DESIGN

A Package for GAP

by

Leonard H. Soicher

School of Mathematical Sciences
Queen Mary University of London

Contents

1 Design

This manual describes the DESIGN 1.8.1 package for GAP. The DESIGN package is for constructing, classifying, partitioning, and studying block designs.

 DESIGN is part of a wider project, which received EPSRC funding under grant GR/R29659/01, to provide a web-based resource for design theory; see

`http://designtheory.org` and [BCD+06]. The development of DESIGN was also partially supported by EPSRC grant EP/M022641/1 (CoDiMa: a Collaborative Computational Project in the area of Computational Discrete Mathematics).

Please reference your use of the DESIGN package in a published work as follows:

L.H. Soicher, The DESIGN package for GAP, Version 1.8.1, 2024,

`https://gap-packages.github.io/design` .

For questions, remarks, suggestions, and issues, please use the issue tracker at

`https://github.com/gap-packages/design/issues` .

1.1 Installing the DESIGN Package

The DESIGN package is included in the standard GAP distribution. You only need to download and install DESIGN if you need to install the package locally or are installing an upgrade of DESIGN to an existing installation of GAP (see the main GAP reference section 76.1). If you do need to download DESIGN, you can find an archive file for the latest release at

`https://github.com/gap-packages/design/releases` . This archive file can then be downloaded and unpacked in the `pkg` subdirectory of an appropriate GAP root directory (see the main GAP reference section 9.2).

The DESIGN package is written entirely in GAP code, and requires no further installation. However, DESIGN makes use of the GRAPE package [Soi24a], which must be fully installed.

1.2 Loading DESIGN

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

```
LoadPackage("design");
```

1.3 The structure of a block design in DESIGN

A **block design** is an ordered 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`, `efficiency`, `id`, `statistical_propertiesXML`, 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) design 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 $*$-construction of [MS07] 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.

Further extended examples of the use of the DESIGN package can be found in [Soi13] and [Soi24b].

```
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,2,4);;
gap> AllTDesignLambdas(Dstar);
[ 105, 20, 3 ]
gap> IsSimpleBlockDesign(Dstar);
false
```

```
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 ] ],
  autGroup := Group( [ ( 2,14,10,18)( 3, 7,19,15)( 4,20, 8,12)( 5,13,17, 9),
      ( 1,17, 5, 9)( 2,10,14, 6)( 4,16,12,20)( 7,15,19,11),
      ( 1,18,19,12)( 2,11, 8, 9)( 3, 4,17, 6)( 5,10,15,20)( 7,16,13,14) ] ),
  blockSizes := [ 4 ],
  isBinary := true,
  allTDesignLambdas := [ 105, 20, 3 ],
```

```
    isSimple := false )
```

2 Information from design parameters

2.1 Information from t-design parameters

For t a non-negative integer and v, k, λ positive integers with $t \leq k \leq v$, a t-**design** with **parameters** t, v, k, λ, or a t-(v, k, λ) **design**, is a binary block design with exactly v points, such that each block has size k and each t-subset of the points is contained in exactly λ blocks.

1 ► `TDesignLambdas( t, v, k, lambda )`

A t-(v, k, λ) design is also an s-(v, k, λ_s) design for $0 \leq s \leq t$, where $\lambda_s = \lambda \binom{v-s}{t-s} / \binom{k-s}{t-s}$.

Given a non-negative integer t, and positive integers v, k, *lambda*, with $t \leq k \leq v$, this function returns a length $t+1$ list whose $(s+1)$-st element is λ_s as defined above, if all the λ_s are integers. Otherwise, `fail` is returned.

```
gap> TDesignLambdas(5,24,8,1);
[ 759, 253, 77, 21, 5, 1 ]
```

2 ► `TDesignLambdaMin( t, v, k )`

Given a non-negative integer t, and positive integers v and k, with $t \leq k \leq v$, this function returns the minimum positive *lambda* such that `TDesignLambdas( t, v, k, lambda )` does not return `fail`.

See 2.1.1.

```
gap> TDesignLambdaMin(5,24,8);
1
gap> TDesignLambdaMin(2,12,4);
3
```

3 ► `TDesignIntersectionTriangle( t, v, k, lambda )`

Suppose D is a t-(v,k,*lambda*) design, let i and j be non-negative integers with $i + j \leq t$, and suppose X and Y are disjoint subsets of the points of D, such that X and Y have respective sizes i and j. The (i, j)-**intersection number** is the number of blocks of D that contain X and are disjoint from Y (this number depends only on t, v, k, *lambda*, i and j).

Given a non-negative integer t, and positive integers v, k and *lambda*, with $t \leq k \leq v$, this function returns the ***t*-design intersection triangle**, which is a two dimensional array whose $(i+1, j+1)$-entry is the (i, j)-intersection number for a t-(v,k,*lambda*) design (assuming such a design exists), such that $i, j \geq 0$, $i + j \leq t$. This function returns `fail` if `TDesignLambdas(`t`,`v`,`k`,`*lambda*`)` does. When *lambda* $= 1$, then more information can be obtained using 2.1.4.

```
gap> TDesignLambdas(2,12,4,3);
[ 33, 11, 3 ]
gap> TDesignIntersectionTriangle(2,12,4,3);
[ [ 33, 22, 14 ], [ 11, 8 ], [ 3 ] ]
gap> TDesignLambdas(2,12,4,2);
fail
gap> TDesignIntersectionTriangle(2,12,4,2);
fail
```

4 ► `SteinerSystemIntersectionTriangle( t, v, k )`

A **Steiner system** is a t-(v,k,1) design, and in this case it is possible to extend the notion of intersection triangle defined in 2.1.3.

Suppose D is a t-(v,k,1) design, with B a block of D, let i and j be non-negative integers with $i+j \leq k$, and suppose X and Y are disjoint subsets of B, such that X and Y have respective sizes i and j. The (i,j)-**intersection number** is the number of blocks of D that contain X and are disjoint from Y (this number depends only on t, v, k, i and j). Note that when $i+j \leq t$, this intersection number is the same as that defined in 2.1.3 for the general t-design case.

Given a non-negative integer t, and positive integers v and k, with $t \leq k \leq v$, this function returns the **Steiner system intersection triangle**, which is a two dimensional array whose $(i+1,j+1)$-entry is the (i,j)-intersection number for a t-(v,k,1) design (assuming such a design exists), such that $i,j \geq 0$, $i+j \leq k$. This function returns `fail` if `TDesignLambdas(`t`,`v`,`k`,1)` does.

See also 2.1.3.

```
gap> SteinerSystemIntersectionTriangle(5,24,8);
[ [ 759, 506, 330, 210, 130, 78, 46, 30, 30 ],
  [ 253, 176, 120, 80, 52, 32, 16, 0 ], [ 77, 56, 40, 28, 20, 16, 16 ],
  [ 21, 16, 12, 8, 4, 0 ], [ 5, 4, 4, 4, 4 ], [ 1, 0, 0, 0 ], [ 1, 0, 0 ],
  [ 1, 0 ], [ 1 ] ]
gap> TDesignIntersectionTriangle(5,24,8,1);
[ [ 759, 506, 330, 210, 130, 78 ], [ 253, 176, 120, 80, 52 ],
  [ 77, 56, 40, 28 ], [ 21, 16, 12 ], [ 5, 4 ], [ 1 ] ]
```

5 ► `TDesignBlockMultiplicityBound( t, v, k, lambda )`

Given a non-negative integer t, and positive integers v, k and *lambda*, with $t \leq k \leq v$, this function returns a non-negative integer which is an upper bound on the multiplicity of any block in any t-(v,k,*lambda*) design (the **multiplicity** of a block in a block design is the number of times that block occurs in the block list). In particular, if the value 0 is returned, then this implies that a t-(v,k,*lambda*) design does not exist.

Although our bounds are reasonably good, we do not claim that the returned bound m is always achieved; that is, there may not exist a t-(v,k,*lambda*) design having a block with multiplicity m.

See also 2.1.6.

```
gap> TDesignBlockMultiplicityBound(5,16,7,5);
2
gap> TDesignBlockMultiplicityBound(2,36,6,1);
0
gap> TDesignBlockMultiplicityBound(2,36,6,2);
2
gap> TDesignBlockMultiplicityBound(2,15,5,2);
0
gap> TDesignBlockMultiplicityBound(2,15,5,4);
2
gap> TDesignBlockMultiplicityBound(2,11,4,6);
3
```

6 ► `ResolvableTDesignBlockMultiplicityBound( t, v, k, lambda )`

A **resolution** of a block design is a partition of the blocks into subsets, each of which forms a partition of the point set, and a block design is **resolvable** if it has a resolution.

Given a non-negative integer t, and positive integers v, k and *lambda*, with $t \leq k \leq v$, this function returns a non-negative integer which is an upper bound on the multiplicity of any block in any resolvable t-(v,k,*lambda*) design (the

multiplicity of a block in a block design is the number of times that block occurs in the block list). In particular, if the value 0 is returned, then this implies that a resolvable *t*-(*v*,*k*,*lambda*) design does not exist.

Although our bounds are reasonably good, we do not claim that the returned bound *m* is always achieved; that is, there may not exist a resolvable *t*-(*v*,*k*,*lambda*) design having a block with multiplicity *m*.

See also 2.1.5.

```
gap> ResolvableTDesignBlockMultiplicityBound(5,12,6,1);
1
gap> ResolvableTDesignBlockMultiplicityBound(2,21,7,3);
0
gap> TDesignBlockMultiplicityBound(2,21,7,3);
1
gap> ResolvableTDesignBlockMultiplicityBound(2,12,4,3);
1
gap> TDesignBlockMultiplicityBound(2,12,4,3);
2
```

2.2 Information from (mixed) orthogonal array parameters

For integers N, k, s, t, with $N, k > 0$, $s > 1$, and $0 \leq t \leq k$, an **orthogonal array** $\mathrm{OA}(N, k, s, t)$ is an $N \times k$ array, in which the entries come from a set of size s of **symbols**, with the property that in any $N \times t$ subarray, every possible t-tuple of symbols occurs as a row equally often (which must be N/s^t times).

For integers $N, k_1, \ldots, k_w, s_1, \ldots, s_w, t$, with $w, N, k_1, \ldots, k_w > 0$, $s_1, \ldots, s_w > 1$, and $0 \leq t \leq k := k_1 + \cdots + k_w$, a **mixed orthogonal array** $\mathrm{OA}(N, s_1^{k_1}, \ldots, s_w^{k_w}, t)$ is an $N \times k$ array, in which the entries in the first k_1 columns come from a set of symbols of size s_1, the entries in the next k_2 columns come from a set of symbols of size s_2, and so on, with the property that in any $N \times t$ subarray, every possible t-tuple of symbols occurs as a row equally often. Clearly, a mixed orthogonal array $\mathrm{OA}(N, s^k, t)$ is the same thing as an orthogonal array $\mathrm{OA}(N, k, s, t)$.

The rows of an orthogonal array or mixed orthogonal array are called **runs**. The **multiplicity** of a run is the number of times it appears as a row in the array.

1 ▶ `OARunMultiplicityBound( N, k, s, t )`

Suppose N, k, s, and t are integers, with N, k positive, $s > 1$, and $0 \leq t \leq k$. Then this function returns an upper bound on the multiplicity of any run in an orthogonal array $\mathrm{OA}(N, k, s, t)$.

An upper bound on the multiplicity of a run in a mixed orthogonal array can be obtained by replacing k and s by non-empty lists of the same length, w say, of positive integers, such that $0 \leq t \leq$ `Sum(k)`, and each entry of s is at least 2. Then the function returns an upper bound on the multiplicity of any run in a mixed orthogonal array $\mathrm{OA}(N, s[1]^{k[1]}, \cdots, s[w]^{k[w]}, t)$.

If the value 0 is returned, then this implies that an orthogonal array or mixed orthogonal array with the given parameters does not exist.

We do not claim that the returned upper bound m is achieved; that is, there may well be no (mixed) orthogonal array with the given parameters having a run with multiplicity m.

```
gap> OARunMultiplicityBound(81,14,3,3);
1
gap> OARunMultiplicityBound(81,15,3,3);
0
gap> OARunMultiplicityBound(36,[18,1,1],[2,3,6],2);
1
gap> OARunMultiplicityBound(72,7,6,2);
2
gap> OARunMultiplicityBound(72,8,6,2);
1
```

2.3 Block intersection polynomials

In [CS07], Cameron and Soicher introduce block intersection polynomials and their applications to the study of block designs. Here we give functions to construct and analyze block intersection polynomials.

1 ▶ `BlockIntersectionPolynomial(`x`, `m`, `*lambdavec*` )`

For k a non-negative integer, define the polynomial $P(x,k) = x(x-1)\cdots(x-k+1)$. Let s and t be non-negative integers, with $s \geq t$, and let $m_0,\ldots,m_s$ and $\lambda_0,\ldots,\lambda_t$ be rational numbers. Then the **block intersection polynomial** for the sequences $[m_0,\ldots,m_s]$, $[\lambda_0,\ldots,\lambda_t]$ is defined to be

$$\sum_{j=0}^{t}\binom{t}{j}P(-x,t-j)[P(s,j)\lambda_j - \sum_{i=j}^{s}P(i,j)m_i],$$

and is denoted by $B(x,[m_0,\ldots,m_s],[\lambda_0,\ldots,\lambda_t])$.

Now suppose x is an indeterminate over the rationals, and m and *lambdavec* are non-empty lists of rational numbers, such that the length of *lambdavec* is not greater than that of m. Then this function returns the block intersection polynomial $B(x,m,$ *lambdavec*$)$.

The importance of a block intersection polynomial is as follows. Let $D=(V,\mathscr{B})$ be a block design, let $S\subseteq V$, with $s=|S|$, and for $i=0,\ldots,s$, suppose that m_i is a non-negative integer with $m_i \leq n_i$, where n_i is the number of blocks intersecting S in exactly i points. Let t be a non-negative **even** integer with $t\leq s$, and suppose that, for $j=0\ldots,t$, we have $\lambda_j = 1/\binom{s}{j}\sum_{T\subseteq S,|T|=j}\lambda_T$, where λ_T is the number of blocks of D containing T. Then the block intersection polynomial $B(x)=B(x,[m_0,\ldots,m_s],[\lambda_0,\ldots,\lambda_t])$ is a polynomial with integer coefficients, and $B(n)\geq 0$ for every integer n. (These conditions can be checked using the function 2.3.2.) In addition, if $B(n)=0$ for some integer n, then $m_i=n_i$ for $i\notin\{n,n+1,\ldots,n+t-1\}$.

For more information on block intersection polynomials and their applications, see [CS07] and [Soi10].

```
gap> x:=Indeterminate(Rationals,1);
x_1
gap> m:=[0,0,0,0,0,0,0,1];;
gap> lambdavec:=TDesignLambdas(6,14,7,4);
[ 1716, 858, 396, 165, 60, 18, 4 ]
gap> B:=BlockIntersectionPolynomial(x,m,lambdavec);
1715*x_1^6-10269*x_1^5+34685*x_1^4-69615*x_1^3+84560*x_1^2-56196*x_1+15120
gap> Factors(B);
[ 1715*x_1-1715,
  x_1^5-1222/245*x_1^4+3733/245*x_1^3-6212/245*x_1^2+5868/245*x_1-432/49 ]
gap> Value(B,1);
0
```

2 ▶ `BlockIntersectionPolynomialCheck(`m`, `*lambdavec*`)`

Suppose m is a list of non-negative integers, and *lambdavec* is a list of non-negative rational numbers, with the length of *lambdavec* odd and not greater than the length of m.

Then, for x an indeterminate over the rationals, this function checks whether `BlockIntersectionPolynomial(`x`,`m`,`*lambdavec*`)` is a polynomial over the integers and has a non-negative value at each integer. The function returns `true` if this is so; else `false` is returned.

See also 2.3.1.

```
gap> m:=[0,0,0,0,0,0,0,1];;
gap> lambdavec:=TDesignLambdas(6,14,7,4);
[ 1716, 858, 396, 165, 60, 18, 4 ]
gap> BlockIntersectionPolynomialCheck(m,lambdavec);
true
gap> m:=[1,0,0,0,0,0,0,1];;
gap> BlockIntersectionPolynomialCheck(m,lambdavec);
false
```

3 Constructing block designs

3.1 Functions to construct block designs

1 ▶ `BlockDesign( v, B )`
▶ `BlockDesign( v, B, G )`

Let v be a positive integer and B a non-empty list of non-empty sorted lists of elements of $\{1,\ldots,v\}$.

The first version of this function returns the block design with point-set $\{1,\ldots,v\}$ and block multiset C, where C is `SortedList(`B`)`.

For the second version of this function, we require G to be a group of permutations of $\{1,\ldots,v\}$, and the function returns the block design with point-set $\{1,\ldots,v\}$ and block multiset C, where C is the sorted list of the concatenation of the G-orbits of the elements of B.

```
gap> BlockDesign( 2, [[1,2],[1],[1,2]] );
rec( isBlockDesign := true, v := 2, blocks := [ [ 1 ], [ 1, 2 ], [ 1, 2 ] ] )
gap> D:=BlockDesign(7, [[1,2,4]], Group((1,2,3,4,5,6,7)));
rec( isBlockDesign := true, v := 7,
  blocks := [ [ 1, 2, 4 ], [ 1, 3, 7 ], [ 1, 5, 6 ], [ 2, 3, 5 ],
      [ 2, 6, 7 ], [ 3, 4, 6 ], [ 4, 5, 7 ] ],
  autSubgroup := Group([ (1,2,3,4,5,6,7) ]) )
gap> AllTDesignLambdas(D);
[ 7, 3, 1 ]
```

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

Let n be positive 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 points of the affine space $AG(n,q)$, and whose blocks are the d-flats of $AG(n,q)$, considering a d-flat as a set of points.

Note that the **affine space** $AG(n,q)$ consists of all the cosets of all the subspaces of the vector space $V(n,q)$, with the **points** being the cosets of the 0-dimensional subspace and the d**-flats** being the cosets of the d-dimensional subspaces. As is usual, we identify the points with the vectors in $V(n,q)$, and these vectors are given as the point-names.

```
gap> D:=AGPointFlatBlockDesign(2,4,1);
rec( isBlockDesign := true, v := 16,
  blocks := [ [ 1, 2, 3, 4 ], [ 1, 5, 9, 13 ], [ 1, 6, 11, 16 ],
      [ 1, 7, 12, 14 ], [ 1, 8, 10, 15 ], [ 2, 5, 12, 15 ], [ 2, 6, 10, 14 ],
      [ 2, 7, 9, 16 ], [ 2, 8, 11, 13 ], [ 3, 5, 10, 16 ], [ 3, 6, 12, 13 ],
      [ 3, 7, 11, 15 ], [ 3, 8, 9, 14 ], [ 4, 5, 11, 14 ], [ 4, 6, 9, 15 ],
      [ 4, 7, 10, 13 ], [ 4, 8, 12, 16 ], [ 5, 6, 7, 8 ], [ 9, 10, 11, 12 ],
      [ 13, 14, 15, 16 ] ],
  autSubgroup := Group([ (5,9,13)(6,10,14)(7,11,15)(8,12,16),
      (2,5,6)(3,9,11)(4,13,16)(7,14,12)(8,10,15),
      (1,5)(2,6)(3,7)(4,8)(9,13)(10,14)(11,15)(12,16),
      (3,4)(7,8)(9,13)(10,14)(11,16)(12,15) ]),
```

```
    pointNames := [ [ 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0 ], [ 0*Z(2), Z(2^2) ],
        [ 0*Z(2), Z(2^2)^2 ], [ Z(2)^0, 0*Z(2) ], [ Z(2)^0, Z(2)^0 ],
        [ Z(2)^0, Z(2^2) ], [ Z(2)^0, Z(2^2)^2 ], [ Z(2^2), 0*Z(2) ],
        [ Z(2^2), Z(2)^0 ], [ Z(2^2), Z(2^2) ], [ Z(2^2), Z(2^2)^2 ],
        [ Z(2^2)^2, 0*Z(2) ], [ Z(2^2)^2, Z(2)^0 ], [ Z(2^2)^2, Z(2^2) ],
        [ Z(2^2)^2, Z(2^2)^2 ] ] )
gap> AllTDesignLambdas(D);
[ 20, 5, 1 ]
```

3 ▶ `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);
rec( isBlockDesign := true, v := 15,
  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 ] ],
  autSubgroup := Group([ (8,12)(9,13)(10,14)(11,15),
      (1,2,4,8)(3,6,12,9)(5,10)(7,14,13,11) ]),
  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)>,
      <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)>,
      <vector space of dimension 1 over GF(2)> ] )
gap> AllTDesignLambdas(D);
[ 35, 7, 1 ]
```

4 ▶ `WittDesign( n )`

Suppose $n \in \{9, 10, 11, 12, 21, 22, 23, 24\}$.

If $n = 24$ then this function returns the **large Witt design** W_{24}, the unique (up to isomorphism) 5-(24,8,1) design. If $n = 24 - i$, where $i \in \{1,2,3\}$, then the i-fold point-derived design of W_{24} is returned; this is the unique (up to isomorphism) $(5-i)$-$(24-i, 8-i, 1)$ design.

If $n = 12$ then this function returns the **small Witt design** W_{12}, the unique (up to isomorphism) 5-(12,6,1) design. If $n = 12 - i$, where $i \in \{1, 2, 3\}$, then the i-fold point-derived design of W_{12} is returned; this is the unique (up to isomorphism) $(5 - i)$-$(12 - i, 6 - i, 1)$ design.

```
gap> W24:=WittDesign(24);;
gap> AllTDesignLambdas(W24);
[ 759, 253, 77, 21, 5, 1 ]
gap> DisplayCompositionSeries(AutomorphismGroup(W24));
G (3 gens, size 244823040)
  M(24)
1 (0 gens, size 1)
gap> W10:=WittDesign(10);;
gap> AllTDesignLambdas(W10);
[ 30, 12, 4, 1 ]
gap> DisplayCompositionSeries(AutomorphismGroup(W10));
G (4 gens, size 1440)
  Z(2)
S (4 gens, size 720)
  Z(2)
S (3 gens, size 360)
  A(6) ~ A(1,9) = L(2,9) ~ B(1,9) = O(3,9) ~ C(1,9) = S(2,9) ~ 2A(1,9) = U(2,\
9)
1 (0 gens, size 1)
```

5 ► `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 ] ]
```

6 ► `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 ]
```

7 ► `DeletedPointsBlockDesign( D, Y )`

Suppose D is a block design and Y is a proper subset of the point-set of D.

Then this function returns the block design DP obtained from D by deleting the points in Y from the point-set, and from each block. It is an error if the resulting design contains an empty block. The points of DP are relabelled $1, 2, \cdots$, preserving the order of the corresponding points of D; the point-names of DP (listed in DP`.pointNames`) are those of these corresponding points of D.

```
gap> D:=BlockDesigns(rec(v:=11,blockSizes:=[5],
>       tSubsetStructure:=rec(t:=2,lambdas:=[2])))[1];
rec( isBlockDesign := true, v := 11,
  blocks := [ [ 1, 2, 3, 4, 5 ], [ 1, 2, 9, 10, 11 ], [ 1, 3, 6, 7, 9 ],
      [ 1, 4, 7, 8, 10 ], [ 1, 5, 6, 8, 11 ], [ 2, 3, 6, 8, 10 ],
      [ 2, 4, 6, 7, 11 ], [ 2, 5, 7, 8, 9 ], [ 3, 4, 8, 9, 11 ],
      [ 3, 5, 7, 10, 11 ], [ 4, 5, 6, 9, 10 ] ],
  tSubsetStructure := rec( t := 2, lambdas := [ 2 ] ), isBinary := true,
  isSimple := true, blockSizes := [ 5 ], blockNumbers := [ 11 ], r := 5,
  autGroup := Group([ (2,4)(3,5)(7,11)(8,9), (1,3)(2,5)(7,9)(10,11),
      (1,5,3)(6,11,7)(8,10,9), (1,10,5,2,11,3)(4,9,7)(6,8) ]) )
gap> AllTDesignLambdas(D);
[ 11, 5, 2 ]
gap> DP:=DeletedPointsBlockDesign(D,[5,8]);
rec( isBlockDesign := true, v := 9,
  blocks := [ [ 1, 2, 3, 4 ], [ 1, 2, 7, 8, 9 ], [ 1, 3, 5, 6, 7 ],
      [ 1, 4, 6, 8 ], [ 1, 5, 9 ], [ 2, 3, 5, 8 ], [ 2, 4, 5, 6, 9 ],
      [ 2, 6, 7 ], [ 3, 4, 7, 9 ], [ 3, 6, 8, 9 ], [ 4, 5, 7, 8 ] ],
  pointNames := [ 1, 2, 3, 4, 6, 7, 9, 10, 11 ] )
gap> PairwiseBalancedLambda(DP);
2
```

8 ► `DeletedBlocksBlockDesign( D, Y)`

Suppose D is a block design, and Y is a proper sublist of the block-list of D (Y need not be sorted).

Then this function returns the block design obtained from D by deleting the blocks in Y (counting repeats) from the block-list of D.

```
gap> D:=BlockDesign(7,[[1,2,4],[1,2,4]],Group((1,2,3,4,5,6,7)));
rec( isBlockDesign := true, v := 7,
  blocks := [ [ 1, 2, 4 ], [ 1, 2, 4 ], [ 1, 3, 7 ], [ 1, 3, 7 ],
      [ 1, 5, 6 ], [ 1, 5, 6 ], [ 2, 3, 5 ], [ 2, 3, 5 ], [ 2, 6, 7 ],
      [ 2, 6, 7 ], [ 3, 4, 6 ], [ 3, 4, 6 ], [ 4, 5, 7 ], [ 4, 5, 7 ] ],
  autSubgroup := Group([ (1,2,3,4,5,6,7) ]) )
gap> DeletedBlocksBlockDesign(D,[[2,3,5],[2,3,5],[4,5,7]]);
rec( isBlockDesign := true, v := 7,
  blocks := [ [ 1, 2, 4 ], [ 1, 2, 4 ], [ 1, 3, 7 ], [ 1, 3, 7 ],
      [ 1, 5, 6 ], [ 1, 5, 6 ], [ 2, 6, 7 ], [ 2, 6, 7 ], [ 3, 4, 6 ],
      [ 3, 4, 6 ], [ 4, 5, 7 ] ] )
```

9 ► `AddedPointBlockDesign( D, Y )`
► `AddedPointBlockDesign( D, Y, pointname )`

Suppose D is a block design, and Y is a sublist of the block-list of D (Y need not be sorted).

Then this function returns the block design obtained from D by adding the new point `D.v+1` to the point-set, and adding this new point (once) to each block of Y (where repeats count).

The optional parameter *pointname* specifies a point-name for the new point.

```
gap> D:=BlockDesign(7,[[1,2,4],[1,2,4]],Group((1,2,3,4,5,6,7)));
rec( isBlockDesign := true, v := 7,
  blocks := [ [ 1, 2, 4 ], [ 1, 2, 4 ], [ 1, 3, 7 ], [ 1, 3, 7 ],
      [ 1, 5, 6 ], [ 1, 5, 6 ], [ 2, 3, 5 ], [ 2, 3, 5 ], [ 2, 6, 7 ],
      [ 2, 6, 7 ], [ 3, 4, 6 ], [ 3, 4, 6 ], [ 4, 5, 7 ], [ 4, 5, 7 ] ],
  autSubgroup := Group([ (1,2,3,4,5,6,7) ]) )
gap> AddedPointBlockDesign(D,[[2,3,5],[2,3,5],[4,5,7]],"infinity");
rec( isBlockDesign := true, v := 8,
  blocks := [ [ 1, 2, 4 ], [ 1, 2, 4 ], [ 1, 3, 7 ], [ 1, 3, 7 ],
      [ 1, 5, 6 ], [ 1, 5, 6 ], [ 2, 3, 5, 8 ], [ 2, 3, 5, 8 ], [ 2, 6, 7 ],
      [ 2, 6, 7 ], [ 3, 4, 6 ], [ 3, 4, 6 ], [ 4, 5, 7 ], [ 4, 5, 7, 8 ] ],
  pointNames := [ 1, 2, 3, 4, 5, 6, 7, "infinity" ] )
```

10 ► `AddedBlocksBlockDesign( D, Y )`

Suppose Y is a list of multisets of points of the block design D. Then this function returns a new block design, whose point-set is that of D, and whose block list is that of D with the elements of Y (including repeats) added.

```
gap> D:=BlockDesign(7,[[1,2,4]],Group((1,2,3,4,5,6,7)));
rec( isBlockDesign := true, v := 7,
  blocks := [ [ 1, 2, 4 ], [ 1, 3, 7 ], [ 1, 5, 6 ], [ 2, 3, 5 ],
      [ 2, 6, 7 ], [ 3, 4, 6 ], [ 4, 5, 7 ] ],
  autSubgroup := Group([ (1,2,3,4,5,6,7) ]) )
gap> AddedBlocksBlockDesign(D,D.blocks);
rec( isBlockDesign := true, v := 7,
  blocks := [ [ 1, 2, 4 ], [ 1, 2, 4 ], [ 1, 3, 7 ], [ 1, 3, 7 ],
      [ 1, 5, 6 ], [ 1, 5, 6 ], [ 2, 3, 5 ], [ 2, 3, 5 ], [ 2, 6, 7 ],
      [ 2, 6, 7 ], [ 3, 4, 6 ], [ 3, 4, 6 ], [ 4, 5, 7 ], [ 4, 5, 7 ] ] )
```

11 ► `DerivedBlockDesign( D, x )`

Suppose D is a block design, and x is a point or block of D. Then this function returns the **derived design** DD of D, with respect to x.

If x is a point then DD is the block design whose blocks are those of D containing x, but with x deleted from these blocks, and the points of DD are those which occur in some block of DD.

If x is a block, then the points of DD are the points in x, and the blocks of DD are the blocks of D other than x containing at least one point of x, but with all points not in x deleted from these blocks. Note that any repeat of x, but not x itself, is a block of DD.

It is an error if the resulting block design DD has no blocks or an empty block.

The points of DD are relabelled $1, 2, \cdots$, preserving the order of the corresponding points of D; the point-names of DD (listed in DD`.pointNames`) are those of these corresponding points of D.

```
gap> D:=BlockDesigns(rec(v:=11,blockSizes:=[5],
>       tSubsetStructure:=rec(t:=2,lambdas:=[2])))[1];
rec( isBlockDesign := true, v := 11,
  blocks := [ [ 1, 2, 3, 4, 5 ], [ 1, 2, 9, 10, 11 ], [ 1, 3, 6, 7, 9 ],
      [ 1, 4, 7, 8, 10 ], [ 1, 5, 6, 8, 11 ], [ 2, 3, 6, 8, 10 ],
      [ 2, 4, 6, 7, 11 ], [ 2, 5, 7, 8, 9 ], [ 3, 4, 8, 9, 11 ],
      [ 3, 5, 7, 10, 11 ], [ 4, 5, 6, 9, 10 ] ],
  tSubsetStructure := rec( t := 2, lambdas := [ 2 ] ), isBinary := true,
  isSimple := true, blockSizes := [ 5 ], blockNumbers := [ 11 ], r := 5,
  autGroup := Group([ (2,4)(3,5)(7,11)(8,9), (1,3)(2,5)(7,9)(10,11),
      (1,5,3)(6,11,7)(8,10,9), (1,10,5,2,11,3)(4,9,7)(6,8) ]) )
gap> AllTDesignLambdas(D);
[ 11, 5, 2 ]
gap> DD:=DerivedBlockDesign(D,6);
rec( isBlockDesign := true, v := 10,
  blocks := [ [ 1, 3, 6, 8 ], [ 1, 5, 7, 10 ], [ 2, 3, 7, 9 ],
      [ 2, 4, 6, 10 ], [ 4, 5, 8, 9 ] ],
  pointNames := [ 1, 2, 3, 4, 5, 7, 8, 9, 10, 11 ] )
gap> AllTDesignLambdas(DD);
[ 5, 2 ]
gap> DD:=DerivedBlockDesign(D,D.blocks[6]);
rec( isBlockDesign := true, v := 5,
  blocks := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ],
      [ 2, 5 ], [ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ],
  pointNames := [ 2, 3, 6, 8, 10 ] )
gap> AllTDesignLambdas(DD);
[ 10, 4, 1 ]
```

12 ▶ `ResidualBlockDesign(` D`,` x `)`

Suppose D is a block design, and x is a point or block of D. Then this function returns the **residual design** RD of D, with respect to x.

If x is a point then RD is the block design whose blocks are those of D not containing x, and the points of RD are those which occur in some block of RD.

If x is a block, then the points of RD are those of D not in x, and the blocks of RD are the blocks of D (including repeats) containing at least one point not in x, but with all points in x deleted from these blocks.

It is an error if the resulting block design RD has no blocks.

The points of RD are relabelled $1, 2, \cdots$, preserving the order of the corresponding points of D; the point-names of RD (listed in RD`.pointNames`) are those of these corresponding points of D.

```
gap> D:=BlockDesigns(rec(v:=11,blockSizes:=[5],
>       tSubsetStructure:=rec(t:=2,lambdas:=[2])))[1];
rec( isBlockDesign := true, v := 11,
  blocks := [ [ 1, 2, 3, 4, 5 ], [ 1, 2, 9, 10, 11 ], [ 1, 3, 6, 7, 9 ],
      [ 1, 4, 7, 8, 10 ], [ 1, 5, 6, 8, 11 ], [ 2, 3, 6, 8, 10 ],
      [ 2, 4, 6, 7, 11 ], [ 2, 5, 7, 8, 9 ], [ 3, 4, 8, 9, 11 ],
      [ 3, 5, 7, 10, 11 ], [ 4, 5, 6, 9, 10 ] ],
  tSubsetStructure := rec( t := 2, lambdas := [ 2 ] ), isBinary := true,
  isSimple := true, blockSizes := [ 5 ], blockNumbers := [ 11 ], r := 5,
  autGroup := Group([ (2,4)(3,5)(7,11)(8,9), (1,3)(2,5)(7,9)(10,11),
      (1,5,3)(6,11,7)(8,10,9), (1,10,5,2,11,3)(4,9,7)(6,8) ]) )
gap> AllTDesignLambdas(D);
[ 11, 5, 2 ]
gap> RD:=ResidualBlockDesign(D,6);
rec( isBlockDesign := true, v := 10,
  blocks := [ [ 1, 2, 3, 4, 5 ], [ 1, 2, 8, 9, 10 ], [ 1, 4, 6, 7, 9 ],
      [ 2, 5, 6, 7, 8 ], [ 3, 4, 7, 8, 10 ], [ 3, 5, 6, 9, 10 ] ],
  pointNames := [ 1, 2, 3, 4, 5, 7, 8, 9, 10, 11 ] )
gap> AllTDesignLambdas(RD);
[ 6, 3 ]
gap> RD:=ResidualBlockDesign(D,D.blocks[6]);
rec( isBlockDesign := true, v := 6,
  blocks := [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 6 ], [ 1, 4, 5 ],
      [ 1, 5, 6 ], [ 2, 3, 5 ], [ 2, 4, 6 ], [ 2, 5, 6 ], [ 3, 4, 5 ],
      [ 3, 4, 6 ] ], pointNames := [ 1, 4, 5, 7, 9, 11 ] )
gap> AllTDesignLambdas(RD);
[ 10, 5, 2 ]
```

13 ▶ `TDesignFromTBD( D, t, k )`

For t a non-negative integer, K a set of positive integers, and v, λ positive integers with $t \leq v$, a t-**wise balanced design**, or a t-(v, K, λ) **design**, is a binary block design with exactly v points, such that each block has size in K and each t-subset of the points is contained in exactly λ blocks.

Now let t and k be positive integers, D be a t-(v, K, λ) design (for some set K), and $t \leq k \leq k_1$, where exactly s distinct block-sizes $k_1 < \cdots < k_s$ occur in D. Then this function returns the t-design $D^* = D^*(t, k)$ described and studied in [MS07].

The point set of D^* is that of D, and the block multiset of D^* consists of, for each $i = 1, \ldots, s$ and each block B of D of size k_i (including repeats), exactly $n/\binom{k_i - t}{k - t}$ copies of every k-subset of B, where $n := \operatorname{lcm}\left(\binom{k_i - t}{k - t} : 1 \leq i \leq s\right)$.

It is shown in [MS07] that D^* is a t-$(v, k, n\lambda)$ design, that $\mathrm{Aut}(D) \subseteq \mathrm{Aut}(D^*)$, and that if $\lambda = 1$ and $t < k$, then $\mathrm{Aut}(D) = \mathrm{Aut}(D^*)$.

```
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, 10 ],
      [ 2, 6, 8 ], [ 2, 7, 9 ], [ 3, 5, 9 ], [ 3, 6, 10 ], [ 3, 7, 8 ],
      [ 4, 5, 8 ], [ 4, 6, 9 ], [ 4, 7, 10 ] ],
  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)(5,7)(8,10),
      (2,3,4)(5,7,6)(8,9,10), (2,3,4)(5,9,6,8,7,10), (2,6,9,3,7,10)(4,5,8) ])
 )
```

```
gap> PairwiseBalancedLambda(D);
1
gap> Dstar:=TDesignFromTBD(D,2,3);
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, 10 ], [ 2, 5, 10 ], [ 2, 6, 8 ], [ 2, 6, 8 ],
      [ 2, 7, 9 ], [ 2, 7, 9 ], [ 3, 5, 9 ], [ 3, 5, 9 ], [ 3, 6, 10 ],
      [ 3, 6, 10 ], [ 3, 7, 8 ], [ 3, 7, 8 ], [ 4, 5, 8 ], [ 4, 5, 8 ],
      [ 4, 6, 9 ], [ 4, 6, 9 ], [ 4, 7, 10 ], [ 4, 7, 10 ], [ 5, 6, 7 ],
      [ 8, 9, 10 ] ],
  autGroup := Group([ (5,6,7)(8,9,10), (2,3)(5,7)(8,10), (2,3,4)(5,7,6)(8,9,
        10), (2,3,4)(5,9,6,8,7,10), (2,6,9,3,7,10)(4,5,8) ]) )
gap> AllTDesignLambdas(Dstar);
[ 30, 9, 2 ]
```

4 Determining basic properties of block designs

4.1 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 ► `BlockDesignPoints(` *D* `)`

This function returns the set of points of the block design *D*, that is `[1..`*D*`.v]`. The returned result is immutable.

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

6 ► `NrBlockDesignPoints( D )`

This function returns the number of points of the block design *D*.

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

7 ► `BlockDesignBlocks( D )`

This function returns the (sorted) list of blocks of the block design *D*. The returned result is immutable.

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

8 ► `NrBlockDesignBlocks( D )`

This function returns the number of blocks of the block design *D*.

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

9 ► `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 ]
```

10 ► `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 ]
```

11 ► `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
```

12 ▶ `PairwiseBalancedLambda( D )`

A binary block design D is **pairwise balanced** if D has at least two points and every pair of distinct points is contained in exactly λ blocks, for some positive constant λ.

Given a binary block design D, this function returns `fail` if D is not pairwise balanced, and otherwise the positive constant λ such that every pair of distinct points of D is in exactly λ blocks.

```
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, 10 ],
      [ 2, 6, 8 ], [ 2, 7, 9 ], [ 3, 5, 9 ], [ 3, 6, 10 ], [ 3, 7, 8 ],
      [ 4, 5, 8 ], [ 4, 6, 9 ], [ 4, 7, 10 ] ],
  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)(5,7)(8,10),
      (2,3,4)(5,7,6)(8,9,10), (2,3,4)(5,9,6,8,7,10), (2,6,9,3,7,10)(4,5,8) ])
 )
gap> PairwiseBalancedLambda(D);
1
```

13 ▶ `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 ]
```

14 ▶ `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 the point-set of D. The returned result is immutable.

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

15 ► `AffineResolvableMu( D )`

A block design is **affine resolvable** if the design is resolvable and any two blocks not in the same parallel class of a resolution meet in a constant number μ of points.

If the block design D is affine resolvable, then this function returns its value of μ; otherwise `fail` is returned.

The value 0 is returned if, and only if, D consists of a single parallel class.

```
gap> P:=PGPointFlatBlockDesign(2,3,1);; # projective plane of order 3
gap> AffineResolvableMu(P);
fail
gap> A:=ResidualBlockDesign(P,P.blocks[1]);; # affine plane of order 3
gap> AffineResolvableMu(A);
1
```

5 Matrices and efficiency measures for block designs

In this chapter we describe functions to calculate certain matrices associated with a block design, and the function `BlockDesignEfficiency` which determines certain statistical efficiency measures of a 1-design.

We also document the utility function `DESIGN_IntervalForLeastRealZero`, which is used in the calculation of E-efficiency measures, but has much wider application.

5.1 Matrices associated with a block design

1 ▶ `PointBlockIncidenceMatrix( D )`

This function returns the point-block incidence matrix N of the block design D. This matrix has rows indexed by the points of D and columns by the blocks of D, with the (i,j)-entry of N being the number of times point i occurs in `D.blocks[`j`]`.

The returned matrix N is immutable.

```
gap> D:=DualBlockDesign(AGPointFlatBlockDesign(2,3,1));;
gap> BlockDesignBlocks(D);
[ [ 1, 2, 3, 4 ], [ 1, 5, 6, 7 ], [ 1, 8, 9, 10 ], [ 2, 5, 8, 11 ],
  [ 2, 7, 9, 12 ], [ 3, 5, 10, 12 ], [ 3, 6, 9, 11 ], [ 4, 6, 8, 12 ],
  [ 4, 7, 10, 11 ] ]
gap> PointBlockIncidenceMatrix(D);
[ [ 1, 1, 1, 0, 0, 0, 0, 0, 0 ], [ 1, 0, 0, 1, 1, 0, 0, 0, 0 ],
  [ 1, 0, 0, 0, 0, 1, 1, 0, 0 ], [ 1, 0, 0, 0, 0, 0, 0, 1, 1 ],
  [ 0, 1, 0, 1, 0, 1, 0, 0, 0 ], [ 0, 1, 0, 0, 0, 0, 1, 1, 0 ],
  [ 0, 1, 0, 0, 1, 0, 0, 0, 1 ], [ 0, 0, 1, 1, 0, 0, 0, 1, 0 ],
  [ 0, 0, 1, 0, 1, 0, 1, 0, 0 ], [ 0, 0, 1, 0, 0, 1, 0, 0, 1 ],
  [ 0, 0, 0, 1, 0, 0, 1, 0, 1 ], [ 0, 0, 0, 0, 1, 1, 0, 1, 0 ] ]
```

2 ▶ `ConcurrenceMatrix( D )`

This function returns the concurrence matrix L of the block design D. This matrix is equal to NN^{T}, where N is the point-block incidence matrix of D (see 5.1.1) and N^{T} is the transpose of N. If D is a binary block design then the (i,j)-entry of its concurrence matrix is the number of blocks containing points i and j.

The returned matrix L is immutable.

```
gap> D:=DualBlockDesign(AGPointFlatBlockDesign(2,3,1));;
gap> BlockDesignBlocks(D);
[ [ 1, 2, 3, 4 ], [ 1, 5, 6, 7 ], [ 1, 8, 9, 10 ], [ 2, 5, 8, 11 ],
  [ 2, 7, 9, 12 ], [ 3, 5, 10, 12 ], [ 3, 6, 9, 11 ], [ 4, 6, 8, 12 ],
  [ 4, 7, 10, 11 ] ]
gap> ConcurrenceMatrix(D);
[ [ 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0 ],
  [ 1, 3, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1 ],
  [ 1, 1, 3, 1, 1, 1, 0, 0, 1, 1, 1, 1 ],
  [ 1, 1, 1, 3, 0, 1, 1, 1, 0, 1, 1, 1 ],
  [ 1, 1, 1, 0, 3, 1, 1, 1, 0, 1, 1, 1 ],
  [ 1, 0, 1, 1, 1, 3, 1, 1, 1, 0, 1, 1 ],
  [ 1, 1, 0, 1, 1, 1, 3, 0, 1, 1, 1, 1 ],
  [ 1, 1, 0, 1, 1, 1, 0, 3, 1, 1, 1, 1 ],
  [ 1, 1, 1, 0, 0, 1, 1, 1, 3, 1, 1, 1 ],
  [ 1, 0, 1, 1, 1, 0, 1, 1, 1, 3, 1, 1 ],
  [ 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 0 ],
  [ 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 3 ] ]
```

3 ► `InformationMatrix( D )`

This function returns the information matrix C of the block design D.

This matrix is defined as follows. Suppose D has v points and b blocks, let R be the $v \times v$ diagonal matrix whose (i,i)-entry is the replication number of the point i, let N be the point-block incidence matrix of D (see 5.1.1), and let K be the $b \times b$ diagonal matrix whose (j,j)-entry is the length of `D.blocks[`j`]`. Then the **information matrix** of D is $C := R - NK^{-1}N^{\mathrm{T}}$. If D is a 1-(v,k,r) design then this expression for C simplifies to $rI - k^{-1}L$, where I is the $v \times v$ identity matrix and L is the concurrence matrix of D (see 5.1.2).

The returned matrix C is immutable.

```
gap> D:=DualBlockDesign(AGPointFlatBlockDesign(2,3,1));;
gap> BlockDesignBlocks(D);
[ [ 1, 2, 3, 4 ], [ 1, 5, 6, 7 ], [ 1, 8, 9, 10 ], [ 2, 5, 8, 11 ],
  [ 2, 7, 9, 12 ], [ 3, 5, 10, 12 ], [ 3, 6, 9, 11 ], [ 4, 6, 8, 12 ],
  [ 4, 7, 10, 11 ] ]
gap> InformationMatrix(D);
[ [ 9/4, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, 0, 0 ],
  [ -1/4, 9/4, -1/4, -1/4, -1/4, 0, -1/4, -1/4, -1/4, 0, -1/4, -1/4 ],
  [ -1/4, -1/4, 9/4, -1/4, -1/4, -1/4, 0, 0, -1/4, -1/4, -1/4, -1/4 ],
  [ -1/4, -1/4, -1/4, 9/4, 0, -1/4, -1/4, -1/4, 0, -1/4, -1/4, -1/4 ],
  [ -1/4, -1/4, -1/4, 0, 9/4, -1/4, -1/4, -1/4, 0, -1/4, -1/4, -1/4 ],
  [ -1/4, 0, -1/4, -1/4, -1/4, 9/4, -1/4, -1/4, -1/4, 0, -1/4, -1/4 ],
  [ -1/4, -1/4, 0, -1/4, -1/4, -1/4, 9/4, 0, -1/4, -1/4, -1/4, -1/4 ],
  [ -1/4, -1/4, 0, -1/4, -1/4, -1/4, 0, 9/4, -1/4, -1/4, -1/4, -1/4 ],
  [ -1/4, -1/4, -1/4, 0, 0, -1/4, -1/4, -1/4, 9/4, -1/4, -1/4, -1/4 ],
  [ -1/4, 0, -1/4, -1/4, -1/4, 0, -1/4, -1/4, -1/4, 9/4, -1/4, -1/4 ],
  [ 0, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, 9/4, 0 ],
  [ 0, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, -1/4, 0, 9/4 ] ]
```

5.2 The function BlockDesignEfficiency

1 ▶ `BlockDesignEfficiency( D )`
▶ `BlockDesignEfficiency( D, eps )`
▶ `BlockDesignEfficiency( D, eps, includeMV )`

Let D be a 1-(v,k,r) design with $v > 1$, let *eps* be a positive rational number (default: 10^{-6}), and let *includeMV* be a boolean (default: `false`). Then this function returns a record *eff* containing information on statistical efficiency measures of D. These measures are defined below. See [CDMS04], [BC09] and [BR97] for further details. All returned results are computed using exact algebraic computation.

The component *eff*`.A` contains the A-efficiency measure for D, *eff*`.Dpowered` contains the D-efficiency measure of D raised to the power $v-1$, and *eff*`.Einterval` is a list $[a,b]$ of non-negative rational numbers such that if x is the E-efficiency measure of D then $a \leq x \leq b$, $b - a \leq eps$, and if x is rational then $a = x = b$. Moreover *eff*`.CEFpolynomial` contains the monic polynomial over the rationals whose zeros (counting multiplicities) are the canonical efficiency factors of the design D. If *includeMV*`=true` then additional work is done to compute the MV- (also called E′-) efficiency measure, and then *eff*`.MV` contains the value of this measure. (This component may be set even if *includeMV*`=false`, as a byproduct of other computation.)

We now define the canonical efficiency factors and the A-, D-, E-, and MV-efficiency measures of a 1-design.

Let D be a 1-(v,k,r) design with $v \geq 2$, let C be the information matrix of D (see 5.1.3), and let $F := r^{-1}C$. The eigenvalues of F are all real and lie in the interval $[0,1]$. At least one of these eigenvalues is zero: an associated eigenvector is the all-1 vector. The remaining eigenvalues $\delta_1 \leq \delta_2 \leq \cdots \leq \delta_{v-1}$ of F are called the **canonical efficiency factors** of D. These are all non-zero if and only if D is connected (that is, the point-block incidence graph of D is a connected graph).

If D is not connected, then the A-, D-, E-, and MV-efficiency measures of D are all defined to be zero. Otherwise, the **A-efficiency measure** is $(v-1)/\sum_{i=1}^{v-1} 1/\delta_i$ (the harmonic mean of the canonical efficiency factors), the **D-efficiency measure** is $(\prod_{i=1}^{v-1} \delta_i)^{1/(v-1)}$ (the geometric mean of the canonical efficiency factors), and the **E-efficiency measure** is δ_1 (the minimum of the canonical efficiency factors).

If D is connected, and the MV-efficiency measure is required, then it is computed as follows. Let $F := r^{-1}C$ be as before, and let $P := v^{-1}J$, where J is the $v \times v$ all-1 matrix. Set $M := (F+P)^{-1} - P$, making M the "Moore-Penrose inverse" of F (see [BC09]). Then the **MV-efficiency measure** of D is the minimum value (over all $i,j \in \{1,\ldots,v\}$, $i \neq j$) of $2/(M_{ii} + M_{jj} - M_{ij} - M_{ji})$.

```
gap> D:=DualBlockDesign(AGPointFlatBlockDesign(2,3,1));;
gap> BlockDesignBlocks(D);
[ [ 1, 2, 3, 4 ], [ 1, 5, 6, 7 ], [ 1, 8, 9, 10 ], [ 2, 5, 8, 11 ],
  [ 2, 7, 9, 12 ], [ 3, 5, 10, 12 ], [ 3, 6, 9, 11 ], [ 4, 6, 8, 12 ],
  [ 4, 7, 10, 11 ] ]
gap> BlockDesignEfficiency(D);
rec( A := 33/41,
  CEFpolynomial := x_1^11-9*x_1^10+147/4*x_1^9-719/8*x_1^8+18723/128*x_1^7-106\
47/64*x_1^6+138159/1024*x_1^5-159813/2048*x_1^4+2067201/65536*x_1^3-556227/655\
36*x_1^2+89667/65536*x_1-6561/65536, Dpowered := 6561/65536,
  Einterval := [ 3/4, 3/4 ] )
gap> BlockDesignEfficiency(D,10^(-4),true);
rec( A := 33/41,
  CEFpolynomial := x_1^11-9*x_1^10+147/4*x_1^9-719/8*x_1^8+18723/128*x_1^7-106\
47/64*x_1^6+138159/1024*x_1^5-159813/2048*x_1^4+2067201/65536*x_1^3-556227/655\
36*x_1^2+89667/65536*x_1-6561/65536, Dpowered := 6561/65536,
  Einterval := [ 3/4, 3/4 ], MV := 3/4 )
```

5.3 Computing an interval for a certain real zero of a rational polynomial

We document a DESIGN package utility function used in the calculation of the `Einterval` component above, but is more widely applicable.

1 ▶ `DESIGN_IntervalForLeastRealZero(` *f*, *a*, *b*, *eps* `)`

Suppose that f is a univariate polynomial over the rationals, a, b are rational numbers with $a \leq b$, and *eps* is a positive rational number.

If f has no real zero in the closed interval $[a, b]$, then this function returns the empty list. Otherwise, let α be the least real zero of f such that $a \leq \alpha \leq b$. Then this function returns a list $[c, d]$ of rational numbers, with $c \leq \alpha \leq d$ and $d - c \leq eps$. Moreover, $c = d$ if and only if α is rational (in which case $\alpha = c = d$).

```
gap> x:=Indeterminate(Rationals,1);
x_1
gap> f:=(x+3)*(x^2-3);
x_1^3+3*x_1^2-3*x_1-9
gap> L:=DESIGN_IntervalForLeastRealZero(f,-5,5,10^(-3));
[ -3, -3 ]
gap> L:=DESIGN_IntervalForLeastRealZero(f,-2,5,10^(-3));
[ -14193/8192, -7093/4096 ]
gap> List(L,Float);
[ -1.73254, -1.73169 ]
gap> L:=DESIGN_IntervalForLeastRealZero(f,0,5,10^(-3));
[ 14185/8192, 7095/4096 ]
gap> List(L,Float);
[ 1.73157, 1.73218 ]
gap> L:=DESIGN_IntervalForLeastRealZero(f,0,5,10^(-5));
[ 454045/262144, 908095/524288 ]
gap> List(L,Float);
[ 1.73204, 1.73205 ]
gap> L:=DESIGN_IntervalForLeastRealZero(f,2,5,10^(-5));
[  ]
```

6 Automorphism groups and isomorphism testing for block designs

The functions in this chapter depend on *nauty* [McK90], [MP14] or *bliss* [JK07] via the GRAPE package, which must be fully installed in order for these functions to work.

6.1 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 $\{1,\ldots,D.\mathtt{v}\}$ 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
```

6.2 Testing isomorphism

1 ► `IsIsomorphicBlockDesign( D1, D2 )`

This boolean function returns `true` if and only if block designs *D1* and *D2* are **isomorphic**, that is, there is a bijection from the point-set of *D1* to that of *D2* which maps the block-multiset of *D1* to that of *D2*.

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

For pairwise isomorphism testing for three or more binary block designs, see 6.2.2.

```
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 ) ]
```

7 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).

7.1 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 the list *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`. 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`, `b`, `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`.b` must be a positive integer, and specifies that each design in *DL* will have exactly *param*`.b` 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*] (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 *B* of size *param*.blockSizes[*i*] with a different block (but possibly a repeat of *B*) 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; 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 an upper bound on the 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 and *param*.isoGroup are both 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 ] ]
```

8 Classifying semi-Latin squares

This chapter describes the function `SemiLatinSquareDuals` which can classify semi-Latin squares with certain given properties, and return a list of their duals as block designs.

8.1 Semi-Latin squares and SOMAs

Let n and k be positive integers. An $(n \times n)/k$ **semi-Latin square** is an n by n array A, whose entries are k-subsets of a kn-set X (the **symbol-set**), such that each element of X occurs exactly once in each row and exactly once in each column of A. (Thus an $(n \times n)/1$ semi-Latin square is the same thing as a Latin square of order n.) For extensive useful information on semi-Latin squares, see

`http://www.maths.qmul.ac.uk/~rab/sls.html` .

A SOMA(k, n) is an $(n \times n)/k$ semi-Latin square A, with $n \geq 2$, in which no 2-subset of the symbol-set is contained in more than one entry of A. For extensive useful information on SOMAs, see

`http://www.maths.qmul.ac.uk/~lsoicher/soma/` .

Let A and B be $(n \times n)/k$ semi-Latin squares. We say that B is **(weakly) isomorphic** to A if B can be obtained from A by applying one or more of: a row permutation; a column permutation; transposing; renaming the symbols. If transposing is not allowed then we get the concept of strong isomorphism. More formally, B is **strongly isomorphic** to A if B can be obtained from A by applying one or more of: a row permutation; a column permutation; renaming the symbols.

Let A be an $(n \times n)/k$ semi-Latin square. Then the dual of A can be represented as a binary block design as follows. The point-set of D is taken to be the Cartesian square of $\{1, \ldots, n\}$, with $[x, y]$ representing the $[x, y]$-entry of A. The blocks of D are in one-to-one correspondence with the symbols of A, with the i-th block of D consisting of the ordered pairs $[x, y]$ such that the i-th symbol of A is contained in the $[x, y]$-entry of A. Given D, the semi-Latin square A can be recovered, up to the naming of its symbols.

8.2 The function SemiLatinSquareDuals

1 ▶ `SemiLatinSquareDuals(` *n*, *k* `)`
▶ `SemiLatinSquareDuals(` *n*, *k*, *maxmult* `)`
▶ `SemiLatinSquareDuals(` *n*, *k*, *maxmult*, *blockintsizes* `)`
▶ `SemiLatinSquareDuals(` *n*, *k*, *maxmult*, *blockintsizes*, *isolevel* `)`

Let n and k be positive integers. Then this function (which makes heavy use of the function `BlockDesigns`) returns a list *DL* of block designs which are the duals of the $(n \times n)/k$ semi-Latin squares whose properties are specified by the given parameters, described below. In practice, depending on the specified properties, this function can be useful for n up to about 6 or 7.

The parameter *maxmult*, if given, must be a positive integer or the string `"default"`. If it is a positive integer, then *maxmult* specifies an upper bound on the multiplicity of each block in each semi-Latin square dual in *DL*. The default value for *maxmult* (if omitted or if given as `"default"`) is k, which poses no constraint on the block multiplicities.

The parameter *blockintsizes*, if given, must be a set of non-negative integers or the string `"default"`. If it is given as a set, then *blockintsizes* specifies, for each semi-Latin square dual in *DL*, the set of possible sizes for the intersection of a block B with a different block (but possibly a repeat of B). The default value for *blockintsizes* (if omitted or if given

as `"default"`) is `[0..n]`, which poses no constraint on the block intersection sizes. Note that block intersection sizes in the dual of a semi-Latin square correspond to concurrencies of points in the semi-Latin square itself. Also note that if $n \geq 2$ and *blockintsizes* is specified to be `[0,1]` then the $(n \times n)/k$ semi-Latin squares being considered are SOMA(k,n)s.

The parameter *isolevel*, if given, must be 0, 1, 2, 3, 4 or the string `"default"` (the default value is 2). The value 0 specifies that *DL* will contain at most one (semi-Latin square dual given as a) block design, and will contain one such block design if and only if a semi-Latin square with the required properties exists. The value 1 specifies that *DL* will contain a list of duals representing all weak isomorphism classes of semi-Latin squares with the required properties (possibly with some classes represented more than once) and the value 2 specifies that *DL* will contain precisely one dual semi-Latin square representative for each weak isomorphism class of semi-Latin squares with the required properties. The values 3 and 4 for *isolevel* play the roles of 1 and 2, respectively, but with weak isomorphism replaced by strong isomorphism. Thus, $isolevel = 3$ specifies that *DL* will contain a list of duals representing all strong isomorphism classes of semi-Latin squares with the required properties (possibly with some classes represented more than once) and $isolevel = 4$ specifies that *DL* will contain precisely one dual semi-Latin square representative for each strong isomorphism class of semi-Latin squares with the required properties.

For example, we determine the numbers of weak and strong isomorphism classes of $(4 \times 4)/k$ semi-Latin squares for $k = 1, \ldots, 6$. (These numbers disagree with P. E. Chigbu's classification for the cases $k = 3, 4$ [BC97].)

```
gap> List([1..6],k->Length(SemiLatinSquareDuals(4,k))); # weak
[ 2, 10, 40, 164, 621, 2298 ]
gap> List([1..6],k->Length(SemiLatinSquareDuals(4,k,"default","default",4))); # strong
[ 2, 11, 46, 201, 829, 3343 ]
```

Next, we determine one SOMA$(3,6)$.

```
gap> SemiLatinSquareDuals(6,3,"default",[0,1],0);
[ rec( isBlockDesign := true, v := 36,
      blocks := [ [ 1, 8, 15, 22, 29, 36 ], [ 1, 9, 16, 23, 30, 32 ],
          [ 1, 12, 14, 21, 28, 35 ], [ 2, 9, 17, 24, 25, 34 ],
          [ 2, 11, 18, 22, 27, 31 ], [ 2, 12, 16, 19, 29, 33 ],
          [ 3, 10, 14, 24, 29, 31 ], [ 3, 11, 16, 20, 25, 36 ],
          [ 3, 12, 13, 23, 26, 34 ], [ 4, 7, 14, 23, 27, 36 ],
          [ 4, 8, 17, 21, 30, 31 ], [ 4, 9, 18, 19, 26, 35 ],
          [ 5, 7, 15, 20, 30, 34 ], [ 5, 8, 13, 24, 28, 33 ],
          [ 5, 10, 18, 21, 25, 32 ], [ 6, 7, 17, 22, 26, 33 ],
          [ 6, 10, 13, 20, 27, 35 ], [ 6, 11, 15, 19, 28, 32 ] ],
      tSubsetStructure := rec( t := 1, lambdas := [ 3 ] ), isBinary := true,
      isSimple := true, blockSizes := [ 6 ], blockNumbers := [ 18 ], r := 3,
      autSubgroup := <permutation group of size 72 with 3 generators>,
      pointNames := [ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ],
          [ 1, 6 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ], [ 2, 4 ], [ 2, 5 ],
          [ 2, 6 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ], [ 3, 5 ],
          [ 3, 6 ], [ 4, 1 ], [ 4, 2 ], [ 4, 3 ], [ 4, 4 ], [ 4, 5 ],
          [ 4, 6 ], [ 5, 1 ], [ 5, 2 ], [ 5, 3 ], [ 5, 4 ], [ 5, 5 ],
          [ 5, 6 ], [ 6, 1 ], [ 6, 2 ], [ 6, 3 ], [ 6, 4 ], [ 6, 5 ],
          [ 6, 6 ] ] ) ]
```

9 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.

9.1 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
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, b, 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.b must be a positive integer, and specifies that each design in each partition in *PL* has exactly *param*.b 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 *B* of size *param*.blockSizes[*i*] with a different block (but possibly a repeat of *B*) 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; 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 an upper bound on the multiplicity of a block 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 permutation 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])));
[  ]
```

9.2 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 ) )
```

10 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]).

10.1 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* (default: empty list), 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 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");
```

10.2 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(
  infoXML := "<info>\n<software>\n[ DESIGN-1.8, GRAPE-4.9.0, GAPDoc-1.6.6, GAP\
-4.12.1 ]\n</software>\n</info>",
  list :=
    [
      rec( blocks := [ [ 1, 2 ], [ 1, 3 ] ], id := "example-0",
          isBinary := true, isBlockDesign := true, v := 3 ),
      rec( blocks := [ [ 1, 2 ], [ 1, 2 ], [ 2, 3 ] ], id := "example-1",
          isBinary := true, isBlockDesign := true, v := 3 ) ],
  pairwiseNonisomorphic := true )
```

Bibliography

[BC97] R. A. Bailey and P. E. Chigbu. Enumeration of semi-latin squares. *Discrete Math.*, 167-168:73–84, 1997.

[BC09] R. A. Bailey and P. J. Cameron. Combinatorics of optimal designs. In S. Huczynska, J. D. Mitchell, and C. M. Roney-Dougal, editors, *Surveys in Combinatorics 2009*, volume 365 of *London Math. Soc. Lecture Notes*, pages 19–73. Cambridge University Press, 2009.

[BCD+06] R. A. Bailey, P. J. Cameron, P. Dobcsányi, J. P. Morgan, and L. H. Soicher. Designs on the web. *Discrete Math.*, 306:3014–3027, 2006.

`https://doi.org/10.1016/j.disc.2004.10.027`.

[BR97] R. A. Bailey and G. Royle. Optimal semi-latin squares with side six and block size two. *Proc. Roy. Soc. London, Ser. A*, 453:1903–1914, 1997.

[CDMS04] P. J. Cameron, P. Dobcsányi, J. P. Morgan, and L. H. Soicher. *The external representation of block designs, Version 2.0*, 2004.

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

[CS07] P. J. Cameron and L. H. Soicher. Block intersection polynomials. *Bull. London Math. Soc.*, 39:559–564, 2007.

`https://doi.org/10.1112/blms/bdm034`.

[JK07] Tommi Juntilla and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In David Applegate et al., editor, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithmics and Combinatorics*, pages 135–149. SIAM, 2007. bliss homepage:

`http://www.tcs.hut.fi/Software/bliss/`.

[McK90] Brendan D. McKay. *nauty user's guide (version 1.5), Technical report TR-CS-90-02.* Australian National University, Computer Science Department, 1990. nauty homepage:

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

[MP14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. *J. Symbolic Comput.*, 60:94–112, 2014.

[MS07] J. P. McSorley and L. H. Soicher. Constructing t-designs from t-wise balanced designs. *European J. Combinatorics*, 28:567–571, 2007.

`https://doi.org/10.1016/j.ejc.2005.02.003`.

[Soi10] L. H. Soicher. More on block intersection polynomials and new applications to graphs and block designs. *J. Comb. Theory, Ser. A*, 117:799–809, 2010.

`https://doi.org/10.1016/j.jcta.2010.03.005`.

[Soi13] Leonard H. Soicher. Designs, groups and computing. In Alla Detinko, Dane Flannery, and Eamonn O'Brien, editors, *Probabilistic Group Theory, Combinatorics, and Computing. Lectures from the Fifth de Brún Workshop*, volume 2070 of *Lecture Notes in Mathematics*, pages 83–107. Springer, Berlin, Heidelberg and New York, 2013.

[Soi24a] L. H. Soicher. *The GRAPE package for GAP, Version 4.9.2*, 2024.

`https://gap-packages.github.io/grape`.

[Soi24b] Leonard H. Soicher. Using GAP packages for research in graph theory, design theory, and finite geometry. In Alexander A. Ivanov, editor, *Algebraic Combinatorics and the Monster Group*, volume 487 of *London Mathematical Society Lecture Note Series*, pages 527–566. Cambridge University Press, 2024. accepted manuscript available at:

`https://webspace.maths.qmul.ac.uk/l.h.soicher/g2g2_final.pdf`.

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".