

PAG

Prescribed Automorphism Groups

0.2.0

27 March 2023

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

© 2023 by Vedran Krcadinac

The PAG package is free software; you can redistribute it and/or modify it under the terms of the [GNU General Public License](#) as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

Acknowledgements

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

Contents

1	The PAG Package	4
1.1	Getting Started	4
1.2	Installation	5
1.3	Examples: Designs	6
1.4	Examples: Latin Squares	10
1.5	Examples: Cubes of Symmetric Designs	10
2	The PAG Functions	14
2.1	Working With Permutation Groups	14
2.2	Generating Orbits	16
2.3	Constructing Objects	17
2.4	Inspecting Objects and Other Functions	19
2.5	Latin Squares	20
2.6	Cubes of Symmetric Designs	23
2.7	Hadamard Matrices	25
2.8	Global Options	25
	References	27
	Index	28

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 present a small example from the paper [Krc18]. In Theorem 8.1, simple 5-(16,7,10) designs with the following automorphism group were 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));
```

They 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 suppressed by setting global options.

Example

```
gap> PAGGlobalOptions.Silent:=true;
true
gap> KramerMesnerSearch(5,16,7,10,g);
[ [ [ 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 ] ] ]
```

The output is a list of base blocks for two designs. There are options to get them in the **Design** package format (**DESIGN: Design**). Then we can also check that they are really 5-designs.

Example

```
gap> d:=KramerMesnerSearch(5,16,7,10,g,rec(Design:=true));
gap> List(d,AllTDesignLambdas);
[ [ 2080, 910, 364, 130, 40, 10 ], [ 2080, 910, 364, 130, 40, 10 ] ]
```

The two designs are in fact isomorphic.

Example

```
gap> d:=KramerMesnerSearch(5,16,7,10,g,rec(NonIsomorphic:=true));
gap> Size(d);
1
```

The option **NonIsomorphic** applies the function **BlockDesignIsomorphismClassRepresentatives** (**DESIGN: BlockDesignIsomorphismClassRepresentatives**) to the constructed designs.

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

To install PAG, copy and unpack the package 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 binaries. 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.

Installations files for PAG are available from the authors. If you are interested, please write to vedran.krcadinac@math.hr.

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.

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 with 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,rec(NonIsomorphic:=true));
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. Both designs have \mathbb{Z}_{13} as their full automorphism group.

Example

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

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 $PTL(2,27)$ 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. Using the FinInG package, we can get $PTL(2,27)$ 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 the group 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);
Computing t-subset orbit representatives...
14
Computing k-subset orbit representatives...
72
Computing the Kramer-Mesner matrix...
.
.
.
Loops: 27732
Total number of solutions: 3

total enumeration time: 0:00:00
gap> Size(d);
4
```

Notice that A. Wassermann's LLL solver [Was98] reports finding 3 solutions, but we get 4 sets of base blocks. That's because the solver may return the same solution more than once. Here is how to get rid of multiple solutions.

Example

```
gap> Size(AsSet(d));
3
```

Most of the CPU time in the example above was used to compute the Kramer-Mesner matrix. The left-hand 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> PAGGlobalOptions.Silent:=true;
true
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's a semidirect product of the additive group of V and a group of order 32.

Example

```
gap> aut:=AutomorphismGroup(d);
<permutation group with 4 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],rec(NonIsomorphic:=true));
gap> List(dd,x->Size(AutomorphismGroup(x)));
[ 1944, 15552, 1296, 2592, 3888 ]
```

Two of the new designs have larger full automorphism groups than 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(NonIsomorphic:=true,
> IntersectionNumbers:=[4,8]));
gap> Size(d);
3
```

We check that they have all the 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,AutomorphismGroup);
gap> List(aut,StructureDescription);
[ "PSL(3,4) : C2", "(C2 x C2 x C2 x C2) : A5", "(C2 x C2 x C2 x C2) : S5" ]
```

1.4 Examples: Latin Squares

1.5 Examples: Cubes of Symmetric Designs

Cubes of symmetric designs were introduced in [KPT23]. Here is the motivational 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, 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, 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 the $(7,3,1)$ design. 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 ] ]
```

For any $d \geq 2$, a d -dimensional cube of symmetric designs can be constructed from a difference set. We use the representation of difference sets from the **DifSets** package (**DifSets: Difference Sets**). For $d = 2$, the cube is simply an incidence matrix of the associated symmetric design.

Example

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

Here is a small 4-dimensional $(3,2,1)$ cube.

Example

```
gap> 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 ] ] ] ]
```

Here are all 3-dimensional difference cubes constructed from the groups 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> Size(CubeAut(c1));
1323
gap> Size(CubeAut(c2));
2646
gap> List([c1, c2], CubeTest);
[ [ [ 21, 5, 1 ] ], [ [ 21, 5, 1 ] ] ]
```

We can make a non-difference cube by the "group cube" construction from [KPT23] (Theorem 4.1). First we find all $(21, 5, 1)$ designs whose blocks are 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. There are 42 designs that 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 any one of these 42 designs.

Example

```
gap> c3:=GroupCube(g[1],nondev[1],3);;  
gap> CubeTest(c3);  
[ [ 21, 5, 1 ] ]  
gap> Size(CubeAut(c3));  
441
```

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 $(1, \dots, n)$.

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 AllSubgroupsConjugation

▷ `AllSubgroupsConjugation(G)` (function)

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

2.1.11 PermRepresentationRight

▷ `PermRepresentationRight(G)` (function)

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

2.1.12 PermRepresentationLeft

▷ `PermRepresentationLeft(G)` (function)

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

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 filtering 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, a list of base blocks for the constructed designs is returned. If $opt.Design$ is defined, the designs are returned in the `Design` package format (**DESIGN: Design**). If $opt.NonIsomorphic$ is defined, the designs are returned in `Design` format and isomorph-rejection is performed. 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 $lambda \leq 3$.
- *IntersectionNumbers*:= lin . Search for designs with block intersection numbers 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, representing 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.

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 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 .

2.3.8 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 .

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 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.

2.4.4 BlockScheme

▷ `BlockScheme(d)` (function)

Returns the block intersection scheme of a schematic block design d . If d is not schematic, returns fail. Uses the package `AssociationSchemes`.

2.4.5 TDesignB

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

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

2.4.6 IversonBracket

▷ `IversonBracket(P)` (function)

Returns 1 if P is true, and 0 otherwise.

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 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 size of the sets s is written first, followed by the matrix entries. Integers are separated by whitespaces.

2.5.3 CayleyTableOfGroup

▷ `CayleyTableOfGroup(G)` (function)

Returns a Cayley table of the group G . The elements are integers $1, \dots, \text{Order}(G)$.

2.5.4 FieldToMOLS

▷ `FieldToMOLS(F)` (function)

Construct a complete set of MOLS from the finite field F .

2.5.5 MOLSAut

▷ `MOLSAut(ls[, opt])` (function)

Compute autotopism, autoparatopism, or automorphism groups of MOLS sets in the list ls . A record with options can be supplied. By default, autotopism groups are computed. If `opt.Paratopism` is defined, autoparatopism groups are computed. If `opt.Isomorphism` is defined, automorphism groups are computed.

2.5.6 MOLSFilter

▷ `MOLSFilter(ls[, opt])` (function)

Returns representatives of isotopism, paratopism, or isomorphism classes of MOLS sets in the list ls . A record with options can be supplied. By default, isotopism class representatives are returned. If `opt.Paratopism` is defined, paratopism class representatives (main class representatives) are returned. If `opt.Isomorphism` is defined, isomorphism class representatives are returned.

2.5.7 IsotopismToPerm

▷ `IsotopismToPerm(n , l)` (function)

Transforms an isotopism, i.e. a list l of three permutations of degree n , to a single permutation of degree $3n$.

2.5.8 PermToIsotopism

▷ `PermToIsotopism(n , p)` (function)

Transforms a permutation p of degree $3n$ to an isotopism, i.e. a list of three permutations of degree n .

2.5.9 MOLSSubsetOrbitRep

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

Computes representatives of pairs and $(s+2)$ -tuples for the construction of MOLS of order n with prescribed autotopism group G . A list containing pairs representatives in the first component and tuples representatives in the second component is returned.

2.5.10 TuplesToMOLS

▷ `TuplesToMOLS(n, s, T)` (function)

Transforms a set of $(s+2)$ -tuples T to a set of MOLS of order n .

2.5.11 KramerMesnerMOLS

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

Search for MOLS sets of order n and size s with prescribed autotopism group G . A record opt with options can be supplied. By default, A.Wassermann's LLL solver `solvediophant` is used and all constructed MOLS are returned, i.e. no filtering is performed. Available options are:

- `Solver:="solvecm"` The backtracing solver `solvecm` is used.
- `Filter:="Isotopism"` Non-isotopic MOLS are returned.
- `Filter:="Paratopism"` Non-paratopic MOLS are returned.
- `Filter:="Isomorphism"` Non-isomorphic MOLS are returned.

2.5.12 KramerMesnerMOLSParatopism

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

Search for MOLS sets of order n and size s with prescribed autoparatopism group G . A record opt with options can be supplied. By default, A.Wassermann's LLL solver `solvediophant` is used and all constructed MOLS are returned, i.e. no filtering is performed. Available options are:

- `Solver:="solvecm"` The backtracing solver `solvecm` is used.
- `Filter:="Isotopism"` Non-isotopic MOLS are returned.
- `Filter:="Paratopism"` Non-paratopic MOLS are returned.
- `Filter:="Isomorphism"` Non-isomorphic MOLS are returned.

2.6 Cubes of Symmetric Designs

2.6.1 DifferenceCube

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

Returns the d -dimensional difference cube constructed from a difference set ds in the group G .

2.6.2 GroupCube

▷ `GroupCube(G , dds , d)` (function)

Returns the d -dimensional 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 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 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 will be supplied. For a d -dimensional cube C of order v , the following calls will return:

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

2.6.5 CubeToOrthogonalArray

▷ `CubeToOrthogonalArray(C)` (function)

Transforms the incidence cube C to an equivalent orthogonal array.

2.6.6 OrthogonalArrayToCube

▷ `OrthogonalArrayToCube(OA)` (function)

Transforms the orthogonal array OA to an equivalent incidence cube.

2.6.7 CubeToTransversalDesign

▷ `CubeToTransversalDesign(C)` (function)

Transforms the incidence cube C to an equivalent transversal design.

2.6.8 TransversalDesignToCube

▷ `TransversalDesignToCube(TD)` (function)

Transforms the transversal design TD to an equivalent incidence cube.

2.6.9 LatinSquareToCube

▷ `LatinSquareToCube(L)` (function)

Transforms the Latin square L to an equivalent incidence cube.

2.6.10 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 $(v,k,lambda)$ cube.

2.6.11 CubeInvariant

▷ `CubeInvariant(C)` (function)

Computes an equivalence invariant of the cube C based on automorphism group sizes of its slices. Cubes equivalent under paratopy have the same invariant.

2.6.12 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 will be forwarded to the `BlockDesignAut (2.4.1)` function; see its documentation.

2.6.13 CubeFilter

▷ `CubeFilter(c1[, opt])` (function)

Eliminates equivalent copies from a list of incidence cubes `c1`. 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 will be forwarded to the `BlockDesignFilter (2.4.2)` function; see its documentation.

2.7 Hadamard Matrices

2.7.1 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`.

2.7.2 HadamardMatFilter

▷ `HadamardMatFilter(h1[, opt])` (function)

Eliminates equivalent copies from a list of Hadamard matrices `h1`. 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 Global Options

2.8.1 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

- [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. 18
- [KPT23] V. Krcadinac, M. O. Pavcevic, and K. Tabak. Cubes of symmetric designs. *preprint*, 2023. <https://arxiv.org/abs/...> 10, 12
- [Krc18] V. Krcadinac. Some new designs with prescribed automorphism groups. *J. Combin. Des.*, 26(4):193–200, 2018. 4
- [KV16] V. Krcadinac and R. Vlahovic. New quasi-symmetric designs by the kramer-mesner method. *Discrete Math.*, 339(12):2884–2890, 2016. 9, 16, 18
- [KVK21] V. Krcadinac and R. Vlahovic Kruc. Quasi-symmetric designs on 56 points. *Adv. Math. Commun.*, 15(4):633–646, 2021. 16
- [McK79] B. McKay. Hadamard equivalence via graph isomorphism. *Discrete Math.*, 27:213–214, 1979. 25
- [MP14] B. McKay and A. Piperno. Practical graph isomorphism, II. *J. Symbolic Comput.*, 60:94–112, 2014. 19, 24, 25
- [Nak21] A. Nakic. The first example of a simple 2-(81,6,2) design. *Examples and Counterexamples*, 1:100005, 2021. 8, 9
- [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
- [Was98] A. Wassermann. Finding simple t -designs with enumeration techniques. *J. Combin. Des.*, 6(2):79–90, 1998. 7, 18

Index

AllSubgroupsConjugation, 15

BaseBlocks, 18

BlockDesignAut, 19

BlockDesignFilter, 19

BlockScheme, 20

CayleyTableOfGroup, 21

CompatibilityMat, 17

CubeAut, 24

CubeFilter, 25

CubeInvariant, 24

CubeSlice, 23

CubeSlices, 23

CubeTest, 24

CubeToOrthogonalArray, 23

CubeToTransversalDesign, 24

CyclicPerm, 14

DifferenceCube, 23

ExpandMatRHS, 18

FieldToMOLS, 21

GroupCube, 23

HadamardMatAut, 25

HadamardMatFilter, 25

IntersectionNumbers, 20

IsGoodSubsetOrbit, 16

IsotopismToPerm, 21

IversonBracket, 20

KramerMesnerMat, 17

KramerMesnerMOLS, 22

KramerMesnerMOLSParatopism, 22

KramerMesnerSearch, 17

LatinSquareToCube, 24

LeftDevelopment, 18

License, 2

MOLSAut, 21

MOLSFilter, 21

MOLSSubsetOrbitRep, 22

MoveGroup, 14

MovePerm, 14

MultiGroup, 15

MultiPerm, 15

OrbitFilter1, 17

OrbitFilter2, 17

OrthogonalArrayToCube, 23

PAG, 4

PAGGlobalOptions, 25

PermRepresentationLeft, 15

PermRepresentationRight, 15

PermToIsotopism, 21

PrimitiveGroupsOfDegree, 15

ReadMOLS, 20

RestrictedGroup, 15

RightDevelopment, 18

SmallLambdaFilter, 16

SolveKramerMesner, 18

SubsetOrbitRep, 16

SubsetOrbitRepIN, 16

SubsetOrbitRepShort1, 16

TDesignB, 20

ToGroup, 14

TransitiveGroupsOfDegree, 15

TransversalDesignToCube, 24

TuplesToMOLS, 22

WriteMOLS, 21