DESIGN 1.1

A GAP 4.4 Package

by

Leonard H. Soicher

School of Mathematical Sciences
Queen Mary, University of London

Contents

1 Design

This manual describes the DESIGN 1.1 package for GAP 4.4. The DESIGN package is for generating, classifying and studying block designs.

All DESIGN functions are written entirely in the GAP language. However, DESIGN requires the GRAPE package [Soi04] to be installed, and makes use of certain GRAPE functions, some of which make use of B. D. McKay's *nauty* (Version 2.0b5) package [McK96]. These GRAPE functions can only be used on a fully installed version of GRAPE in a UNIX environment. DESIGN also requires the GAPDoc package [LN04], if you want to read lists of designs in the

`http://designtheory.org` external representation format (see [CDMS04]).

Except for the function `SmallestImageSet`, which is Copyright © Steve Linton 2003, the DESIGN package is Copyright © Leonard H. Soicher 2003–2004. DESIGN is part of an EPSRC funded project to provide a web-based resource for design theory; see

`http://designtheory.org` .

DESIGN is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. For details, see

`http://www.gnu.org/licenses/gpl.html`

If you use DESIGN to solve a problem then please send a short email about it to `L.H.Soicher@qmul.ac.uk`, and reference the DESIGN package as follows:

Leonard H. Soicher, The DESIGN package for GAP,

`http://designtheory.org/software/gap_design/` .

1.1 Installing the DESIGN Package

The DESIGN package only runs properly on a UNIX system. Before installing DESIGN (on your UNIX system), you must make sure that the GRAPE and GAPDoc packages are fully installed.

To install DESIGN 1.1 (after installing GAP, GRAPE and GAPDoc), first obtain the DESIGN archive file `design1r1.tar.gz`, available from

`http://designtheory.org/software/gap_design/` and then copy this archive file into the `pkg` directory of the GAP root directory. Actually, it is possible to have several GAP root directories, and so it is easy to install DESIGN locally even if you have no permission to add files to the main GAP installation (see 9.2). Now go to the appropriate `pkg` directory containing `design1r1.tar.gz`, and then run

```
gunzip design1r1.tar.gz
tar -xf design1r1.tar
```

That's all there is to do.

Both dvi and pdf versions of the DESIGN manual are available (as `manual.dvi` and `manual.pdf` respectively) in the `doc` directory of the home directory of DESIGN.

If you install DESIGN, then please tell `L.H.Soicher@qmul.ac.uk`, where you should also send any comments or bug reports.

1.2 Loading DESIGN

Before using DESIGN you must load the package within GAP by calling the statement

```
gap> LoadPackage("design");
true
```

1.3 The structure of a block design in DESIGN

A **block design** is a pair (X, B), where X is a non-empty finite set whose elements are called **points**, and B is a non-empty finite multiset whose elements are called **blocks**, such that each block is a non-empty finite multiset of points.

DESIGN deals with arbitrary block designs. However, at present, some DESIGN functions only work for **binary** block designs (i.e. those with no repeated element in any block of the design), but these functions will check if an input block design is binary.

In DESIGN, a block design D is stored as a record, with mandatory components `isBlockDesign`, `v`, and `blocks`. The points of a block design D are always 1,2,...,D`.v`, but they may also be given **names** in the optional component `pointNames`, with D`.pointNames[`i`]` the name of point i. The `blocks` component must be a sorted list of the blocks of D (including any repeats), with each block being a sorted list of points (including any repeats).

A block design record may also have some optional components which store information about the design. At present these optional components include `isSimple`, `isBinary`, `isConnected`, `r`, `blockSizes`, `blockNumbers`, `resolutions`, `autGroup`, `autSubgroup`, `tSubsetStructure`, `allTDesignLambdas`, and `pointNames`.

A non-expert user should only use functions in the DESIGN package to create block design records and their components.

1.4 Example of the use of DESIGN

To give you an idea of the capabilities of this package, we now give an extended example of an application of the DESIGN package, in which a nearly resolvable non-simple 2-(21,4,3) is constructed (for Donald Preece) via a pairwise-balanced design. All the DESIGN functions used here are described in this manual.

The program first discovers the unique (up to isomorphism) pairwise-balanced 2-(21,$\{4,5\}$,1) design D invariant under $H = \langle(1, 2, \ldots, 20)\rangle$, and then applies the McSorley-Soicher construction to this design D to obtain a non-simple 2-(21,4,3) design *Dstar* with automorphism group of order 80. The program then classifies the near-resolutions of *Dstar* invariant under the subgroup of order 5 of H, and finds exactly two such (up to the action of Aut (*Dstar*)). Finally, *Dstar* is printed.

```
gap> H:=CyclicGroup(IsPermGroup,20);
Group([ (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20) ])
gap> D:=BlockDesigns(rec(v:=21,blockSizes:=[4,5],
>       tSubsetStructure:=rec(t:=2,lambdas:=[1]),
>       requiredAutSubgroup:=H ));;
gap> Length(D);
1
gap> D:=D[1];;
gap> BlockSizes(D);
[ 4, 5 ]
gap> BlockNumbers(D);
[ 20, 9 ]
gap> Size(AutGroupBlockDesign(D));
80
```

```
gap> Dstar:=TDesignFromTBD(D,2,BlockSizes(D));;
gap> AllTDesignLambdas(Dstar);
[ 105, 20, 3 ]
gap> Size(AutGroupBlockDesign(Dstar));
80
gap> near_resolutions:=PartitionsIntoBlockDesigns(rec(
>    blockDesign:=Dstar,
>    v:=21,blockSizes:=[4],
>    tSubsetStructure:=rec(t:=0,lambdas:=[5]),
>    blockIntersectionNumbers:=[[ [0] ]],
>    requiredAutSubgroup:=SylowSubgroup(H,5) ));;
gap> Length(near_resolutions);
2
gap> List(near_resolutions,x->Size(x.autGroup));
[ 5, 20 ]
gap> Print(Dstar,"\n");
rec(
  isBlockDesign := true,
  v := 21,
  blocks := [ [ 1, 2, 4, 15 ], [ 1, 2, 4, 15 ], [ 1, 2, 4, 15 ],
      [ 1, 3, 14, 20 ], [ 1, 3, 14, 20 ], [ 1, 3, 14, 20 ], [ 1, 5, 9, 13 ],
      [ 1, 5, 9, 17 ], [ 1, 5, 13, 17 ], [ 1, 6, 11, 16 ], [ 1, 6, 11, 21 ],
      [ 1, 6, 16, 21 ], [ 1, 7, 8, 10 ], [ 1, 7, 8, 10 ], [ 1, 7, 8, 10 ],
      [ 1, 9, 13, 17 ], [ 1, 11, 16, 21 ], [ 1, 12, 18, 19 ],
      [ 1, 12, 18, 19 ], [ 1, 12, 18, 19 ], [ 2, 3, 5, 16 ], [ 2, 3, 5, 16 ],
      [ 2, 3, 5, 16 ], [ 2, 6, 10, 14 ], [ 2, 6, 10, 18 ], [ 2, 6, 14, 18 ],
      [ 2, 7, 12, 17 ], [ 2, 7, 12, 21 ], [ 2, 7, 17, 21 ], [ 2, 8, 9, 11 ],
      [ 2, 8, 9, 11 ], [ 2, 8, 9, 11 ], [ 2, 10, 14, 18 ], [ 2, 12, 17, 21 ],
      [ 2, 13, 19, 20 ], [ 2, 13, 19, 20 ], [ 2, 13, 19, 20 ],
      [ 3, 4, 6, 17 ], [ 3, 4, 6, 17 ], [ 3, 4, 6, 17 ], [ 3, 7, 11, 15 ],
      [ 3, 7, 11, 19 ], [ 3, 7, 15, 19 ], [ 3, 8, 13, 18 ], [ 3, 8, 13, 21 ],
      [ 3, 8, 18, 21 ], [ 3, 9, 10, 12 ], [ 3, 9, 10, 12 ], [ 3, 9, 10, 12 ],
      [ 3, 11, 15, 19 ], [ 3, 13, 18, 21 ], [ 4, 5, 7, 18 ], [ 4, 5, 7, 18 ],
      [ 4, 5, 7, 18 ], [ 4, 8, 12, 16 ], [ 4, 8, 12, 20 ], [ 4, 8, 16, 20 ],
      [ 4, 9, 14, 19 ], [ 4, 9, 14, 21 ], [ 4, 9, 19, 21 ], [ 4, 10, 11, 13 ],
      [ 4, 10, 11, 13 ], [ 4, 10, 11, 13 ], [ 4, 12, 16, 20 ],
      [ 4, 14, 19, 21 ], [ 5, 6, 8, 19 ], [ 5, 6, 8, 19 ], [ 5, 6, 8, 19 ],
      [ 5, 9, 13, 17 ], [ 5, 10, 15, 20 ], [ 5, 10, 15, 21 ],
      [ 5, 10, 20, 21 ], [ 5, 11, 12, 14 ], [ 5, 11, 12, 14 ],
      [ 5, 11, 12, 14 ], [ 5, 15, 20, 21 ], [ 6, 7, 9, 20 ], [ 6, 7, 9, 20 ],
      [ 6, 7, 9, 20 ], [ 6, 10, 14, 18 ], [ 6, 11, 16, 21 ],
      [ 6, 12, 13, 15 ], [ 6, 12, 13, 15 ], [ 6, 12, 13, 15 ],
      [ 7, 11, 15, 19 ], [ 7, 12, 17, 21 ], [ 7, 13, 14, 16 ],
      [ 7, 13, 14, 16 ], [ 7, 13, 14, 16 ], [ 8, 12, 16, 20 ],
      [ 8, 13, 18, 21 ], [ 8, 14, 15, 17 ], [ 8, 14, 15, 17 ],
      [ 8, 14, 15, 17 ], [ 9, 14, 19, 21 ], [ 9, 15, 16, 18 ],
      [ 9, 15, 16, 18 ], [ 9, 15, 16, 18 ], [ 10, 15, 20, 21 ],
      [ 10, 16, 17, 19 ], [ 10, 16, 17, 19 ], [ 10, 16, 17, 19 ],
      [ 11, 17, 18, 20 ], [ 11, 17, 18, 20 ], [ 11, 17, 18, 20 ] ],
  blockSizes := [ 4 ],
  isBinary := true,
  allTDesignLambdas := [ 105, 20, 3 ],
```

```
isSimple := false,
autGroup :=
 Group( [ ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20),
    ( 2,18,10,14)( 3,15,19, 7)( 4,12, 8,20)( 5, 9,17,13) ] ) )
```

2 Constructing block designs

2.1 Functions to construct block designs

1 ▶ `BlockDesign( v, B )`

Let v be a positive integer. Then this function returns the block design with point-set $\{1,\ldots,v\}$ and block multiset B, which must be a non-empty sorted list of non-empty sorted lists of elements of $\{1,\ldots,v\}$.

```
gap> BlockDesign( 2, [[1],[1,2],[1,2]] );
rec( isBlockDesign := true, v := 2, blocks := [ [ 1 ], [ 1, 2 ], [ 1, 2 ] ] )
```

2 ▶ `PGPointFlatBlockDesign( n, q, d )`

Let n be a non-negative integer, q a prime-power, and d a non-negative integer less than or equal to n. Then this function returns the block design whose points are the (projective) points of the projective space $PG(n,q)$, and whose blocks are the d-flats of $PG(n,q)$, considering a d-flat as a set of projective points.

Note that the **projective space** $PG(n,q)$ consists of all the subspaces of the vector space $V(n+1,q)$, with the **projective points** being the 1-dimensional subspaces and the **d-flats** being the $(d+1)$-dimensional subspaces.

```
gap> D:=PGPointFlatBlockDesign(3,2,1);;
gap> Print(D,"\n");
rec(
  isBlockDesign := true,
  v := 15,
  pointNames :=
   [ VectorSpace( GF(2), [ [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ] ),
      VectorSpace( GF(2), [ [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ] ),
      VectorSpace( GF(2), [ [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ] ] ),
      VectorSpace( GF(2), [ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ] ),
      VectorSpace( GF(2), [ [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ] ] ),
      VectorSpace( GF(2), [ [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ] ] ),
      VectorSpace( GF(2), [ [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ] ] ),
      VectorSpace( GF(2), [ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ] ] ),
      VectorSpace( GF(2), [ [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ] ] ),
      VectorSpace( GF(2), [ [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ] ] ),
      VectorSpace( GF(2), [ [ Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0 ] ] ),
      VectorSpace( GF(2), [ [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ] ] ),
      VectorSpace( GF(2), [ [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ] ] ),
      VectorSpace( GF(2), [ [ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ] ] ),
      VectorSpace( GF(2), [ [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ] ] ) ],
  blocks := [ [ 1, 2, 3 ], [ 1, 4, 5 ], [ 1, 6, 7 ], [ 1, 8, 9 ],
      [ 1, 10, 11 ], [ 1, 12, 13 ], [ 1, 14, 15 ], [ 2, 4, 6 ], [ 2, 5, 7 ],
      [ 2, 8, 10 ], [ 2, 9, 11 ], [ 2, 12, 14 ], [ 2, 13, 15 ], [ 3, 4, 7 ],
      [ 3, 5, 6 ], [ 3, 8, 11 ], [ 3, 9, 10 ], [ 3, 12, 15 ], [ 3, 13, 14 ],
```

```
        [ 4, 8, 12 ], [ 4, 9, 13 ], [ 4, 10, 14 ], [ 4, 11, 15 ], [ 5, 8, 13 ],
        [ 5, 9, 12 ], [ 5, 10, 15 ], [ 5, 11, 14 ], [ 6, 8, 14 ], [ 6, 9, 15 ],
        [ 6, 10, 12 ], [ 6, 11, 13 ], [ 7, 8, 15 ], [ 7, 9, 14 ],
        [ 7, 10, 13 ], [ 7, 11, 12 ] ] )
gap> Size(AutGroupBlockDesign(D));
20160
```

3 ▶ `TDesignFromTBD( D, t, K )`

Assuming that D is a t-wise balanced t-(v, K, λ) design, with t a positive integer and K a set contained in `[`t`..`D`.v]`, this function returns the t-design obtained by applying the McSorley-Soicher construction to D. The returned design is a t-$(v, K[1], n\lambda)$, where n is the least common multiple of $\{\binom{k-t}{K[1]-t} : k \in K\}$.

```
gap> D:=BlockDesigns(rec(v:=10,blockSizes:=[3,4],
>          tSubsetStructure:=rec(t:=2,lambdas:=[1])))[1];
rec( isBlockDesign := true, v := 10,
  blocks := [ [ 1, 2, 3, 4 ], [ 1, 5, 6, 7 ], [ 1, 8, 9, 10 ], [ 2, 5, 8 ],
      [ 2, 6, 9 ], [ 2, 7, 10 ], [ 3, 5, 9 ], [ 3, 6, 10 ], [ 3, 7, 8 ],
      [ 4, 5, 10 ], [ 4, 6, 8 ], [ 4, 7, 9 ] ],
  tSubsetStructure := rec( t := 2, lambdas := [ 1 ] ), isBinary := true,
  isSimple := true, blockSizes := [ 3, 4 ], blockNumbers := [ 9, 3 ],
  autGroup := Group([ (5,6,7)(8,9,10), (2,3)(5,6)(8,10), (2,3,4)(8,9,10),
      (2,3,4)(5,8,7,9,6,10), (2,6,10)(3,7,9)(4,5,8) ]) )
gap> Dstar:=TDesignFromTBD(D,2,[3,4]);
rec( isBlockDesign := true, v := 10,
  blocks := [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 4 ], [ 1, 5, 6 ],
      [ 1, 5, 7 ], [ 1, 6, 7 ], [ 1, 8, 9 ], [ 1, 8, 10 ], [ 1, 9, 10 ],
      [ 2, 3, 4 ], [ 2, 5, 8 ], [ 2, 5, 8 ], [ 2, 6, 9 ], [ 2, 6, 9 ],
      [ 2, 7, 10 ], [ 2, 7, 10 ], [ 3, 5, 9 ], [ 3, 5, 9 ], [ 3, 6, 10 ],
      [ 3, 6, 10 ], [ 3, 7, 8 ], [ 3, 7, 8 ], [ 4, 5, 10 ], [ 4, 5, 10 ],
      [ 4, 6, 8 ], [ 4, 6, 8 ], [ 4, 7, 9 ], [ 4, 7, 9 ], [ 5, 6, 7 ],
      [ 8, 9, 10 ] ] )
gap> AllTDesignLambdas(Dstar);
[ 30, 9, 2 ]
```

4 ▶ `DualBlockDesign( D )`

Suppose D is a block design for which every point lies on at least one block. Then this function returns the dual of D, the block design in which the roles of points and blocks are interchanged, but incidence (including repeated incidence) stays the same. Note that, since the list of blocks of a block design is always sorted, the block list of the dual of the dual of D may not be equal to the block list of D.

```
gap> D:=BlockDesign(4,[[1,3],[2,3,4],[3,4]]);;
gap> dualD:=DualBlockDesign(D);
rec( isBlockDesign := true, v := 3,
  blocks := [ [ 1 ], [ 1, 2, 3 ], [ 2 ], [ 2, 3 ] ],
  pointNames := [ [ 1, 3 ], [ 2, 3, 4 ], [ 3, 4 ] ] )
gap> DualBlockDesign(dualD).blocks;
[ [ 1, 2 ], [ 2, 3, 4 ], [ 2, 4 ] ]
```

5 ▶ `ComplementBlocksBlockDesign( D )`

Suppose D is a binary incomplete-block design. Then this function returns the block design on the same point-set as D, whose blocks are the complements of those of D (complemented with respect to the point-set).

```
gap> D:=PGPointFlatBlockDesign(2,2,1);
rec( isBlockDesign := true, v := 7,
  pointNames := [ <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)> ],
  blocks := [ [ 1, 2, 3 ], [ 1, 4, 5 ], [ 1, 6, 7 ], [ 2, 4, 6 ],
      [ 2, 5, 7 ], [ 3, 4, 7 ], [ 3, 5, 6 ] ] )
gap> AllTDesignLambdas(D);
[ 7, 3, 1 ]
gap> C:=ComplementBlocksBlockDesign(D);
rec( isBlockDesign := true, v := 7,
  blocks := [ [ 1, 2, 4, 7 ], [ 1, 2, 5, 6 ], [ 1, 3, 4, 6 ], [ 1, 3, 5, 7 ],
      [ 2, 3, 4, 5 ], [ 2, 3, 6, 7 ], [ 4, 5, 6, 7 ] ],
  pointNames := [ <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)>,
      <vector space of dimension 1 over GF(2)> ] )
gap> AllTDesignLambdas(C);
[ 7, 4, 2 ]
```

3 Determining basic properties of block designs

3.1 The functions for basic properties

1 ▶ `IsBlockDesign( obj )`

This boolean function returns `true` if and only if *obj*, which can be an object of arbitrary type, is a block design.

```
gap> IsBlockDesign(5);
false
gap> IsBlockDesign( BlockDesign(2,[[1],[1,2],[1,2]]) );
true
```

2 ▶ `IsBinaryBlockDesign( D )`

This boolean function returns `true` if and only if the block design D is **binary**, that is, if no block of D has a repeated element.

```
gap> IsBinaryBlockDesign( BlockDesign(2,[[1],[1,2],[1,2]]) );
true
gap> IsBinaryBlockDesign( BlockDesign(2,[[1],[1,2],[1,2,2]]) );
false
```

3 ▶ `IsSimpleBlockDesign( D )`

This boolean function returns `true` if and only if the block design D is **simple**, that is, if no block of D is repeated.

```
gap> IsSimpleBlockDesign( BlockDesign(2,[[1],[1,2],[1,2]]) );
false
gap> IsSimpleBlockDesign( BlockDesign(2,[[1],[1,2],[1,2,2]]) );
true
```

4 ▶ `IsConnectedBlockDesign( D )`

This boolean function returns `true` if and only if the block design D is **connected**, that is, if its incidence graph is a connected graph.

```
gap> IsConnectedBlockDesign( BlockDesign(2,[[1],[2]]) );
false
gap> IsConnectedBlockDesign( BlockDesign(2,[[1,2]]) );
true
```

5 ▶ `ReplicationNumber( D )`

If the block design D is equireplicate, then this function returns its replication number; otherwise `fail` is returned.

A block design D is **equireplicate** with **replication number** r if, for every point x of D, r is equal to the sum over the blocks of the multiplicity of x in a block. For a binary block design this is the same as saying that each point x is contained in exactly r blocks.

```
gap> ReplicationNumber(BlockDesign(4,[[1],[1,2],[2,3,3],[4,4]]));
2
gap> ReplicationNumber(BlockDesign(4,[[1],[1,2],[2,3],[4,4]]));
fail
```

6 ► `BlockSizes( D )`

This function returns the set of sizes (actually list-lengths) of the blocks of the block design D.

```
gap> BlockSizes( BlockDesign(3,[[1],[1,2,2],[1,2,3],[2],[3]]) );
[ 1, 3 ]
```

7 ► `BlockNumbers( D )`

Let D be a block design. Then this function returns a list of the same length as `BlockSizes(`D`)`, such that the i-th element of this returned list is the number of blocks of D of size `BlockSizes(`D`)[`i`]`.

```
gap> D:=BlockDesign(3,[[1],[1,2,2],[1,2,3],[2],[3]]);
rec( isBlockDesign := true, v := 3,
  blocks := [ [ 1 ], [ 1, 2, 2 ], [ 1, 2, 3 ], [ 2 ], [ 3 ] ] )
gap> BlockSizes(D);
[ 1, 3 ]
gap> BlockNumbers(D);
[ 3, 2 ]
```

8 ► `TSubsetLambdasVector( D, t )`

Let D be a block design, t a non-negative integer, and v=`D.v`. Then this function returns an integer vector L whose positions correspond to the t-subsets of $\{1,\ldots,v\}$. The i-th element of L is the sum over all blocks B of D of the number of times the i-th t-subset (in lexicographic order) is contained in B. (For example, if $t=2$ and $B=[1,1,2,3,3,4]$, then B contains $[1,2]$ twice, $[1,3]$ four times, $[1,4]$ twice, $[2,3]$ twice, $[2,4]$ once, and $[3,4]$ twice.) In particular, if D is binary then $L[i]$ is simply the number of blocks of D containing the i-th t-subset (in lexicographic order).

```
gap> D:=BlockDesign(3,[[1],[1,2,2],[1,2,3],[2],[3]]);;
gap> TSubsetLambdasVector(D,0);
[ 5 ]
gap> TSubsetLambdasVector(D,1);
[ 3, 4, 2 ]
gap> TSubsetLambdasVector(D,2);
[ 3, 1, 1 ]
gap> TSubsetLambdasVector(D,3);
[ 1 ]
```

9 ► `AllTDesignLambdas( D )`

If the block design D is not a t-design for some $t \geq 0$ then this function returns an empty list. Otherwise D is a binary block design with constant block size k, say, and this function returns a list L of length $T+1$, where T is the maximum $t \leq k$ such that D is a t-design, and, for $i = 1,\ldots,T+1$, $L[i]$ is equal to the (constant) number of blocks of D containing an $(i-1)$-subset of $\{1,\ldots,D\cdot$`v`$\}$.

```
gap> AllTDesignLambdas(PGPointFlatBlockDesign(3,2,1));
[ 35, 7, 1 ]
```

4 Automorphism groups and isomorphism testing for block designs

The functions in this chapter depend on *nauty* via the GRAPE package, which must be fully installed on a computer running UNIX in order for these functions to work.

4.1 Computing automorphism groups

1 ▶ `AutGroupBlockDesign( D )`

This function returns the automorphism group of the block design D. The **automorphism group** $\mathrm{Aut}(D)$ of D is the group consisting of all the permutations of the points $\{1, \ldots, D \cdot \mathrm{v}\}$ which preserve the block-multiset of D.

This function is not yet implemented for non-binary block designs.

This function can also be called via `AutomorphismGroup(`D`)`.

```
gap> D:=PGPointFlatBlockDesign(2,3,1);; # projective plane of order 3
gap> Size(AutGroupBlockDesign(D));
5616
```

4.2 Testing isomorphism

1 ▶ `IsIsomorphicBlockDesign( D1, D2 )`

This boolean function returns `true` if and only if block designs *D1* and *D2* are isomorphic.

This function is not yet implemented for non-binary block designs.

For pairwise isomorphism testing for three or more binary block designs, see 4.2.2.

```
gap> D1:=BlockDesign(3,[[1],[1,2,3],[2]]);;
gap> D2:=BlockDesign(3,[[1],[1,2,3],[3]]);;
gap> IsIsomorphicBlockDesign(D1,D2);
true
gap> D3:=BlockDesign(4,[[1],[1,2,3],[3]]);;
gap> IsIsomorphicBlockDesign(D2,D3);
false
gap> # block designs with different numbers of points are not isomorphic
```

2 ▶ `BlockDesignIsomorphismClassRepresentatives( L )`

Given a list L of binary block designs, this function returns a list consisting of pairwise non-isomorphic elements of L, representing all the isomorphism classes of elements of L. The order of the elements in the returned list may differ from their order in L.

```
gap> D1:=BlockDesign(3,[[1],[1,2,3],[2]]);;
gap> D2:=BlockDesign(3,[[1],[1,2,3],[3]]);;
gap> D3:=BlockDesign(4,[[1],[1,2,3],[3]]);;
gap> BlockDesignIsomorphismClassRepresentatives([D1,D2,D3]);
[ rec( isBlockDesign := true, v := 4, blocks := [ [ 1 ], [ 1, 2, 3 ], [ 3 ] ],
      isBinary := true ),
  rec( isBlockDesign := true, v := 3, blocks := [ [ 1 ], [ 1, 2, 3 ], [ 2 ] ],
      isBinary := true ) ]
```

5 Classifying block designs

This chapter describes the function `BlockDesigns` which can classify block designs with given properties. The possible properties a user can specify are many and varied, and are described below. Depending on the properties, this function can handle block designs with up to about 20 points (sometimes more and sometimes less, depending on the problem).

5.1 The function BlockDesigns

1 ▶ `BlockDesigns(` *param* `)`

This function returns a list *DL* of block designs whose properties are specified by the user in the record *param*. The precise interpretation of the output depends on *param*, described below. Only binary designs are generated by this function, if *param*`.blockDesign` is unbound or is a binary design.

The required components of *param* are `v`, `blockSizes`, and `tSubsetStructure`.

param`.v` must be a positive integer, and specifies that for each block design in *DL*, the points are 1,...,*param*`.v`.

param`.blockSizes` must be a set of positive integers, and specifies that the block sizes of each block design in *DL* will be contained in *param*`.blockSizes`.

param`.tSubsetStructure` must be a record, having components `t`, `partition`, and `lambdas`. Let t be equal to *param*`.tSubsetStructure.t`, *partition* be *param*`.tSubsetStructure.partition`, and *lambdas* be *param*`.tSubsetStructure.lambdas`. Then t must be a non-negative integer, *partition* must be a list of non-empty sets of t-subsets of `[1..`*param*`.v]`, forming an ordered partition of all the t-subsets of `[1..`*param*`.v]`, and *lambdas* must be a list of distinct non-negative integers (not all zero) of the same length as *partition*. This specifies that for each design in *DL*, each t-subset in *partition*`[`i`]` will occur exactly *lambdas*`[`i`]` times, counted over all blocks of the design. For binary designs, this means that each t-subset in *partition*`[`i`]` is contained in exactly *lambdas*`[`i`]` blocks. The `partition` component is optional if *lambdas* has length 1. We require that t is less than or equal to each element of *param*`.blockSizes`, and if *param*`.blockDesign` is bound, then each block of *param*`.blockDesign` must contain at least t distinct elements. Note that if *param*`.tSubsetStructure` is equal to `rec(t:=0,lambdas:=[`b`])`, for some positive integer b, then all that is being specified is that each design in *DL* must have exactly b blocks.

The optional components of *param* are used to specify additional constraints on the designs in *DL* or to change default parameter values. These optional components are `blockDesign`, `r`, `blockNumbers`, `blockIntersectionNumbers`, `blockMaxMultiplicities`, `isoGroup`, `requiredAutSubgroup`, and `isoLevel`.

param`.blockDesign` must be a block design with *param*`.blockDesign.v` equal to *param*`.v`. Then each block multiset of a design in *DL* will be a submultiset of *param*`.blockDesign.blocks` (that is, each block of a design D in *DL* will be a block of *param*`.blockDesign`, and the multiplicity of a block of D will be less than or equal to that block's multiplicity in *param*`.blockDesign`). The `blockDesign` component is useful for the computation of subdesigns, such as parallel classes.

param`.r` must be a positive integer, and specifies that in each design in *DL*, each point will occur exactly *param*`.r` times in the list of blocks. In other words, each design in *DL* will have replication number *param*`.r`.

param`.blockNumbers` must be a list of non-negative integers, the i-th element of which specifies the number of blocks whose size is equal to *param*`.blockSizes[`i`]` (for each design in *DL*). The length of

param.`blockNumbers` must equal that of *param*.`blockSizes`, and at least one entry of *param*.`blockNumbers` must be positive.

param.`blockIntersectionNumbers` must be a symmetric matrix of sets of non-negative integers, the `[`i`][`j`]`-element of which specifies the set of possible sizes for the intersection of a block of size *param*.`blockSizes[`i`]` with one of size *param*.`blockSizes[`j`]` (for each design in DL). In the case of multisets, we take the multiplicity of an element in the intersection to be the minimum of its multiplicities in the multisets being intersected, so, for example, the intersection of `[1,1,1,2,2,3]` with `[1,1,2,2,2,4]` is `[1,1,2,2]`, having size 4. The dimension of *param*.`blockIntersectionNumbers` must equal the length of *param*.`blockSizes`.

param.`blockMaxMultiplicities` must be a list of non-negative integers, the i-th element of which specifies the maximum multiplicity of a block whose size is equal to *param*.`blockSizes[`i`]` (for each design in DL). The length of *param*.`blockMaxMultiplicities` must equal that of *param*.`blockSizes`.

Let G be the automorphism group of *param*.`blockDesign` if bound, and G be `SymmetricGroup(`*param*.`v)` otherwise. Let H be the subgroup of G stabilizing *param*.`tSubsetStructure.partition` (as an ordered list of sets of sets) if bound, and H be equal to G otherwise.

param.`isoGroup` must be a subgroup of H, and specifies that we consider two designs with the required properties to be **equivalent** if their block multisets are in the same orbit of *param*.`isoGroup` (in its action on multisets of multisets of `[1..`*param*.`v]`). The default for *param*.`isoGroup` is H. Thus, if *param*.`blockDesign` is unbound, equivalence is the same as block-design isomorphism for the required designs.

param.`requiredAutSubgroup` must be a subgroup of *param*.`isoGroup`, and specifies that each design in DL must be invariant under *param*.`requiredAutSubgroup` (in its action on multisets of multisets of `[1..`*param*.`v]`). The default for *param*.`requiredAutSubgroup` is the trivial permutation group.

param.`isoLevel` must be 0, 1, or 2 (the default is 2). The value 0 specifies that DL will contain at most one block design, and will contain one block design with the required properties if and only if such a block design exists; the value 1 specifies that DL will contain (perhaps properly) a list of *param*.`isoGroup` orbit-representatives of the required designs; the value 2 specifies that DL will consist precisely of *param*.`isoGroup`-orbit representatives of the required designs.

For an example, we classify up to isomorphism the 2-(15,3,1) designs invariant under a semi-regular group of automorphisms of order 5, and then classify all parallel classes of these designs, up to the action of the automorphism groups of these designs.

```
gap> DL:=BlockDesigns(rec(
>    v:=15,blockSizes:=[3],
>    tSubsetStructure:=rec(t:=2,lambdas:=[1]),
>    requiredAutSubgroup:=
>       Group((1,2,3,4,5)(6,7,8,9,10)(11,12,13,14,15))));;
gap> List(DL,AllTDesignLambdas);
[ [ 35, 7, 1 ], [ 35, 7, 1 ], [ 35, 7, 1 ] ]
gap> List(DL,D->Size(AutGroupBlockDesign(D)));
[ 20160, 5, 60 ]
gap> parclasses:=List(DL,D->
>    BlockDesigns(rec(
>       blockDesign:=D,
>       v:=15,blockSizes:=[3],
>       tSubsetStructure:=rec(t:=1,lambdas:=[1]))));
[ [ rec( isBlockDesign := true, v := 15,
          blocks := [ [ 1, 2, 6 ], [ 3, 4, 8 ], [ 5, 7, 14 ], [ 9, 12, 15 ],
              [ 10, 11, 13 ] ],
          tSubsetStructure := rec( t := 1, lambdas := [ 1 ] ),
          isBinary := true, isSimple := true, blockSizes := [ 3 ],
          blockNumbers := [ 5 ], r := 1,
```

```
            autSubgroup := Group([ (2,6)(3,11)(4,10)(5,14)(8,13)(12,15),
                (2,6)(4,8)(5,12)(7,9)(10,13)(14,15),
                (2,6)(3,12)(4,9)(7,14)(8,15)(11,13),
                (3,12,5)(4,15,7)(8,9,14)(10,11,13),
                (1,6,2)(3,4,8)(5,7,14)(9,12,15)(10,11,13),
                (1,8,11,2,3,10)(4,13,6)(5,15,14,9,7,12) ]) ) ],
  [ rec( isBlockDesign := true, v := 15,
            blocks := [ [ 1, 7, 12 ], [ 2, 8, 13 ], [ 3, 9, 14 ],
                [ 4, 10, 15 ], [ 5, 6, 11 ] ],
            tSubsetStructure := rec( t := 1, lambdas := [ 1 ] ),
            isBinary := true, isSimple := true, blockSizes := [ 3 ],
            blockNumbers := [ 5 ], r := 1,
            autSubgroup := Group([ (1,5,4,3,2)(6,10,9,8,7)(11,15,14,13,12) ]) )
     ],
  [ rec( isBlockDesign := true, v := 15, blocks := [ [ 1, 2, 6 ], [ 3, 10, 13
                  ], [ 4, 11, 12 ], [ 5, 7, 15 ], [ 8, 9, 14 ] ],
            tSubsetStructure := rec( t := 1, lambdas := [ 1 ] ),
            isBinary := true, isSimple := true, blockSizes := [ 3 ],
            blockNumbers := [ 5 ], r := 1,
            autSubgroup := Group([ (1,2)(3,5)(7,10)(8,9)(11,12)(13,15),
                (1,11,8)(2,12,9)(3,13,10)(4,14,6)(5,15,7) ]) ),
       rec( isBlockDesign := true, v := 15,
            blocks := [ [ 1, 8, 11 ], [ 2, 9, 12 ], [ 3, 10, 13 ],
                [ 4, 6, 14 ], [ 5, 7, 15 ] ],
            tSubsetStructure := rec( t := 1, lambdas := [ 1 ] ),
            isBinary := true, isSimple := true, blockSizes := [ 3 ],
            blockNumbers := [ 5 ], r := 1,
            autSubgroup := Group([ (1,2)(3,5)(7,10)(8,9)(11,12)(13,15),
                (1,3,4,2)(6,9,8,10)(11,13,14,12),
                (1,3,5,2,4)(6,8,10,7,9)(11,13,15,12,14),
                (1,11,8)(2,12,9)(3,13,10)(4,14,6)(5,15,7) ]) ) ] ]
gap> List(parclasses,Length);
[ 1, 1, 2 ]
gap> List(parclasses,L->List(L,parclass->Size(parclass.autSubgroup)));
[ [ 360 ], [ 5 ], [ 6, 60 ] ]
```

6 Partitioning block designs

This chapter describes the function `PartitionsIntoBlockDesigns` which can classify partitions of (the block multiset of) a given block design into (the block multisets of) block designs having user-specified properties. We also describe `MakeResolutionsComponent` which is useful for the special case when the desired partitions are resolutions.

6.1 Partitioning a block design into block designs

1 ▶ `PartitionsIntoBlockDesigns(` *param* `)`

Let D equal *param*`.blockDesign`. This function returns a list PL of partitions of (the block multiset of) D. Each element of PL is a record with one component `partition`, and, in most cases, a component `autGroup`. The `partition` component gives a list P of block designs, all with the same point set as D, such that the list of (the block multisets of) the designs in P`.partition` forms a partition of (the block multiset of) D. The component P`.autGroup`, if bound, gives the automorphism group of the partition, which is the stabilizer of the partition in the automorphism group of D. The precise interpretation of the output depends on *param*, described below.

The required components of *param* are `blockDesign`, `v`, `blockSizes`, and `tSubsetStructure`.

param`.blockDesign` is the block design to be partitioned.

param`.v` must be a positive integer, and specifies that for each block design in each partition in PL, the points are 1,...,*param*`.v`. It is required that *param*`.v` be equal to *param*`.blockDesign.v`.

param`.blockSizes` must be a set of positive integers, and specifies that the block sizes of each block design in each partition in PL will be contained in *param*`.blockSizes`.

param`.tSubsetStructure` must be a record, having components `t`, `partition`, and `lambdas`. Let t be equal to *param*`.tSubsetStructure.t`, *partition* be *param*`.tSubsetStructure.partition`, and *lambdas* be *param*`.tSubsetStructure.lambdas`. Then t must be a non-negative integer, *partition* must be a list of non-empty sets of t-subsets of `[1..`*param*`.v]`, forming an ordered partition of all the t-subsets of `[1..`*param*`.v]`, and *lambdas* must be a list of distinct non-negative integers (not all zero) of the same length as *partition*. This specifies that for each design in each partition in PL, each t-subset in *partition*`[`i`]` will occur exactly *lambdas*`[`i`]` times, counted over all blocks of the design. For binary designs, this means that each t-subset in *partition*`[`i`]` is contained in exactly *lambdas*`[`i`]` blocks. The `partition` component is optional if *lambdas* has length 1. We require that t is less than or equal to each element of *param*`.blockSizes`, and that each block of *param*`.blockDesign` contains at least t distinct elements.

The optional components of *param* are used to specify additional constraints on the partitions in PL, or to change default parameter values. These optional components are `r`, `blockNumbers`, `blockIntersectionNumbers`, `blockMaxMultiplicities`, `isoGroup`, `requiredAutSubgroup`, and `isoLevel`. Note that the last three of these optional components refer to the partitions and not to the block designs in a partition.

param`.r` must be a positive integer, and specifies that in each design in each partition in PL, each point must occur exactly *param*`.r` times in the list of blocks.

param`.blockNumbers` must be a list of non-negative integers, the i-th element of which specifies the number of blocks whose size is equal to *param*`.blockSizes[`i`]` (in each design in each partition in PL). The length of

param.`blockNumbers` must equal that of *param*.`blockSizes`, and at least one entry of *param*.`blockNumbers` must be positive.

param.`blockIntersectionNumbers` must be a symmetric matrix of sets of non-negative integers, the `[`i`][`j`]`-element of which specifies the set of possible sizes for the intersection of a block of size *param*.`blockSizes[`i`]` with one of size *param*.`blockSizes[`j`]` (in each design in each partition in PL). In the case of multisets, we take the multiplicity of an element in the intersection to be the minimum of its multiplicities in the multisets being intersected, so, for example, the intersection of `[1,1,1,2,2,3]` with `[1,1,2,2,2,4]` is `[1,1,2,2]`, having size 4. The dimension of *param*.`blockIntersectionNumbers` must equal the length of *param*.`blockSizes`.

param.`blockMaxMultiplicities` must be a list of non-negative integers, the i-th element of which specifies the maximum multiplicity of a block whose whose size is equal to *param*.`blockSizes[`i`]` (for each design in each partition in PL). The length of *param*.`blockMaxMultiplicities` must equal that of *param*.`blockSizes`.

param.`isoGroup` must be a subgroup of the automorphism group of *param*.`blockDesign`. We consider two elements of PL to be **equivalent** if they are in the same orbit of *param*.`isoGroup` (in its action on multisets of block multisets). The default for *param*.`isoGroup` is the automorphism group of *param*.`blockDesign`.

param.`requiredAutSubgroup` must be a subgroup of *param*.`isoGroup`, and specifies that each partition in PL must be invariant under *param*.`requiredAutSubgroup` (in its action on multisets of block multisets). The default for *param*.`requiredAutSubgroup` is the trivial premutation group.

param.`isoLevel` must be 0, 1, or 2 (the default is 2). The value 0 specifies that PL will contain at most one partition, and will contain one partition with the required properties if and only if such a partition exists; the value 1 specifies that PL will contain (perhaps properly) a list of *param*.`isoGroup` orbit-representatives of the required partitions; the value 2 specifies that PL will consist precisely of *param*.`isoGroup`-orbit representatives of the required partitions.

For an example, we first classify up to isomorphism the 2-(15,3,1) designs invariant under a semi-regular group of automorphisms of order 5, and then use `PartitionsIntoBlockDesigns` to classify all the resolutions of these designs, up to the actions of the respective automorphism groups of the designs.

```
gap> DL:=BlockDesigns(rec(
>    v:=15,blockSizes:=[3],
>    tSubsetStructure:=rec(t:=2,lambdas:=[1]),
>    requiredAutSubgroup:=
>       Group((1,2,3,4,5)(6,7,8,9,10)(11,12,13,14,15))));;
gap> List(DL,D->Size(AutGroupBlockDesign(D)));
[ 20160, 5, 60 ]
gap> PL:=PartitionsIntoBlockDesigns(rec(
>       blockDesign:=DL[1],
>       v:=15,blockSizes:=[3],
>       tSubsetStructure:=rec(t:=1,lambdas:=[1])));
[ rec(
      partition := [ rec( isBlockDesign := true, v := 15, blocks := [ [ 1, 2,
                     6 ], [ 3, 4, 8 ], [ 5, 7, 14 ], [ 9, 12, 15 ],
                 [ 10, 11, 13 ] ] ),
          rec( isBlockDesign := true, v := 15, blocks :=
                [ [ 1, 3, 11 ], [ 2, 4, 12 ], [ 5, 6, 8 ], [ 7, 13, 15 ],
                  [ 9, 10, 14 ] ] ),
          rec( isBlockDesign := true, v := 15, blocks :=
                [ [ 1, 4, 14 ], [ 2, 5, 15 ], [ 3, 10, 12 ], [ 6, 7, 11 ],
                  [ 8, 9, 13 ] ] ),
          rec( isBlockDesign := true, v := 15, blocks :=
                [ [ 1, 5, 10 ], [ 2, 9, 11 ], [ 3, 14, 15 ], [ 4, 6, 13 ],
```

```
                  [ 7, 8, 12 ] ] ),
          rec( isBlockDesign := true, v := 15, blocks :=
                [ [ 1, 7, 9 ], [ 2, 8, 10 ], [ 3, 5, 13 ], [ 4, 11, 15 ],
                  [ 6, 12, 14 ] ] ),
          rec( isBlockDesign := true, v := 15, blocks :=
                [ [ 1, 8, 15 ], [ 2, 13, 14 ], [ 3, 6, 9 ], [ 4, 7, 10 ],
                  [ 5, 11, 12 ] ] ),
          rec( isBlockDesign := true, v := 15, blocks :=
                [ [ 1, 12, 13 ], [ 2, 3, 7 ], [ 4, 5, 9 ], [ 6, 10, 15 ],
                  [ 8, 11, 14 ] ] ) ],
      autGroup := Group([ (1,10)(2,11)(3,8)(6,13)(7,14)(12,15),
          (1,13)(2,11)(3,14)(4,5)(6,10)(7,8),
          (1,13,7)(2,11,5)(6,10,14)(9,12,15),
          (2,11,5,15,4,9,12)(3,10,8,14,7,13,6) ]) ),
  rec( partition := [ rec( isBlockDesign := true, v := 15,
              blocks := [ [ 1, 2, 6 ], [ 3, 4, 8 ], [ 5, 7, 14 ],
                  [ 9, 12, 15 ], [ 10, 11, 13 ] ] ),
          rec( isBlockDesign := true, v := 15,
              blocks := [ [ 1, 3, 11 ], [ 2, 4, 12 ], [ 5, 6, 8 ],
                  [ 7, 13, 15 ], [ 9, 10, 14 ] ] ),
          rec( isBlockDesign := true, v := 15,
              blocks := [ [ 1, 4, 14 ], [ 2, 5, 15 ], [ 3, 10, 12 ],
                  [ 6, 7, 11 ], [ 8, 9, 13 ] ] ),
          rec( isBlockDesign := true, v := 15,
              blocks := [ [ 1, 5, 10 ], [ 2, 13, 14 ], [ 3, 6, 9 ],
                  [ 4, 11, 15 ], [ 7, 8, 12 ] ] ),
          rec( isBlockDesign := true, v := 15,
              blocks := [ [ 1, 7, 9 ], [ 2, 8, 10 ], [ 3, 14, 15 ],
                  [ 4, 6, 13 ], [ 5, 11, 12 ] ] ),
          rec( isBlockDesign := true, v := 15,
              blocks := [ [ 1, 8, 15 ], [ 2, 9, 11 ], [ 3, 5, 13 ],
                  [ 4, 7, 10 ], [ 6, 12, 14 ] ] ),
          rec( isBlockDesign := true, v := 15,
              blocks := [ [ 1, 12, 13 ], [ 2, 3, 7 ], [ 4, 5, 9 ],
                  [ 6, 10, 15 ], [ 8, 11, 14 ] ] ) ],
      autGroup := Group([ (1,15)(2,9)(3,4)(5,7)(6,12)(10,13),
          (1,12)(2,9)(3,5)(4,7)(6,15)(8,14),
          (1,14)(2,5)(3,8)(6,7)(9,12)(10,13),
          (1,8,10)(2,5,15)(3,14,13)(4,9,12) ]) ) ]
gap> List(PL,resolution->Size(resolution.autGroup));
[ 168, 168 ]
gap> PL:=PartitionsIntoBlockDesigns(rec(
>       blockDesign:=DL[2],
>       v:=15,blockSizes:=[3],
>       tSubsetStructure:=rec(t:=1,lambdas:=[1])));
[  ]
gap> PL:=PartitionsIntoBlockDesigns(rec(
>       blockDesign:=DL[3],
>       v:=15,blockSizes:=[3],
>       tSubsetStructure:=rec(t:=1,lambdas:=[1])));
[  ]
```

6.2 Computing resolutions

1 ▶ `MakeResolutionsComponent( D )`
▶ `MakeResolutionsComponent( D, isolevel )`

This function computes resolutions of the block design D, and stores the result in D`.resolutions`. If D`.resolutions` already exists then it is ignored and overwritten. This function returns no value.

A **resolution** of a block design D is a partition of the blocks into subsets, each of which forms a partition of the point set. We say that two resolutions R and S of D are **isomorphic** if there is an element g in the automorphism group of D, such that the g-image of R is S. (Isomorphism defines an equivalence relation on the set of resolutions of D.)

The parameter *isolevel* (default 2) determines how many resolutions are computed: *isolevel*=2 means to classify up to isomorphism, *isolevel*=1 means to determine at least one representative from each isomorphism class, and *isolevel*=0 means to determine whether or not D has a resolution.

When this function is finished, D`.resolutions` will have the following three components:

`list`: a list of distinct partitions into block designs forming resolutions of D;

`pairwiseNonisomorphic`: `true`, `false` or `"unknown"`, depending on the resolutions in `list` and what is known. If *isolevel*=0 or *isolevel*=2 then this component will be `true`;

`allClassesRepresented`: `true`, `false` or `"unknown"`, depending on the resolutions in `list` and what is known. If *isolevel*=1 or *isolevel*=2 then this component will be `true`.

Note that D`.resolutions` may be changed to contain more information as a side-effect of other functions in the DESIGN package.

```
gap> L:=BlockDesigns(rec(v:=9,blockSizes:=[3],
>          tSubsetStructure:=rec(t:=2,lambdas:=[1])));;
gap> D:=L[1];;
gap> MakeResolutionsComponent(D);
gap> D;
rec( isBlockDesign := true, v := 9,
  blocks := [ [ 1, 2, 3 ], [ 1, 4, 5 ], [ 1, 6, 7 ], [ 1, 8, 9 ],
      [ 2, 4, 6 ], [ 2, 5, 8 ], [ 2, 7, 9 ], [ 3, 4, 9 ], [ 3, 5, 7 ],
      [ 3, 6, 8 ], [ 4, 7, 8 ], [ 5, 6, 9 ] ],
  tSubsetStructure := rec( t := 2, lambdas := [ 1 ] ), isBinary := true,
  isSimple := true, blockSizes := [ 3 ], blockNumbers := [ 12 ], r := 4,
  autGroup := Group([ (1,2)(5,6)(7,8), (1,3,2)(4,8,7)(5,6,9), (1,2)(4,7)(5,9),
      (1,2)(4,9)(5,7)(6,8), (1,4,8,6,9,2)(3,5,7) ]),
  resolutions := rec( list := [ rec( partition :=
                [ rec( isBlockDesign := true, v := 9,
                      blocks := [ [ 1, 2, 3 ], [ 4, 7, 8 ], [ 5, 6, 9 ] ] ),
                  rec( isBlockDesign := true, v := 9,
                      blocks := [ [ 1, 4, 5 ], [ 2, 7, 9 ], [ 3, 6, 8 ] ] ),
                  rec( isBlockDesign := true, v := 9,
                      blocks := [ [ 1, 6, 7 ], [ 2, 5, 8 ], [ 3, 4, 9 ] ] ),
                  rec( isBlockDesign := true, v := 9,
                      blocks := [ [ 1, 8, 9 ], [ 2, 4, 6 ], [ 3, 5, 7 ] ] ) ],
              autGroup := Group(
                [ (2,3)(4,5)(6,7)(8,9), (1,3,2)(4,8,7)(5,6,9),
                  (1,8,9)(2,4,6)(3,7,5), (1,2)(5,6)(7,8), (1,2)(4,7)(5,9),
                  (1,2,9,6,8,4)(3,7,5) ]) ) ], pairwiseNonisomorphic := true,
      allClassesRepresented := true ) )
```

7 XML I/O of block designs

This chapter describes functions to write and read lists of binary block designs in the

`http://designtheory.org` external representation XML-format (see [CDMS04]).

7.1 Writing lists of block designs and their properties in XML-format

1 ▶ `BlockDesignsToXMLFile(` *filename*`,` *designs* `)`
▶ `BlockDesignsToXMLFile(` *filename*`,` *designs*`,` *include* `)`
▶ `BlockDesignsToXMLFile(` *filename*`,` *designs*`,` *include*`,` *list_id* `)`

This function writes a list of (assumed distinct) binary block designs (given in DESIGN package format) to a file in external representation XML-format (version 2.0).

The parameter *filename* is a string giving the name of the file, and *designs* is a record whose component `list` contains the list of block designs (*designs* can also be a list, in which case it is replaced by `rec(list:=`*designs*`)`.

The record *designs* should have the following components:

`list`: the list of distinct binary block designs in DESIGN package format;

`pairwiseNonisomorphic` (optional): should be `true` or `false` or the string `"unknown"`, specifying the pairwise-nonisomorphism status of the designs in *designs*`.list`;

`infoXML` (optional): should contain a string in XML format for the *info* element of the *list_of_designs* which is written.

The combinatorial and group-theoretical properties output for each design depend on *include*, which should be a list containing zero or more of the strings `"indicators"`, `"resolvable"`, `"combinatorial_properties"`, `"automorphism_group"`, and `"resolutions"`. A shorthand for the list containing all these strings is `"all"`. The default for *include* is the empty list (which is a change from versions of DESIGN previous to 1.1). The strings `"indicators"`, `"combinatorial_properties"`, `"automorphism_group"`, and `"resolutions"` are used to specify that those subtrees of the external representation of each design are to be expanded and written out. In the case of `"resolutions"` being in *include*, **all** resolutions up to isomorphism will be determined and written out. The string `"resolvable"` is used to specify that the `resolvable` indicator must be set (usually this is not forced), if the `indicators` subtree is written out, and also that if a design is resolvable but `"resolutions"` is not in *include*, then one and only one resolution should be written out in the `resolutions` subtree.

If *list_id* is given then the id's of the output designs will be *list_id*`-0`, *list_id*`-1`, *list_id*`-2`, ...

```
gap> D:=[ BlockDesign(3, [[1,2],[1,3]]),
>         BlockDesign(3, [[1,2],[1,2],[2,3]]) ];;
gap> designs:=rec(list:=D, pairwiseNonisomorphic:=true);;
gap> BlockDesignsToXMLFile("example.xml",designs,[],"example");
```

7.2 Reading lists of block designs in XML-format

1 ▶ `BlockDesignsFromXMLFile( filename )`

This function reads a file with name *filename*, containing a list of distinct binary block designs in external representation XML-format, and returns a record *designs* in DESIGN package format containing the essential information in this file.

The record *designs* contains the following components:

`list`: a list of block designs in DESIGN package format of the list of block designs in the file (certain elements such as *statistical_properties* are stored verbatim as strings; certain other elements are not stored since it is usually easier and more reliable to recompute them – this can be done when the block designs are written out in XML format);

`pairwiseNonisomorphic` is set according to the attribute `pairwise_nonisomorphic` of the XML element *list_of_designs*. The component `pairwiseNonisomorphic` is `false` if this attribute is `false`, `true` if this attribute is `true`, and `"unknown"` otherwise;

`infoXML` is bound iff the *info* element occurs as a child of the XML *list_of_designs* element, and if bound, contains this *info* element in a string.

```
gap> BlockDesignsFromXMLFile("example.xml");
rec(
  list := [ rec( isBlockDesign := true, v := 3, id := "example-0", blocks :=
            [ [ 1, 2 ], [ 1, 3 ] ], isBinary := true ),
      rec( isBlockDesign := true, v := 3, id := "example-1",
          blocks := [ [ 1, 2 ], [ 1, 2 ], [ 2, 3 ] ], isBinary := true ) ],
  pairwiseNonisomorphic := true )
```

Bibliography

[CDMS04] Peter J. Cameron, Peter Dobcsányi, John P. Morgan, and Leonard H. Soicher. *The external representation of block designs (version 2.0)*, 2004.

`http://designtheory.org/library/extrep/`.

[LN04] Frank Lübeck and Max Neunhöffer. *The GAPDoc package for GAP (version 0.9999)*, 2004.

`http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc/`.

[McK96] Brendan D. McKay. *The nauty package (version 2.0b5)*, 1996.

`http://cs.anu.edu.au/people/bdm/nauty/`.

[Soi04] Leonard H. Soicher. *The GRAPE package for GAP (version 4.2)*, 2004.

`http://www.maths.qmul.ac.uk/~leonard/grape/`.