.

Let *csdata* be a list of records as used for `SigInvariant` (i.e. containing *.cosets* and *.signatures*). Then `ReducedStartsets(`*startsets*`,`*autlist*`,`*csdata*`,`*Gdata*`)` `SigInvariant` is used for *func*.

13 ► `maxAutsizeForOrbitCalculation` V

In `ReducedStartsets`, a bound is needed to decide if `Orbit` or `RepresentativeAction` should be used. If the group is larger than *maxAutsizeForOrbitCalculation*, `RepresentativeAction` is used. The default value for `maxAutsizeForOrbitCalculation` is 10^6. If you want to change it, you will have to edit the file `sigs.gd`.

4.2 Blackbox functions

Here are a few functions used in chapter 2. These are meant as black boxes for quick tests. Some of them make choices for you which might not be suitable to the chase you consider, so for serious studies, consider using the more complicated-looking functions above (an example for this comprises chapter 5).

1 ► `SignatureData(` *Gdata*`,` *forbiddenSet*`,` *k*`,` *lambda*`,` *maxtest* `)` O

Let *Gdata* be a record as returned by `PermutationRepForDiffsetCalculations`. Let *forbiddenSet* the forbidden set (as set or group).

k is the length of the relative difference set to be constructed and *lambda* the usual parameter. *maxtest* is the Then `SignatureData` calls `SignatureDataForNormalSubgroups` for normal subgroups of order at least `RootInt(Gdata.G)`. Here *maxtest* is an integer which determines how many permutations of a possible signature are checked to be a sorted signature. Choose a value of at least 10^5. Larger numbers here normaly result in better results when generating difference sets (making reduction more effective).

2 ► `NormalSgsHavingAtMostNSigs(` *sigdata*`,` *n*`,` *lengthlist* `)` F

Let *sigdata* be a list as returned by 'SignatureDataForNormalSubgroups', an integer *n* and a list of integers *lengthlist*. `NormalSgsHavingAtMostKSigs` filters *sigdata* and returns a list of records with components .subgroup and .sigs is returned, such that for every entry .subgroup is a normal subgroup of index in *lengthlist* having at most *n* signatures.

3 ► `SuitableAutomorphismsForReduction(` *Gdata*`,` *normalsg* `)` F

Given a normal subgroup *normalsg* of *Gdata.G*, the function returns a list containing the group of automorphisms of *Gdata.G* which stabilizes all cosets modulo *normalsg*. This group is returned as a group of permutations on *Gdata.Glist* (which is actually the right regular representation). The returned list can be used with `StartsetsInCoset`.

4 ▶ `StartsetsInCoset(` *ssets*`,` *coset*`,` *forbiddenSet*`,` *aim*`,` *autlist*`,` *sigdat*`,` *data*`,` *lambda* `)` F

Assume, we want to generate difference sets "coset by coset" modulo some normal subgroup. Let *ssets* be a (possibly empty) set of startsets, *coset* the coset from which to take the elements to append to the startsets from *ssets*. Furthermore, let *aim* be the size of the generated partial difference sets (that is, the size of the elements from *ssets* plus the number of elements to be added from *coset*). Let *autlist* be a list of groups of automorphisms (in permutation representation) to use with the reduction algorithm. Here the output from `SuitableAutomorphismsForReduction` can be used. And *data* and sigdat are the records as returned by `PermutationRepForDiffsetCalculations` and `SignatureDataForNormalSubgroups` (or `SignatureData`, alternatively). The parameter *lambda* is the usual one for difference sets (the number of ways of expressing elements outside the forbidden set as quotients).

Then `StartsetsInCoset` returns a list of partial difference sets (a list of lists of integers) of length *aim*.

5 An Example Program

Here is a similar example to that in chapter 2. But now we do a little more handwork to see how things work. Now we will calculate the relative difference sets of "Dembowski-Piper type d" and order 16. These difference sets represent projective planes which admit a quasiregular collineation group such that the fixed structure is an anti-flag. See [DP67], [Dem68] or [Röd06] for details.

To have a little more output, you may want to increase `InfoRDS`:

```
gap> SetInfoLevel(InfoRDS,3);
```

First, define some parameters and calculate the data needed:

```
gap> k:=16;;lambda:=1;;groupOrder:=255;; #Diffset parameters
gap> forbiddenGroupOrder:=15;;
gap> maxtest:=10^6;;                     #Bound for sig testing
gap> G:=CyclicGroup(groupOrder);
<pc group of size 255 with 3 generators>
gap> Gdata:=PermutationRepForDiffsetCalculations(G);;
gap> MakeImmutable(Gdata);;
```

Now the forbidden group is calculated in a very ineffective way. Then we calculate admissible signatures. As there are only few normal subgroups in G, we calculate them all. For other groups, one should choose more wisely.

```
gap> N:=First(NormalSubgroups(Gdata.G),i->Size(i)=forbiddenGroupOrder);
Group([ f1*f3^9, f2*f3^10 ])
gap> globalSigData:=[];;
gap> normals:=Filtered(NormalSubgroups(Gdata.G),n->Size(n) in [2..groupOrder-1]);;
gap> sigdat:=SignatureDataForNormalSubgroups(normals,globalSigData,
>                            N,Gdata,[k,lambda,maxtest,true]);;
```

The last step gives better results, if a larger *maxtest* is chosen. But it also takes more time. To find a suitable *maxtest*, the output of `SignatureDataForNormalSubgroups` can be used, if `InfoLevel(InfoRDS)` is at least 2. Note that for recalculating signatures, you will have to reset *globalSigData* to `[]`. Try experimenting with *maxtest* to see the effect of signatures for the generation of startsets.

Now examine the signatures found. Look if there is a normal subgroup which has only one admissible signature (of course, you can also use `NormalSgsHavingAtMostNSigs` here):

```
gap> Set(Filtered(sigdat,s->Size(s.sigs)=1 and Size(s.sigs[1])<=5),i->i.sigs);
[ [ [ 0, 4, 4, 4, 4 ] ], [ [ 4, 4, 8 ] ] ]
```

Great! we'll take the subgroup of index 3. The cosets modulo this group will be used to generate startsets and we assume that the trivial coset contains 8 elements of the difference set. So we generate startsets in U and make a first reduction. For this, we need U and N as lists of integers (recall that difference sets are asumed to be lists of integers). We will call these lists Up and Np. Furthermore, we will have to choose a suitable group of automorphisms for reduction. As G is cyclic, we may take $Gdata \cdot Aac$ here. A good choice is the stabilizer of all cosets modulo U. Yet sometimes larger groups may be possible. For example if we want to generate start sets in U and then do a brute force search. In this case, we may take the stabilizer of U for reduction.

```
gap> U:=First(sigdat,s->s.sigs=[ [ 4, 4, 8 ] ]).subgroup;
Group([ f2, f3 ])
gap> cosets:=RightCosets(G,U);
gap> U1:=cosets[2];;U2:=cosets[3];;
gap> Up:=GroupList2PermList(Set(U),Gdata);;
gap> Np:=GroupList2PermList(Set(N),Gdata);
[ 1, 12, 25, 43, 78, 97, 115, 116, 134, 151, 169, 188, 207, 238, 249 ]
gap> comps:=Difference(Up,Np);;
gap> ssets:=List(comps,i->[i]);;
gap> ssets:=ReducedStartsets(ssets,[Gdata.Aac],sigdat,Gdata.diffTable);
#I  Size 80
#I  2/ 0 @ 0:00:00.061
[ [ 3 ], [ 4 ] ]
```

Observe that 1 is assumed to be element of every difference set and is not recorded explicitly. So the partial difference sets represented by *ssets* at this point are `[ [ 1, 3 ], [ 1, 4 ] ]`. Now the startsets are extended to size 7 using elements of *Up*. The runtime varies depending on the output of `Signature-DataForNormalSubgroups` and hence on *maxtest*.

```
gap> repeat
>     ssets:=ExtendedStartsets(ssets,comps,Np,7,Gdata);
>     ssets:=ReducedStartsets(ssets,[Gdata.Aac],sigdat,Gdata.diffTable);;
> until ssets=[] or Size(ssets[1])=7;
#I  Size 133
#I  3/ 0 @ 0:00:00.133
#I  Size 847
#I  4/ 0 @ 0:00:00.949
#I  Size 6309
#I  4/ 0 @ 0:00:07.692
#I  Size 21527
#I  5/ 0 @ 0:00:28.337
#I  Size 15884
#I  4/ 0 @ 0:00:21.837
#I  Size 1216
#I  4/ 0 @ 0:00:01.758
gap> Size(ssets);
192
```

At a higher level of `InfoRDS`, the number of start sets which are discarded because of wrong signatures are also shown. Now the cosets are changed. Here we use the `NoSort` version of `RaiseStartSetLevel`. This leads to a lot of start sets in the first step. If the number of start sets in U is very large, this could be too much for a reduction. Then the only option is using the brute force method. But also for the brute force search, `RaiseStartSetLevelNoSort` must be called first (remember that we chos an enumeration of G and assume the last element from each startset to be the largeset “interesting” one for further completions).

```
gap> comps:=Difference(GroupList2PermList(Set(U1),Gdata),Np);;
gap> ssets:=ExtendedStartsetsNoSort(ssets,comps,Np,11,Gdata);;
gap> ssets:=ReducedStartsets(ssets,[Gdata.Aac],sigdat,Gdata.diffTable);;
#I  Size 8640
#I  9/ 0 @ 0:00:14.159
gap> Size(ssets);
6899
```

And as above, we continue:

```
repeat
    ssets:=ExtendedStartsets(ssets,comps,Np,11,Gdata);
    ssets:=ReducedStartsets(ssets,[Gdata.Aac],sigdat,Gdata.diffTable);;
until ssets=[] or Size(ssets[1])=11;
comps:=Difference(GroupList2PermList(Set(U2),Gdata),Np);
RaiseStartSetLevelNoSort(ssets,comps,Np,15,Gdata);
repeat
    ssets:=ExtendedStartsets(ssets,comps,Np,15,Gdata);
    ssets:=ReducedStartsets(ssets,[Gdata.Aac],sigdat,Gdata.diffTable);;
until ssets=[] or Size(ssets[1])=15;
```

6 Ordered Signatures

In this chapter, we will discuss two methods to calculate ordered signatures. The first one can be used for relative difference sets with forbidden set, while the second one does only work for ordinary difference sets.

The methods introduced here can only be used in some special cases.

6.1 Ordered signatures by quotient images

Let $D \subseteq G$ be a relative difference set with parameters $(v/n, n, k, \lambda)$ and forbidden set $N \subseteq G$. Let $U \leq G$ be a normal subgroup such that $U \subseteq N$.

Then the coset signature $(v_1, \ldots, v_{|G:U|})$ of D has only the entries 1 (k- times) and 0 ($|G : U| - k$- times). And as in chapter 4 we have

$$\sum_j v_j v_{ij} = \lambda(|U| - |g_i U \cap N|) \quad \text{for } g_i \notin U$$

where $v_{ij} = |D \cap g_i g_j U|$. If the forbidden set N is a subgroup of G we have $|g_i U \cap N|$ is either 0 or equal to $|U \cap N| = |U|$.

Let $\phi: G \to G/U$ be the canonical epimorphism. Then D^ϕ is a relative difference set in G/U with forbidden set N^ϕ and parameters $(v/n, n/|U|, k, |U|\lambda)$.

So the ordered signatures with respect to U are equivalent to the relative difference sets in G/U. Observe that we may not apply reduction in G/U using the full automorphismgroup of G/U but only the group induced by the stabiliser of U in the automorphism group of G. This is due to the fact that we use an "induced" notion of equivalence in G/U because we are interested in signatures and not primarily in difference sets in G/U.

1 ► `NormalSgsForQuotientImages(` *forbidden*`,` *Gdata* `)` O

calculates all normal subgroups of *Gdata.G* which lie in *forbidden*. The returned value is a list of normal subgroups which define pairwise non-isomorphic factor groups.

2 ► `DataForQuotientImage(` *normal*`,` *forbidden*`,` *k*`,` *lambda*`,` *Gdata* `)` O

Let *Gdata* be the usual record for a group G. And let k and *lambda* be the parameters of the relative difference set we want to find. Let then *forbidden* be the forbidden set (as a group or a list of group elements or integers) and *normal* a normal subgroup of G which is contained in *forbidden*.

Then `DataForQuotientImage` returns a record containing the record *.Gdata* of the factor group G/U where the automorphism group is the one induced by the stabiliser of *normal* in the automorphism group of G. Furthermore the returned record contains the forbidden set *.forbidden* in G/U and the new parameter *.lambda* for the difference set in G/U.

The data returned by `DataForQuotientImage` can be used to calculate difference sets in G/U in the way outlined in chapter 2. A quotient image of a relative difference set has a larger λ than the initial difference set. So the following invariant can be used for the generation of difference sets:

3 ▶ `MultiplicityInvariantLargeLambda(` *set*, *Gdata* `)` O

Let *set* be a partial relative difference set with $\lambda > 1$. Set P`:=AllPresentables(`*set*`,`*Gdata*`)` then the set of multiplicities of P is an invariant for partial relative difference sets.

`MultiplicityInvariantLargeLambda` returns a List in a form as `Collected` does.

```
gap> G:=CyclicGroup(7);;Gdata:=PermutationRepForDiffsetCalculations(G);;
gap> AllPresentables([2,3],Gdata);
[ 2, 3, 7, 2, 7, 6 ]
gap> MultiplicityInvariantLargeLambda([2,3],Gdata);
[ [ 1, 2 ], [ 2, 2 ] ]
```

This invariant can be used for `ReducedStartSets` complementary to the signature invariant by defining

```
gap> partfunc:=function(list)
> local sig;
> if sig=fail
> then return fail;
> fi;
> return [MultiplicityInvariantLargeLambda(list,Gdata),SigInvariant(list,sigdata)];
> end;
function( list ) ... end
```

and then passing *partfunc* to `ReducedStartSets`. Of course, sigdata has to be the list of records defining the coset signatures (see section 4.1)

After all difference sets are known, they must be converted into ordered signatures. This is done by the following function:

4 ▶ `OrderedSigsFromQuotientImages(` *fGroupData*, *qimages*, *forbidden*, *normal*, *Gdata* `)` O

Let *Gdata* be the usual record for a group G and *normal* a normal subgroup of G which lies in the forbidden set *forbidden*. Let then *fGroupData* be the record *.Gdata* describing $G/normal$ as returned by `DataForQuotientImage` and *qimages* a set of difference sets in $G/normal$.

Then `OrderedSigsFromQuotientImages` returns a record containing a list of ordered signatures *.orderedSigs* and a list of cosets *.cosets* as well as the factor group *.fg* defined by *fGroupData* and its full automorphism group *fgaut* and the image of *forbidden* in *.fg* is returned as *.Nfg*.

5 ▶ `MatchingFGDataForOrderedSigs(` *forbidden*, *Gdata*, *normalsgs*, *fgdata* `)` O

Let *fgdata* be a list of records of the form returned by `OrderedSigsFromQuotientImages` and *normalsgs* a list of normal subgroups of the group *Gdata.G*. Furthermore let *forbidden* be the forbidden set as a list of group elements or integers or a subgroup of *Gdata.G*.

Then `MatchingFGDataForOrderedSigs` retruns all elements of *fgdata* which match a normal subgroup of *normalsgs*. The returned value is a record containing the normal subgroup *.normal* from *normalsgs*, the record *.sigdata* from *fgdata* and a homomorphism *.hom* which maps *Gdata.G* onto *.sigdata.Gdata.G* and takes *forbidden* to *.sigdata.Nfg*.

6 ▶ `OrderedSigInvariant(` *set*, *data* `)` O

does the same as `SigInvariant`, but for ordered signatures. Here *data* has to be a list of records containing ordered signatures called *.orderedSigs* and cosets *.cosets* just as returned by `OrderedSigsFromQuotientImages`.

Assume we have calculated ordered signatures and have stored them in a record *.osigs* and a list *normalSubgroupsData* as returned by `SignatureData` containing the admissible signatures. A function for partitioning partial relative difference sets as required by `ReducedStartsets` can be defined as follows:

```
partitionfunc:=function(list)
 local si, osi;
  si:=SigInvariant(Union(list,[1]),normalSubgroupsData);
  osi:=OrderedSigInvariant(Union(list,[1]),[osigs]);
  if osi=fail or si=fail
   then
    return fail;
  else
    return si;
  fi;
end;
```

6.2 Ordered signatures using representations

This section contains some methods for ordered signatures in ordinary difference sets. Unfortunately, these methods are not as comfortable as those for unordered signatures. The reason for this is simply that I didn't have any time to tie them together to high-level functions. If you need help here, don't hesitate to contact me.

6.3 Definition

Let $R \subseteq G$ be a (partial) ordinary difference set (for definition see 3.1). Let $U \leq G$ be a normal subgroup and $C = \{g_1, \ldots, g_{|G:U|}\}$ be a system of representatives of G/U.

As in 4.1 we may define the coset signature of R relative to U.

Let $U = g_1, \ldots, g_{|G:U|}$ be an enumeration of G/U. An "admissible ordered signature" for U is a tuple $(v_1, \ldots, v_{|G:U|})$ such that

$$\begin{array}{rl} \sum v_i = k & \\ \sum v_i^2 = \lambda(|U|-1)+k & \\ \sum_j v_j v_{ij} = \lambda(|U|-1) & \quad \text{for } g_i \notin U \end{array}$$

holds where we index the v_i by elements of G/U, so $v_i = v_{g_i}$ and write $v_{ij} = v_{g_i g_j}$. Observe that the third equation is a restriction on the ordering of the tuple $(v_1, \ldots, v_{|G:U|})$. If v is an admissible ordered signature, then the multiset of v is an unordered signature.

Getting ordered admissible signatures from unordered ones can be done by taking all permutations of the unordered signature and verifying the above equations. Obviously, this method isn't very satisfying (nevertheless, the methods for testing unordered signatures from section 4.1 do this to find out if there is an ordered signature at all. Except that they stop when they find an ordered signature).

For ordinary difference sets in extensions of semidirect products of cyclic groups, ordered signatures may be calculated a lot easier (see [Röd06] for details).

6.4 Methods for calculating ordered signatures

1 ► `NormalSubgroupsForRep(` *groupdata*`,` *divisor* `)` O

Let *groupdata* be the output of `PermutationRepForDiffsetCalculations` and *divisor* an integer. Then `NormalSubgroupsForRep` calculates all normal subgroups of *groupdata.G* such that the size of the factor group is divisible by *divisor* and the factor group is a semidirect product of cyclic groups.

The output is a record consisting of

1. a normal subgroup *.Nsg* of G
2. the factor group *.fgrp*$:=G/Nsg$
3. the epimorphism *.epi* from G to *.fgrp*
4. a root of unity *.root*
5. a galois automorphism *.alpha*

6.+7. generators of the factor group $G/.Nsg$ named $.a$ and $.b$ such that $.a$ is normalized by $.b$.

8 a list *.int2pairtable* such that the i^{th} entry ist the pair *[m,n]* with that *Glist[i]^epi=a^(m-1)*b^(n-1)*

.alpha and *.root* may be used as input for `OrderedSigs`

2 ► `OrderedSigs(` *coeffSums*`,` *absSum*`,` *alpha*`,` *root* `)` O

Let G be group which contains a normal subgroup of index s such that the coset signature for a difference set for this normal subgroup is *coeffSums*. Let N be a normal subgroup of G such that G/N is a semidirect product of cyclic group of orders s, q and i divides the order of G/N.

Then `OrderedSigs(`*coeffSums*`,`*absSum*`,`*alpha*`,`*root*`)` calculates all ordered signatures for N. Here *root* is a primitive q-th root of unity and *alpha* is a Galois- automorphism of $CS(q)$ with order dividing s. *absSum* is the order of the difference set. (i.e. $order = k - \lambda$).

`OrderedSigs` is based on calculations using an s-dimensional unitary representation of G/N. In this representation a subset of G induces a semi-circular matrix. The returned value is a list of lists s-tuples The entries of the s-tuples are coefficients of numbers in $\mathbb{Z}[root]$ such that the semi-circular matrix defined by these numbers together with *alpha* meets necessary conditions for matrices induced by difference sets. To gain the algebraic numbers from the s-tuple *tup*, use `List(`*tup*`,i->CoeffList2CyclotomicList(i,`*root*`))`

Each |*coeffSums*|-tuple returned defines an ordered signature. The ordering of G/N is chosen to fit to the data returned by `NormalSubgroupsForRep`:

$[a^0, a^1, \dots, a^{q-1}], [a^0 b, a^1 b, \dots, a^{q-1} b], \dots, [a^0 b^{s-1}, \dots, a^{q-1} b^{s-1}]$

So for the calculation of ordered signatures, smaller ordered signatures *coeffSums* have to be known. But this is not so bad, as small signatures are easy to calculate. The following example shows an application.

```
gap> G:=SmallGroup(273,3);
<pc group of size 273 with 3 generators>
gap> Gdata:=PermutationRepForDiffsetCalculations(G);;
gap> CosetSignatures(273,273/3,16);
[ [ 3, 7, 7 ] ]
gap> nsgs:=NormalSubgroupsForRep(Gdata,3);
[ rec( Nsg := Group([ f2 ]), alpha := ANFAutomorphism( CF(13), 3 ),
      root := E(13), fgrp := Group([ f1, <identity> of ..., f2 ]),
      epi := [ f1, f2, f3 ] -> [ f1, <identity> of ..., f2 ], a := f2,
      b := f1,
      int2pairtable := [ [ 1, 1 ], [ 1, 2 ], [ 1, 1 ], [ 2, 1 ], [ 1, 3 ],
...
          [ 8, 3 ], [ 11, 3 ], [ 5, 2 ], [ 11, 3 ] ] ),
```

```
  rec( Nsg := Group([ f3 ]), alpha := ANFAutomorphism( CF(7), 2 ),
      root := E(7), fgrp := Group([ f1, f2, <identity> of ... ]),
      epi := [ f1, f2, f3 ] -> [ f1, f2, <identity> of ... ], a := f2,
      b := f1,
      int2pairtable := [ [ 1, 1 ], [ 1, 2 ], [ 2, 1 ], [ 1, 1 ], [ 1, 3 ],
...
          [ 6, 3 ], [ 4, 3 ], [ 4, 2 ], [ 6, 3 ] ] ) ]
gap> osigs:=OrderedSigs([3,7,7],16,nsgs[2].alpha,nsgs[2].root);
[ [ [ 0, 0, 0, 1, 0, 1, 1 ], [ 0, 0, 1, 2, 2, 0, 2 ], [ 2, 2, 0, 2, 0, 0, 1 ] ],
  [ [ 0, 0, 0, 1, 0, 1, 1 ], [ 0, 1, 2, 2, 0, 2, 0 ], [ 2, 0, 0, 1, 2, 2, 0 ] ],
...
   [ [ 1, 1, 0, 1, 0, 0, 0 ], [ 2, 2, 1, 0, 0, 2, 0 ], [ 2, 1, 0, 0, 2, 0, 2 ] ] ]
gap> Size(osigs);
98
gap> Set(osigs,g->SortedList(Concatenation(g)));
[ [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2 ] ]
```

Note that the signature `[3, 7, 7]` can be assumed to be ordered (by passing to a suitable translate). So even if we are not interested in **ordered** signatures, we have found out that there is only one admissible unordered signature for this normal subgroup. To get this result using `TestedSignatures` would have taken a **very** long time.

Of course, ordered signatures can also be used directly.

3 ► `OrderedSignatureOfSet( set, normal_data )` O

takes a set *set* of integers (meant to be a partial difference set) and a list of records as returned by `NormalSubgroupsForRep`. The returned value is a list of lists which is the ordered signature of the partial difference set *set* and can be compared to the output of `OrderedSigs`

```
gap> OrderedSignatureOfSet([2,3,4,5],nsgs[2]);
[ [ 1, 1, 1, 0, 0, 0, 0 ], [ 1, 0, 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0, 0, 0 ] ]
```

7 Determining the Isomorphism Class of Projective Planes

The methods in this chapter do not deal with relative difference sets. Instead, they help studying projective planes. So if you have a relative difference set, you must first generate the projective plane it defines (if it does).

Projective planes are always assumed to consist of positive integers (as points) and sets of integers (as blocks). The incidence relation is assumed to be the element relation. The blocks of a projective plane must be **sets**.

The following methods generate a record characterising the projective plane. As most of the functions in this chapter need this data, the record returned by `ElationPrecalc` or `ElationPrecalcSmall` is the recommended representation of projective planes.

1 ► `ElationPrecalc(` *blocks* `)` F
► `ElationPrecalcSmall(` *blocks* `)` F

Given the blocks *blocks* of a projective plane, `ElationPrecalc(` *blocks* `)` returns a record conatining

.points the points of the projective plane (immutable)

.blocks the blocks as passed to the function (immutable)

.jpoint a matrix with ij-th entry the point meeting the i-th and the j-th block.

.jblock a matrix with ij-th entry the position of the block connecting the point i to the point j in *blocks*.

`ElationPrecalcSmall(` *blocks* `)` returns a record which does only contain *.points*, *.blocks* and *.jblock*. Hence the name.

In the following sections, some of the functions have two versions. The versions which have a `Small` appended to it's name do not depend on the data generated by `ElationPrecalc`, but rather on the data structure provided by `ElationPrecalcSmall`. The `Small` versions are generally much slower than the other ones.

2 ► `DualPlane(` *blocks* `)` O

For a projective plane given by *blocks*, `DualPlane(` *blocks* `)` returns a record containing a set of blocks defining the dual plane and a List *image* containing the same blocks such that *image[p]* is the image of the point p under duality. It is not tested, if the design defined by *blocks* is actually a projective plane.

3 ► `ProjectiveClosureOfPointSet(` *points*, *maxsize*, *data* `)` O

Let P be a projective plane given by the record *data* as returned by `ElationPrecalcSmall`. Let *points* be a set of points (integers). Then `ProjectiveClosureOfPointSet` returns the projective colsure of *points* in P (the smallest subplane of P containing the points *points*). The closure is returned as a list of points. If $maxsize \neq 0$, calculations are stopped if the closure is known to have at least *maxsize* points and *data.points* is returned. Observe that this is a “small” function, in the sense that it does not need the data from `ElationPrecalc` but merely the data generated by `ElationPrecalcSmall`.

7.1 Isomorphisms and Collineations

Isomorphisms of projective planes are mappings which take points to points and blocks to blocks and respect incidence. A **collineation** of a projective plane P is a collineation from P to P (an automorphism).

As projective planes are assumed to live on the integers, isomorphisms of projective planes are represented by permutations. To test if a permutation on points is actually an isomorphism of projective planes, the following methods can be used.

1 ▶ `IsIsomorphismOfProjectivePlanes(` *perm*`,` *blocks1*`,` *blocks2* `)` O

Let *blocks1*, *blocks2* be two sets of blocks of projective planes on the same points. `IsIsomorphismOfProjectivePlanes(` *perm*`,`*blocks1*`,`*blocks2* `)` test if the permutation *perm* on points defines an isomorphism of the projective planes defined by *blocks1* and *blocks2*.

2 ▶ `IsCollineationOfProjectivePlane(` *perm*`,` *blocks* `)` O
▶ `IsCollineationOfProjectivePlane(` *perm*`,` *data* `)` O

Let *blocks* be the blocks of a projective plane and *perm* a permutation on the points of this plane. `IsCollineationOfProjectivePlane(`*perm*`,`*blocks*`)` returns `true`, if *perm* induces a collineation of the projective plane.

If *data* as returned by `ElationPrecalc` is given instead of *blocks*, the calculation should be faster.

3 ▶ `IsomorphismProjPlanesByGenerators(` *gens1*`,` *data1*`,` *gens2*`,` *data2* `)` O
▶ `IsomorphismProjPlanesByGeneratorsNC(` *gens1*`,` *data1*`,` *gens2*`,` *data2* `)` O

Let *gens1* be a list of points generating the projective plane defined by *data1* and *gens2* a list of generating points for *data2*. Then a permutation is returned representing a mapping from the *data1.points* to *data2.points* and mapping the list *gens1* to the list *gens2*. If there is no such mapping which defines an isomorphism of projective planes, `fail` is returned. Note that this is a "small" function, in the sense that *data1* and *data2* are as returned by `ElationPrecalcSmall` rather than by `ElationPrecalc`.

`IsomorphismProjPlanesByGeneratorsNC` does **not** checked whether *gens1* and *gens2* really generate the planes given by *data1* and *data2*.

```
# Assume that <blocks> contains a list of lines of a projective plane
# of order 16
gap> data:=ElationPrecalc(blocks);;
gap> Size(ProjectiveClosureOfPointSet([1,2,3,5],16,data));
4
gap> Size(ProjectiveClosureOfPointSet([1,2,60,268],16,data));time;
273
0
gap> Size(ProjectiveClosureOfPointSet([1,2,60,268],0,data));time;
273
184
gap> IsomorphismProjPlanesByGenerators([1,2,3,5],data,[1,2,60,268],data);
fail
gap> IsomorphismProjPlanesByGenerators([1,2,60,268],data,[1,2,60,268],data);
()
gap> IsomorphismProjPlanesByGenerators([1,2,60,268],data,[1,3,146,268],data);
(2,3)(5,10)(6,12)(7,9)(8,11)(13,16)(17,249)(18,251)(19,250)( [...] )
gap> Order(last);
2
```

7.2 Central Collineations

Let ϕ be a collineation of a projective plane which fixes one point block-wise (the so-called **centre**) and one block point-wise (the so-called **axis**). If the centre is contained in the axis, ϕ is called **elation**. Otherwise, ϕ is called **homology**. The group of elations with given axis is called **translation group** of the plane (relative to the chosen axis). A projective plane with transitive translation group is called **translation plane**. Here transitivity is on the points outside the axis.

1 ▶ `ElationsByPairs(` *centre*`,` *axis*`,` *pairs*`,` *data* `)` O
▶ `ElationsByPairs(` *centre*`,` *axis*`,` *pairs*`,` *blocks* `)` O
▶ `ElationsByPairsSmall(` *centre*`,` *axis*`,` *pairs*`,` *data* `)` O

Let *centre* be a point and *axis* a block of a projective plane defined by *blocks* (or by *data* as returned by `ElationPrecalc`). The list *pairs* must contain pairs of points outside *axis*. `ElationsByPairs` returns a collineation fixing *axis* pointwise and *centre* blockwise (an elation) such that for each pair *p* of *pairs* *p[1]* is mapped on *p[2]*. If no such elation exists, `fail` is returned.

`ElationsByPairsSmall` uses *data* as returned by `ElationPrecalcSmall`

2 ▶ `AllElationsCentAx(` *centre*`,` *axis*`,` *data*`[, "generators"] )` O
▶ `AllElationsCentAx(` *centre*`,` *axis*`,` *blocks*`[, "generators"] )` O
▶ `AllElationsCentAxSmall(` *centre*`,` *axis*`,` *data*`[, "generators"] )` O

Let *centre* be a point and *axis* a block of a projective plane defined by *blocks* (or by *data* as returned by `ElationPrecalc`). `AllElationsCentAx` returns a list of all non-trivial elations with centre *centre* and axis *axis*. If "generators" is set, a list of generators of the translation group is returned.

3 ▶ `AllElationsAx(` *axis*`,` *data*`[, "generators"] )` O
▶ `AllElationsAx(` *axis*`,` *blocks* `)` O
▶ `AllElationsAxSmall(` *axis*`,` *data*`[, "generators"] )` O

Let *axis* be a block of a projective plane defined by *blocks* (or by *data* as returned by `ElationPrecalc`). `AllElationsAx` returns a list of all non-trivial elations with axis *axis*.

4 ▶ `IsTranslationPlane(` *infline*`,` *planedata* `)` O
▶ `IsTranslationPlaneSmall(` *infline*`,` *planedata* `)` O

If the group of elations with axis *infline* is (sharply) transitive on the affine points (the points outside *infline*), `IsTranslationPlane` returns `true`, otherwise it returns `false`. This is faster than calculating the full translation group if the projective plane is not a translation plane.

5 ▶ `HomologyByPairSmall(` *centre*`,` *axis*`,` *pair*`,` *data* `)` O

`HomologyByPairSmall` returns the homology defined by the pair *pair* fixing *centre* blockwise and *axis* pointwise. The returned permutation fixes *axis* pointwise and *centre* linewise and takes *pair[1]* to *pair[2]*.

6 ▶ `GroupOfHomologiesSmall(` *centre*`,` *axis*`,` *data* `)` O

returns the group of homologies with centre *centre* and axis *axis*.

7.3 Collineations on Baer Subplanes

Let P be a projective plane of order n^2. A subplane B of order n of P is called **Baer subplane**. Baer suplanes are exactly the maximal subplanes of P.

1 ► `InducedCollineation(` *baerdata*, *baercoll*, *point*, *image*, *planedata*, *liftingperm* `)` O

If a projective plane contains a Baer subplane, collineations of the subplane may be lifted to the full plane. Here *baercoll* is a collineation of the subplane given by *baerdata* (as returned by `ElationPrecalc`. Be careful, as the enumeration for the subplane is not the same as for the whole plane). *liftingperm* is a permutation on the points of the full pane which converts the enumeration of the subplane to that of the full plane. This means that the image of *baerdata.points* under *liftingperm* is a subset of *planedata.points*. Namely the one representing the Baer plane in the enumeration used for the whole plane. *point* and *image* are points outside the Baer plane.

`InducedCollineation` returns a collineation of the full plane (as a permutation on *planedata.points*) which takes *point* to *image* and acts on the Baer plane as *baercoll* does.

Just to make this clear again, *baerdata* has points $[1, \ldots, n^2 + n + 1]$ and *planedata* has points $[1, \ldots, n^4 + n^2 + 1]$. *baercoll* lives on *baerdata.points* (and hence on $n^2 + n + 1$ points) and *point* and *image* live on *planedata.points*. Anything can happen if you mix something up here.

7.4 Invariants for Projective Planes

The functions `NrFanoPlanesAtPoints`, `pRank`, `FingerprintAntiFlag` and `FingerprintProjPlane` calculate invariants for finite projective planes. For more details see [Röd06] and [Moo95]. The values of some of these invariants are available from the homepages of [Moo] and [Roy] for many planes.

1 ► `NrFanoPlanesAtPoints(` *points*, *data* `)` O

For a projective plane defined by the blocks *data* as returned by `ElationPrecalc`, `NrFanoPlanesAtPoints(`*points*`,`*data*`)` calculates the so-called Fano invariant. That is, for each point in *points*, the number of subplanes of order 2 (so-called Fano planes) containing this point is calculated. The method returns a list of pairs of the form [*point*, *number*] where *number* is the number of Fano sub-planes in *point*.

2 ► `NrFanoPlanesAtPointsSmall(` *pointlist*, *data* `)` O

Uses *data* as returned by `ElationPrecalcSmall`. Only use this, if you want to do a quick experiment in a plane of **small** order and don't like to generate a new set of data with `ElationPrecalc`. The difference between `NrFanoPlanesAtPoints` and `NrFanoPlanesAtPointsSmall` is that the "small" version does some operations for lists (like `Intersection`) whereas the "large" version does only read matrix entries. This makes quite a difference as for a plane of order n, there are $\binom{n+1}{3}\binom{n}{2}n$ quadrangles to be tested per point.

3 ► `IncidenceMatrix(` *points*, *blocks* `)` O
► `IncidenceMatrix(` *data* `)` O

returns a matrix I, where the columns are numbered by the blocks and the rows are numbered by points. And *I[i][j]=1* if and only if *points[i]* is incident (contained in) *blocks[j]*.

4 ► `pRank(` *blocklist*, *p* `)` O
► `pRank(` *data*, *p* `)` O

Let I be the incidence matrix of the projective plane given by the list of blocks *blocklist* or the record *data* as returned by `ElationPrecalc`. The rank of $I \cdot I^t$ as a matrix over $GF(p)$ is called p-rank of the projective plane. Here I^t denotes the transposed matrix.

As `pRank` calls `IncidenceMatrix`, the list *blocklist* has to be a list of lists of integers.

5 ► `FingerprintProjPlane(` *blocks* `)` O
► `FingerprintProjPlane(` *data* `)` O

For each anti-flag (p, l) of a projective plane of order n, define an arbitrary but fixed enumeration of the lines through p and the points on l. Say $l_1, \ldots, l_{n+1}$ and $p_1, \ldots, p_{n+1}$ The incidence relation defines a canonical bijection between the l_i and the p_i and hence a permutation on the indices $1, \ldots, n+1$. Let $\sigma_{(p,l)}$ be this permutation.

Denote the points and lines of the plane by $q_1, \ldots q_{n^2+n+1}$ and $e_1, \ldots, e_{n^2+n+1}$. Define the sign matrix as $A_{ij} = sgn(\sigma_{(q_i,e_j)})$ if (q_i, e_j) is an anti-flag and $= 0$ if it is a flag. Then the fingerprint is defnied as the multiset of the entries of $|AA^t|$. Here *data* is a record as returned by `ElationPrecalcSmall`.

6 ► `FingerprintAntiFlag(` *point*`,` *linenr*`,` *data* `)` O

Let $m_1, \ldots, m_{n+1}$ be the lines containing *point* and $E_1, \ldots, E_{n+1}$ the points on the line given by *linenr* such that E_i is incident with m_i. Now label the points of m_i as $point = P_{i,1}, \ldots, P_{i,n+1} = E_i$ and the lines of E_i as $line = l_1, \ldots, l_{i,n+1} = m_i$. For $i \neq j$, each $P_{j,k}$ lies on exactly one line $l_{i,k\sigma_{i,j}}$ containing E_i for some permutation $\sigma_{i,j}$

Define a matrix A, where $A_{i,j}$ is the sign of $\sigma_{i,j}$ if $i \neq j$ and $A_{i,i} = 0$ for all i. The partial fingerprint is the multiset of entries of $|AA^t|$ where A^t denotes the transposed matrix of A.

this is a "small" function.

8 Some functions for everyday use

This chapter contains a number of functions that did not fit in anywhere else. Some of them might be useful for other people, too, so they were included here.

8.1 Groups and actions

1 ► `OnSubgroups( subgroup, aut )` F

For a group G and an automorphism *aut* of G, `OnSubgroups(`*subgroup*`,`*aut*`)` is the image of *subgroup* under *aut*

```
gap> G:=Group((1,2,3),(2,3));
Group([ (1,2,3), (2,3) ])
gap> alpha:=InnerAutomorphism(G,(1,2,3));
^(1,2,3)
gap> OnSubgroups(Subgroup(G,[(2,3)]),alpha);
Group([ (1,3) ])
```

2 ► `RepsCClassesGivenOrder( group, order )` O

`RepsCClassesGivenOrder(` *group*, *order* `)` returns all elements of order *order* up to conjugacy. Note that the representatives are **not** always the smallest elements of each conjugacy class.

```
gap> RepsCClassesGivenOrder(SymmetricGroup(5),2);
[ (4,5), (2,3)(4,5) ]
```

8.2 Iterators

1 ► `CartesianIterator( tuplelist )` O

Returns an iterator for `Cartesian(`*tuplelist*`)`

2 ► `ConcatenationOfIterators( iterlist )` F

`ConcatenationOfIterators(`*iterlist*`)` returns an iterator which runs through all iterators in *iterlist*. Note that the returned iterator loops over the iterators in *iterlist* **sequentially** beginning with the first one.

```
gap> it:=Iterator([1,2,3]);;
gap> it2:=CartesianIterator([[9,10],[11]]);;
gap> cit:=ConcatenationOfIterators([it,it2]);;
gap> repeat
> Print(NextIterator(cit),",\c");
> until IsDoneIterator(cit);
1,2,3,[ 9, 11 ],[ 10, 11 ],
```

8.3 Lists and Matrices

1 ► `IsListOfIntegers(` *list* `)` P

`IsListOfIntegers(` *list* `)` returns `IsSubset(Integers,` *list* `)` if *list* is a dense list and `false` otherwise.

2 ► `List2Tuples(` *list*`,` *int* `)` O

If `Size(` *list* `)` is divisible by *int*, `List2Tuples(` *list*`,`*int*`)` returns a list *list2* of size *int* such that `Concatenation(` *list2* `)=` *list* and every element of *list2* has the same size.

```
gap> List2Tuples([1..6],2);
[ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
```

3 ► `MatTimesTransMat(` *mat* `)` O

does the same as *mat*`*TransposedMat(` *mat* `)` but uses slightly less space and time for large matrices.

4 ► `PartitionByFunctionNF(` *list*`,` *f* `)` O

`PartitionByFunctionNF(` *list*`,` *f* `)` partitions the list *list* according to the values of the function *f* defined on *list*. If *f* returns `fail` for some element of *list*, `PartitionByFunctionNF(` *list*`,` *f* `)` enters a break loop. Leaving the break loop with 'return;' is safe because `PartitionByFunctionNF` treats `fail` as all other results of *f*.

5 ► `PartitionByFunction(` *list*`,` *f* `)` O

`PartitionByFunction(` *list*`,` *f* `)` partitions the list *list* according to the values of the function *f* defined on *list*. All elements, for which *f* returns `fail` are omitted, so `PartitionByFunction` does not necessarily return a partition. If `InfoLevel(InfoRDS)` is at least 2, the number of elements for which *f* returns `fail` is shown (if `fail` is returned at all).

```
gap> PartitionByFunctionNF([-1..5],x->x^2);
[ [ 0 ], [ -1, 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ] ]
gap> test:=function(x)
> if x>0 then return Sqrt(x);
>  else return fail;
> fi;
> end;
function( x ) ... end
gap> PartitionByFunction([-1..5],test);
[ [ 1 ], [ 4 ], [ 5 ], [ 2 ], [ 3 ] ]
gap> SetInfoLevel(InfoRDS,2);
gap> PartitionByFunction([-1..5],test);
#I  -2-
[ [ 1 ], [ 4 ], [ 5 ], [ 2 ], [ 3 ] ]
gap> PartitionByFunctionNF([-1..5],test);
Error, function returned <fail> called from
...
brk> return;
[ [ 1 ], [ 4 ], [ 5 ], [ 2 ], [ 3 ], [ -1, 0 ] ]
```

8.4 Cyclotomic numbers

1 ▶ `IsRootOfUnity(` *cyc* `)` P

`IsRootOfUnity` tests if a given cyclotomic is actually a root of unity.

2 ▶ `CoeffList2CyclotomicList(` *list*`,` *root* `)` O

`CoeffList2CyclogomicList(` *list*`,` *root* `)` takes a list of integers *list* and a root of unity *root* and returns a list *list2*, where *list2[i]=list[i]* root^(i-1)*.

3 ▶ `AbssquareInCyclotomics(` *list*`,` *root* `)` O

For a list of integers and a root of unity, `AbssquareInCyclotomics(` *list*`,` *root* `)` returns the modulus of `Sum(CoeffList2CyclotomicList(` *list*`,` *root* `))`.

4 ▶ `CycsGivenCoeffSum(` *sum*`,` *root* `)` O

`CycsGivenCoeffSum(` *sum*`,` *root* `)` returns all elements of $\mathbb{Z}[root]$ such that the coefficient sum is *sum* and all coefficients are non-negative. The returned list has the following form: The cyclotomic numbers are represented by coefficients. `CoeffList2CyclotomicList` can be used to get the algebraic number represented by *list*. The list is partitioned into equivalence classes of elements having the same modulus. For each class the modulus is returned. This means that `CycsGivenCoeffSum` returns a list of pairs where the first entry of each pair is the square of the modulus of an element of the second entry. And the second entry is a list of coefficient lists of cyclotomics in $\mathbb{Z}[root]$ having the coefficient sum *sum*.

```
gap> CycsGivenCoeffSum(3,E(3));
[ [ 0, [ [ 1, 1, 1 ] ] ],
  [ 3, [ [ 0, 1, 2 ], [ 0, 2, 1 ], [ 1, 0, 2 ], [ 1, 2, 0 ], [ 2, 0, 1 ],
          [ 2, 1, 0 ] ] ], [ 9, [ [ 0, 0, 3 ], [ 0, 3, 0 ], [ 3, 0, 0 ] ] ] ]
gap> CycsGivenCoeffSum(2,E(2));
[ [ 0, [ [ 1, 1 ] ] ], [ 4, [ [ 0, 2 ], [ 2, 0 ] ] ] ]
```

8.5 Filters and Categories

The following was originally posted at the GAP forum by Thomas Breuer [Bre05].

Each filter in GAP is either a simple filter or a meet of filters. For example, `IsInt` and `IsPosRat` are simple filters, and `IsPosInt` is defined as their meet `IsInt and IsPosRat`.

Each **simple filter** is of one of the following kinds.

1. property: Such a filter is an operation, the filter value can be computed. The (unary) methods of this operation must return `true` or `false`, and the return value is stored in the argument, except if the argument is of a basic data type such as cyclotomic (including rationals and integers), finite field element, permutation, or internally represented list –the latter with a few exceptions. Examples of properties are `IsFinite`, `IsAbelian`, `IsSSortedList`.

2. attribute tester: Such a filter is associated to an operation that has been created via `DeclareAttribute`, in the sense that the value is `true` if and only if a return value for (a unary method of) this operation is stored in the argument. Currently, attribute values are stored in objects in the filter `IsAttributeStoringRep`. Examples of attribute testers are `HasSize`, `HasCentre`, `HasDerivedSubgroup`.

2.' property tester: Such a filter is similar to an attribute tester, but the associated operation is a property. So property testers can return `true` also if the argument is not in the filter `IsAttributeStoringRep`. Examples of property testers are `HasIsFinite`, `HasIsAbelian`, `HasIsSSortedList`.

3. category or representation: These filters are not associated to operations, their values cannot be computed but are set upon creation of an object and should not be changed later, such that for a filter of this kind,

one can rely on the fact that if the value is `true` then it was `true` already when the object in question was created.

The distinction between representation and category is intended to express dependency on or independence of the way how the object is stored internally. For example, `IsPositionalObjectRep`, `IsComponentObjectRep`, and `IsInternalRep` are filters of the representation kind; the idea is that such filters are used in low level methods, and that higher level methods can be implemented without referring to these filters.

Examples of categories are `IsInt`, `IsRat`, `IsPerm`, `IsFFE`, and filters expressing algebraic structures, such as `IsMagma`, `IsMagmaWithOne`, `IsAdditiveMagma`. When one calls such a filter, one can be sure that no computation is triggered. For example, whenever a quotient of two integers is formed, the result is clearly in the filter `IsRat`, but the system also stores the value of `IsInt`, i.e., GAP does not support “unevaluated rationals” for which the `IsInt` value is computed on demand and then stored.

4. other filters: Some filters do not belong to the above kinds, they are not associated to operations but they are intended to be set (or even reset) by the user or by functions also after the creation of objects. Examples are `IsQuickPositionList`, `CanEasilyTestMembership`, `IsHandledByNiceBasis`.

Each **meet of filters** can involve computable simple filters (properties, attribute and property testers) and not computable simple filters (categories, representations, other filters). When one calls a meet of two filters then the two filters from which the meet was formed are evaluated (if necessary). So a meet of filters is computable only if at least one computable simple filter is involved.

1 ► `IsComputableFilter(` *filter* `)` F

'IsComputableFilter(*filter*)' returns *true* if a the filter *filter* is computable. Filters for which 'IsComputable-Filter' returns *false* may be used in 'DeclareOperation'.

```
gap> IsComputableFilter( IsFinite );
true
gap> IsComputableFilter( HasSize );
true
gap> IsComputableFilter( HasIsFinite );
true
gap> IsComputableFilter( IsPositionalObjectRep );
false
gap> IsComputableFilter( IsInt );
false
gap> IsComputableFilter( IsQuickPositionList );
false
gap> IsComputableFilter( IsInt and IsPosRat );
false
gap> IsComputableFilter( IsMagma );
false
```

Bibliography

[Bre05] Thomas Breuer. Re: Filter trouble. Posting at the GAP forum, Jun 2005.

[Bru55] Richard H. Bruck. Difference sets in a finite group. *Transactions of the American Mathematical Society*, 78(78):464–481, 1955.

[Dem68] Peter Dembowski. *Finite Geometries*. Number 44 in Ergebnisse der Mathematik und ihrer Genzgebiete. Springer-Verlag, Berlin Heidelberg, 1968.

[DP67] Peter Dembowski and Fred Piper. Quasiregular collineation groups of finite projective planes. *Mathematische Zeitschrift*, 99:53–75, 1967.

[Moo] G. Eric Moorhouse. Data for projective planes.

`http://www.uwyo.edu/moorhouse/`.

[Moo95] G. Eric Moorhouse. Two-graphs and skew two-graphs in finite geometries. *Linear Algebra and its Applications*, 226–228:529–551, 1995.

[Röd06] Marc Röder. *Quasiregular Projective Planes of Order* 16 – *A Computational Approach*. PhD thesis, Technische Universität Kaiserslautern, 2006.

[Roy] Gordon Royle. Combinatorial catalogues.

`http://www.csse.uwa.edu.au/~gordon/data.html`.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "PermutationCharacter" comes before "permutation group".