PAG

Prescribed Automorphism Groups

0.2.4

12 November 2024

Vedran Krcadinac

Vedran Krcadinac
Email: vedran.krcadinac@math.hr
Homepage: https://web.math.pmf.unizg.hr/~krcko/homepage.html
Address: University of Zagreb, Faculty of Science,
Department of Mathematics
Bijenicka cesta 30, HR-10000 Zagreb, Croatia

Abstract

PAG is a GAP package for constructing combinatorial objects with prescribed automorphism groups.

Copyright

Acknowledgements

Development of the PAG package has been supported by the Croatian Science Foundation under the project IP-2020-02-9752.

Contents

Chapter 1

The PAG Package

Prescribed Automorphism Groups (PAG) is a GAP package for constructing combinatorial objects with prescribed automorphism groups.

1.1 Getting Started

The package is loaded by

Example
```
gap> LoadPackage("PAG");
```

Let us start with a small example from the paper [Krc18]. In Theorem 8.1, a simple 5-(16,7,10) design with the following automorphism group was constructed.

Example
```
gap> g:=Group((2,3,4)(5,6,7,8,9,10)(11,12,13,14,15,16),
> (1,5)(2,12)(3,15)(4,8)(6,14)(7,16)(9,10)(11,13));
```

The design can be obtained by typing

Example
```
gap> KramerMesnerSearch(5,16,7,10,g);
Computing t-subset orbit representatives...
28
Computing k-subset orbit representatives...
71
Computing the Kramer-Mesner matrix...
[ 29, 72 ]
Starting solver...
No BOUNDS
The RHS is fixed !
No upper bounds: 0/1 variables are assumed

Orthogonal defect: 26.953339
First reduction successful
Orthogonal defect: 20.216092
Second reduction successful
.
.
.
```

Comments during the calculation can be supressed by setting global options.

Example
```
gap> PAGGlobalOptions.Silent:=true;
true
gap> d:=KramerMesnerSearch(5,16,7,10,g);
[ rec( autSubgroup := Group([ (2,3,4)(5,6,7,8,9,10)(11,12,13,14,15,16),
          (1,5)(2,12)(3,15)(4,8)(6,14)(7,16)(9,10)(11,13) ]),
      blocks := [ [ 1, 2, 3, 4, 5, 6, 13 ], [ 1, 2, 3, 4, 5, 6, 14 ],
          [ 1, 2, 3, 4, 5, 7, 9 ], [ 1, 2, 3, 4, 5, 7, 12 ],
          [ 1, 2, 3, 4, 5, 9, 16 ], [ 1, 2, 3, 4, 5, 10, 12 ],
          [ 1, 2, 3, 4, 5, 10, 13 ], [ 1, 2, 3, 4, 5, 11, 12 ],
          [ 1, 2, 3, 4, 5, 11, 16 ], [ 1, 2, 3, 4, 5, 12, 14 ],
          [ 1, 2, 3, 4, 6, 7, 14 ], [ 1, 2, 3, 4, 6, 7, 15 ],
.
.
.
```

The output is a list of non-isomorphic designs in the Design package format (**DESIGN: Design**). We can check that it is really a 5-design.

Example
```
gap> List(d,AllTDesignLambdas);
[ [ 2080, 910, 364, 130, 40, 10 ] ]
```

The output is large because the Design format includes a list of all blocks, and 5-(16,7,10) designs have 2080 blocks. Instead, we can ask just for the base blocks.

Example
```
gap> bb:=KramerMesnerSearch(5,16,7,10,g,rec(BaseBlocks:=true));
[ [ [ 1, 2, 3, 4, 5, 6, 13 ], [ 1, 2, 3, 4, 5, 6, 14 ],
      [ 1, 2, 3, 5, 6, 7, 11 ], [ 1, 2, 3, 5, 6, 8, 9 ],
      [ 1, 2, 3, 5, 6, 9, 10 ], [ 1, 2, 3, 5, 6, 9, 12 ],
      [ 1, 2, 3, 5, 6, 10, 15 ], [ 1, 2, 3, 5, 6, 14, 16 ],
      [ 1, 2, 3, 5, 8, 11, 12 ], [ 1, 2, 5, 6, 7, 8, 16 ],
      [ 1, 2, 5, 6, 7, 9, 14 ], [ 1, 2, 5, 6, 7, 12, 13 ],
      [ 1, 2, 5, 6, 7, 14, 15 ] ],
  [ [ 1, 2, 3, 4, 5, 6, 8 ], [ 1, 2, 3, 4, 5, 6, 14 ],
      [ 1, 2, 3, 5, 6, 7, 11 ], [ 1, 2, 3, 5, 6, 9, 12 ],
      [ 1, 2, 3, 5, 6, 10, 12 ], [ 1, 2, 3, 5, 6, 10, 16 ],
      [ 1, 2, 3, 5, 6, 12, 13 ], [ 1, 2, 3, 5, 6, 14, 15 ],
      [ 1, 2, 3, 5, 8, 11, 12 ], [ 1, 2, 5, 6, 7, 8, 9 ],
      [ 1, 2, 5, 6, 7, 9, 14 ], [ 1, 2, 5, 6, 7, 12, 13 ],
      [ 1, 2, 5, 6, 11, 14, 16 ] ] ]
```

In this case isomorph rejection is not performed and we get two sets of base blocks. They can be turned into designs by calling the `BlockDesign` (**DESIGN: BlockDesign**) function: `List(bb,x->BlockDesign(16,x,g));`.

1.2 Installation

The PAG package requires GAP 4.11 and the following packages:

- Images 1.3

- GRAPE 4.8
- Design 1.7

The following packages are also loaded, if available. They are needed for a limited number of PAG functions.

- AssociationSchemes 2.0
- DifSets 2.3.1
- GUAVA 3.15
- FinInG 1.4.1

The current installation file for PAG is available at https://vkrcadinac.github.io/PAG/. To install PAG, unpack it to the pkg directory of your local GAP installation. The package uses external binaries. To compile them on UNIX-like environments, change to the pkg/PAG-* directory and call

Example
```
$ ./configure.sh
```

This produces a Makefile in the current directory. Now call

Example
```
$ make all
```

to compile the binares. They are placed in the bin subdirectory. Documentation in the doc subdirectory is already compiled and can be read in PDF, html or from within GAP. To recompile the documentation, call GAP with the makedoc.g file.

1.3 Examples: Designs

The PAG function KramerMesnerSearch performs a search for t-designs with given parameters and a given permutation group as group of automorphisms. See the paper by B. Schmalz [Sch93] for an introduction to the Kramer-Mesner approach to constructing t-designs. Our first two examples are from this paper. The original paper of Earl Kramer and Dale Mesner is [KM76].

1.3.1 6-(14,7,4) Designs

The summary about known 6-designs on page 130 of [Sch93] mentions that there are exactly two 6-(14,7,4) designs with cyclic derived designs. This means that the two 6-designs have automorphisms of order 13. They can be constructed by the following GAP commands.

Example
```
gap> g:=Group(CyclicPerm(13));
Group([ (1,2,3,4,5,6,7,8,9,10,11,12,13) ])
gap> d:=KramerMesnerSearch(6,14,7,4,g);;
gap> List(d,AllTDesignLambdas);
[ [ 1716, 858, 396, 165, 60, 18, 4 ], [ 1716, 858, 396, 165, 60, 18, 4 ] ]
```

The solver quickly finds 24 solutions of the Kramer-Mesner system. Most of the computation time is used to eliminate isomorphic designs. This can be turned off:

Example
```
gap> d2:=KramerMesnerSearch(6,14,7,4,g,rec(NonIsomorphic:=false));;
gap> Size(d2);
30
gap> Size(AsSet(d2));
24
```

Now we get a list of 30 designs. By default, A. Wassermann's LLL solver [Was98] is used; it may return the same solution more than once. The number of distinct designs is 24. The two non-isomorphic designs have $\mathbb{Z}_{13}$ as their full automorphism group.

Example
```
gap> List(d,BlockDesignAut);
[ Group([ (1,2,3,4,5,6,7,8,9,10,11,12,13) ]),
  Group([ (1,2,3,4,5,6,7,8,9,10,11,12,13) ]) ]
```

1.3.2 6-(28,8,λ) Designs

In [Sch93], the existence of 6-(28,8,λ) designs was established for $\lambda = 42$, 63, 84, and 105. The exact numbers of these designs with automorphism group $P\Gamma L(2,27)$ were computed. While the projective general linear groups are readily available in GAP through the PGL command, there seems to be no equivalent command for semilinear groups. We can get $P\Gamma L(2,27)$ using the FinInG package, as the collineation group of the projective line over $GF(27)$.

Example
```
gap> LoadPackage("FinInG");
gap> g1:=CollineationGroup(ProjectiveSpace(1,27));
The FinInG collineation group PGammaL(2,27)
```

We need a permutation representation of this group on 28 points.

Example
```
gap> g:=Image(ActionOnAllProjPoints(g1));
Group([ (3,28,27,26,25,24,23,22,21,20,19,18,17,4,16,15,14,13,12,11,10,9,8,7,6,5),
  (1,2,4)(5,8,24)(6,21,10)(7,16,15)(9,25,28)(11,13,14)(12,27,23)(17,26,18)
  (19,20,22), (5,7,13)(6,10,21)(8,16,14)(9,18,22)(11,24,15)(12,27,23)(17,19,25)
  (20,28,26) ])
```

Alternatively, we can get $P\Gamma L(2,27)$ from the library of small primitive permutation groups.

Example
```
gap> PrimitiveGroupsOfDegree(28);
[ PGL(2, 7), PSL(2, 8), PGammaL(2, 8), PSU(3, 3), PGammaU(3, 3), PSp(6, 2), A(8),
  S(8), PSL(2, 27), PGL(2, 27), PSL(2, 27):3, PGammaL(2, 27), A(28), S(28) ]
```

Now we can construct the designs with $\lambda = 42$.

Example
```
gap> d:=KramerMesnerSearch(6,28,8,42,g,rec(BaseBlocks:=true));;
gap> Size(AsSet(d));
3
```

Most of the CPU time in the example above was used to compute the Kramer-Mesner matrix. The left side of the Kramer-Mesner system is the same matrix for all λ, so we can compute it once and reuse it to save time.

Example
```
gap> tsub:=SubsetOrbitRep(g,28,6);;
gap> ksub:=SubsetOrbitRep(g,28,8);;
gap> m:=KramerMesnerMat(g,tsub,ksub);;
```

Now we can quickly get the exact numbers of designs from the paper [Sch93].

Example
```
gap> Size(AsSet(SolveKramerMesner(ExpandMatRHS(m,42))));
3
gap> Size(AsSet(SolveKramerMesner(ExpandMatRHS(m,63))));
367
gap> Size(AsSet(SolveKramerMesner(ExpandMatRHS(m,84))));
21743
gap> Size(AsSet(SolveKramerMesner(ExpandMatRHS(m,105))));
38277
```

1.3.3 2-(81,6,2) Designs

The first simple 2-(81,6,2) design was recently found by A. Nakic [Nak21]. Here are the base blocks of this design copy-pasted from the paper.

Example
```
gap> bb:=[[[0,0,0,0],[0,0,0,1],[0,0,0,2],[0,1,0,0],[0,1,0,1],[0,1,0,2]],
> [[0,0,0,0],[0,0,1,1],[0,0,2,2],[2,1,0,0],[2,1,1,1],[2,1,2,2]],
> [[0,0,0,0],[0,1,1,1],[0,2,2,2],[0,0,1,0],[0,1,2,1],[0,2,0,2]],
> [[0,0,0,0],[0,1,2,0],[0,2,1,0],[2,0,2,1],[2,1,1,1],[2,2,0,1]],
> [[0,0,0,0],[1,0,0,0],[2,0,0,0],[0,2,2,1],[1,2,2,1],[2,2,2,1]],
> [[0,0,0,0],[1,0,1,0],[2,0,2,0],[0,1,0,0],[1,1,1,0],[2,1,2,0]],
> [[0,0,0,0],[1,0,1,1],[2,0,2,2],[0,0,2,0],[1,0,0,1],[2,0,1,2]],
> [[0,0,0,0],[1,0,2,0],[2,0,1,0],[0,2,1,1],[1,2,0,1],[2,2,2,1]],
> [[0,0,0,0],[1,0,2,2],[2,0,1,1],[0,1,2,1],[1,1,1,0],[2,1,0,2]],
> [[0,0,0,0],[1,1,0,0],[2,2,0,0],[0,2,0,1],[1,0,0,1],[2,1,0,1]],
> [[0,0,0,0],[1,1,0,1],[2,2,0,2],[0,2,2,0],[1,0,2,1],[2,1,2,2]],
> [[0,0,0,0],[1,1,2,0],[2,2,1,0],[0,0,2,1],[1,1,1,1],[2,2,0,1]],
> [[0,0,0,0],[1,1,2,1],[2,2,1,2],[0,2,1,1],[1,0,0,2],[2,1,2,0]],
> [[0,0,0,0],[1,1,2,2],[2,2,1,1],[0,2,2,0],[1,0,1,2],[2,1,0,1]],
> [[0,0,0,0],[1,2,1,2],[2,1,2,1],[0,0,2,1],[1,2,0,0],[2,1,1,2]],
> [[0,0,0,0],[1,2,2,0],[2,1,1,0],[0,2,2,1],[1,1,1,1],[2,0,0,1]]]*Z(3)^0;;
```

The points of this design are elements of the 4-dimensional vector space V over $GF(3)$. Here is how to get the desing in the Design package format.

Example
```
gap> V:=Tuples([0,1,2],4)*Z(3)^0;;
gap> d1:=Union(List(bb,y->List(V,x->AsSet(x+y))));;
gap> d:=BlockDesign(81,List(d1,y->List(y,x->Position(V,x))));;
gap> AllTDesignLambdas(d);
[ 432, 32, 2 ]
```

The full automorphism group of the design is of order 2592. It is a semidirect product of the additive group of V and a group of order 32.

Example

```
gap> aut:=BlockDesignAut(d);
<permutation group with 5 generators>
gap> Size(aut);
2592
gap> StructureDescription(aut);
"(C3 x C3 x C3 x C3) : (C16 : C2)"
```

This group has three subgroups of order 648 up to conjugation. We can use the second subgroup to construct four more simple 2-(81,6,2) designs.

Example

```
gap> g:=Filtered(AllSubgroupsConjugation(aut),x->Size(x)=648);
[ <permutation group of size 648 with 7 generators>,
  <permutation group of size 648 with 7 generators>,
  <permutation group of size 648 with 7 generators> ]
gap> dd:=KramerMesnerSearch(2,81,6,2,g[2]);;
gap> List(dd,x->Size(AutomorphismGroup(x)));
[ 1296, 2592, 3888, 1944, 15552 ]
```

Two of the new designs have larger full automorphism groups than the design from [Nak21]. Using their subgroups, more simple 2-(81,6,2) designs can be constructed.

1.3.4 Quasi-symmetric 2-(56,16,18) Designs

Here is how the quasi-symmetric 2-(56,16,18) designs with intersection numbers $x = 4$, $y = 8$ from the paper [KV16] can be constructed.

Example

```
gap> g:=Group((1,2,3,4,5)(6,7,8,9,10)(11,12,13,14,15)(16,17,18,19,20)
>    (21,22,23,24,25)(26,27,28,29,30)(31,32,33,34,35)(36,37,38,39,40)
>    (41,42,43,44,45)(46,47,48,49,50)(51,52,53,54,55),
> (1,6,8)(2,21,26)(3,32,34)(4,11,5)(7,15,22)(9,16,13)(10,29,17)
>    (12,33,30)(14,19,31)(18,23,35)(24,28,36)(25,37,39)(27,38,40)
>    (42,51,49)(43,52,45)(44,46,47)(48,54,53)(50,56,55));
<permutation group with 2 generators>
gap> d:=KramerMesnerSearch(2,56,16,18,g,rec(IntersectionNumbers:=[4,8]));;
gap> Size(d);
3
```

We check that they have all required properties and compute their full automorphism groups:

Example

```
gap> List(d,AllTDesignLambdas);
[ [ 231, 66, 18 ], [ 231, 66, 18 ], [ 231, 66, 18 ] ]
gap> List(d,IntersectionNumbers);
[ [ 4, 8 ], [ 4, 8 ], [ 4, 8 ] ]
gap> aut:=List(d,BlockDesignAut);;
gap> List(aut,StructureDescription);
[ "(C2 x C2 x C2 x C2) : S5", "(C2 x C2 x C2 x C2) : A5", "PSL(3,4) : C2" ]
```

1.4 Examples: Latin Squares

See [KD15] for an introduction to Latin squares and definitions of isotopy, paratopy, etc. Multiplication tables of groups are examples of Latin squares.

Example
```
gap> MultiplicationTable(CyclicGroup(7));
[ [ 1, 2, 3, 4, 5, 6, 7 ],
  [ 2, 3, 4, 5, 6, 7, 1 ],
  [ 3, 4, 5, 6, 7, 1, 2 ],
  [ 4, 5, 6, 7, 1, 2, 3 ],
  [ 5, 6, 7, 1, 2, 3, 4 ],
  [ 6, 7, 1, 2, 3, 4, 5 ],
  [ 7, 1, 2, 3, 4, 5, 6 ] ]
```

We can construct more examples by prescribing symmetry groups. The PAG function `KramerMesnerMOLS` performs a search for sets of s mutually orthogonal Latin squares (MOLS) of order n and a given permutation group as autotopy or autoparatopy group. The group must act on the $s+2$ point classes of the corresponding transversal design. By [Fal12] and [SVW12], an autotopy of order 5 of a Latin square of order 7 must have the following cycle structure.

Example
```
gap> a:=MultiPerm(CyclicPerm(5),[1..7],3);
(1,2,3,4,5)(8,9,10,11,12)(15,16,17,18,19)
```

There are two main classes of such Latin squares. They are multiplication tables of non-associative quasigroups.

Example
```
gap> KramerMesnerMOLS(7,1,Group(a));
[ [ [ [ 1, 3, 2, 6, 7, 4, 5 ],
        [ 7, 2, 4, 3, 6, 5, 1 ],
        [ 6, 7, 3, 5, 4, 1, 2 ],
        [ 5, 6, 7, 4, 1, 2, 3 ],
        [ 2, 1, 6, 7, 5, 3, 4 ],
        [ 3, 4, 5, 1, 2, 6, 7 ],
        [ 4, 5, 1, 2, 3, 7, 6 ] ] ],
  [ [ [ 1, 3, 5, 6, 7, 2, 4 ],
        [ 7, 2, 4, 1, 6, 3, 5 ],
        [ 6, 7, 3, 5, 2, 4, 1 ],
        [ 3, 6, 7, 4, 1, 5, 2 ],
        [ 2, 4, 6, 7, 5, 1, 3 ],
        [ 4, 5, 1, 2, 3, 6, 7 ],
        [ 5, 1, 2, 3, 4, 7, 6 ] ] ] ]
```

Single Latin squares are treated as MOLS sets of size $s=1$, hence the excess brackets. When the order n is a prime power, complete sets of $s=n-1$ MOLS are easily constructed from finite fields.

Example
```
gap> ls4:=FieldToMOLS(GF(4));
[ [ [ 1, 2, 3, 4 ],
      [ 2, 1, 4, 3 ],
      [ 3, 4, 1, 2 ],
      [ 4, 3, 2, 1 ] ],
  [ [ 1, 2, 3, 4 ],
```

```
        [ 3, 4, 1, 2 ],
        [ 4, 3, 2, 1 ],
        [ 2, 1, 4, 3 ] ],
    [ [ 1, 2, 3, 4 ],
        [ 4, 3, 2, 1 ],
        [ 2, 1, 4, 3 ],
        [ 3, 4, 1, 2 ] ] ]
gap> AreMOLS(ls4);
true
```

The package Guava contains a function `AreMOLS` (**GUAVA: AreMOLS**) to test sets of MOLS. A famous problem is to find MOLS of order 10. The Handbook of Combinatorial Designs [CD07], III.5.6 contains an example of a 1-diagonally cyclic self-orthogonal Latin square L of order 10. Self-orthogonal means that L is orthogonal to its transpose. In other words, the MOLS set $\{L, L^t\}$ is invariant under the following conjugation, simultaneously exchanging rows–columns and the two Latin squares.

Example
```
gap> c:=Sortex(Concatenation([11..20],[1..10],[31..40],[21..30]));
(1,11)(2,12)(3,13)(4,14)(5,15)(6,16)(7,17)(8,18)(9,19)(10,20)(21,
31)(22,32)(23,33)(24,34)(25,35)(26,36)(27,37)(28,38)(29,39)(30,40)
```

Furthermore, the example from [CD07] has an autotopy of order 9.

Example
```
gap> a:=MultiPerm(CyclicPerm(9),[1..10],4);
(1,2,3,4,5,6,7,8,9)(11,12,13,14,15,16,17,18,19)(21,22,23,24,25,26,
27,28,29)(31,32,33,34,35,36,37,38,39)
```

The permutations a and c generate an autoparatopy group of order 18 we can use to construct the example.

Example
```
gap> g:=Group(a,c);;
gap> Size(g);
18
gap> ls10:=KramerMesnerMOLS(10,2,g);;
gap> List(ls10,AreMOLS);
[ true, true, true, true, true ]
```

We see that there are 5 inequivalent pairs of MOLS with g as autoparatopy group. Here is one pair.

Example
```
gap> ls10[1];
[ [ [ 1, 3, 6, 9, 2, 10, 5, 7, 4, 8 ],
        [ 5, 2, 4, 7, 1, 3, 10, 6, 8, 9 ],
        [ 9, 6, 3, 5, 8, 2, 4, 10, 7, 1 ],
        [ 8, 1, 7, 4, 6, 9, 3, 5, 10, 2 ],
        [ 10, 9, 2, 8, 5, 7, 1, 4, 6, 3 ],
        [ 7, 10, 1, 3, 9, 6, 8, 2, 5, 4 ],
        [ 6, 8, 10, 2, 4, 1, 7, 9, 3, 5 ],
        [ 4, 7, 9, 10, 3, 5, 2, 8, 1, 6 ],
        [ 2, 5, 8, 1, 10, 4, 6, 3, 9, 7 ],
        [ 3, 4, 5, 6, 7, 8, 9, 1, 2, 10 ] ],
    [ [ 1, 5, 9, 8, 10, 7, 6, 4, 2, 3 ],
```

```
        [ 3, 2, 6, 1, 9, 10, 8, 7, 5, 4 ],
        [ 6, 4, 3, 7, 2, 1, 10, 9, 8, 5 ],
        [ 9, 7, 5, 4, 8, 3, 2, 10, 1, 6 ],
        [ 2, 1, 8, 6, 5, 9, 4, 3, 10, 7 ],
        [ 10, 3, 2, 9, 7, 6, 1, 5, 4, 8 ],
        [ 5, 10, 4, 3, 1, 8, 7, 2, 6, 9 ],
        [ 7, 6, 10, 5, 4, 2, 9, 8, 3, 1 ],
        [ 4, 8, 7, 10, 6, 5, 3, 1, 9, 2 ],
        [ 8, 9, 1, 2, 3, 4, 5, 6, 7, 10 ] ] ]
```

1.5 Examples: Cubes of Symmetric Designs

Cubes of symmetric designs are studied in the paper [KPT24]. Here is an example.

Example

```
gap> c:=DifferenceCube(Group((1,2,3,4,5,6,7)),[1,2,4],3);
[ [ [ 1, 1, 0, 1, 0, 0, 0 ],
      [ 1, 0, 1, 0, 0, 0, 1 ],
      [ 0, 1, 0, 0, 0, 1, 1 ],
      [ 1, 0, 0, 0, 1, 1, 0 ],
      [ 0, 0, 0, 1, 1, 0, 1 ],
      [ 0, 0, 1, 1, 0, 1, 0 ],
      [ 0, 1, 1, 0, 1, 0, 0 ] ],
  [ [ 1, 0, 1, 0, 0, 0, 1 ],
      [ 0, 1, 0, 0, 0, 1, 1 ],
      [ 1, 0, 0, 0, 1, 1, 0 ],
      [ 0, 0, 0, 1, 1, 0, 1 ],
      [ 0, 0, 1, 1, 0, 1, 0 ],
      [ 0, 1, 1, 0, 1, 0, 0 ],
      [ 1, 1, 0, 1, 0, 0, 0 ] ],
  [ [ 0, 1, 0, 0, 0, 1, 1 ],
      [ 1, 0, 0, 0, 1, 1, 0 ],
      [ 0, 0, 0, 1, 1, 0, 1 ],
      [ 0, 0, 1, 1, 0, 1, 0 ],
      [ 0, 1, 1, 0, 1, 0, 0 ],
      [ 1, 1, 0, 1, 0, 0, 0 ],
      [ 1, 0, 1, 0, 0, 0, 1 ] ],
  [ [ 1, 0, 0, 0, 1, 1, 0 ],
      [ 0, 0, 0, 1, 1, 0, 1 ],
      [ 0, 0, 1, 1, 0, 1, 0 ],
      [ 0, 1, 1, 0, 1, 0, 0 ],
      [ 1, 1, 0, 1, 0, 0, 0 ],
      [ 1, 0, 1, 0, 0, 0, 1 ],
      [ 0, 1, 0, 0, 0, 1, 1 ] ],
  [ [ 0, 0, 0, 1, 1, 0, 1 ],
      [ 0, 0, 1, 1, 0, 1, 0 ],
      [ 0, 1, 1, 0, 1, 0, 0 ],
      [ 1, 1, 0, 1, 0, 0, 0 ],
      [ 1, 0, 1, 0, 0, 0, 1 ],
      [ 0, 1, 0, 0, 0, 1, 1 ],
      [ 1, 0, 0, 0, 1, 1, 0 ] ],
  [ [ 0, 0, 1, 1, 0, 1, 0 ],
```

```
        [ 0, 1, 1, 0, 1, 0, 0 ],
        [ 1, 1, 0, 1, 0, 0, 0 ],
        [ 1, 0, 1, 0, 0, 0, 1 ],
        [ 0, 1, 0, 0, 0, 1, 1 ],
        [ 1, 0, 0, 0, 1, 1, 0 ],
        [ 0, 0, 0, 1, 1, 0, 1 ] ],
    [ [ 0, 1, 1, 0, 1, 0, 0 ],
        [ 1, 1, 0, 1, 0, 0, 0 ],
        [ 1, 0, 1, 0, 0, 0, 1 ],
        [ 0, 1, 0, 0, 0, 1, 1 ],
        [ 1, 0, 0, 0, 1, 1, 0 ],
        [ 0, 0, 0, 1, 1, 0, 1 ],
        [ 0, 0, 1, 1, 0, 1, 0 ] ] ]
```

This is a 3-dimensional array of zeros and ones such that all 2-dimensional slices are incidence matrices of $(7,3,1)$ designs. For example, here is a slice obtained by varying coordinates $1,3$ and setting coordinate 2 to 7.

Example

```
gap> m:=CubeSlice(c,1,3,[7]);
[ [ 0, 1, 1, 0, 1, 0, 0 ],
  [ 1, 1, 0, 1, 0, 0, 0 ],
  [ 1, 0, 1, 0, 0, 0, 1 ],
  [ 0, 1, 0, 0, 0, 1, 1 ],
  [ 1, 0, 0, 0, 1, 1, 0 ],
  [ 0, 0, 0, 1, 1, 0, 1 ],
  [ 0, 0, 1, 1, 0, 1, 0 ] ]
gap> m*TransposedMat(m);
[ [ 3, 1, 1, 1, 1, 1, 1 ],
  [ 1, 3, 1, 1, 1, 1, 1 ],
  [ 1, 1, 3, 1, 1, 1, 1 ],
  [ 1, 1, 1, 3, 1, 1, 1 ],
  [ 1, 1, 1, 1, 3, 1, 1 ],
  [ 1, 1, 1, 1, 1, 3, 1 ],
  [ 1, 1, 1, 1, 1, 1, 3 ] ]
```

A cube of arbitrary dimension $n \geq 2$ can be constructed from a difference set in a group by calling `DifferenceCube` (2.6.1). The function uses the representation of difference sets from the DifSets package (**DifSets: Difference Sets**). For $n=2$, the difference cube is simply an incidence matrix of the associated symmetric design, i.e. the development of the difference set.

Example

```
gap> g:=SmallGroup(15,1);
<pc group of size 15 with 2 generators>
gap> StructureDescription(g);
"C15"
gap> ds:=DifferenceSets(g);
[ [ 1, 2, 3, 4, 8, 11, 12 ] ]
gap> m:=DifferenceCube(g,ds[1],2);
[ [ 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0 ],
  [ 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1 ],
  [ 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1 ],
  [ 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0 ],
  [ 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1 ],
```

```
  [ 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0 ],
  [ 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1 ],
  [ 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1 ],
  [ 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0 ],
  [ 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0 ],
  [ 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0 ],
  [ 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0 ],
  [ 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1 ],
  [ 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0 ],
  [ 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1 ] ]
gap> d:=BlockDesign(15,List(m,x->Positions(x,1)));;
gap> AllTDesignLambdas(d);
[ 15, 7, 3 ]
```

The function `DifferenceSets` (**DifSets: DifferenceSets**) returns a list of all difference sets up to equivalence in a given group. Here is a small 4-dimensional $(3,2,1)$ cube.

Example

```
gap> c:=DifferenceCube(Group((1,2,3)),[1,2],4);
[ [ [ [ 1, 1, 0 ], [ 1, 0, 1 ], [ 0, 1, 1 ] ],
      [ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 0 ] ],
      [ [ 0, 1, 1 ], [ 1, 1, 0 ], [ 1, 0, 1 ] ] ],
  [ [ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 0 ] ],
      [ [ 0, 1, 1 ], [ 1, 1, 0 ], [ 1, 0, 1 ] ],
      [ [ 1, 1, 0 ], [ 1, 0, 1 ], [ 0, 1, 1 ] ] ],
  [ [ [ 0, 1, 1 ], [ 1, 1, 0 ], [ 1, 0, 1 ] ],
      [ [ 1, 1, 0 ], [ 1, 0, 1 ], [ 0, 1, 1 ] ],
      [ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 0 ] ] ] ]
gap> CubeTest(c);
[ [ 3, 2, 1 ] ]
```

The function `CubeTest` (2.6.13) looks at all possible slices and checks if they are incidence matrices of (v,k,λ) designs. In the next example we construct all 3-dimensional difference cubes of order 21.

Example

```
gap> g:=AllSmallGroups(21);;
gap> List(g,StructureDescription);
[ "C7 : C3", "C21" ]
gap> ds:=List(g,DifferenceSets);
[ [ [ 1, 2, 3, 9, 10 ] ], [ [ 1, 2, 7, 10, 16 ] ] ]
gap> c1:=DifferenceCube(g[1],ds[1][1],3);;
gap> c2:=DifferenceCube(g[2],ds[2][1],3);;
gap> List([c1,c2],CubeTest);
[ [ [ 21, 5, 1 ] ], [ [ 21, 5, 1 ] ] ]
gap> Size(CubeAut(c1));
1323
gap> Size(CubeAut(c2));
2646
```

The function `CubeAut` (2.6.15) computes the full autotopy group of a cube. By setting options, full autoparatopy groups can also be obtained. We can make a non-difference cube by the "group cube" construction of Theorem 4.1 from [KPT24]. First we search for all $(21,5,1)$ designs with blocks being difference sets in the Frobenius group of order 21.

Example
```
gap> allds:=Filtered(Combinations([1..21],5),x->IsDifferenceSet(g[1],x));;
gap> Size(allds);
294
gap> A:=KramerMesnerMat(Group(()),Combinations([1..21],2),allds,1,21);;
gap> PAGGlobalOptions.Silent:=true;;
gap> sol:=AsSet(SolveKramerMesner(A));;
gap> des:=List(sol,x->BaseBlocks(allds,x));;
gap> Size(des);
70
```

Among these 70 designs, 14 are left developments, and 14 are right developments. The remaining 42 designs are not developments, but all of their blocks are difference sets.

Example
```
gap> dev1:=AsSet(List(allds,x->LeftDevelopment(g[1],x).blocks));;
gap> Size(dev1);
14
gap> dev2:=AsSet(List(allds,x->RightDevelopment(g[1],x).blocks));;
gap> Size(dev2);
14
gap> nondev:=Difference(des,Union(dev1,dev2));;
gap> Size(nondev);
42
```

Now we apply the group cube construction to these 42 designs. The obtained cubes are equivalent.

Example
```
gap> cc:=List(nondev,x->GroupCube(g[1],x,3));;
gap> Size(CubeFilter(cc));
1
```

The function `CubeFilter` (2.6.16) eliminates equivalent copies from a list of cubes. Our new cube is not equivalent with the two $(21,5,1)$ difference cubes.

Example
```
gap> c3:=cc[1];;
gap> CubeTest(c3);
[ [ 21, 5, 1 ] ]
gap> Size(CubeFilter([c1,c2,c3]));
3
gap> Size(CubeAut(c3));
441
```

However, the three cubes have the same slice invariant; see [KPT24] for the definition.

Example
```
gap> List([c1,c2,c3],SliceInvariant);
[ [ [ [ [ 120960, 21 ] ], 3 ] ], [ [ [ [ 120960, 21 ] ], 3 ] ],
  [ [ [ [ 120960, 21 ] ], 3 ] ] ]
```

Cubes with slice invariants different from any difference cube can be constructed for parameters of the form $(4^m, 2^{m-1}(2^m-1), 2^{m-1}(2^{m-1}-1))$, $m \geq 2$.

Example
```
gap> m:=2;; n:=3;;
gap> cl:=List([1,2,3],i->GroupCube(SDPSeriesGroup(m),SDPSeriesDesign(m,i),n));;
gap> List(cl,CubeTest);
[ [ [ 16, 6, 2 ] ], [ [ 16, 6, 2 ] ], [ [ 16, 6, 2 ] ] ]
gap> List(cl,SliceInvariant);
[ [ [ [ [ 11520, 16 ] ], 3 ] ],
  [ [ [ [ 768, 16 ] ], 2 ], [ [ [ 11520, 16 ] ], 1 ] ],
  [ [ [ [ 384, 16 ] ], 2 ], [ [ [ 11520, 16 ] ], 1 ] ] ]
```

The first cube in the list `cl` is a difference cube. The other two cubes are not, because they have non-isomorphic slices in different directions. This construction works for all $m \geq 2$ and dimensions $n \geq 3$, but it takes a lot of time and memory for bigger values of m and n. We classified all 3-dimensional group cubes of $(16,6,2)$ designs; they are available at https://web.math.pmf.unizg.hr/~krcko/results/cubes.html. A list of 1423 non-group cubes of $(16,6,2)$ designs is also provided.

The package DifSets contains precomputed lists of difference sets up to equivalence. They are loaded by the function `LoadDifferenceSets` (**DifSets: LoadDifferenceSets**). We can use them to compute all difference cubes up to equivalence.

Example
```
gap> v:=27;
27
gap> l1:=Concatenation(List([1..NrSmallGroups(v)],
> i->List(LoadDifferenceSets(v,i),x->[i,x])));
[ [ 4, [ 1, 2, 3, 4, 5, 6, 9, 12, 16, 19, 20, 23, 26 ] ],
  [ 4, [ 1, 2, 3, 4, 5, 7, 8, 9, 13, 15, 18, 19, 23 ] ],
  [ 5, [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 15, 23, 25, 27 ] ] ]
```

The list `l1` now contains all inequivalent difference sets in groups of order 27. The first entry is the group ID from the GAP library of small groups, followed by the difference set.

Example
```
gap> StructureDescription(SmallGroup(27,4));
"C9 : C3"
gap> StructureDescription(SmallGroup(27,5));
"C3 x C3 x C3"
gap> l2:=List(l1,x->DifferenceCube(SmallGroup(v,x[1]),x[2],3));;
gap> l3:=l1{CubeFilter(l2,rec(Positions:=true))};
[ [ 4, [ 1, 2, 3, 4, 5, 6, 9, 12, 16, 19, 20, 23, 26 ] ],
  [ 5, [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 15, 23, 25, 27 ] ] ]
```

The list `l3` contains difference sets giving 3-cubes that are inequivalent (not paratopic). Notice that the two cubes arising from difference sets in $\mathbb{Z}_9 \rtimes \mathbb{Z}_3$ (group ID 4) are paratopic, but not isotopic:

Example
```
gap> l4:=l1{CubeFilter(l2,rec(Positions:=true,Isotopy:=true))};
[ [ 4, [ 1, 2, 3, 4, 5, 6, 9, 12, 16, 19, 20, 23, 26 ] ],
  [ 4, [ 1, 2, 3, 4, 5, 7, 8, 9, 13, 15, 18, 19, 23 ] ],
  [ 5, [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 15, 23, 25, 27 ] ] ]
```

We will now construct some non-difference group cubes in $\mathbb{Z}_9 \rtimes \mathbb{Z}_3$. Here is an way to get all difference sets, including equivalent ones.

Example
```
gap> g:=SmallGroup(v,4);
<pc group of size 27 with 3 generators>
gap> ds:=LoadDifferenceSets(v,4);
[ [ 1, 2, 3, 4, 5, 6, 9, 12, 16, 19, 20, 23, 26 ],
  [ 1, 2, 3, 4, 5, 7, 8, 9, 13, 15, 18, 19, 23 ] ]
gap> allds:=EquivalentDifferenceSets(g,ds);;
gap> Size(allds);
972
```

For parameters $(21,5,1)$ we could search for all designs with difference sets as blocks. This would take too much time for $(27,13,6)$, so we prescribe an automorphism group of order 3.

Example
```
gap> ge:=ExtendedPermRepresentation(g);
<permutation group with 7 generators>
gap> sub:=AllSubgroupsConjugation(ge);;
gap> h:=sub[4];
Group([ (1,10,4)(2,15,7)(3,17,9)(5,20,12)(6,22,14)(8,23,16)
  (11,25,19)(13,26,21)(18,27,24) ])
gap> alldsorb:=List(Orbits(h,allds,OnSets),Representative);;
gap> Size(alldsorb);
324
gap> pairsorb:=List(Orbits(h,Combinations([1..27],2),OnSets),Representative);;
gap> Size(pairsorb);
117
gap> A:=KramerMesnerMat(h,pairsorb,alldsorb,6,27);;
gap> sol:=AsSet(SolveKramerMesner(A));;
gap> des:=List(sol,x->BlockDesign(27,BaseBlocks(alldsorb,x),h).blocks);;
gap> Size(des);
288
```

We get 288 designs with difference sets as blocks. Let us remove the ones which are developments of their blocks.

Example
```
gap> dev1:=AsSet(List(allds,x->LeftDevelopment(g,x).blocks));;
gap> dev2:=AsSet(List(allds,x->RightDevelopment(g,x).blocks));;
gap> nondev:=List(Difference(des,Union(dev1,dev2)),x->[4,x]);;
gap> Size(nondev);
216
```

Next, we remove the ones leading to equivalent 3-cubes.

Example
```
gap> cc:=List(nondev,x->GroupCube(SmallGroup(v,x[1]),x[2],3));;
gap> l5:=nondev{CubeFilter(cc,rec(Positions:=true))};
[ [ 4,
      [ [ 1, 2, 3, 4, 5, 6, 9, 12, 16, 19, 20, 23, 26 ],
          [ 1, 2, 3, 4, 5, 7, 10, 13, 14, 19, 21, 22, 24 ],
          [ 1, 2, 3, 7, 11, 12, 13, 15, 20, 23, 24, 25, 27 ],
          [ 1, 2, 4, 6, 10, 11, 13, 14, 15, 17, 18, 20, 26 ],
          [ 1, 2, 4, 8, 9, 12, 13, 16, 17, 18, 22, 24, 27 ],
          [ 1, 2, 9, 10, 11, 14, 16, 17, 19, 21, 23, 25, 27 ],
          [ 1, 3, 4, 7, 8, 11, 17, 18, 19, 22, 23, 25, 26 ],
```

```
            [ 1, 3, 5, 8, 9, 10, 14, 15, 18, 23, 24, 26, 27 ],
            [ 1, 3, 5, 8, 10, 11, 12, 15, 16, 17, 20, 21, 22 ],
            [ 1, 4, 6, 7, 9, 10, 12, 15, 21, 22, 25, 26, 27 ],
            [ 1, 5, 6, 7, 9, 11, 14, 16, 18, 20, 22, 24, 25 ],
            [ 1, 5, 6, 8, 13, 17, 19, 20, 21, 24, 25, 26, 27 ],
            [ 1, 6, 7, 8, 12, 13, 14, 15, 16, 18, 19, 21, 23 ],
            [ 2, 3, 5, 6, 9, 13, 15, 17, 18, 21, 22, 23, 25 ],
            [ 2, 3, 6, 7, 8, 9, 11, 12, 14, 17, 21, 24, 26 ],
            [ 2, 3, 6, 8, 10, 12, 14, 18, 19, 20, 22, 25, 27 ],
            [ 2, 4, 5, 7, 8, 9, 11, 15, 18, 19, 20, 21, 27 ],
            [ 2, 4, 8, 14, 15, 16, 20, 21, 22, 23, 24, 25, 26 ],
            [ 2, 5, 6, 7, 8, 10, 11, 13, 16, 22, 23, 26, 27 ],
            [ 2, 5, 7, 10, 12, 15, 16, 17, 18, 19, 24, 25, 26 ],
            [ 3, 4, 5, 11, 12, 13, 14, 16, 18, 21, 25, 26, 27 ],
            [ 3, 4, 6, 7, 10, 16, 17, 18, 20, 21, 23, 24, 27 ],
            [ 3, 4, 6, 8, 9, 10, 11, 13, 15, 16, 19, 24, 25 ],
            [ 3, 7, 9, 13, 14, 15, 16, 17, 19, 20, 22, 26, 27 ],
            [ 4, 5, 6, 11, 12, 14, 15, 17, 19, 22, 23, 24, 27 ],
            [ 4, 5, 7, 8, 9, 10, 12, 13, 14, 17, 20, 23, 25 ],
            [ 9, 10, 11, 12, 13, 18, 19, 20, 21, 22, 23, 24, 26 ] ] ],
  [ 4,
      [ [ 1, 2, 3, 4, 5, 6, 9, 12, 16, 19, 20, 23, 26 ],
            [ 1, 2, 3, 5, 7, 11, 14, 15, 18, 20, 23, 24, 25 ],
            [ 1, 2, 3, 7, 9, 13, 14, 17, 19, 20, 21, 22, 27 ],
            [ 1, 2, 4, 6, 7, 8, 10, 11, 13, 18, 20, 22, 26 ],
            [ 1, 2, 4, 10, 12, 14, 15, 21, 22, 23, 25, 26, 27 ],
            [ 1, 2, 5, 8, 12, 13, 17, 18, 19, 21, 24, 25, 26 ],
            [ 1, 3, 4, 6, 8, 11, 13, 15, 17, 19, 23, 25, 27 ],
            [ 1, 3, 5, 8, 10, 11, 12, 15, 16, 17, 20, 21, 22 ],
            [ 1, 3, 6, 7, 8, 9, 10, 12, 18, 21, 23, 24, 27 ],
            [ 1, 4, 5, 6, 7, 10, 13, 14, 15, 16, 19, 21, 24 ],
            [ 1, 4, 8, 9, 14, 15, 16, 17, 18, 20, 24, 26, 27 ],
            [ 1, 5, 6, 9, 11, 12, 13, 14, 16, 18, 22, 25, 27 ],
            [ 1, 7, 9, 10, 11, 16, 17, 19, 22, 23, 24, 25, 26 ],
            [ 2, 3, 4, 5, 10, 13, 16, 17, 18, 22, 23, 24, 27 ],
            [ 2, 3, 4, 8, 9, 10, 11, 14, 16, 18, 19, 21, 25 ],
            [ 2, 3, 6, 9, 10, 11, 12, 13, 14, 15, 17, 24, 26 ],
            [ 2, 4, 5, 7, 8, 9, 11, 12, 15, 19, 22, 24, 27 ],
            [ 2, 5, 6, 7, 8, 11, 14, 16, 17, 21, 23, 26, 27 ],
            [ 2, 6, 7, 10, 12, 15, 16, 17, 18, 19, 20, 25, 27 ],
            [ 2, 6, 8, 9, 13, 15, 16, 20, 21, 22, 23, 24, 25 ],
            [ 3, 4, 5, 6, 7, 9, 15, 17, 18, 21, 22, 25, 26 ],
            [ 3, 4, 7, 11, 12, 13, 16, 20, 21, 24, 25, 26, 27 ],
            [ 3, 5, 6, 8, 10, 14, 19, 20, 22, 24, 25, 26, 27 ],
            [ 3, 7, 8, 12, 13, 14, 15, 16, 18, 19, 22, 23, 26 ],
            [ 4, 5, 7, 8, 9, 10, 12, 13, 14, 17, 20, 23, 25 ],
            [ 4, 6, 11, 12, 14, 17, 18, 19, 20, 21, 22, 23, 24 ],
            [ 5, 9, 10, 11, 13, 15, 18, 19, 20, 21, 23, 26, 27 ] ] ] ]
```

We have constructed two $(27,13,6)$ designs with blocks being difference sets in $\mathbb{Z}_9 \rtimes \mathbb{Z}_3$, which are not their developments. Here are the slice invariants of the difference and non-difference group 3-cubes constructed so far.

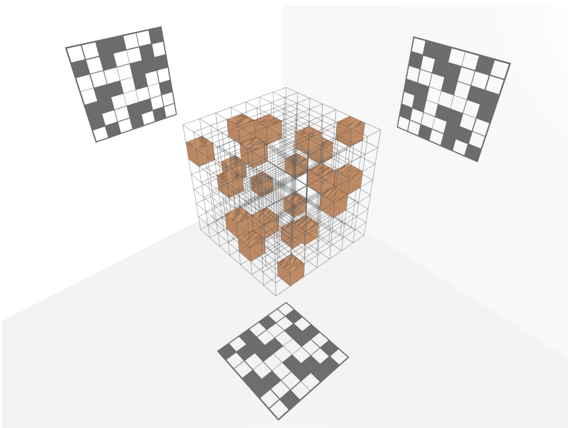
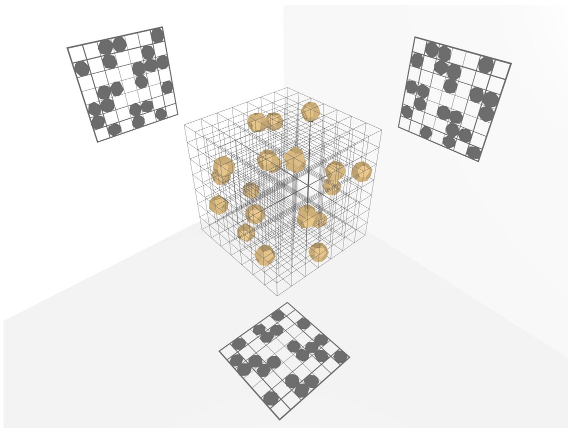
Example

```
gap> dc:=List(l3,x->DifferenceCube(SmallGroup(v,x[1]),x[2],3));;
gap> gc:=List(l5,x->GroupCube(SmallGroup(v,x[1]),x[2],3));;
gap> List(dc,SliceInvariant);
[ [ [ [ [ 1053, 27 ] ], 3 ] ], [ [ [ [ 1053, 27 ] ], 3 ] ] ]
gap> List(gc,SliceInvariant);
[ [ [ [ [ 27, 27 ] ], 2 ], [ [ [ 1053, 27 ] ], 1 ] ],
  [ [ [ [ 27, 27 ] ], 2 ], [ [ [ 1053, 27 ] ], 1 ] ] ]
```

More examples of difference and non-difference group cubes are available on our web page:
`https://web.math.pmf.unizg.hr/~krcko/results/cubes.html`

1.6 Examples: Projection Cubes of Symmetric Designs

Projection cubes of symmetric designs are introduced and studied in [KR24]. They are n-dimensional matrices of zeros and ones such that all 2-dimensional projections are incidence matrices of symmetric (v,k,λ) designs. The set of all such matrices is denoted $P^n(v,k,\lambda)$. Here are two pictures of $P^3(7,3,1)$-cubes.

These are the cubes C_1 and C_3 from [KR24]. Incidence cubes can be represented as sets of indices of the 1-entries. The two cubes above have the following “orthogonal array representations”.

Example

```
gap> C1:=[[1,2,3],[1,4,5],[1,6,7],[2,3,1],[2,4,6],[2,7,5],[3,1,2],[3,6,5],
> [3,7,4],[4,3,7],[4,5,1],[4,6,2],[5,1,4],[5,2,7],[5,3,6],[6,2,4],[6,5,3],
> [6,7,1],[7,1,6],[7,4,3],[7,5,2]];;
gap> C3:=[[1,2,4],[1,4,6],[1,6,2],[2,3,7],[2,4,3],[2,7,4],[3,1,6],[3,6,7],
> [3,7,1],[4,3,6],[4,5,3],[4,6,5],[5,1,2],[5,2,3],[5,3,1],[6,2,7],[6,5,2],
> [6,7,5],[7,1,4],[7,4,5],[7,5,1]];;
```

We can switch between this representation and n-dimensional matrices with the functions `OrthogonalArrayToCube` (2.6.8) and `CubeToOrthogonalArray` (2.6.7).

Example

```
gap> C1c:=OrthogonalArrayToCube(C1);
[ [ [ 0, 0, 0, 0, 0, 0, 0 ],
      [ 0, 0, 1, 0, 0, 0, 0 ],
      [ 0, 0, 0, 0, 0, 0, 0 ],
```

```
        [ 0, 0, 0, 0, 1, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 1 ],
        [ 0, 0, 0, 0, 0, 0, 0 ] ],
    [ [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 1, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 1, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 1, 0, 0 ] ],
    [ [ 0, 1, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 1, 0, 0 ],
        [ 0, 0, 0, 1, 0, 0, 0 ] ],
    [ [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 1 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 1, 0, 0, 0, 0, 0, 0 ],
        [ 0, 1, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ] ],
    [ [ 0, 0, 0, 1, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 1 ],
        [ 0, 0, 0, 0, 0, 1, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ] ],
    [ [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 1, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 1, 0, 0, 0, 0, 0, 0 ] ],
    [ [ 0, 0, 0, 0, 0, 1, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 1, 0, 0, 0, 0 ],
        [ 0, 1, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ],
        [ 0, 0, 0, 0, 0, 0, 0 ] ] ]
gap> CubeProjectionTest(C1c);
[ [ 7, 3, 1 ] ]
```

The function `CubeProjectionTest` (2.7.3) checks if an n-dimensional matrix is a $P^n(v,k,\lambda)$-cube. The result should be `[[v,k,lambda]]`. Anything else means it does not satisfy the requirements. There is a faster function `OrthogonalArrayProjectionTest` (2.7.6) that works directly with the orthogonal array representation.

Example
```
gap> OrthogonalArrayProjectionTest(C3);
[ [ 7, 3, 1 ] ]
```

Functions `CubeFilter` (2.6.16) and `CubeAut` (2.6.15) also have versions that work with orthogonal arrays.

Example
```
gap> Size(OrthogonalArrayFilter([C1,C3]));
2
```

This means that C_1 and C_3 are not equivalent (paratopic). They are distinguished by the size of the full autoparatopy group.

Example
```
gap> Size(OrthogonalArrayAut(C1,rec(Paratopy:=true)));
63
gap> Size(OrthogonalArrayAut(C3,rec(Paratopy:=true)));
42
```

Projection cubes can be constructed from n-dimensional difference sets. The family of Paley difference sets extends naturally to higher dimensions.

Example
```
gap> D4:=PaleyDifferenceSet(7);
[ [ 0*Z(7), Z(7)^0, Z(7), Z(7)^2, Z(7)^3, Z(7)^4, Z(7)^5 ],
  [ 0*Z(7), Z(7)^2, Z(7)^3, Z(7)^4, Z(7)^5, Z(7)^0, Z(7) ],
  [ 0*Z(7), Z(7)^4, Z(7)^5, Z(7)^0, Z(7), Z(7)^2, Z(7)^3 ] ]
gap> C4:=DifferenceSetToOrthogonalArray(D4);
[ [ 1, 2, 3, 4, 5, 6, 7 ], [ 2, 4, 6, 3, 1, 7, 5 ], [ 3, 6, 5, 7, 4, 1, 2 ],
  [ 4, 3, 7, 6, 2, 5, 1 ], [ 5, 1, 4, 2, 7, 3, 6 ], [ 6, 7, 1, 5, 3, 2, 4 ],
  [ 7, 5, 2, 1, 6, 4, 3 ], [ 1, 4, 5, 6, 7, 2, 3 ], [ 2, 3, 1, 7, 5, 4, 6 ],
  [ 3, 7, 4, 1, 2, 6, 5 ], [ 4, 6, 2, 5, 1, 3, 7 ], [ 5, 2, 7, 3, 6, 1, 4 ],
  [ 6, 5, 3, 2, 4, 7, 1 ], [ 7, 1, 6, 4, 3, 5, 2 ], [ 1, 6, 7, 2, 3, 4, 5 ],
  [ 2, 7, 5, 4, 6, 3, 1 ], [ 3, 1, 2, 6, 5, 7, 4 ], [ 4, 5, 1, 3, 7, 6, 2 ],
  [ 5, 3, 6, 1, 4, 2, 7 ], [ 6, 2, 4, 7, 1, 5, 3 ], [ 7, 4, 3, 5, 2, 1, 6 ] ]
gap> OrthogonalArrayProjectionTest(C4);
[ [ 7, 3, 1 ] ]
```

This is a 7-dimensional analog of the Fano plane. The cubes C_1 and C_3 are its restrictions.

Example
```
gap> AsSet(List(C4,x->x{[1,2,3]}))=C1;
true
gap> AsSet(List(C4,x->x{[1,2,4]}))=C3;
true
```

The cyclotomic difference sets (4th and 8th powers in finite fields of appropriate order) and the twin prime power difference sets also have higher-dimensional versions.

Example
```
gap> C5:=DifferenceSetToOrthogonalArray(PowerDifferenceSet(37,4));;
gap> OrthogonalArrayProjectionTest(C5);
[ [ 37, 9, 2 ] ]
gap> C6:=DifferenceSetToOrthogonalArray(PowerDifferenceSet(73,8));;
```

```
gap> OrthogonalArrayProjectionTest(C6);
[ [ 73, 9, 1 ] ]
gap> C7:=DifferenceSetToOrthogonalArray(TwinPrimePowerDifferenceSet(5));;
gap> OrthogonalArrayProjectionTest(C7);
[ [ 35, 17, 8 ] ]
```

These are projection cubes in $P^{37}(37,9,2)$, $P^{73}(73,9,1)$, and $P^5(35,17,8)$. In the paper [KR24] the following 3-dimensional $(16,6,2)$ difference set in $\mathbb{Z}_4 \times \mathbb{Z}_4$ is considered.

Example
```
gap> D4:=[[[0,0],[0,0],[1,0]],[[0,0],[1,0],[0,0]],[[0,0],[0,1],[2,0]],
> [[0,0],[2,0],[0,1]],[[0,0],[1,2],[0,3]],[[0,0],[2,3],[3,2]]];;
gap> g:=SmallGroup(16,2);
<pc group of size 16 with 4 generators>
gap> StructureDescription(g);
"C4 x C4"
```

We first convert D_4 to the DifSets package format.

Example
```
gap> toel:=x->g.1^x[1]*g.2^x[2];;
gap> D4a:=List(D4,x->List(x,toel));
[ [ <identity> of ..., <identity> of ..., f1 ],
  [ <identity> of ..., f1, <identity> of ... ], [ <identity> of ..., f2, f3 ],
  [ <identity> of ..., f3, f2 ], [ <identity> of ..., f1*f4, f2*f4 ],
  [ <identity> of ..., f2*f3*f4, f1*f3*f4 ] ]
gap> e:=Elements(g);
[ <identity> of ..., f1, f2, f3, f4, f1*f2, f1*f3, f1*f4, f2*f3, f2*f4, f3*f4,
f1*f2*f3, f1*f2*f4, f1*f3*f4, f2*f3*f4, f1*f2*f3*f4 ]
gap> D4b:=List(D4a,x->List(x,y->Position(e,y)));
[ [ 1, 1, 2 ], [ 1, 2, 1 ], [ 1, 3, 4 ], [ 1, 4, 3 ], [ 1, 8, 10 ],
  [ 1, 15, 14 ] ]
```

The function `DifferenceSetToOrthogonalArray` (2.7.7) takes either a group and an n-dimensional difference set in DifSets format, or an additive difference set containing finite field elements. The development of D_4 over $G = \mathbb{Z}_4 \times \mathbb{Z}_4$ is a projection cube in $P^3(16,6,2)$ with full autoparatopy group isomorphic to G.

Example
```
gap> C8:=DifferenceSetToOrthogonalArray(g,D4b);;
gap> OrthogonalArrayProjectionTest(C8);
[ [ 16, 6, 2 ] ]
gap> IsomorphismGroups(g,OrthogonalArrayAut(C8,rec(Paratopy:=true)));
[ f1, f2 ] -> [ (1,7,4,2)(3,12,9,6)(5,14,11,8)(10,16,15,13)(17,23,20,18)(19,28,
    25,22)(21,30,27,24)(26,32,31,29)(33,39,36,34)(35,44,41,38)(37,46,43,40)(42,
    48,47,45), (1,10,5,3)(2,13,8,6)(4,15,11,9)(7,16,14,12)(17,26,21,19)(18,29,
    24,22)(20,31,27,25)(23,32,30,28)(33,42,37,35)(34,45,40,38)(36,47,43,41)(39,
    48,46,44) ]
```

More examples of projection cubes are available on the following web page,

`https://web.math.pmf.unizg.hr/~krcko/results/pcubes.html`

including 102 cubes in $P^3(16,6,2)$ that cannot be constructed from 3-dimensional difference sets. Four of these cubes are shown in Figure 2 of [KR24]. They have the interesting property that the projections are non-isomorphic $(16,6,2)$ designs.

Example
```
gap> Read("examples.oa");
gap> fig2:=[Crrg,Crgg,Crrb,Crbb];;
gap> List(fig2,OrthogonalArrayProjectionTest);
[ [ [ 16, 6, 2 ] ], [ [ 16, 6, 2 ] ], [ [ 16, 6, 2 ] ], [ [ 16, 6, 2 ] ] ]
gap> p:=oa->List(OrthogonalArrayProjections(oa),x->Size(OrthogonalArrayAut(x)));;
gap> List(fig2,p);
[ [ 768, 11520, 11520 ], [ 768, 768, 11520 ], [ 384, 11520, 11520 ],
  [ 384, 384, 11520 ] ]
```

The four cubes are not equivalent with cubes constructed from difference sets because their autotopy groups act non-transitively on the indices.

Example
```
gap> aut:=List(fig2,OrthogonalArrayAut);
[ <permutation group with 3 generators>, <permutation group with 3 generators>,
  <permutation group with 3 generators>, <permutation group with 3 generators> ]
gap> List(aut,x->List(Orbits(x),Size));
[ [ 12, 4, 12, 4, 12, 4 ], [ 12, 4, 12, 4, 12, 4 ], [ 8, 8, 8, 8, 8, 8 ],
  [ 8, 8, 8, 8, 8, 8 ] ]
```

1.7 Examples: Mosaics of Combinatorial Designs

Mosaics of combinatorial designs were introduced in [GGP18] and a contruction from resolvable designs was proved. This construction of mosaics is implemented in PAG for affine designs.

Example
```
gap> ag123:=AffineMosaic(1,2,3);
[ [ 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3 ],
  [ 2, 3, 1, 1, 2, 3, 2, 3, 1, 2, 3, 1 ],
  [ 3, 1, 2, 1, 2, 3, 3, 1, 2, 3, 1, 2 ],
  [ 1, 2, 3, 2, 3, 1, 3, 1, 2, 2, 3, 1 ],
  [ 2, 3, 1, 2, 3, 1, 1, 2, 3, 3, 1, 2 ],
  [ 3, 1, 2, 2, 3, 1, 2, 3, 1, 1, 2, 3 ],
  [ 1, 2, 3, 3, 1, 2, 2, 3, 1, 3, 1, 2 ],
  [ 2, 3, 1, 3, 1, 2, 3, 1, 2, 1, 2, 3 ],
  [ 3, 1, 2, 3, 1, 2, 1, 2, 3, 2, 3, 1 ] ]
gap> MosaicParameters(ag123);
"2-(9,3,1) + 2-(9,3,1) + 2-(9,3,1)"
gap> ag232:=AffineMosaic(2,3,2);
[ [ 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 ],
  [ 2, 1, 1, 2, 2, 1, 1, 2, 1, 2, 2, 1, 2, 1 ],
  [ 1, 2, 2, 1, 2, 1, 1, 2, 2, 1, 1, 2, 2, 1 ],
  [ 2, 1, 2, 1, 1, 2, 1, 2, 2, 1, 2, 1, 1, 2 ],
  [ 1, 2, 1, 2, 1, 2, 2, 1, 2, 1, 2, 1, 2, 1 ],
  [ 2, 1, 1, 2, 2, 1, 2, 1, 2, 1, 1, 2, 1, 2 ],
  [ 1, 2, 2, 1, 2, 1, 2, 1, 1, 2, 2, 1, 1, 2 ],
  [ 2, 1, 2, 1, 1, 2, 2, 1, 1, 2, 1, 2, 2, 1 ] ]
gap> MosaicParameters(ag232);
"3-(8,4,1) + 3-(8,4,1)"
```

The command `AffineMosaic` uses the FinInG package and is not loaded if the package is not present.

Tilings of groups with difference sets [CKZ15] give rise to mosaics of symmetric designs. Here is an example of a $(31,6,1)$ tiling and the corresponding mosaic.

Example
```
gap> t:=[ [ 1, 5, 11, 24, 25, 27 ],
> [ 2, 10, 17, 19, 22, 23 ],
> [ 3, 4, 7, 13, 15, 20 ],
> [ 6, 8, 9, 14, 26, 30 ],
> [ 12, 16, 18, 21, 28, 29 ] ];;
gap> m:=DifferenceMosaic(CyclicGroup(31), t);;
gap> MosaicParameters(m);
"2-(31,6,1) + 2-(31,6,1) + 2-(31,6,1) + 2-(31,6,1) + 2-(31,6,1)"
```

The paper [Krc24] gives some interesting small examples of mosaics. Files containing these examples are available on the web page

https://web.math.pmf.unizg.hr/~krcko/results/mosaics.html

Example
```
gap> m:=ReadMat("13-346ex.txt")[1];
[ [ 1, 3, 3, 3, 2, 3, 2, 2, 3, 1, 3, 2, 1, 1, 3, 2, 2, 3, 3, 1, 3, 3, 3, 2, 1, 2 ],
  [ 1, 1, 3, 3, 3, 2, 3, 2, 2, 3, 1, 3, 2, 2, 1, 3, 2, 2, 3, 3, 1, 3, 3, 3, 2, 1 ],
  [ 2, 1, 1, 3, 3, 3, 2, 3, 2, 2, 3, 1, 3, 1, 2, 1, 3, 2, 2, 3, 3, 1, 3, 3, 3, 2 ],
  [ 3, 2, 1, 1, 3, 3, 3, 2, 3, 2, 2, 3, 1, 2, 1, 2, 1, 3, 2, 2, 3, 3, 1, 3, 3, 3 ],
  [ 1, 3, 2, 1, 1, 3, 3, 3, 2, 3, 2, 2, 3, 3, 2, 1, 2, 1, 3, 2, 2, 3, 3, 1, 3, 3 ],
  [ 3, 1, 3, 2, 1, 1, 3, 3, 3, 2, 3, 2, 2, 3, 3, 2, 1, 2, 1, 3, 2, 2, 3, 3, 1, 3 ],
  [ 2, 3, 1, 3, 2, 1, 1, 3, 3, 3, 2, 3, 2, 3, 3, 3, 2, 1, 2, 1, 3, 2, 2, 3, 3, 1 ],
  [ 2, 2, 3, 1, 3, 2, 1, 1, 3, 3, 3, 2, 3, 1, 3, 3, 3, 2, 1, 2, 1, 3, 2, 2, 3, 3 ],
  [ 3, 2, 2, 3, 1, 3, 2, 1, 1, 3, 3, 3, 2, 3, 1, 3, 3, 3, 2, 1, 2, 1, 3, 2, 2, 3 ],
  [ 2, 3, 2, 2, 3, 1, 3, 2, 1, 1, 3, 3, 3, 3, 3, 1, 3, 3, 3, 2, 1, 2, 1, 3, 2, 2 ],
  [ 3, 2, 3, 2, 2, 3, 1, 3, 2, 1, 1, 3, 3, 2, 3, 3, 1, 3, 3, 3, 2, 1, 2, 1, 3, 2 ],
  [ 3, 3, 2, 3, 2, 2, 3, 1, 3, 2, 1, 1, 3, 2, 2, 3, 3, 1, 3, 3, 3, 2, 1, 2, 1, 3 ],
  [ 3, 3, 3, 2, 3, 2, 2, 3, 1, 3, 2, 1, 1, 3, 2, 2, 3, 3, 1, 3, 3, 3, 2, 1, 2, 1 ] ]
gap> MosaicParameters(m);
"2-(13,3,1) + 2-(13,4,2) + 2-(13,6,5)"
```

This is the first example of an inhomogenous mosaic, containing designs with distinct parameters.

Example
```
gap> m1:=ReadMat("9-3-2ex1.txt")[1];
[ [ 1, 2, 1, 1, 2, 1, 1, 3, 3, 1, 2, 3, 1, 3, 2, 1, 3, 3, 2, 2, 3, 2, 3, 2 ],
  [ 1, 1, 2, 1, 1, 2, 3, 1, 3, 3, 1, 2, 2, 1, 3, 3, 1, 3, 3, 2, 2, 2, 2, 3 ],
  [ 2, 1, 1, 2, 1, 1, 3, 3, 1, 2, 3, 1, 3, 2, 1, 3, 3, 1, 2, 3, 2, 3, 2, 2 ],
  [ 1, 3, 2, 2, 3, 3, 1, 2, 1, 3, 3, 1, 2, 1, 2, 2, 2, 3, 1, 3, 1, 1, 3, 2 ],
  [ 2, 1, 3, 3, 2, 3, 1, 1, 2, 1, 3, 3, 2, 2, 1, 3, 2, 2, 1, 1, 3, 2, 1, 3 ],
  [ 3, 2, 1, 3, 3, 2, 2, 1, 1, 3, 1, 3, 1, 2, 2, 2, 3, 2, 3, 1, 1, 3, 2, 1 ],
  [ 2, 3, 3, 1, 3, 2, 3, 2, 2, 2, 2, 1, 1, 3, 3, 2, 1, 1, 1, 2, 3, 3, 1, 1 ],
  [ 3, 2, 3, 2, 1, 3, 2, 3, 2, 1, 2, 2, 3, 1, 3, 1, 2, 1, 3, 1, 2, 1, 3, 1 ],
  [ 3, 3, 2, 3, 2, 1, 2, 2, 3, 2, 1, 2, 3, 3, 1, 1, 1, 2, 2, 3, 1, 1, 1, 3 ] ]
gap> MosaicParameters(m1);
"2-(9,3,2) + 2-(9,3,2) + 2-(9,3,2)"
gap> m2:=ReadMat("9-3-2ex2.txt")[1];
[ [ 1, 2, 1, 1, 2, 1, 1, 3, 3, 1, 3, 3, 1, 3, 2, 1, 2, 3, 3, 2, 2, 3, 2, 2 ],
  [ 1, 1, 2, 1, 1, 2, 3, 1, 3, 3, 1, 3, 2, 1, 3, 3, 1, 2, 2, 3, 2, 2, 3, 2 ],
  [ 2, 1, 1, 2, 1, 1, 3, 3, 1, 3, 3, 1, 3, 2, 1, 2, 3, 1, 2, 2, 3, 2, 2, 3 ],
  [ 1, 3, 2, 3, 3, 1, 2, 2, 1, 2, 1, 3, 3, 3, 2, 2, 3, 2, 1, 2, 1, 1, 3, 1 ],
```

```
  [ 2, 1, 3, 1, 3, 3, 1, 2, 2, 3, 2, 1, 2, 3, 3, 2, 2, 3, 1, 1, 2, 1, 1, 3 ],
  [ 3, 2, 1, 3, 1, 3, 2, 1, 2, 1, 3, 2, 3, 2, 3, 3, 2, 2, 2, 1, 1, 3, 1, 1 ],
  [ 2, 3, 3, 2, 3, 2, 2, 3, 1, 1, 2, 2, 1, 1, 2, 3, 1, 1, 1, 3, 3, 3, 1, 2 ],
  [ 3, 2, 3, 2, 2, 3, 1, 2, 3, 2, 1, 2, 2, 1, 1, 1, 3, 1, 3, 1, 3, 2, 3, 1 ],
  [ 3, 3, 2, 3, 2, 2, 3, 1, 2, 2, 2, 1, 1, 2, 1, 1, 1, 3, 3, 3, 1, 1, 2, 3 ] ]
gap> MosaicParameters(m2);
"2-(9,3,2) + 2-(9,3,2) + 2-(9,3,2)"
```

These two mosaics cannot be obtained by the construction from [GGP18]. The first mosaic contains three isomorphic copies of a 2-$(9,3,2)$ design that is not resolvable.

Example
```
gap> d1:=BlockDesignFilter(MosaicToBlockDesigns(m1));
[ rec( blocks := [ [ 1, 2, 4 ], [ 1, 2, 7 ], [ 1, 3, 6 ], [ 1, 3, 9 ],
          [ 1, 4, 5 ], [ 1, 5, 8 ], [ 1, 6, 7 ], [ 1, 8, 9 ], [ 2, 3, 5 ],
          [ 2, 3, 8 ], [ 2, 4, 8 ], [ 2, 5, 6 ], [ 2, 6, 9 ], [ 2, 7, 9 ],
          [ 3, 4, 6 ], [ 3, 4, 7 ], [ 3, 5, 9 ], [ 3, 7, 8 ], [ 4, 5, 7 ],
          [ 4, 6, 9 ], [ 4, 8, 9 ], [ 5, 6, 8 ], [ 5, 7, 9 ], [ 6, 7, 8 ] ],
      isBlockDesign := true, v := 9 ) ]
gap> MakeResolutionsComponent(d1[1]);
gap> d1[1].resolutions.list;
[  ]
```

The second mosaic contains three non-isomorphic designs, one resolvable and two not resolvable.

Example
```
gap> d2:=BlockDesignFilter(MosaicToBlockDesigns(m2));
[ rec( blocks := [ [ 1, 2, 4 ], [ 1, 2, 5 ], [ 1, 3, 4 ], [ 1, 3, 6 ],
          [ 1, 5, 8 ], [ 1, 6, 7 ], [ 1, 7, 9 ], [ 1, 8, 9 ], [ 2, 3, 5 ],
          [ 2, 3, 6 ], [ 2, 4, 8 ], [ 2, 6, 9 ], [ 2, 7, 8 ], [ 2, 7, 9 ],
          [ 3, 4, 7 ], [ 3, 5, 9 ], [ 3, 7, 8 ], [ 3, 8, 9 ], [ 4, 5, 7 ],
          [ 4, 5, 9 ], [ 4, 6, 8 ], [ 4, 6, 9 ], [ 5, 6, 7 ], [ 5, 6, 8 ] ],
      isBlockDesign := true, v := 9 ),
  rec( blocks := [ [ 1, 2, 5 ], [ 1, 2, 7 ], [ 1, 3, 4 ], [ 1, 3, 9 ],
          [ 1, 4, 7 ], [ 1, 5, 6 ], [ 1, 6, 8 ], [ 1, 8, 9 ], [ 2, 3, 6 ],
          [ 2, 3, 8 ], [ 2, 4, 6 ], [ 2, 4, 9 ], [ 2, 5, 8 ], [ 2, 7, 9 ],
          [ 3, 4, 5 ], [ 3, 5, 7 ], [ 3, 6, 9 ], [ 3, 7, 8 ], [ 4, 5, 8 ],
          [ 4, 6, 7 ], [ 4, 8, 9 ], [ 5, 6, 9 ], [ 5, 7, 9 ], [ 6, 7, 8 ] ],
      isBlockDesign := true, v := 9 ),
  rec( blocks := [ [ 1, 2, 4 ], [ 1, 2, 8 ], [ 1, 3, 6 ], [ 1, 3, 7 ],
          [ 1, 4, 5 ], [ 1, 5, 9 ], [ 1, 6, 7 ], [ 1, 8, 9 ], [ 2, 3, 5 ],
          [ 2, 3, 9 ], [ 2, 4, 8 ], [ 2, 5, 6 ], [ 2, 6, 7 ], [ 2, 7, 9 ],
          [ 3, 4, 6 ], [ 3, 4, 8 ], [ 3, 5, 9 ], [ 3, 7, 8 ], [ 4, 5, 7 ],
          [ 4, 6, 9 ], [ 4, 7, 9 ], [ 5, 6, 8 ], [ 5, 7, 8 ], [ 6, 8, 9 ] ],
      isBlockDesign := true, v := 9 ) ]
gap> MakeResolutionsComponent(d2[1]);
gap> MakeResolutionsComponent(d2[2]);
gap> MakeResolutionsComponent(d2[3]);
gap> d2[1].resolutions.list;
[ rec( autGroup := Group([ (1,5,8)(2,6,9)(3,4,7), (1,7,6)(2,8,4)(3,9,5), (1,2)
          (4,5)(7,9) ]),
      partition :=
        [
          rec( blocks := [ [ 1, 2, 4 ], [ 3, 8, 9 ], [ 5, 6, 7 ] ],
```

```
            isBlockDesign := true, v := 9 ),
        rec( blocks := [ [ 1, 2, 5 ], [ 3, 7, 8 ], [ 4, 6, 9 ] ],
            isBlockDesign := true, v := 9 ),
        rec( blocks := [ [ 1, 3, 4 ], [ 2, 7, 9 ], [ 5, 6, 8 ] ],
            isBlockDesign := true, v := 9 ),
        rec( blocks := [ [ 1, 3, 6 ], [ 2, 7, 8 ], [ 4, 5, 9 ] ],
            isBlockDesign := true, v := 9 ),
        rec( blocks := [ [ 1, 5, 8 ], [ 2, 6, 9 ], [ 3, 4, 7 ] ],
            isBlockDesign := true, v := 9 ),
        rec( blocks := [ [ 1, 6, 7 ], [ 2, 4, 8 ], [ 3, 5, 9 ] ],
            isBlockDesign := true, v := 9 ),
        rec( blocks := [ [ 1, 7, 9 ], [ 2, 3, 5 ], [ 4, 6, 8 ] ],
            isBlockDesign := true, v := 9 ),
        rec( blocks := [ [ 1, 8, 9 ], [ 2, 3, 6 ], [ 4, 5, 7 ] ],
            isBlockDesign := true, v := 9 ) ] ) ]
gap> d2[2].resolutions.list;
[  ]
gap> d2[3].resolutions.list;
[  ]
```

Finally, here is a mosaic of projective planes of order 3 from [Krc24].

Example

```
gap> m:=ReadMat("13-4-1.txt")[1];
[ [ 0, 1, 2, 1, 3, 2, 3, 1, 1, 3, 3, 2, 2 ],
  [ 3, 0, 2, 3, 2, 1, 2, 1, 2, 3, 1, 1, 3 ],
  [ 3, 1, 0, 2, 1, 3, 3, 3, 2, 2, 1, 2, 1 ],
  [ 3, 3, 1, 0, 1, 1, 2, 2, 1, 2, 3, 3, 2 ],
  [ 2, 1, 1, 2, 0, 2, 2, 3, 3, 1, 3, 1, 3 ],
  [ 2, 3, 2, 3, 3, 0, 1, 3, 1, 2, 2, 1, 1 ],
  [ 1, 2, 2, 2, 3, 3, 0, 2, 1, 1, 1, 3, 3 ],
  [ 3, 2, 3, 1, 3, 1, 2, 0, 3, 1, 2, 2, 1 ],
  [ 1, 1, 3, 2, 2, 1, 1, 3, 0, 3, 2, 3, 2 ],
  [ 1, 3, 3, 1, 1, 2, 3, 2, 2, 0, 2, 1, 3 ],
  [ 1, 2, 1, 3, 2, 2, 3, 1, 3, 2, 0, 3, 1 ],
  [ 2, 2, 3, 3, 1, 3, 1, 1, 2, 1, 3, 0, 2 ],
  [ 2, 3, 1, 1, 2, 3, 1, 2, 3, 3, 1, 2, 0 ] ]
gap> MosaicParameters(m);
"2-(13,4,1) + 2-(13,4,1) + 2-(13,4,1)"
gap> aut:=MatAut(m);
Group([ (1,3,2)(4,6,5)(7,9,8)(10,12,11)(14,16,15)(17,19,18)(20,22,21)
  (23,25,24)(28,30,29) ])
gap> Size(aut);
3
```

The full automorphism group of this mosaic is of order 3, so it cannot be obtained by tiling groups with $(13,4,1)$ difference sets.

Chapter 2

The PAG Functions

The following functions are available in the PAG package.

2.1 Working With Permutation Groups

2.1.1 CyclicPerm

▷ `CyclicPerm(n)` (function)

Returns the cyclic permutation (1,...,*n*).

2.1.2 ToGroup

▷ `ToGroup(G, f)` (function)

Apply function *f* to each generator of the group *G*.

2.1.3 MovePerm

▷ `MovePerm(p, from, to)` (function)

Moves permutation *p* acting on the set *from* to a permutation acting on the set *to*. The arguments *from* and *to* should be lists of integers of the same size. Alternatively, if instead of *from* and *to* just one integer argument *by* is given, the permutation is moved from `MovedPoints(p)` to `MovedPoints(p)+by`; see `MovedPoints` (**Reference: MovedPoints for a permutation**).

2.1.4 MoveGroup

▷ `MoveGroup(G, from, to)` (function)

Apply `MovePerm` (2.1.3) to each generator of the group *G*.

2.1.5 MultiPerm

▷ `MultiPerm(p, set, m)` (function)

Repeat the action of a permutation *m* times. The new permutation acts on *m* disjoint copies of *set*.

2.1.6 MultiGroup

▷ `MultiGroup(G, set, m)` (function)

Apply `MultiPerm` (2.1.5) to each generator of the group *G*.

2.1.7 RestrictedGroup

▷ `RestrictedGroup(G, set)` (function)

Apply `RestrictedPerm` (**Reference: RestrictedPerm**) to each generator of the group *G*.

2.1.8 PrimitiveGroupsOfDegree

▷ `PrimitiveGroupsOfDegree(v)` (function)

Returns a list of all primitive permutation groups on *v* points.

2.1.9 TransitiveGroupsOfDegree

▷ `TransitiveGroupsOfDegree(v)` (function)

Returns a list of all transitive permutation groups on *v* points.

2.1.10 Homogeneity

▷ `Homogeneity(G)` (function)

Returns the degree of homogeneity of the permutation group *G*, i.e. the largest integer k such that *G* is k-homogeneous. This means that every k-subset of points can be mapped to every other. Kantor [Kan72] classified all groups that are k-homogenous but not k-transitive.

2.1.11 AllSubgroupsConjugation

▷ `AllSubgroupsConjugation(G)` (function)

Returns a list of all subgroups of *G* up to conjugation.

2.1.12 PermRepresentationRight

▷ `PermRepresentationRight(G)` (function)

Returns the regular permutation representation of a group `G` by right multiplication.

2.1.13 PermRepresentationLeft

▷ `PermRepresentationLeft(G)` (function)

Returns the regular permutation representation of a group `G` by left multiplication.

2.1.14 ExtendedPermRepresentation

▷ `ExtendedPermRepresentation(G)` (function)

Returns the extended permutation representation of a group `G` including right multiplication, left multiplication, and group automorphisms.

2.2 Generating Orbits

2.2.1 SubsetOrbitRep

▷ `SubsetOrbitRep(G, v, k[, opt])` (function)

Computes orbit representatives of `k`-subsets of [1..`v`] under the action of the permutation group `G`. The basic algorithm is described in [KVK21]. The algorithm for short orbits is described in [KV16]. The last argument is a record `opt` for options. The possible components of `opt` are:

- *SizeLE*:=*n* If defined, only representatives of orbits of size less or equal to `n` are computed.
- *IntesectionNumbers*:=*lin* If defined, only representatives of good orbits are returned. These are orbits with intersection numbers in the list of integers `lin`.

2.2.2 SubsetOrbitRepShort1

▷ `SubsetOrbitRepShort1(G, v, k, size)` (function)

Computes `G`-orbit representatives of `k`-subsets of [1..`v`] of size less or equal `size`. Here, `size` is an integer smaller than the order of the group `G`. The algorithm is described in [KV16].

2.2.3 SubsetOrbitRepIN

▷ `SubsetOrbitRepIN(G, v, k, lin[, opt])` (function)

Computes orbit representatives of `k`-subsets of [1..`v`] under the action of the permutation group `G` with intersection numbers in the list `lin`. Parts of the search tree with partial subsets intersecting in more than the largest number in `lin` are skipped. Short orbits are computed separately. The algorithm is described in [KVK21]. The last (optional) argument `opt` is a record for options. The possible components are:

- `Verbose:=true/false` Print comments reporting the progress of the calculation.
- `FilteringLevel:=n` Apply filrering of the search tree up to subsets of size `n`. By default, `n=k`.

2.2.4 IsGoodSubsetOrbit

▷ `IsGoodSubsetOrbit(G, rep, lin)` (function)

Check if the subset orbit generated by the permutation group `G` and the representative `rep` is a good orbit with respect to the list of intersection numbers `lin`. This means that the intersection size of any pair of sets from the orbit is an integer in `lin`.

2.2.5 SmallLambdaFilter

▷ `SmallLambdaFilter(G, tsub, ksub, lambda)` (function)

Remove `k`-subset representatives from `ksub` such that the corresponding `G`-orbit covers some of the `t`-subset representatives from `tsub` more than `lambda` times.

2.2.6 OrbitFilter1

▷ `OrbitFilter1(G, obj, action)` (function)

Takes a list of objects `obj` and returns one representative from each orbit of the group `G` acting by `action`. The result is a sublist of `obj`. The algorithm uses the GAP function `Orbit` (**Reference: Orbit**).

2.2.7 OrbitFilter2

▷ `OrbitFilter2(G, obj, action)` (function)

Takes a list of objects `obj` and returns one representative from each orbit of the group `G` acting by `action`. Canonical representatives are returned, so the result is not a sublist of `obj`. The algorithm uses the `CanonicalImage` (**images: CanonicalImage**) function from the package Images.

2.3 Constructing Objects

2.3.1 KramerMesnerSearch

▷ `KramerMesnerSearch(t, v, k, lambda, G[, opt])` (function)

Performs a search for `t`-(`v`,`k`,`lambda`) designs with prescribed automorphism group `G` by the Kramer-Mesner method. A record with options can be supplied. By default, designs are returned in the Design package format (**DESIGN: Design**) and isomorph-rejection is performed by calling

BlockDesignFilter (2.4.2). It can be turned off by setting *opt.NonIsomorphic*:=false. By setting *opt.BaseBlocks*:=true, base blocks are returned instead of designs. This automatically turns off isomorph-rejection. Other available options are:

- *SmallLambda*:=true/false. Perform the "small lambda filter", i.e. remove *k*-orbits covering some of the *t*-orbits more than *lambda* times. By default, this is done if *lambda*<=3.
- *IntersectionNumbers*:=*lin*/false. Search for designs with block intersection nubers in the list of integers *lin* (e.g. quasi-symmetric designs).

2.3.2 KramerMesnerMat

▷ KramerMesnerMat(*G, tsub, ksub[, lambda][, b]*) (function)

Returns the Kramer-Mesner matrix for a permutation group *G*. The rows are labelled by t-subset orbits represented by *tsub*, and the columns by k-subset orbits represented by *ksub*. A column of constants *lambda* is added if the optional argument *lambda* is given. Another row is added if the optional argument *b* is given, repesenting the constraint that sizes of the chosen k-subset orbits must sum up to the number of blocks *b*.

2.3.3 CompatibilityMat

▷ CompatibilityMat(*G, ksub, lin*) (function)

Returns the compatibility matrix of the k-subset representatives *ksub* with respect to the group *G* and list of intersection numbers *lin*. Entries are 1 if intersection sizes of subsets in the corresponding *G*-orbits are all integers in *lin*, and 0 otherwise.

2.3.4 SolveKramerMesner

▷ SolveKramerMesner(*mat[, cm][, opt]*) (function)

Solve a system of linear equations determined by the matrix *mat* over $\{0,1\}$. By default, A.Wassermann's LLL solver solvediophant [Was98] is used. If the second argument is a compatibility matrix *cm*, the backtracking program solvecm from the papers [KNP11] and [KV16] is used. The solver can also be chosen explicitly in the record *opt*. Possible components are:

- *Solver*:="solvediophant" If defined, solvediophant is used.
- *Solver*:="solvecm" If defined, solvecm is used.
- *Solver*:="libexact" If defined, libexact is used. This is P. Kaski and O. Pottonen's implementation of the Dancing Links algorithm, see [KP08]. For this solver the coefficients of *mat* must be in $\{0,1\}$!

2.3.5 BaseBlocks

▷ BaseBlocks(*ksub, sol*) (function)

Returns base blocks of design(s) from solution(s) *sol* by picking them from k-subset orbit representatives *ksub*.

2.3.6 ExpandMatRHS

▷ `ExpandMatRHS(mat, lambda)` (function)

Add a column of `lambda`'s to the right of the matrix `mat`.

2.3.7 CameronSeidelSet

▷ `CameronSeidelSet(m)` (function)

Returns a list of $2^{m/2}$ symplectic $m \times m$ matrices over $GF(2)$ such that the difference of any two of them is a regular matrix. Here `m` is an even integer. The construction is described on page 6 of the paper [CS73].

2.3.8 OrthogonalNormalBasis

▷ `OrthogonalNormalBasis(k)` (function)

Attempts to find a basis for the field $GF(2^k)$ over $GF(2)$ that is orthogonal with respect to the trace inner product $Tr(xy)$. This should work for odd integers `k`, but might fail for even integers.

2.3.9 KerdockSet

▷ `KerdockSet(m)` (function)

Returns a Kerdock set of 2^{m-1} symplectic $m \times m$ matrices over $GF(2)$ such that the difference of any two of them is a regular matrix. Here `m` is an even integer. The construction is based on Example 2.4 in the paper [Kan95].

2.3.10 SingerDifferenceSets

▷ `SingerDifferenceSets(q, n)` (function)

Returns the classical Singer difference sets in the cyclic group of order $v = (q^n - 1)/(q - 1)$, e.g. `Group(CyclicPerm(v))`. The difference sets are subsets of `[1..v]` to make them compatible with the DifSets package. For each D returned, $D - 1$ is a difference set in the integers modulo v (a subset of `[0..v-1]`).

2.3.11 NormalizedSingerDifferenceSets

▷ `NormalizedSingerDifferenceSets(q, n)` (function)

Returns the classical Singer difference sets in the cyclic group of order $v = (q^n - 1)/(q - 1)$ that are normalized. If D is a difference set, this means that the elements of $D - 1$ sum up to 0 modulo v.

2.3.12 RightDevelopment

▷ `RightDevelopment(G, ds)` (function)

Returns a block design that is the development of the difference set `ds` by right multiplication in the group `G`. If `ds` is a tiling of the group `G` or a list of disjoint difference sets, a mosaic of symmetric designs is returned.

2.3.13 LeftDevelopment

▷ `LeftDevelopment(G, ds)` (function)

Returns a block design that is the development of the difference set `ds` by left multiplication in the group `G`. If `ds` is a tiling of the group `G` or a list of disjoint difference sets, a mosaic of symmetric designs is returned.

2.3.14 EquivalentDifferenceSets

▷ `EquivalentDifferenceSets(g, D)` (function)

Given a difference set or list of difference sets `D` in a group `g`, returns the set of all difference sets equivalent to the ones in `D`.

2.4 Inspecting Objects and Other Functions

2.4.1 BlockDesignAut

▷ `BlockDesignAut(d[, opt])` (function)

Computes the full automorphism group of a block design `d`. Uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. This is an alternative for the `AutGroupBlockDesign` function from the Design package (**DESIGN: Automorphism groups and isomorphism testing for block designs**). The optional argument `opt` is a record for options. Possible components of `opt` are:

- `Traces:=true/false` Use `Traces`. This is the default.
- `SparseNauty:=true/false` Use `nauty` for sparse graphs.
- `DenseNauty:=true/false` Use `nauty` for dense graphs. This is usually the slowest program, but it allows vertex invariants. Vertex invariants are ignored by the other programs.
- `BlockAction:=true/false` If set to `true`, the action of the automorphisms on blocks is also given. In this case automorphisms are permutations of degree $v+b$. By default, only the action on points is given, i.e. automorphisms are permutations of degree v.
- `Dual:=true/false` If set to `true`, dual automorphisms (correlations) are also included. They will appear only for self-dual symmetric designs (with the same number of points and blocks). The default is `false`.

- *PointClasses*:=*s* Color the points into classes of size *s* that cannot be mapped onto each other. By default all points are in the same class.
- *VertexInvariant*:=*n* Use vertex invariant number *n*. The numbering is the same as in `dreadnaut`, e.g. *n*=1: `twopaths`, *n*=2: `adjtriang`, etc. The default is `twopaths`. Vertex invariants only work with dense `nauty`. They are ignored by sparse `nauty` and `Traces`.
- *Mininvarlevel*:=*n* Set `mininvarlevel` to *n*. The default is *n*=0.
- *Maxinvarlevel*:=*n* Set `maxinvarlevel` to *n*. The default is *n*=2.
- *Invararg*:=*n* Set `invararg` to *n*. The default is *n*=0.

2.4.2 BlockDesignFilter

▷ `BlockDesignFilter(dl[, opt])` (function)

Eliminates isomorphic copies from a list of block designs *dl*. Uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. This is an alternative for the `BlockDesignIsomorphismClassRepresentatives` function from the Design package (**DESIGN: Automorphism groups and isomorphism testing for block designs**). The optional argument *opt* is a record for options. Possible components of *opt* are:

- *Traces*:=`true/false` Use `Traces`. This is the default.
- *SparseNauty*:=`true/false` Use `nauty` for sparse graphs.
- *PointClasses*:=*s* Color the points into classes of size *s* that cannot be mapped onto each other. By default all points are in the same class.
- *Positions*:=`true/false` Return positions of nonisomorphic designs instead of the designs themselves.

2.4.3 Cliquer

▷ `Cliquer(g[, opt])` (function)

Searches for cliques in the graph *g*. Uses `Cliquer` by S.Niskanen and P.Ostergard [NO03]. The graph can either be given in GRAPE format, or as a list `[v,elist]` where `v` is the number of vertices and `elist` is a list of edges (2-element subsets of `[1..v]`). The optional argument *opt* is a record for options. Possible components are:

- *Silent*:=`true/false` Work silently, or report progress. The default is taken from `PAGGlobalOptions`.
- *FindAll*:=`true/false` Find all cliques, or search for a single clique. The default is `true`.
- *CliqueSize*:=`n` or `[min,max]` Search for cliques of size `n`, or size from `min` to `max`. By default, searches for cliques of maximum size.
- *Order*:=`n` Reorder vertices by ordering function number `n`. Available functions are `n`= 1 `ident`, `n`= 2 `reverse`, `n`= 3 `degree`, `n`= 4 `random`, and `n`= 5 `greedy` (default).

2.4.4 DisjointCliques

▷ `DisjointCliques(L[, opt])` (function)

Given a list `L` of k-sets of integers, searches for cliques of mutually disjoint k-sets from the list. The sets must be of equal size k. Uses `Cliquer` by S.Niskanen and P.Ostergard [NO03]. The optional argument `opt` is a record for options with possible components:

- *Silent*:=`true/false` Work silently, or report progress. The default is taken from `PAGGlobalOptions`.
- *FindAll*:=`true/false` Find all cliques, or search for a single clique. The default is `true`.
- *CliqueSize*:=`n` or `[min,max]` Search for cliques of size `n`, or size from `min` to `max`. By default, searches for cliques of maximum size.
- *Order*:=`n` Reorder vertices by ordering function number `n`. Available functions are `n`= 1 `ident`, `n`= 2 `reverse`, `n`= 3 `degree`, `n`= 4 `random`, and `n`= 5 `greedy` (default).

2.4.5 IntersectionNumbers

▷ `IntersectionNumbers(d[, opt])` (function)

Returns the list of intersection numbers of the block design `d`. The optional argument `opt` is a record for options. Possible components of `opt` are:

- *Frequencies*:=`true/false` If set to `true`, frequencies of the intersection numbers are also returned.

2.4.6 BlockScheme

▷ `BlockScheme(d[, opt])` (function)

Returns the block intersection association scheme of a block design `d`, or `fail` if `d` is not block schematic. The optional argument `opt` is a record for options. If it contains the component *Matrix*:=`true`, the block intersection matrix is returned instead. Uses the package AssociationSchemes. If the package is not available, `BlockScheme` always returns the block intersection matrix and does not check if it defines an association scheme.

2.4.7 PointPairScheme

▷ `PointPairScheme(d[, opt])` (function)

Returns the point pair association scheme of a block design `d`, or `fail` if `d` is not point pair schematic. The optional argument `opt` is a record for options. If it contains the component *Matrix*:=`true`, the point pair inclusion matrix is returned instead. The point pair scheme was defined by Cameron [Cam75] for Steiner 3-designs. This command is a slight generalisation that works for arbitrary designs. Uses the package AssociationSchemes. If the package is not available, `PointPairScheme` always returns the point pair inclusion matrix and does not check if it defines an association scheme.

2.4.8 TDesignB

▷ `TDesignB(t, v, k, lambda)` (function)

The number of blocks of a `t`-(`v`,`k`,`lambda`) design.

2.4.9 IversonBracket

▷ `IversonBracket(P)` (function)

Returns 1 if `P` is true, and 0 otherwise.

2.4.10 SymmetricDifference

▷ `SymmetricDifference(X, Y)` (function)

Returns the symmetric difference of two sets `X` and `Y`.

2.4.11 AddWeights

▷ `AddWeights(wd)` (function)

Makes a weight distribudion `wd` more readable by adding the weights and skipping zero components. The argument `wd` is the weight distribution of a code returned by the `WeightDistribution` command from the GUAVA package.

2.4.12 AdjacencyMat

▷ `AdjacencyMat(g)` (function)

Returns the adjacency matrix of the graph `g` in GRAPE format.

2.5 Latin Squares

2.5.1 ReadMOLS

▷ `ReadMOLS(filename)` (function)

Read a list of MOLS sets from a file. The file starts with the number of rows m, columns n, and the size of the sets s, followed by the matrix entries. Integers in the file are separated by whitespaces.

2.5.2 WriteMOLS

▷ `WriteMOLS(filename, list)` (function)

Write a list of MOLS sets to a file. The number of rows m, columns n, and the size of the sets s is written first, followed by the matrix entries. Integers are separated by whitespaces.

2.5.3 FieldToMOLS

▷ `FieldToMOLS(F)` (function)

Construct a complete set of MOLS from the finite field `F`. A similar function is `MOLS` (**GUAVA: MOLS**) from the package Guava.

2.5.4 MOLSToOrthogonalArray

▷ `MOLSToOrthogonalArray(ls)` (function)

Transforms the set of MOLS `ls` to an equivalent orthogonal array.

2.5.5 OrthogonalArrayToMOLS

▷ `OrthogonalArrayToMOLS(oa)` (function)

Transforms the orthogonal array `oa` to an equivalent set of MOLS.

2.5.6 MOLSToTransversalDesign

▷ `MOLSToTransversalDesign(ls)` (function)

Transforms the set of MOLS `ls` to an equivalent transversal design.

2.5.7 TransversalDesignToMOLS

▷ `TransversalDesignToMOLS(td)` (function)

Transforms the transversal design `td` to an equivalent set of MOLS.

2.5.8 MOLSAut

▷ `MOLSAut(ls[, opt])` (function)

Computes the full auto(para)topy group of a set of MOLS `ls`. Uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. The optional argument `opt` is a record for options. Possible components are:

- *Isotopy*:=`true/false` Compute the full autotopy group of `ls`. This is the default.
- *Paratopy*:=`true/false` Compute the full autoparatopy group of `ls`.

Any other components are forwarded to the `BlockDesignAut` (2.4.1) function; see its documentation.

2.5.9 MOLSFilter

▷ `MOLSFilter(ls[, opt])` (function)

Eliminates isotopic/paratopic copies from a list of MOLS sets `ls`. Uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. The optional argument `opt` is a record for options. Possible components are:

- `Paratopy:=true/false` Eliminate paratopic MOLS sets. This is the default.
- `Isotopy:=true/false` Eliminate isotopic MOLS sets.

Any other components are forwarded to the `BlockDesignFilter` (2.4.2) function; see its documentation.

2.5.10 IsAutotopyGroup

▷ `IsAutotopyGroup(n, s, G)` (function)

Check if `G` is an autotopy group for transversal designs with $s+2$ point classes of order `n`.

2.5.11 MOLSSubsetOrbitRep

▷ `MOLSSubsetOrbitRep(n, s, G)` (function)

Computes representatives of pairs and transversals of the $s+2$ point classes for the construction of MOLS of order `n` with prescribed autotopy group `G`. A list containing pair representatives in the first component and transversal representatives in the second component is returned.

2.5.12 KramerMesnerMOLS

▷ `KramerMesnerMOLS(n, s, G[, opt])` (function)

If the function `IsAutotopyGroup` (2.5.10)(`G`) returns `true` for the group `G`, call `KramerMesnerMOLSAutotopy` (2.5.13); otherwise call `KramerMesnerMOLSAutoparatopy` (2.5.14).

2.5.13 KramerMesnerMOLSAutotopy

▷ `KramerMesnerMOLSAutotopy(n, s, G[, opt])` (function)

Search for MOLS sets of order `n` and size `s` with prescribed autotopy group `G`. By default, A.Wassermann's LLL solver `solvediophant` is used for $s=1$, and the backtracking solver `solvecm` is used for $s>1$. This can be changed by setting options in the record `opt`. Available options are:

- `Solver:="solvediophant"` Use `solvediophant`.
- `Solver:="solvecm"` Use `solvecm`.
- `Paratopy:=true/false` Eliminate paratopic solutions. This is the default.
- `Isotopy:=true/false` Eliminate isotopic solutions. All solutions are returned if either option is set to `false`.

2.5.14 KramerMesnerMOLSAutoparatopy

▷ `KramerMesnerMOLSAutoparatopy(n, s, G[, opt])` (function)

Search for MOLS sets of order *n* and size *s* with prescribed autoparatopy group *G*. By default, A.Wassermann's LLL solver `solvediophant` is used for $s = 1$, and the backtracking solver `solvecm` is used for $s > 1$. This can be changed by setting options in the record *opt*. Available options are:

- *Solver*:=`"solvediophant"` Use `solvediophant`.
- *Solver*:=`"solvecm"` Use `solvecm`.
- *Paratopy*:=`true/false` Eliminate paratopic solutions. This is the default.
- *Isotopy*:=`true/false` Eliminate isotopic solutions. All solutions are returned if either option is set to `false`.

2.6 Cubes of Symmetric Designs

2.6.1 DifferenceCube

▷ `DifferenceCube(G, ds, n)` (function)

Returns the *n*-dimenional difference cube constructed from a difference set *ds* in the group *G*.

2.6.2 GroupCube

▷ `GroupCube(G, dds, n)` (function)

Returns the *n*-dimenional group cube constructed from a symmetric design *dds* such that the blocks are difference sets in the group *G*.

2.6.3 CubeSlice

▷ `CubeSlice(C, x, y, fixed)` (function)

Returns a 2-dimensional slice of the incidence cube *C* obtained by varying coordinates in positions *x* and *y*, and taking fixed values for the remaining coordinates given in a list *fixed*.

2.6.4 CubeSlices

▷ `CubeSlices(C[, x, y][, fixed])` (function)

Returns 2-dimensional slices of the incidence cube *C*. Optional arguments are the varying coordinates *x* and *y*, and values of the fixed coordinates in a list *fixed*. If optional arguments are not given, all possibilities are supplied. For an n-dimensional cube *C* of order v, the following calls will return:

- CubeSlices(*C*, *x*, *y*) ... v^{n-2} slices obtained by varying values of the fixed coordinates.
- CubeSlices(*C*, *fixed*) ... $\binom{n}{2}$ slices obtained by varying the non-fixed coordinates $x < y$.

- CubeSlices(`C`) ... $\binom{n}{2} \cdot v^{n-2}$ slices obtained by varying both the non-fixed coordinates $x < y$ and values of the fixed coordinates.

2.6.5 CubeLayer

▷ `CubeLayer(C, x, fixed)` (function)

Returns an $(n-1)$-dimensional layer of the n-dimensional cube `C` obtained by setting coordinate `x` to the value `fixed` and varying the remaining coordinates.

2.6.6 CubeLayers

▷ `CubeLayers(C, x)` (function)

Returns the $(n-1)$-dimensional layers of the n-dimensional cube `C` obtained by fixing coordinate `x`.

2.6.7 CubeToOrthogonalArray

▷ `CubeToOrthogonalArray(C)` (function)

Transforms the incidence cube `C` to an equivalent orthogonal array.

2.6.8 OrthogonalArrayToCube

▷ `OrthogonalArrayToCube(oa)` (function)

Transforms the orthogonal array `oa` to an equivalent incidence cube.

2.6.9 OrthogonalArrayToTransversalDesign

▷ `OrthogonalArrayToTransversalDesign(oa)` (function)

Transforms the orthogonal array `oa` to an equivalent transversal design.

2.6.10 CubeToTransversalDesign

▷ `CubeToTransversalDesign(C)` (function)

Transforms the incidence cube `C` to an equivalent transversal design.

2.6.11 TransversalDesignToCube

▷ `TransversalDesignToCube(td)` (function)

Transforms the transversal design `td` to an equivalent incidence cube.

2.6.12 LatinSquareToCube

▷ `LatinSquareToCube(L)` (function)

Transforms the Latin square `L` to an equivalent incidence cube.

2.6.13 CubeTest

▷ `CubeTest(C)` (function)

Test whether an incidence cube `C` is a cube of symmetric designs. The result should be `[[v,k,lambda]]`. Anything else means that `C` is not a (v,k,λ) cube.

2.6.14 SliceInvariant

▷ `SliceInvariant(C)` (function)

Computes a paratopy invariant of the cube `C` based on automorphism group sizes of parallel slices. Cubes equivalent under paratopy have the same invariant.

2.6.15 CubeAut

▷ `CubeAut(C[, opt])` (function)

Computes the full auto(para)topy group of an incidence cube `C`. Uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. The optional argument `opt` is a record for options. Possible components are:

- `Isotopy:=true/false` Compute the full autotopy group of `C`. This is the default.
- `Paratopy:=true/false` Compute the full autoparatopy group of `C`.

Any other components are forwarded to the `BlockDesignAut` (2.4.1) function; see its documentation.

2.6.16 CubeFilter

▷ `CubeFilter(cl[, opt])` (function)

Eliminates equivalent copies from a list of incidence cubes `cl`. Uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. The optional argument `opt` is a record for options. Possible components are:

- `Paratopy:=true/false` Eliminate paratopic cubes. This is the default.
- `Isotopy:=true/false` Eliminate isotopic cubes.

Any other components are forwarded to the `BlockDesignFilter` (2.4.2) function; see its documentation.

2.6.17 SDPSeriesGroup

▷ `SDPSeriesGroup(m)` (function)

Returns a group for the designs of `SDPSeriesDesign` (2.6.18). This is the elementary Abelian group of order 4^m.

2.6.18 SDPSeriesDesign

▷ `SDPSeriesDesign(m, i)` (function)

Returns a symmetric block design with parameters $(4^m, 2^{m-1}(2^m-1), 2^{m-1}(2^{m-1}-1))$. The argument `i` must be 1, 2, or 3. If $i = 1$, the design is the symplectic design of Kantor [Kan75]. This design has the symmetric difference property (SDP). If $i = 2$ or $i = 3$, two other non-isomorphic designs with the same parameters are returned. They are not SDP designs, but have the property that all their blocks are difference sets in the group returned by `SDPSeriesGroup` (2.6.17). Developments of these blocks are isomorphic to the design for $i = 1$, so the two other designs are not developments of their blocks.

2.7 Projection Cubes of Symmetric Designs

2.7.1 CubeProjection

▷ `CubeProjection(C, p)` (function)

Returns the projection of the n-dimensional cube `C` on a pair of coordinates `p`.

2.7.2 CubeProjections

▷ `CubeProjections(C)` (function)

Returns the projections of the n-dimensional cube `C` on all pairs of coordinates.

2.7.3 CubeProjectionTest

▷ `CubeProjectionTest(C)` (function)

Test whether an incidence cube `C` is a projection cube of symmetric designs. The result should be `[[v,k,lambda]]`. Anything else means that `C` is not a (v, k, λ) projection cube. The function `OrthogonalArrayProjectionTest` (2.7.6) is usually much faster.

2.7.4 OrthogonalArrayProjection

▷ `OrthogonalArrayProjection(oa, t)` (function)

Returns the projection of the orthogonal array `oa` on a tuple of coordinates `t`.

2.7.5 OrthogonalArrayProjections

▷ `OrthogonalArrayProjections(oa[, k])` (function)

Returns the projections of the orthogonal array `oa` on all `k`-tuples of coordinates. If the second argument is not given, `k`$=2$ is assumed.

2.7.6 OrthogonalArrayProjectionTest

▷ `OrthogonalArrayProjectionTest(oa)` (function)

Test whether an orthogonal array `oa` corresponds to a projection cube of symmetric (v,k,λ) designs. The result should be `[[v,k,lambda]]`. Anything else means that `oa` does not correspond to a projection cube.

2.7.7 DifferenceSetToOrthogonalArray

▷ `DifferenceSetToOrthogonalArray([G, ]ds)` (function)

Transforms a (higher-dimensional) difference set to an orthogonal array. The argument `G` is a group and `ds` is a difference set in the DifSets package format, with positive integers as elements. If the first argument is not given, `ds` contains finite field elements and the operation is addition. This is used for Paley difference sets and twin prime power difference sets.

2.7.8 PaleyDifferenceSet

▷ `PaleyDifferenceSet(q)` (function)

Returns the `q`-dimensional Paley difference set in $GF(q)$. This is a $(q,(q-1)/2,(q-3)/4)$ difference set in the additive group of $GF(q)$. See [KR24] for more details.

2.7.9 PowerDifferenceSet

▷ `PowerDifferenceSet(q, m)` (function)

Returns the `q`-dimensional difference set constructed from the `m`-th powers in $GF(q)$. Paley difference sets are power difference sets for $m=2$. See [KR24] for more details.

2.7.10 TwinPrimePowerDifferenceSet

▷ `TwinPrimePowerDifferenceSet(q)` (function)

Returns the `q`-dimensional twin prime power difference set. For $n=(q+1)^2/4$, this is a $(4n-1,2n-1,n-1)$ difference set in the direct product $GF(q)\times GF(q+2)$. Both `q` and `q+2` must be powers of primes. See [KR24] for more details.

2.7.11 OrthogonalArrayAut

▷ `OrthogonalArrayAut(oa[, opt])` (function)

Computes the full auto(para)topy group of an orthogonal array *oa*. Uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. The optional argument *opt* is a record for options. Possible components are:

- *Isotopy*:=`true/false` Compute the full autotopy group of *oa*. This is the default.
- *Paratopy*:=`true/false` Compute the full autoparatopy group of *oa*.

Any other components are forwarded to the `BlockDesignAut` (2.4.1) function; see its documentation.

2.7.12 OrthogonalArrayFilter

▷ `OrthogonalArrayFilter(oal[, opt])` (function)

Eliminates equivalent copies from a list of orthogonal arrays *oal*. Uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. The optional argument *opt* is a record for options. Possible components are:

- *Paratopy*:=`true/false` Eliminate paratopic orthogonal arrays. This is the default.
- *Isotopy*:=`true/false` Eliminate isotopic orthogonal arrays.

Any other components are forwarded to the `BlockDesignFilter` (2.4.2) function; see its documentation.

2.8 Hadamard Matrices

2.8.1 IsHadamardMat

▷ `IsHadamardMat(H)` (function)

Returns `true` if *H* is an n-dimensional Hadamard matrix and `false` otherwise.

2.8.2 IsProperHadamardMat

▷ `IsProperHadamardMat(H)` (function)

Returns `true` if *H* is a proper n-dimensional Hadamard matrix and `false` otherwise.

2.8.3 Paley1Mat

▷ `Paley1Mat(q)` (function)

Returns a Paley type I Hadamard matrix of order $q+1$ constructed from the squares in $GF(q)$. The argument should be a prime power $q \equiv 3 \pmod 4$.

2.8.4 Paley2Mat

▷ `Paley2Mat(q)` (function)

Returns a Paley type II Hadamard matrix of order $2(q+1)$ constructed from the squares in $GF(q)$. The argument should be a prime power $q \equiv 1 \pmod 4$.

2.8.5 Paley3DMat

▷ `Paley3DMat(v)` (function)

Returns a three-dimensional Hadamard matrix of order v obtained by the Paley-like construction introduced in [KPT23]. The argument should be an even number v such that $v-1$ is a prime power.

2.8.6 SDPSeriesHadamardMat

▷ `SDPSeriesHadamardMat(m, i)` (function)

Returns a Hadamard matrix of order 4^m for the SDP series of designs. The argument i must be 1, 2, or 3. See documentation for the `SDPSeriesDesign` (2.6.18) function.

2.8.7 AllOnesMat

▷ `AllOnesMat(v[, n])` (function)

Returns the n-dimensional matrix of order v with all entries 1. By default, $n = 2$.

2.8.8 ProductConstructionMat

▷ `ProductConstructionMat(H, n)` (function)

Given a 2-dimensional Hadamard matrix H, returns the n-dimensional proper Hadamard matrix obtained by the product construction of Yang [Yan86].

2.8.9 DigitConstructionMat

▷ `DigitConstructionMat(H, s)` (function)

Given a 2-dimensional Hadamard matrix H of order $(2t)^s$, returns the $2s$-dimensional Hadamard matrix of order $2t$ obtained by Theorem 6.1.4 of [YNX10].

2.8.10 CyclicDimensionIncrease

▷ `CyclicDimensionIncrease(H)` (function)

Given an n-dimensional Hadamard matrix H, returns the $(n+1)$-dimensional Hadamard matrix obtained by Theorem 6.1.5 of [YNX10]. The construction also works for cyclic cubes of symmetric designs.

2.8.11 HadamardMatAut

▷ `HadamardMatAut(H[, opt])` (function)

Computes the full automorphism group of a Hadamard matrix *H*. Represents the matrix by a colored graph (see [McK79]) and uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. The optional argument *opt* is a record for options. Possible components of *opt* are:

- *Dual*:=`true/false` If set to `true`, dual automorphisms (transpositions) are also allowed. The default is `false`.

2.8.12 HadamardMatFilter

▷ `HadamardMatFilter(hl[, opt])` (function)

Eliminates equivalent copies from a list of Hadamard matrices *hl*. Represents the matrices by colored graphs (see [McK79]) and uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. The optional argument *opt* is a record for options. Possible components of *opt* are:

- *Dual*:=`true/false` If set to `true`, dual equivalence is allowed (i.e. the matrices can be transposed). The default is `false`.
- *Positions*:=`true/false` Return positions of inequivalent Hadamard matrices instead of the matrices themselves.

2.8.13 HadamardToIncidence

▷ `HadamardToIncidence(M)` (function)

Transforms the Hadamard matrix *M* to an incidence matrix by replacing all -1 entries by 0.

2.8.14 IncidenceToHadamard

▷ `IncidenceToHadamard(M)` (function)

Transforms the incidence matrix *M* to a $(1,-1)$-matrix by replacing all 0 entries by -1.

2.9 Mosaics of Combinatorial Designs

2.9.1 MosaicParameters

▷ `MosaicParameters(M)` (function)

Returns a string with the parameters of the mosaic of combinatorial designs *M*. See [GGP18] for the definition. Entries 0 in the matrix *M* are considered empty, and other integers are considered as incidences of distinct designs.

2.9.2 BlocksToIncidenceMat

▷ `BlocksToIncidenceMat(d)` (function)

Transforms a list of blocks `d` to an incidence matrix. Points correspond to rows, and blocks to columns.

2.9.3 IncidenceMatToBlocks

▷ `IncidenceMatToBlocks(M)` (function)

Transforms an incidence matrix `M` to a list of blocks. Rows correspond to points, and columns to blocks.

2.9.4 MosaicToBlockDesigns

▷ `MosaicToBlockDesigns(M)` (function)

Transforms a mosaic of combinatorial designs `M` with c colors to a list of c block designs in the Design package format.

2.9.5 ReadMat

▷ `ReadMat(filename)` (function)

Reads a list of $m \times n$ integer matrices from a file. The file starts with the number of rows m and columns n followed by the matrix entries. Integers in the file are separated by whitespaces.

2.9.6 WriteMat

▷ `WriteMat(filename, list)` (function)

Writes a list of $m \times n$ integer matrices to a file. The number of rows m and columns n is written first, followed by the matrix entries. Integers are separated by whitespaces.

2.9.7 AffineMosaic

▷ `AffineMosaic(k, n, q)` (function)

Returns a mosaic of designs with blocks being `k`-dimensional subspaces of the affine space $AG(n,q)$. Uses the FinInG package. If the package is not available, the function is not loaded.

2.9.8 DifferenceMosaic

▷ `DifferenceMosaic(G, dds)` (function)

Returns the mosaic of symmetric designs obtained from a list of disjoint difference sets `dds` in the group `G`.

2.9.9 PowersMosaic

▷ `PowersMosaic(q, n)` (function)

Returns the mosaic of symmetric designs constructed from n-th powers in the field $GF(q)$.

2.9.10 MatAut

▷ `MatAut(M)` (function)

Computes the full autotopy group of a matrix M. It is assumed that the entries of M are consecutive integers. Permutations of rows, columns and symbols are allowed. Represents the matrix by a colored graph and uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14].

2.9.11 MatFilter

▷ `MatFilter(ml[, opt])` (function)

Eliminates equivalent copies from a list of matrices `ml`. It is assumed that all of the matrices have the same set of consecutive integers as entries. Two matrices are equivalent (isotopic) if one can be transformed into the other by permutating rows, columns and symbols. Represents the matrices by colored graphs and uses `nauty/Traces 2.8` by B.D.McKay and A.Piperno [MP14]. The optional argument `opt` is a record for options. Possible components of `opt` are:

- *Positions*:=`true/false` Return positions of inequivalent matrices instead of the matrices themselves.

2.10 Global Options

2.10.1 PAGGlobalOptions

▷ `PAGGlobalOptions` (global variable)

A record with global options for the PAG package. Components are:

- *Silent*:=`true/false` If set to `true`, functions such as SolveKramerMesner will not print comments reporting the progress of the calculation.

- *TempDir*:=`directory object` Temporary directory used to communicate with external programs.

References

[Cam75] P. J. Cameron. Two remarks on Steiner systems. *Geometriae Dedicata*, 4:403–418, 1975. 35

[CD07] C. J. Colbourn and J. H. Dinitz, editors. *Handbook of combinatorial designs. Second edition*. Chapman & Hall/CRC, 2007. 11

[CKZ15] A. Custic, V. Krcadinac, and Y. Zhou. Tiling groups with difference sets. *Electron. J. Combin.*, 22(2):P2.56, 2015. https://doi.org/10.37236/5157. 24

[CS73] P. J. Cameron and J. J. Seidel. Quadratic forms over $GF(2)$. *Indag. Math.*, 35:1–8, 1973. 32

[Fal12] R. M. Falcon. Cycle structures of autotopisms of the latin squares of order up to 11. *Ars Combin.*, 103:239–256, 2012. 10

[GGP18] O. W. Gnilke, M. Greferath, and M. O. Pavcevic. Mosaics of combinatorial designs. *Des. Codes Cryptogr.*, 86(1):85–95, 2018. 23, 25, 46

[Kan72] W. M. Kantor. k-homogenous groups. *Math. Z.*, 124:261–265, 1972. 28

[Kan75] W. M. Kantor. Symplectic groups, symmetric designs, and line ovals. *J. Algebra*, 33:43–58, 1975. 42

[Kan95] W. M. Kantor. *Codes, quadratic forms and finite geometries. In: Different aspects of coding theory (Proc. Sympos. Appl. Math., San Francisco, 1995)*. American Mathematical Society, 1995. 32

[KD15] A. D. Keedwell and J. Denes. *Latin squares and their applications. Second edition*. Elsevier/North-Holland, 2015. 10

[KM76] E. S. Kramer and D. M. Mesner. t-designs on hypergraphs. *Discrete Math.*, 15(3):263–296, 1976. 6

[KNP11] V. Krcadinac, A. Nakic, and M. O. Pavcevic. The Kramer-Mesner method with tactical decompositions: some new unitals on 65 points. *J. Combin. Des.*, 19(4):290–303, 2011. https://doi.org/10.1002/jcd.20277. 31

[KP08] P. Kaski and O. Pottonen. *libexact User's Guide, Version 1.0, HIIT Technical Reports 2008-1*. Helsinki Institute for Information Technology HIIT, 2008. 31

[KPT23] V. Krcadinac, M. O. Pavcevic, and K. Tabak. Three-dimensional Hadamard matrices of Paley type. *Finite Fields Appl.*, 92:102306, 2023. https://doi.org/10.1016/j.ffa.2023.102306. 45

[KPT24] V. Krcadinac, M. O. Pavcevic, and K. Tabak. Cubes of symmetric designs. *Ars Math. Contemp.*, 2024. https://doi.org/10.26493/1855-3974.3222.e53. 12, 14, 15

[KR24] V. Krcadinac and L. Relic. Projection cubes of symmetric designs. *Preprint*, 2024. https://arxiv.org/abs/2411.06936. 19, 22, 43

[Krc18] V. Krcadinac. Some new designs with prescribed automorphism groups. *J. Combin. Des.*, 26(4):193–200, 2018. https://doi.org/10.1002/jcd.21587. 4

[Krc24] V. Krcadinac. Small examples of mosaics of combinatorial designs. *Examples and Counterexamples*, 6:6100163, 2024. https://doi.org/10.1016/j.exco.2024.100163. 24, 26

[KV16] V. Krcadinac and R. Vlahovic. New quasi-symmetric designs by the Kramer-Mesner method. *Discrete Math.*, 339(12):2884–2890, 2016. https://doi.org/10.1016/j.disc.2016.05.030. 9, 29, 31

[KVK21] V. Krcadinac and R. Vlahovic Kruc. Quasi-symmetric designs on 56 points. *Adv. Math. Commun.*, 15(4):633–646, 2021. https://doi.org/10.3934/amc.2020086. 29, 30

[McK79] B. McKay. Hadamard equivalence via graph isomorphism. *Discrete Math.*, 27:213–214, 1979. 46

[MP14] B. McKay and A. Piperno. Practical graph isomorphism, II. *J. Symbolic Comput.*, 60:94–112, 2014. 33, 34, 37, 38, 41, 44, 46, 48

[Nak21] A. Nakic. The first example of a simple 2-(81,6,2) design. *Examples and Counterexamples*, 1:100005, 2021. https://doi.org/10.1016/j.exco.2021.100005. 8, 9

[NO03] S. Niskanen and P. Ostergard. *Cliquer User's Guide, Version 1.0, Tech. Rep. T48*. Helsinki University of Technology, 2003. 34, 35

[Sch93] B. Schmalz. The t-designs with prescribed automorphism group, new simple 6-designs. *J. Combin. Des.*, 1(2):125–170, 1993. 6, 7, 8

[SVW12] D. S. Stones, P. Vojtechovsky, and I. M. Wanless. Cycle structure of autotopisms of quasigroups and Latin squares. *J. Combin. Des.*, 20(5):227–263, 2012. 10

[Was98] A. Wassermann. Finding simple t-designs with enumeration techniques. *J. Combin. Des.*, 6(2):79–90, 1998. 7, 31

[Yan86] Y. X. Yang. Proofs of some conjectures about higher-dimensional Hadamard matrices (Chinese). *Kexue Tongbao*, 31(2):85–88, 1986. 45

[YNX10] Y. X. Yang, X. X. Niu, and C. Q. Xu. *Theory and applications of higher-dimensional Hadamard matrices. Second edition*. CRC Press, 2010. 45

Index