Using GAP packages for research in graph theory, design theory, and finite geometry

Leonard H. Soicher
School of Mathematical Sciences
Queen Mary University of London
Mile End Road, London E1 4NS, UK

January 19, 2023

Abstract

The GAP system is a freely available, open-source computer system for algebra and discrete mathematics, with an emphasis on computational group theory. This article provides a tutorial introduction to the GRAPE, DESIGN, and FinInG packages for GAP. The GRAPE package is used to construct and study graphs related to groups, designs, and finite geometries. The DESIGN package is used to construct, classify, partition, and study block designs. The FinInG package has comprehensive functionality for many types of finite incidence structures and their groups of automorphisms, and we focus on its application to projective spaces and coset geometries. Instructive examples are given throughout, including some research applications of the author.

1 Introduction

This article is based on selected material from the author's (online) minicourse [36] given at the Graphs and Groups, Geometries and GAP (G2G2) Summer School, Rogla, Slovenia, 2021. Its purpose is to help researchers find out about and use certain packages of the GAP system [15] in their work on groups, graphs, designs, and finite geometries, and the fruitful interplay of these areas.

We will first focus on the GRAPE package [35], to construct and study graphs related to groups, designs, and finite geometries. Then we will look at the

DESIGN package [34], to construct, classify, partition, and study block designs. After that, we will demonstrate certain functionality for projective spaces and coset geometries of the FinInG package [3] for finite incidence geometry. Groups play fundamental roles in the construction, classification, and study of many types of combinatorial structures, and we will see the heavy involvement of groups in all three packages presented.

We will also see many examples of the use of these packages, including some extended examples demonstrating research applications of the author. The given examples come from the log-file of a single continuous GAP computation (run from a file), which takes a total of about 13 minutes run-time on an i5 laptop computer running Linux. The research applications include showing that the Haemer's partial geometry [18] is uniquely determined, up to isomorphism as a partial linear space, by its point graph and parameters, a construction of the Moscow-Soicher graph [11, Section 3.2.5], a classification and study of certain Sylvester designs [1], the determination that the Cohen-Tits near octagon [7, Section 13.6] has no ovoid, but does have a resolution, classifications of maximal partial spreads in $\mathrm{PG}(3, 4)$, and the construction and study of a flag-transitive coset geometry for the Hall-Janko sporadic simple group J_2.

1.1 GAP and its packages

The GAP system is an internationally developed computer system for algebra and discrete mathematics, with an emphasis on computational group theory. GAP is used in research and teaching for studying groups and their representations, rings, vector spaces, algebras, graphs, designs, finite geometries, and more. The GAP system is open source and freely available. It has a kernel written in the C language, but is mainly written in the higher-level GAP programming language. GAP has a library of thousands of functions written in the GAP language, and also provides large data libraries of mathematical objects. The packages described in this article make extensive use of GAP functionality, in particular its powerful group-theoretic machinery.

The main GAP website is `https://www.gap-system.org`, from which you can download the latest version of GAP to install on your computer. You will find extensive documentation, including the GAP reference manual, and learning resources at `https://www.gap-system.org/Doc/doc.html`.

GAP packages are structured open-source extensions to GAP, usually user-contributed. They provide extra functionality or data to GAP, usually have their own separate manual, integrate smoothly with GAP and its help system, and are distributed with GAP. Package authors get full credit and are usually

responsible for the maintenance of their packages. Packages may also be formally refereed, which has been the case with the DESIGN and FinInG packages.

The reference manuals for the GAP packages described in this article are available from their entries in `https://www.gap-system.org/Packages/packages.html`, and via online help in GAP. The present author made significant use of these manuals in the preparation of this article. In particular, the descriptions of the data structures and functions of the GRAPE and DESIGN packages that are given here are largely derived from their respective manuals. The reader should consult the GRAPE, DESIGN, and FinInG manuals for thorough documentation of these packages, including further optional function parameters, and much further functionality not covered here. For even more in-depth understanding of GAP and its packages, see also their (open) source code.

1.2 Prerequisites

The author's G2G2 minicourse lectures and exercises [36] include an introduction to the use of GAP, but for this article, it is assumed that the reader is already familiar with the GAP system, including basic programming in GAP, and working with integers, lists, sets, records, user-defined functions, permutation groups, and group actions. The tutorial [16] included with the GAP distribution covers all the background in GAP that is needed here, and more. You may also find the introductory GAP course [22] helpful.

No particular familiarity with the GRAPE, DESIGN, and FinInG packages is assumed. However, it is assumed that the reader has some basic knowledge in graph theory and permutation group theory, including the theory of group actions. A good reference for (algebraic) graph theory is [17], and for permutation groups, [9] is recommended. Some familiarity with design theory and finite geometry is useful, but not essential. For background in design theory, see [10], and for finite geometry, see [2] and [26].

2 The GRAPE package

The GRAPE package for GAP provides functionality for graphs, and is designed primarily for applications in algebraic graph theory, permutation group theory, design theory, and finite geometry.

Within GAP, a package is loaded using the `LoadPackage` command. For example:

```
gap> LoadPackage("grape",false);
true
```

The (optional) second parameter of the `LoadPackage` command was set to `false` to suppress the printing of the package banner. The return value of `true` indicates that the package was loaded correctly.

In general, GRAPE deals with finite directed graphs, which may have loops, but have no multiple edges. For the mathematical purposes of GRAPE, a **graph** consists of a finite set of **vertices**, together with a set of ordered pairs of vertices called **edges**. An edge $[x, y]$ in a graph is a **loop** if $x = y$. A graph is **simple** if it has no loops and whenever $[x, y]$ is an edge, then so is $[y, x]$. A simple graph can thus be considered to be a finite undirected graph having no loops and no multiple edges. Some GRAPE functions only work for simple graphs, but these functions will check if an input graph is simple.

Graphs Γ and Δ are **isomorphic** if there is an **isomorphism** from Γ to Δ, that is, a bijection ϕ from the vertex-set of Γ to that of Δ, such that, if v, w are vertices of Γ then $[v, w]$ is an edge of Γ if and only if $[v^\phi, w^\phi]$ is an edge of Δ. An **automorphism** of a graph Γ is an isomorphism from Γ to itself. The set of all automorphisms of Γ forms a group of permutations of the vertices of Γ, called the **automorphism group** of Γ.

In GRAPE, a graph always comes together with an associated group of automorphisms. This is an important and fundamental feature of GRAPE. This group is set by GRAPE when the graph is constructed. It is used by GRAPE to store the graph compactly, to speed up computations with the graph, and can affect the output of certain GRAPE functions. Often, but not always, this group is the full automorphism group of the graph. To have a new group (of automorphisms) associated with a graph in GRAPE, the user should construct a new graph with the new group, using the function `NewGroupGraph`.

2.1 The structure of a graph in GRAPE

In GRAPE, a graph *gamma* is stored as a record, with mandatory components `isGraph`, `order`, `group`, `schreierVector`, `representatives`, and `adjacencies`. Usually, the user need not be aware of this record structure, and is strongly advised only to use GRAPE functions to construct and modify graphs.

The `isGraph` component is set to `true` to specify that the record is a GRAPE graph. The `order` component gives the number of vertices of *gamma*.

The vertices of *gamma* are always $\{1, 2, \ldots, gamma.\texttt{order}\}$, but they may also be given names, either by a user (using `AssignVertexNames`) or by a

function constructing a graph (e.g. `Graph`, `InducedSubgraph`, `CayleyGraph`, `QuotientGraph`). The `names` component, if present, records these names, with *gamma*`.names[`i`]` being the name of vertex i. If the `names` component is not present, then the names are taken to be $1, 2, \ldots,$ *gamma*`.order`.

The `group` component records the GAP permutation group associated with *gamma*. This group must be a subgroup of the automorphism group of *gamma*.

The `representatives` component records a set of orbit representatives for the action of *gamma*`.group` on the vertices of *gamma*, with *gamma*`.adjacencies[`i`]` being the set of vertices (out)adjacent to *gamma*`.representatives[`i`]`.

`GeneratorsOfGroup(`*gamma*`.group)`, together with the `schreierVector` and `adjacencies` components, are used to compute the (out)adjacency-set of a given vertex of *gamma* when needed. See [30] for details on how this is done, as well as other insights into the algorithms used by GRAPE.

The only mandatory component which may change once a graph is initially constructed is `adjacencies` (when an edge-orbit of *gamma*`.group` is added to, or removed from, *gamma*). A graph record may also have some additional optional components which record information about that graph.

Here is a very simple example of the use of GRAPE, in which we construct the famous Petersen graph and compute some of its properties. More explanation of the functions used will be given later.

```
gap> Petersen := Graph( SymmetricGroup([1..5]),
>    Combinations([1..5],2), OnSets,
>    function(x,y) return Intersection(x,y)=[]; end,
>    true );
rec( adjacencies := [ [ 8, 9, 10 ] ],
  group := Group([ (1,5,8,10,4)(2,6,9,3,7), (2,5)(3,6)(4,7) ]),
  isGraph := true,
  names := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ],
      [ 2, 4 ], [ 2, 5 ], [ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ],
  order := 10, representatives := [ 1 ],
  schreierVector := [ -1, 2, 2, 1, 1, 1, 2, 1, 1, 1 ] )
gap> Vertices(Petersen);
[ 1 .. 10 ]
gap> VertexNames(Petersen);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 2, 4 ],
  [ 2, 5 ], [ 3, 4 ], [ 3, 5 ], [ 4, 5 ] ]
gap> DirectedEdges(Petersen);
[ [ 1, 8 ], [ 1, 9 ], [ 1, 10 ], [ 2, 6 ], [ 2, 7 ], [ 2, 10 ],
  [ 3, 5 ], [ 3, 7 ], [ 3, 9 ], [ 4, 5 ], [ 4, 6 ], [ 4, 8 ],
  [ 5, 3 ], [ 5, 4 ], [ 5, 10 ], [ 6, 2 ], [ 6, 4 ], [ 6, 9 ],
  [ 7, 2 ], [ 7, 3 ], [ 7, 8 ], [ 8, 1 ], [ 8, 4 ], [ 8, 7 ],
  [ 9, 1 ], [ 9, 3 ], [ 9, 6 ], [ 10, 1 ], [ 10, 2 ], [ 10, 5 ] ]
```

```
gap> UndirectedEdges(Petersen);
[ [ 1, 8 ], [ 1, 9 ], [ 1, 10 ], [ 2, 6 ], [ 2, 7 ], [ 2, 10 ],
  [ 3, 5 ], [ 3, 7 ], [ 3, 9 ], [ 4, 5 ], [ 4, 6 ], [ 4, 8 ],
  [ 5, 10 ], [ 6, 9 ], [ 7, 8 ] ]
gap> Adjacency(Petersen,5);  # for example
[ 3, 4, 10 ]
gap> Diameter(Petersen);
2
gap> Girth(Petersen);
5
gap> autgrp:=AutomorphismGroup(Petersen);
Group([ (3,4)(6,7)(8,9), (2,3)(5,6)(9,10), (2,5)(3,6)(4,7), (1,2)(6,8)
  (7,9) ])
gap> Size(autgrp);
120
gap> CliqueNumber(Petersen);
2
gap> MaximumClique(Petersen);
[ 1, 8 ]
gap> ChromaticNumber(Petersen);
3
gap> MinimumVertexColouring(Petersen);
[ 1, 1, 2, 3, 1, 2, 3, 2, 3, 2 ]
```

2.2 The function `Graph` in GRAPE

This is the most general and useful way of constructing a graph in GRAPE. If you learn just one GRAPE function to construct graphs, this is it!

A basic call to this function has the form

`Graph(` *G*`,` *L*`,` *act*`,` *rel* `).`

The parameter L should be a list of elements of a set S on which the group G acts, with the action given by the function *act*. The parameter *rel* should be a boolean function defining a G-invariant relation on S (so that for g in G, x, y in S, $rel(x, y)$ if and only if $rel(act(x, g), act(y, g))$).

Then the function `Graph` returns a graph *gamma* which has as vertex-names (an immutable copy of)

`Concatenation( Orbits(` *G*`,` *L*`,` *act* `) ),`

and for vertices v, w of *gamma*, $[v, w]$ is a (directed) edge if and only if

rel`( VertexName(` *gamma*`,` *v* `), VertexName(` *gamma*`,` *w* `) ).`

There is an additional optional (boolean) parameter, *invt* (default: `false`), which if set to `true`, asserts that L is a duplicate-free list invariant (setwise) under G with respect to action *act*, and then the function `Graph` behaves as above, except that the vertex-names of *gamma* become (an immutable copy of) L. In particular, this allows a user to specify the ordering of the vertices in the returned graph, as GAP makes no guarantees about the order of the orbits returned by the function `Orbits`, nor about the order of the elements within these orbits.

The group associated with the graph *gamma* returned is the image of the action homomorphism for G acting via act on `VertexNames(gamma)`.

We now present Peter Cameron's construction of the Hoffman-Singleton graph using the function `Graph`, closely following [9, Section 3.6]. The vertices consist of the 35 3-subsets of $\{1,\ldots,7\}$, together with one orbit of 15 projective planes of order 2 on the points $\{1,\ldots,7\}$, under the action of the alternating group A_7. The adjacencies are described by the function `HoffmanSingletonAdjacency` below.

```
gap> projectiveplane:=Set(Orbit(Group((1,2,3,4,5,6,7)),[1,2,4],OnSets));
[ [ 1, 2, 4 ], [ 1, 3, 7 ], [ 1, 5, 6 ], [ 2, 3, 5 ], [ 2, 6, 7 ],
  [ 3, 4, 6 ], [ 4, 5, 7 ] ]
gap> #
gap> # Now  projectiveplane  is (the set of lines of) a projective plane
gap> # of order 2.
gap> #
gap> HoffmanSingletonAction:=function(x,g)
> #
> # This function gives the action of AlternatingGroup([1..7])
> # on the vertices of the Hoffman-Singleton graph, in
> # Peter Cameron's construction.
> #
> if Size(x)=3 then               # x is a 3-set
>    return OnSets(x,g);
> else                            # x is a projective plane
>    return OnSetsSets(x,g);
> fi;
> end;;
gap> HoffmanSingletonAdjacency:=function(x,y)
> #
> # This boolean function returns  true  iff vertices  x  and  y
> # are adjacent in the Hoffman-Singleton graph, in
> # Peter Cameron's construction.
> #
> if Size(x)=3 then                  # x is a 3-set
>    if Size(y)=3 then               # y is a 3-set
>       return Intersection(x,y)=[]; # join iff x and y disjoint
```

```
>    else                            # y is a projective plane
>       return x in y;               # join iff x is a line of y
>    fi;
> else                               # x is a projective plane
>    if Size(y)=3 then               # y is a 3-set
>       return y in x;               # join iff y is a line of x
>    else                            # y is a projective plane
>       return false;                # don't join
>    fi;
> fi;
> end;;
gap> HoffmanSingleton:=Graph( AlternatingGroup([1..7]),
>     [[1,2,3], projectiveplane],
>     HoffmanSingletonAction,
>     HoffmanSingletonAdjacency);;
gap> IsSimpleGraph(HoffmanSingleton);
true
gap> OrderGraph(HoffmanSingleton);  # number of vertices
50
gap> VertexDegrees(HoffmanSingleton);  # set of vertex degrees
[ 7 ]
gap> Diameter(HoffmanSingleton);
2
gap> Girth(HoffmanSingleton);
5
gap> Size(HoffmanSingleton.group);
2520
gap> autgrp:=AutomorphismGroup(HoffmanSingleton);;
gap> Size(autgrp);
252000
gap> HoffmanSingleton:=NewGroupGraph(autgrp,HoffmanSingleton);;
gap> # So now the group associated with  HoffmanSingleton  is its
gap> # full automorphism group.
gap> Size(HoffmanSingleton.group);
252000
gap> DisplayCompositionSeries(HoffmanSingleton.group);
G (5 gens, size 252000)
 | C2
S (2 gens, size 126000)
 | U3(5)
1 (0 gens, size 1)
```

2.3 EdgeOrbitsGraph

A common way to construct a graph in GRAPE is via the function `EdgeOrbitsGraph`. Where n is a non-negative integer, G is a permutation group on $\{1,\ldots,n\}$, and *edges* is a list of ordered pairs of elements of $\{1,\ldots,n\}$, the function call

`EdgeOrbitsGraph(` G`,` *edges*`,` n `)`

returns the (directed) graph with vertex-set $\{1, \ldots, n\}$, and edge-set the union of the G-orbits of the ordered pairs in *edges*. The group associated with the returned graph is G. The parameter n may be omitted, in which case n is taken to be the largest point moved by G.

Note that G may be the trivial permutation group, `Group( () )`, in which case the (directed) edges of the returned graph are precisely those in the list *edges* (not including any repeats).
Here is an example.

```
gap> G:=AllPrimitiveGroups(NrMovedPoints,275,Size,2025*443520*2);
[ McL:2 ]
gap> G:=G[1];  # the automorphism group of the McLaughlin group
McL:2
gap> H:=Stabilizer(G,1);
<permutation group of size 6531840 with 3 generators>
gap> orbs:=List(Orbits(H,[1..275]),Set);;
gap> List(orbs,Length);
[ 1, 112, 162 ]
gap> y:=First(orbs,x->Length(x)=112)[1];
2
gap> McLaughlin:=EdgeOrbitsGraph(G,[[1,y]]);;  # the McLaughlin graph
gap> IsSimpleGraph(McLaughlin);
true
gap> OrderGraph(McLaughlin);
275
gap> VertexDegrees(McLaughlin);
[ 112 ]
```

Some other useful GRAPE functions to construct graphs are `NullGraph`, `CompleteGraph`, `CayleyGraph`, `JohnsonGraph`, and `HammingGraph`.

2.4 Automorphism groups and isomorphism testing in GRAPE

GRAPE contains functionality to test graph isomorphism and to calculate the automorphism group of a graph. This functionality can also be applied to graphs with ordered vertex partitions (see `Graphs with colour-classes` in the GRAPE manual).

To do all this, by default GRAPE uses its included version (currently final patched version 2.2) of B.D. McKay's nauty software [25]. A Linux user may instead choose to have GRAPE use their own installed copy of T. Junttila's

and P. Kaski's bliss software [20]. The nauty (and nauty/traces) and bliss software packages are powerful tools using "partition backtrack" for computing the automorphism group of a graph and for canonically labelling a graph for the purpose of isomorphism testing. A basic introduction to these techniques is given in [30], with much more detailed expositions in [25, 20]. The GRAPE interfaces to nauty and bliss are seamless and transparent to the user.

As we have already seen, where *gamma* is a graph in GRAPE, the function call

`AutomorphismGroup(` *gamma* `)`

returns the automorphism group of *gamma*, which can then be studied using the permutation group functionality in GAP.

Where *gamma* and *delta* are graphs in GRAPE, the function call

`GraphIsomorphism(` *gamma*`,` *delta* `)`

returns a permutation giving an isomorphism from *gamma* to *delta* if these graphs are isomorphic, and returns the special value `fail` if the graphs are not isomorphic, whereas

`IsIsomorphicGraph(` *gamma*`,` *delta* `)`

returns `true` if the graphs are isomorphic, and `false` otherwise.
For example:

```
gap> J:=JohnsonGraph(7,3);;
gap> Size(AutomorphismGroup(J));
5040
gap> IsIsomorphicGraph(J,JohnsonGraph(7,4));
true
```

When comparing more than two graphs for pairwise isomorphism, you should use the function `GraphIsomorphismClassRepresentatives`. Where L is a list of graphs in GRAPE, the function call

`GraphIsomorphismClassRepresentatives(` L `)`

returns a list consisting of pairwise non-isomorphic elements of L (in some order), representing all the isomorphism classes of elements of L.

2.5 Local and global regularity parameters

Let *gamma* be a simple connected graph, and let V be a singleton vertex or set of vertices of *gamma*.

We say that *gamma* has the **local parameter** $c_i(V)$ (respectively $a_i(V)$, $b_i(V)$), with respect to V, if the number of vertices at distance $i-1$ (respectively i, $i+1$) from V that are adjacent to a vertex w at distance i from V is the constant $c_i(V)$ (respectively $a_i(V)$, $b_i(V)$) depending only on i and V (and not the choice of w). See `Distance` in the GRAPE manual.

We say that *gamma* has the **global parameter** c_i (respectively a_i, b_i) if the number of vertices at distance $i-1$ (respectively i, $i+1$) from a vertex v that are adjacent to a vertex w at distance i from v is the constant c_i (respectively a_i, b_i) depending only on i (and not the choice of v or w).

In GRAPE, the function call

`LocalParameters(` *gamma*`,` V `)`

returns a list whose i-th element is the list

$$[c_{i-1}(V), a_{i-1}(V), b_{i-1}(V)],$$

except that if some local parameter does not exist then -1 is put in its place.

The function call

`LocalParameters(` *gamma*`,` V`,` G `)`

does the same, except that the additional parameter G is assumed to be a subgroup of the automorphism group of *gamma* fixing V setwise. Including such a G can result in a performance gain.

The function call

`GlobalParameters(` *gamma* `)`

returns a list of length one more than the diameter of *gamma* (see `Diameter` in the GRAPE manual), and whose i-th element is the list

$$[c_{i-1}, a_{i-1}, b_{i-1}],$$

except that if some global parameter does not exist then -1 is put in its place.

Note that (the simple connected graph) *gamma* is **distance-regular** if and only if this function returns no -1 in place of a global parameter (see [7]).

See also `IsDistanceRegular` and `CollapsedAdjacencyMat` in the GRAPE manual.
Here are some examples.

```
gap> GlobalParameters(HoffmanSingleton);
[ [ 0, 0, 7 ], [ 1, 0, 6 ], [ 1, 6, 0 ] ]
gap> IsDistanceRegular(HoffmanSingleton);
true
gap> edge:=[1,Adjacency(HoffmanSingleton,1)[1]];
[ 1, 4 ]
gap> edge_stab:=Stabilizer(HoffmanSingleton.group,edge,OnSets);
<permutation group of size 1440 with 4 generators>
gap> LocalParameters(HoffmanSingleton,edge,edge_stab);
[ [ 0, 1, 6 ], [ 1, 0, 6 ], [ 2, 5, 0 ] ]
gap> GlobalParameters(McLaughlin);
[ [ 0, 0, 112 ], [ 1, 30, 81 ], [ 56, 56, 0 ] ]
gap> GlobalParameters(EdgeGraph(McLaughlin));
[ [ 0, 0, 222 ], [ 1, -1, -1 ], [ -1, -1, -1 ], [ 184, 38, 0 ] ]
```

2.6 The Sylvester graph

The **Sylvester graph** is the unique distance-regular graph with intersection array $\{5, 4, 2; 1, 1, 4\}$, that is, with global parameters

`[ [0,0,5],[1,0,4],[1,2,2],[4,1,0] ]`.

This graph can be constructed as the induced subgraph on the 36 vertices at distance 2 from a fixed edge in the Hoffman-Singleton graph (see [7, Section 13.1.A]). We do this now using GRAPE.

```
gap> Sylvester:=DistanceSetInduced(HoffmanSingleton,2,edge,edge_stab);;
gap> GlobalParameters(Sylvester);
[ [ 0, 0, 5 ], [ 1, 0, 4 ], [ 1, 2, 2 ], [ 4, 1, 0 ] ]
```

2.7 Construction of a distance-regular graph from a transitive group

Let G be a transitive permutation group on $V := \{1, \ldots, n\}$. A **generalized orbital graph** for G is a simple graph with vertex-set V on which G acts naturally as a (vertex-transitive) group of automorphisms.

Here we construct the (non-null) generalized orbital graphs for the group M_{22}:2 (the automorphism group of the Mathieu group M_{22}) in its primitive action on 672 points. This includes the distance-regular so-called Moscow-Soicher graph (see [28] and [11, Section 3.2.5]).

```
gap> n:=672;
672
gap> G:=AllPrimitiveGroups(NrMovedPoints,n,Size,443520*2);
[ M(22).2 ]
gap> G:=G[1];  # the automorphism group of the Mathieu group $M_{22}$
M(22).2
gap> RankAction(G,[1..n]);
6
gap> L:=GeneralizedOrbitalGraphs(G);;
gap> # the non-null generalized orbital graphs
gap> Length(L);
31
gap> List(L,GlobalParameters);
[ [ [ 0, 0, 55 ], [ 1, 8, 46 ], [ -1, -1, -1 ], [ 45, 10, 0 ] ],
  [ [ 0, 0, 385 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 440 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 605 ], [ 1, -1, -1 ], [ 540, 65, 0 ] ],
  [ [ 0, 0, 671 ], [ 1, 670, 0 ] ],
  [ [ 0, 0, 506 ], [ 1, -1, -1 ], [ 398, 108, 0 ] ],
  [ [ 0, 0, 550 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 616 ], [ 1, -1, -1 ], [ 562, 54, 0 ] ],
  [ [ 0, 0, 451 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 110 ], [ 1, 28, 81 ], [ 18, 80, 12 ], [ 90, 20, 0 ] ],
  [ [ 0, 0, 275 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 341 ], [ 1, -1, -1 ], [ 170, 171, 0 ] ],
  [ [ 0, 0, 176 ], [ 1, 40, 135 ], [ 48, 128, 0 ] ],
  [ [ 0, 0, 220 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 286 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 121 ], [ 1, 20, 100 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 330 ], [ 1, 158, 171 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 385 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 550 ], [ 1, -1, -1 ], [ 450, 100, 0 ] ],
  [ [ 0, 0, 616 ], [ 1, -1, -1 ], [ 570, 46, 0 ] ],
  [ [ 0, 0, 451 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 495 ], [ 1, 366, 128 ], [ 360, 135, 0 ] ],
  [ [ 0, 0, 561 ], [ 1, -1, -1 ], [ 480, 81, 0 ] ],
  [ [ 0, 0, 396 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 55 ], [ 1, 0, 54 ], [ -1, -1, -1 ], [ 45, 10, 0 ] ],
  [ [ 0, 0, 220 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 286 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 121 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 165 ], [ 1, 56, 108 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 231 ], [ 1, -1, -1 ], [ -1, -1, 0 ] ],
  [ [ 0, 0, 66 ], [ 1, 0, 65 ], [ -1, -1, 0 ] ] ]
gap> MoscowSoicher:=First(L,x->VertexDegrees(x)=[110]);;
gap> IsDistanceRegular(MoscowSoicher);
true
gap> GlobalParameters(MoscowSoicher);
[ [ 0, 0, 110 ], [ 1, 28, 81 ], [ 18, 80, 12 ], [ 90, 20, 0 ] ]
```

```
gap> AutomorphismGroup(MoscowSoicher) = G;
true
```

See also `VertexTransitiveDRGs` and `OrbitalGraphColadjMats` in the GRAPE manual for further study of vertex-transitive graphs and the (homogeneous) coherent configurations associated to transitive permutation groups.

2.8 Cliques

Let Γ be a simple graph.

A **clique** of Γ is a set of pairwise adjacent vertices. A **maximal** clique of Γ is a clique which is not properly contained in any clique of Γ, while a **maximum** clique of Γ is a clique of largest size in Γ. The **clique number** of Γ is the size of a maximum clique of Γ. A **co-clique** (or **independent set**) of Γ is a set of pairwise non-adjacent vertices. Clearly, the co-cliques of Γ are precisely the cliques of the complement of Γ. The **independence number** of Γ is the size of a largest co-clique of Γ.

Now let *gamma* be a simple graph in GRAPE, with associated group $G :=$ *gamma*`.group`. Then GRAPE functions can compute the maximal cliques of *gamma*, or the cliques of given size in *gamma*, or the maximal cliques of given size in *gamma*, such that one such clique is determined or it is determined that no such cliques exist, or G-orbit generators of all such cliques are determined, or G-orbit representatives of all such cliques are determined. There is also the functionality to determine a maximum clique, and hence to determine the clique number of *gamma*. These are computationally HARD problems!

The main function for clique classification in GRAPE is `CompleteSubgraphsOfGivenSize`. We shall now study the use of this function for non-weighted cliques, but for a complete specification, see `CompleteSubgraphsOfGivenSize` in the GRAPE manual. Also see `CompleteSubgraphs`, `MaximumClique`, and `CliqueNumber`.

In GRAPE, where *gamma* is a simple graph, k is a non-negative integer, *alls* is 0, 1, or 2, and *maxi* is `true` or `false`, the function call

`CompleteSubgraphsOfGivenSize(` *gamma*`,` k`,` *alls*`,` *maxi* `)`

returns a set K (possibly empty) of cliques of size k of *gamma*. These are all maximal cliques if *maxi*=`true`, and not necessarily maximal cliques if *maxi*=`false`. The parameter *maxi* is optional, and has default value `false`.

The parameter *alls* controls how many cliques are returned.

First, suppose $alls = 0$. Then K will contain at most one element. If $maxi =$ `false` then K will contain a clique of size k if and only if *gamma* has such a

clique, and if *maxi* = `true` then K will contain a maximal clique of size k if and only if *gamma* has a maximal clique of that size.

In the above function call with *alls* = 0, if *gamma*.`group` is not the full automorphism group of *gamma*, it may be (much) more efficient to replace *gamma* by a copy whose associated group is its full automorphism group.

If *alls* = 1 and *maxi* = `false`, then K will contain (perhaps properly) a set of *gamma*.`group` orbit-representatives of the size k cliques of *gamma*. If *alls* = 1 and *maxi* = `true`, then K will contain (perhaps properly) a set of *gamma*.`group` orbit-representatives of the size k maximal cliques of *gamma*.

If *alls* = 2 and *maxi* = `false`, then K will be a set of *gamma*.`group` orbit-representatives of all the size k cliques of *gamma*. If *alls* = 2 and *maxi* = `true`, then K will be a set of *gamma*.`group` orbit-representatives of the size k maximal cliques of *gamma*.

Here are some illustrative examples.

```
gap> gamma:=MoscowSoicher;;
gap> CompleteSubgraphsOfGivenSize(gamma,5,0,false);
[ [ 1, 2, 18, 35, 41 ] ]
gap> CompleteSubgraphsOfGivenSize(gamma,5,0,true);
[ [ 1, 2, 80, 124, 455 ] ]
gap> CompleteSubgraphsOfGivenSize(gamma,5,1,false);
[ [ 1, 2, 18, 35, 41 ], [ 1, 2, 18, 35, 377 ], [ 1, 2, 18, 119, 161 ],
  [ 1, 2, 18, 119, 277 ], [ 1, 2, 18, 161, 277 ],
  [ 1, 2, 18, 281, 287 ], [ 1, 2, 18, 281, 366 ],
  [ 1, 2, 80, 124, 455 ], [ 1, 2, 124, 183, 461 ] ]
gap> CompleteSubgraphsOfGivenSize(gamma,5,1,true);
[ [ 1, 2, 80, 124, 455 ] ]
gap> CompleteSubgraphsOfGivenSize(gamma,5,2,false);
[ [ 1, 2, 18, 35, 41 ], [ 1, 2, 18, 119, 161 ],
  [ 1, 2, 18, 281, 287 ], [ 1, 2, 80, 124, 455 ] ]
gap> CompleteSubgraphsOfGivenSize(gamma,5,2,true);
[ [ 1, 2, 80, 124, 455 ] ]
gap> CompleteSubgraphsOfGivenSize(gamma,7,0,false);
[  ]
gap> C:=CompleteSubgraphsOfGivenSize(gamma,6,2,true);
[ [ 1, 2, 18, 35, 41, 377 ], [ 1, 2, 18, 119, 161, 277 ],
  [ 1, 2, 18, 281, 287, 366 ] ]
gap> List(C,clique->Size(Stabilizer(gamma.group,clique,OnSets)));
[ 360, 144, 36 ]
gap> CliqueNumber(McLaughlin);  # the size of a largest clique
5
gap> MaximumClique(McLaughlin); # a clique of largest size
[ 1, 2, 17, 45, 193 ]
gap> CliqueNumber(ComplementGraph(McLaughlin));
22
gap> # This is the independence number of the McLaughlin graph,
```

```
gap> # the size of a largest co-clique.
```

The function `CompleteSubgraphsOfGivenSize` can also compute and classify cliques with given vertex-weight sum in a vertex-weighted graph, where the weights can be positive integers or non-zero d-vectors of non-negative integers (satisfying certain conditions with respect to the group associated with the graph). This type of clique functionality in GRAPE is used by the DESIGN package for GAP, for functionality to construct and classify block designs of many types (including those invariant under a specified group), as well as to construct and classify parallel classes and resolutions of block designs. We will see some of this DESIGN package functionality later.

2.9 Proper vertex-colourings

Let Γ be a simple graph. A **proper vertex-colouring** of Γ is a labelling of its vertices by elements from a set of **colours**, such that adjacent vertices are labelled with different colours. Where k is a non-negative integer, a **vertex k-colouring** of Γ is a proper vertex-colouring using at most k colours. A **minimum vertex-colouring** of Γ is a vertex k-colouring with k as small as possible, and the **chromatic number** $\chi(\Gamma)$ of Γ is the number of colours used in a minimum vertex-colouring of Γ.

In GRAPE, a proper vertex-colouring of a simple graph is given as a list of positive integers (the colours), indexed by the vertices of the graph, such that the i-th list element is the colour of vertex i.

In GRAPE, where *gamma* is a simple graph and k is a non-negative integer, the function call

`VertexColouring(` *gamma*`,` k `)`

returns a vertex k-colouring of *gamma* if such a colouring exists; otherwise, the special value `fail` is returned. In general, this is a computationally HARD problem. See also `MinimumVertexColouring` and `ChromaticNumber` in the GRAPE manual.

2.9.1 On colouring the McLaughlin graph

In the code below, we first make the induced subgraph in the McLaughlin graph on the vertices at distance 1 from a fixed vertex (here the vertex 1). This is the so-called first subconstituent of the McLaughlin graph. We then verify that its chromatic number is 8. After that, we make the second

subconstituent of the McLaughlin graph, and determine that its chromatic number is 10.

```
gap> gamma:=DistanceSetInduced(McLaughlin,1,1);;
gap> # This is the induced subgraph on the vertices of the
gap> # McLaughlin graph at distance 1 from the vertex 1.
gap> GlobalParameters(gamma);
[ [ 0, 0, 30 ], [ 1, 2, 27 ], [ 10, 20, 0 ] ]
gap> VertexColouring(gamma,7);
fail
gap> VertexColouring(gamma,8)<>fail;
true
gap> delta:=DistanceSetInduced(McLaughlin,2,1);;
gap> # This is the induced subgraph on the vertices of the
gap> # McLaughlin graph at distance 2 from the vertex 1.
gap> GlobalParameters(delta);
[ [ 0, 0, 56 ], [ 1, 10, 45 ], [ 24, 32, 0 ] ]
gap> ChromaticNumber(delta);
10
```

Now here is a challenging problem. Let M be the McLaughlin graph. It follows from the preceding computation that $\chi(M) \leq 18$. In fact, using the "ant colony optimization" program supplied as supplementary material to [23], the author found a vertex 16-colouring of M. On the other hand, the eigenvalue-based lower bound given in [8, Corollary 3.6.4] shows that $\chi(M) \geq 15$. Thus, the chromatic number of the McLaughlin graph M is 15 or 16. The problem is to either find a vertex 15-colouring of M or to show there is no such colouring.

2.10 Partial linear spaces and partial geometries

Where s and t are positive integers, a **partial linear space** with **parameters** (s, t) consists of a finite set of **points**, together with a set of $(s+1)$-subsets of the points, called **lines**, such that every point is on exactly $t+1$ lines, and any two distinct points lie on (i.e. are contained in) at most one line (equivalently, any two distinct lines meet in at most one point).

Two partial linear spaces are **isomorphic** if there is a bijection from the point-set of the first to that of the second which applied to the set of lines of the first yields the set of lines of the second.

The **point graph** (or **collinearity graph**) of a partial linear space is the graph whose vertices are the points of the space, with two distinct points joined by an edge exactly when they lie on a common line.

Now, where *gamma* is a simple graph in GRAPE, and s and t are positive integers, the function call

`PartialLinearSpaces(` *gamma*`,` s`,` t `)`

returns a list of representatives of the distinct isomorphism classes of partial linear spaces with parameters (s, t) and point graph *gamma*. (This may take a (very) long time.) In the output of this function, a partial linear space S is given by its incidence graph *delta*, such that the group *delta*`.group` associated with *delta* is the automorphism group of S, acting on point-vertices and line-vertices, and preserving both sets. See the GRAPE manual entry for `PartialLinearSpaces` for more information, including details of optional parameters.
For example:

```
gap> K7:=CompleteGraph(SymmetricGroup([1..7]));;
gap> P:=PartialLinearSpaces(K7,2,2);
[ rec( adjacencies := [ [ 8, 9, 10 ], [ 1, 2, 3 ] ],
      group := Group([ (1,2)(5,6)(9,11)(10,12), (1,2,3)(5,6,7)
          (9,11,13)(10,12,14), (1,2,3)(4,7,6)(9,12,14)(10,11,13),
          (1,4,7,6,2,5,3)(8,9,13,10,11,12,14) ]), isGraph := true,
      isSimple := true,
      names := [ 1, 2, 3, 4, 5, 6, 7, [ 1, 2, 3 ], [ 1, 4, 5 ],
          [ 1, 6, 7 ], [ 2, 4, 6 ], [ 2, 5, 7 ], [ 3, 4, 7 ],
          [ 3, 5, 6 ] ], order := 14, representatives := [ 1, 8 ],
      schreierVector := [ -1, 1, 2, 4, 4, 1, 3, -2, 4, 1, 1, 3, 4, 2
         ] ) ]
gap> Length(P);
1
gap> GlobalParameters(P[1]);
[ [ 0, 0, 3 ], [ 1, 0, 2 ], [ 1, 0, 2 ], [ 3, 0, 0 ] ]
gap> Size(P[1].group);
168
gap> T:=ComplementGraph(JohnsonGraph(10,2));;
gap> P:=PartialLinearSpaces(T,4,6);;
gap> List(P,x->Size(x.group));
[ 216, 1512 ]
```

2.10.1 The Haemers partial geometry

The Haemers partial geometry [18] is a partial geometry with parameters $s = 4, t = 17, \alpha = 2$ and point graph the distance-2 graph of the edge graph of the Hoffman-Singleton graph. (The **distance**-k graph Γ_k of a simple graph Γ has the same vertex set as Γ, but with two vertices joined by an edge in Γ_k if and only if they have distance k in Γ. The **edge graph** $L(\Gamma)$ of a simple

graph Γ has as vertices the edges of Γ, with two such vertices joined by an edge in $L(\Gamma)$ if and only if they have exactly one common vertex in Γ.)

As done in the GRAPE manual, we now use the GRAPE function `PartialLinearSpaces` to construct the Haemers partial geometry and its dual, prove that they are determined up to isomorphism (as partial linear spaces) by their respective point graphs and parameters, and determine their full automorphism groups.

```
gap> pointgraph:=DistanceGraph(EdgeGraph(HoffmanSingleton),2);;
gap> GlobalParameters(pointgraph);
[ [ 0, 0, 72 ], [ 1, 20, 51 ], [ 36, 36, 0 ] ]
gap> VertexName(pointgraph,1);  # example vertex-name
[ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
gap> spaces:=PartialLinearSpaces(pointgraph,4,17);;
gap> Length(spaces);
1
gap> HaemersGeometry:=spaces[1];;
gap> #
gap> # Now  HaemersGeometry  is the incidence graph of the Haemers
gap> # partial geometry.
gap> #
gap> DisplayCompositionSeries(HaemersGeometry.group);
G (2 gens, size 2520)
 | A7
1 (0 gens, size 1)
gap> linevertex:=Adjacency(HaemersGeometry,1)[1];
176
gap> linegraph:=PointGraph(HaemersGeometry,linevertex);;
gap> GlobalParameters(linegraph);
[ [ 0, 0, 85 ], [ 1, 20, 64 ], [ 10, 75, 0 ] ]
gap> spaces:=PartialLinearSpaces(linegraph,17,4);;
gap> Length(spaces);
1
gap> DualHaemersGeometry:=spaces[1];;
gap> DisplayCompositionSeries(DualHaemersGeometry.group);
G (3 gens, size 2520)
 | A7
1 (0 gens, size 1)
```

2.11 Steve Linton's function `SmallestImageSet`

An important behind-the-scenes workhorse included in GRAPE is Steve Linton's function `SmallestImageSet`. This function is used in GRAPE in the classification of cliques and the classification of partial linear spaces with given point graph and parameters, as well as in the DESIGN package in the classification of block designs. The function is of use in many other situations

when classifying objects up to the action of a given permutation group G, when the objects can be represented as subsets of the permutation domain of G.

Let G be a permutation group on $X := \{1, \ldots, n\}$, and let $S \subseteq X$. Then the function call

`SmallestImageSet( `G`, `S` )`

returns the lexicographically least set in `Orbit(`G`,`S`,OnSets)`, without explicitly computing this (possibly huge) orbit. Thus, if T is any subset of X, then S and T are equivalent under the action of G if and only if

`SmallestImageSet( `G`, `S` ) = SmallestImageSet( `G`, `T` ).`

Typically, we set things up so that certain subsets of X represent the objects we are classifying, with two objects equivalent if and only if their representing sets are in the same G-orbit of sets. Then, if C is a list of subsets of X (say certain cliques of a given size in a graph), and we want to determine a set of (canonical) representatives for the distinct G-orbits of the elements of C, we can do this as:

`Set( List( `C`, c->SmallestImageSet( `G`,c ) ) ).`

Steve Linton's algorithm for `SmallestImageSet` is given in [24]. Further developments in the computation of minimal and canonical images with respect to a group action are given in [19].

3 The DESIGN package

The DESIGN package for GAP provides functionality for constructing, classifying, partitioning, and studying block designs, including the computation of statistical efficiency measures for these designs. For the use of the DESIGN package for statistical designs, see [31] and [1]. Here we concentrate on the combinatorial aspects of block designs. DESIGN makes use of the GRAPE package.

3.1 Block designs

A **block design** is an ordered pair (P, B), where P is a non-empty finite set whose elements are called **points**, and B is a non-empty finite multiset

whose elements are called **blocks**, such that each block is a non-empty finite multiset of points. For our purposes, a **multiset** is a list, where order does not matter, but the number of times an element appears does. We represent a multiset as an ordered list in GAP.

A **parallel class** (or **spread**) of a block design $D = (P, B)$ is a sub(multi)set of B forming a partition of P, and a **resolution** of D is a partition of B into parallel classes. We say that a block design is **resolvable** if it has a resolution.

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

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

A t-(v, k, λ) design is also an s-(v, k, λ_s) design for $0 \leq s \leq t$, where

$$\lambda_s = \lambda \binom{v-s}{t-s} \Big/ \binom{k-s}{t-s}.$$

For example, we compute these λ_s for a 5-$(24, 8, 1)$ design.

```
gap> LoadPackage("design",false);
true
gap> TDesignLambdas(5,24,8,1);
[ 759, 253, 77, 21, 5, 1 ]
```

The DESIGN package has extensive functionality for t-designs and their parameters. For example, you can compute an upper bound on the multiplicity of a block in any t-design with given parameters t, v, k, λ or in any resolvable such t-design. See the DESIGN manual (or online help in GAP) for specifications of the functions used below.

```
gap> TDesignLambdaMin(2,12,4);
3
gap> TDesignLambdas(2,12,4,3);
[ 33, 11, 3 ]
gap> TDesignBlockMultiplicityBound(2,12,4,3);
2
gap> ResolvableTDesignBlockMultiplicityBound(2,12,4,3);
```

```
1
gap> #
gap> # The DESIGN package knows the Bruck-Ryser-Chowla Theorem,
gap> # and so knows there is no projective plane of order 6.
gap> #
gap> TDesignBlockMultiplicityBound(2,43,7,1);
0
```

3.2 The function `BlockDesign`

The DESIGN package function `BlockDesign` gives a straightforward way of constructing a block design.

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

`BlockDesign(` v`,` B `)`

returns the block design with point-set V and block multiset C, where C is `SortedList(` B `)`. Moreover, where G is a group of permutations on V, the function call

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

returns the block design with point-set V and block multiset C, where now C is the sorted list of the concatenation of each of the G-orbits of the elements in the list B. For example:

```
gap> G:=Group((1,2,3,4,5,6,7));;
gap> design:=BlockDesign(7,[[1,2,4],[1,2,4]],G);
rec( autSubgroup := Group([ (1,2,3,4,5,6,7) ]),
  blocks := [ [ 1, 2, 4 ], [ 1, 2, 4 ], [ 1, 3, 7 ], [ 1, 3, 7 ],
      [ 1, 5, 6 ], [ 1, 5, 6 ], [ 2, 3, 5 ], [ 2, 3, 5 ],
      [ 2, 6, 7 ], [ 2, 6, 7 ], [ 3, 4, 6 ], [ 3, 4, 6 ],
      [ 4, 5, 7 ], [ 4, 5, 7 ] ], isBlockDesign := true, v := 7 )
gap> IsSimpleBlockDesign(design);
false
gap> AllTDesignLambdas(design);
[ 14, 6, 2 ]
gap> Size(AutomorphismGroup(design));
168
```

We note that, given a block design D, the function call

`AllTDesignLambdas(` D `)`

returns the empty list if D is not a t-design. Otherwise, D is a binary block design with constant block size k, say, and this function returns an immutable list L of length $T+1$, where T is the maximum $t \leq k$ such that D is a t-design, and, for $i = 1, \ldots, T+1$, the i-th element of L is equal to the (constant) number of blocks of D containing an $(i-1)$-subset of the point-set of D.

3.3 The structure of a block design in DESIGN

In DESIGN, a block design D is stored as a record, with mandatory components `isBlockDesign`, `v`, and `blocks`.

The `isBlockDesign` component is set to `true` to specify that the record is a DESIGN package block design. The component `v` gives the number of points of D. The points of D are $\{1, 2, \ldots, D.\texttt{v}\}$, but they may also be given names in the optional component `pointNames`, with $D.\texttt{pointNames}[i]$ the name of point i. The `blocks` component must be a sorted list of the blocks of D (including any repeats), with each block being a sorted list of points (including any repeats). A block design record may also have some optional components which store information about the design.

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

3.4 Automorphism groups and isomorphism testing in DESIGN

Let A and B be block designs. Then A and B are **isomorphic** if there is an **isomorphism** from A to B, that is, a bijection from the points of A to those of B which applied to the block multiset of A yields the block multiset of B. An **automorphism** of A is an isomorphism from A to itself. The set of all automorphisms of A forms a group, the **automorphism group** of A.

The DESIGN package contains functionality to test block design isomorphism and to calculate the automorphism group of a block design, but currently only for binary block designs (equivalently, block designs where every block is a set). To do this, the DESIGN package uses the graph automorphism group and isomorphism testing functionality in GRAPE.

Now suppose that A and B are binary block designs in the DESIGN package. Then the function call

`AutomorphismGroup(` A `)`

returns the automorphism group of A, which can then be studied using the permutation group functionality in GAP. The function call

`IsIsomorphicBlockDesign(` A`,` B `)`

returns `true` if the block designs are isomorphic, and `false` otherwise.

When comparing more than two binary block designs for pairwise isomorphism, you should use the function `BlockDesignIsomorphismClassRepresentatives`. Where L is a list of binary block designs in DESIGN package format, the function call

`BlockDesignIsomorphismClassRepresentatives(` L `)`

returns a list consisting of pairwise non-isomorphic elements of L, representing all the isomorphism classes of elements of L.

3.5 The function `BlockDesigns`

The most important DESIGN package function is `BlockDesigns`, which can construct and classify block designs satisfying a wide range of user-specified properties (although this may require a very large amount of store or time depending on the problem). The calling syntax is

`BlockDesigns(` *param* `)`

and the function returns a list of block designs whose properties are specified by the user in the record *param*, as described thoroughly in the full function documentation in the DESIGN manual. Only binary block designs with the given properties are generated if *param*`.blockDesign` is unbound or is a binary block design, which we assume to be the case in this article.

The required components of the record *param* are as follows:

- *param*`.v` should be a positive integer specifying the number of points (in each returned design)
- *param*`.blockSizes` should be a set of positive integers, specifying the allowable block size(s)
- *param*`.tSubsetStructure` should be a record, used to specify for one given $t \geq 0$, for each t-subset T of the points, the number of blocks containing T. This number may be a positive constant *lambda*, not depending on T, in which case, *param*`.tSubsetStructure` can simply be set to

`rec( t :=` *t*`, lambdas := [`*lambda*`] ).`

We will see an example using the more general form of *param*`.tSubsetStructure` later.

The default is to classify the required designs up to isomorphism. As an example, we now classify the 5-(12,6,1) designs, verifying the well-known result that the "small Witt design" is the unique such design (up to isomorphism).

```
gap> designs := BlockDesigns( rec(
>    v:=12,  # there are exactly 12 points
>    blockSizes:=[6], # each block has size 6
>    tSubsetStructure:=rec(t:=5, lambdas:=[1])
>        # every 5-subset of the points is contained in exactly one block
>    ) );;
gap> Length(designs);
1
gap> AllTDesignLambdas(designs[1]); # as a check
[ 132, 66, 30, 12, 4, 1 ]
gap> Size(AutomorphismGroup(designs[1]));
95040
```

We next classify, up to isomorphism, the binary block designs having 11 points, such that each block has size 4 or 5, and every pair of distinct points is contained in exactly two blocks.

```
gap> designs:=BlockDesigns( rec( v:=11, # there are exactly 11 points
>    blockSizes:=[4,5],  # each block has size 4 or 5
>    tSubsetStructure:=rec(t:=2, lambdas:=[2])
>        # every 2-subset of the points is contained in exactly two blocks
> ) );;
gap> Length(designs);
5
gap> List(designs,BlockSizes);
[ [ 5 ], [ 4, 5 ], [ 4, 5 ], [ 4, 5 ], [ 4, 5 ] ]
gap> List(designs,BlockNumbers);
[ [ 11 ], [ 10, 5 ], [ 10, 5 ], [ 10, 5 ], [ 10, 5 ] ]
gap> List(designs,AllTDesignLambdas);
[ [ 11, 5, 2 ], [  ], [  ], [  ], [  ] ]
gap> List(designs,d->Size(AutomorphismGroup(d)));
[ 660, 6, 8, 12, 120 ]
```

Further properties may optionally be specified to the function `BlockDesigns` via the record *param*. How to do this is precisely detailed in the DESIGN package documentation for `BlockDesigns`. These further properties include:

- a total number b of blocks, specifying that every returned design has exactly b blocks

- a replication number r, specifying that in every returned design, every point is in exactly r blocks
- a list giving, for each specified possible block size, a corresponding maximum multiplicity of a block of that size
- a list giving, for each specified possible block size, a corresponding number of blocks of that size
- the possible sizes of intersections of pairs of blocks of given sizes
- a permutation group G on the point set $\{1, \ldots, v\}$, such that two designs are considered to be isomorphic if one is in the G-orbit of the other (if *param*`.blockDesign` is unbound, the default is a group G giving the usual notion of block design isomorphism)
- a subgroup H of G such that H is required to be a subgroup of the automorphism group of each returned design (the default is for H to be the trivial permutation group)
- whether the user wants a single design with the specified properties (if one exists), a list of G-orbit representatives of all such designs (i.e. isomorphism class representatives as determined by G; this is the default), or a list of such designs containing at least one representative from each G-orbit.

For example, we now construct one simple 2-(20,5,4) design invariant under a group of order 19 (a block design is **simple** if it has no repeated block).

```
gap> H:=CyclicGroup(IsPermGroup,19);
Group([ (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19) ])
gap> D:=BlockDesigns( rec( v:=20,
>    blockSizes:=[5],
>    tSubsetStructure:=rec(t:=2, lambdas:=[4]),
>    blockMaxMultiplicities:=[1], # since we want a simple design
>    requiredAutSubgroup:=H,
>       # since we want a design whose automorphism group contains H
>    isoLevel:=0
>       # since we want just one design (if such a design exists)
>    ) );;
gap> Length(D);
1
gap> AllTDesignLambdas(D[1]); # a check
[ 76, 19, 4 ]
gap> Size(AutomorphismGroup(D[1]));
19
```

3.6 Computing subdesigns using the function `BlockDesigns`

Here a **subdesign** of a block design D means a block design with the same point set as D and whose block multiset is a submultiset of the blocks of D.

Setting the optional parameter *param*`.blockDesign` to be a block design D asks `BlockDesigns(` *param* `)` to construct subdesigns of D with the other given properties. In this case, the default is to determine the subdesigns up to the action of the automorphism group of D.

We now give an example of the use of subdesigns, where we determine certain "Sylvester designs", which are highly efficient block designs from a statistical design viewpoint. See [1]. A **Sylvester design** is a 1-(36,6,8) design, such that

- if a and b are distinct points then $\{a, b\}$ is contained in just 1 or 2 blocks, and
- the graph on the 36 points whose edges are those $\{a, b\}$ contained in exactly 2 blocks is isomorphic to the Sylvester graph.

```
gap> Sylvester_3:=DistanceGraph(Sylvester,3);;
gap> GlobalParameters(Sylvester_3);
[ [ 0, 0, 10 ], [ 1, 4, 5 ], [ 2, 8, 0 ] ]
gap> IsIsomorphicGraph(Sylvester_3,HammingGraph(2,6));
true
gap> K:=CompleteSubgraphsOfGivenSize(Sylvester_3,6,2);
[ [ 1, 10, 13, 16, 22, 23 ] ]
gap> K:=Union(Orbits(Sylvester_3.group,K,OnSets));;
gap> Length(K);
12
gap> #
gap> # Now  K  is the set of cliques of size  6  of  Sylvester_3,
gap> # which is isomorphic to the Hamming graph  H(2,6).
gap> #
gap> complement_3:=ComplementGraph(Sylvester_3);;
gap> L:=CompleteSubgraphsOfGivenSize(complement_3,6,2);
[ [ 1, 2, 4, 8, 9, 35 ], [ 1, 2, 4, 8, 19, 28 ],
  [ 1, 2, 4, 9, 11, 30 ], [ 1, 2, 4, 11, 19, 36 ],
  [ 1, 2, 5, 19, 24, 36 ] ]
gap> L:=Union(Orbits(complement_3.group,L,OnSets));;
gap> Length(L);
720
gap> #
gap> # Now  L  is the set of independent sets of size  6  in Sylvester_3.
gap> #
gap> blocks:=Union(K,L);;
gap> BD:=BlockDesign(36,blocks);;
```

```
gap> #
gap> # We are here interested in those Sylvester designs that are
gap> # subdesigns of  BD.
gap> #
gap> edges:=UndirectedEdges(Sylvester);;
gap> nonedges:=UndirectedEdges(ComplementGraph(Sylvester));;
gap> Sdesigns := BlockDesigns(rec(v:=36, blockSizes:=[6],
>    tSubsetStructure:=
>       rec(t:=2, partition:=[edges,nonedges],lambdas:=[2,1]),
>    r:=8, # replication number
>    blockDesign:=BD, # to get subdesigns of  BD
>    isoGroup:=AutomorphismGroup(Sylvester)));;
gap> Length(Sdesigns);
4
gap> #
gap> # The elements of  Sdesigns  are (up to the action of the
gap> # automorphism group of the Sylvester graph, and so up to
gap> # isomorphism) the Sylvester designs whose blocks come from
gap> # the rows, columns, and transversals of the 6x6 array
gap> # defined by the distance-3 graph of the Sylvester graph.
gap> #
gap> List(Sdesigns,design->Size(AutomorphismGroup(design)));
[ 144, 16, 72, 1440 ]
```

Challenge problem: Classify *all* the Sylvester designs.

3.7 Partitioning block designs

The DESIGN package function `PartitionsIntoBlockDesigns` constructs partitions of (the block multiset of) a given block design D, such that the subdesigns of D whose block multisets are the parts of this partition each have the same user-specified properties. See `PartitionsIntoBlockDesigns` in the DESIGN manual for full details.

For example, given a block design D having v points and each of whose blocks has size k, we can classify the resolutions of D by classifying the partitions of D into 1-$(v, k, 1)$ designs, up to the action of the automorphism group of D. We now do this for the Sylvester designs constucted above.

```
gap> L:=List(Sdesigns, design->
>    PartitionsIntoBlockDesigns(rec(v:=36, blockSizes:=[6],
>       blockDesign:=design,
>       tSubsetStructure:=rec(t:=1,lambdas:=[1]) ) ) );;
gap> List(L,Length);
[ 1, 0, 1, 1 ]
gap> # This gives the respective numbers of resolutions of the designs
```

```
gap> # in  Sdesigns,  up to the actions of the respective automorphism
gap> # groups of these designs.
```

4 Study of the Cohen-Tits near octagon

There is a unique distance-regular graph with intersection array $\{10, 8, 8, 2; 1, 1, 4, 5\}$, that is, with global parameters

```
[[0,0,10], [1,1,8], [1,1,8], [4,4,2], [5,5,0]].
```

The partial linear space CT with parameters $(2, 4)$ having this graph as point graph is the **Cohen-Tits near octagon**. The lines of CT consist of all the 525 triangles (cliques of size 3) in this point graph. See [7, Sections 6.4 and 13.6].

In the extended example of this section, we show that CT does not have an ovoid (where an **ovoid** is a set of points such that every line lies on just one of these points), but CT does have a resolution, both of which appear to be new results. We start by constructing the point graph of CT on the conjugacy class of size 315 of involutions in the Hall-Janko sporadic simple group J_2.

```
gap> n:=315;
315
gap> L:=AllPrimitiveGroups(NrMovedPoints,100,Size,604800,IsSimple,true);
[ J_2 ]
gap> J2:=L[1];
J_2
gap> CC:=Filtered(ConjugacyClasses(J2),x->Size(x)=n);;
gap> Length(CC);
1
gap> CC:=CC[1];;
gap> Order(Representative(CC));
2
gap> pointgraph:=Graph(J2,AsSet(CC),OnPoints,
>    function(x,y) return x*y=y*x and x<>y; end,
>    true);;
gap> GlobalParameters(pointgraph);
[ [ 0, 0, 10 ], [ 1, 1, 8 ], [ 1, 1, 8 ], [ 4, 4, 2 ], [ 5, 5, 0 ] ]
gap> #
gap> # Now  pointgraph  is the point graph of the Cohen-Tits near octagon.
gap> #
gap> A:=AutomorphismGroup(pointgraph);;
gap> Size(A);
1209600
```

```
gap> pointgraph:=NewGroupGraph(A,pointgraph);;
gap> K:=CompleteSubgraphsOfGivenSize(pointgraph,3,2);
[ [ 1, 2, 5 ] ]
gap> CT:=BlockDesign(n,K,A);;
gap> #
gap> # Now  CT  is the Cohen-Tits near octagon as a block design
gap> # in DESIGN package format.
gap> #
gap> NrBlockDesignPoints(CT);
315
gap> BlockSizes(CT);
[ 3 ]
gap> AllTDesignLambdas(CT);
[ 525, 5 ]
gap> dualCT:=DualBlockDesign(CT);;
gap> NrBlockDesignPoints(dualCT);
525
gap> BlockSizes(dualCT);
[ 5 ]
gap> AllTDesignLambdas(dualCT);
[ 315, 3 ]
gap> #
gap> # Does  dualCT  have a spread (parallel class)?
gap> #
gap> spreadsofdual:=BlockDesigns(rec(v:=NrBlockDesignPoints(dualCT),
>    blockSizes:=BlockSizes(dualCT),
>    blockDesign:=dualCT,
>    tSubsetStructure:=rec(t:=1, lambdas:=[1])));
[  ]
gap> #
gap> # So the dual of  CT  has no  1-(525,5,1)  subdesign,
gap> # i.e. no spread. Equivalently,  CT  has no ovoid.
gap> #
gap> # What about possible spreads and resolutions of  CT?
gap> #
gap> # We shall first consider the spreads of  CT  invariant under
gap> # a well-chosen subgroup  S  of its automorphism group  A.
gap> #
gap> CC:=Set(ConjugacyClasses(A),x->Group(Representative(x)));;
gap> S:=Filtered(CC,x->Size(x)=5 and Size(Centralizer(A,x))=300);;
gap> Length(S);
1
gap> S:=S[1];
<permutation group of size 5 with 1 generator>
gap> #
gap> # We now classify the  S-invariant  spreads of  CT,  up to the
gap> # action of the normalizer in  A  of  S.
gap> #
gap> spreads:=BlockDesigns(rec(v:=NrBlockDesignPoints(CT),
```

```
>    blockSizes:=BlockSizes(CT),
>    blockDesign:=CT,
>    requiredAutSubgroup:=S,
>    isoGroup:=Normalizer(A,S),
>    tSubsetStructure:=rec(t:=1, lambdas:=[1])));;
gap> Collected(List(spreads,AllTDesignLambdas));  # a check
[ [ [ 105, 1 ], 39 ] ]
gap> #
gap> # Now make each spread into the set of the indices of its blocks
gap> # in  CT.
gap> #
gap> spreads:=List(spreads,
>    spread->List(spread.blocks,B->PositionSorted(CT.blocks,B)));;
gap> #
gap> # We now make a graph whose vertices correspond to the spreads
gap> # invariant under a conjugate in  A  of  S,  with two such
gap> # spreads joined by an edge iff they are disjoint.
gap> #
gap> spreadsgraph:=Graph(Image(ActionHomomorphism(A,CT.blocks,OnSets)),
>    spreads, OnSets,
>    function(x,y) return Intersection(x,y)=[]; end);;
gap> OrderGraph(spreadsgraph);
2645160
gap> VertexDegrees(spreadsgraph);
[ 51, 53, 158, 163, 166, 167, 173, 176, 179, 188, 196, 201, 207, 208,
  210, 212, 223, 226, 248, 252, 269, 312, 327, 357, 383, 395, 438,
  832, 900, 1369, 1423, 1432, 1561, 1879, 4948 ]
gap> Size(spreadsgraph.group);
1209600
gap> #
gap> # Now a clique of size  5  in spreadsgraph would yield a resolution
gap> # of  CT.
gap> #
gap> CompleteSubgraphsOfGivenSize(spreadsgraph,5,0,true);
[ [ 309961, 458110, 501557, 828391, 1047808 ] ]
gap> #
gap> # Thus, the Cohen-Tits near octagon  CT  has a resolution.
gap> #
```

5 The FinInG Package

FinInG [3] is a large and powerful GAP package for finite incidence geometry. It requires the GAP packages GAPDoc, Forms, cvec, Orb, GenSS, and GRAPE, and can also make use of DESIGN.

A (finite) **incidence structure** consists of a finite set of **elements**, a reflexive symmetric **incidence** relation on the set of elements, and a function,

called a **type** function, from the set of elements to a finite set of **types**, such that no two distinct elements having the same type are incident. An **incidence geometry** is an incidence structure such that every maximal set of pairwise incident elements contains an element of each type.

FinInG provides efficient functionality for constructing and studying the following incidence structures:

- projective and affine spaces
- classical polar spaces
- generalised polygons
- coset incidence structures.

FinInG also provides functionality for:

- groups of automorphisms of incidence structures
- specialised actions, orbits, and stabilizers
- morphisms of incidence geometries
- algebraic varieties over finite fields.

See [4] for a good overview of the FinInG package. The extensive FinInG manual includes much more detailed information, together with many useful examples. In this article, however, we look only into some FinInG functionality for projective spaces and coset incidence structures, concentrating on applications by the author.

5.1 The projective space $\mathbf{PG}(d, q)$

Let V be a $(d+1)$-dimensional vector space over the finite field $\mathrm{GF}(q)$.

In FinInG, the **projective space** $\mathrm{PG}(d, q)$ is the incidence geometry whose elements are the proper non-zero subspaces of V, with two elements being incident exactly when one is contained in the other, and having the same type if and only if they have the same dimension.

The **points** and **lines** of $\mathrm{PG}(d, q)$ are respectively the 1- and 2-dimensional subspaces of V. A **partial spread** (of lines) in a projective space is a set of lines whose pairwise intersection is the zero-subspace (projectively, the **empty subspace**). A **maximal** partial spread is one which is not contained in any larger partial spread of the projective space.

We now use FinInG and GRAPE to classify the maximal partial spreads of lines in the projective space $\mathrm{PG}(3,4)$, first, up to the action of the projective general linear group $\mathrm{PGL}(4,4)$ (which was done much more slowly in [29]), and then, up to the action of the full collineation group $\mathrm{P\Gamma L}(4,4)$ of the projective space.

```
gap> LoadPackage("fining",false);
true
gap> d:=3;;
gap> q:=4;;
gap> pg:=PG(d,q);
ProjectiveSpace(3, 4)
gap> IsIncidenceStructure(pg);
true
gap> TypesOfElementsOfIncidenceStructure(pg);
[ "point", "line", "plane" ]
gap> lines:=Lines(pg);
<lines of ProjectiveSpace(3, 4)>
gap> Size(lines);
357
gap> points:=Points(pg);
<points of ProjectiveSpace(3, 4)>
gap> Size(points);
85
gap> lineset:=Set(List(lines));;
gap> G:=ProjectivityGroup(pg);
The FinInG projectivity group PGL(4,4)
gap> Size(G);
987033600
gap> #
gap> # Now construct the graph  gamma,  whose vertices are labelled
gap> # by the lines of  PG(d,q),  with two vertices joined by an edge
gap> # iff the corresponding lines have no point in common.
gap> #
gap> # The cliques of  gamma  give the partial spreads of  PG(d,q),
gap> # with the maximal cliques giving the maximal partial spreads.
gap> #
gap> gamma:=Graph(G, lineset, OnProjSubspaces,
>    function(x,y) return Meet(x,y)=EmptySubspace(pg); end,
>    true);;
gap> GlobalParameters(gamma);
[ [ 0, 0, 256 ], [ 1, 180, 75 ], [ 192, 64, 0 ] ]
gap> Size(gamma.group);
987033600
gap> maximalpartialspreads:=CompleteSubgraphs(gamma,-1,2);;
gap> # The maximal complete subgraphs of  gamma,  up to the action of
gap> # gamma.group.  See the GRAPE documentation on  CompleteSubgraphs.
gap> #
```

```
gap> L:=List(maximalpartialspreads,Length);
[ 13, 14, 11, 13, 11, 12, 11, 12, 13, 13, 14, 14, 14, 13, 11, 13, 11,
  11, 13, 12, 17, 13, 13, 13, 13, 17, 13, 13, 17 ]
gap> Collected(L);
[ [ 11, 6 ], [ 12, 3 ], [ 13, 13 ], [ 14, 4 ], [ 17, 3 ] ]
gap> #
gap> # Repeat the classification with  G  now being the collineation
gap> # group of  pg.
gap> #
gap> G:=CollineationGroup(pg);
The FinInG collineation group PGammaL(4,4)
gap> Size(G);
1974067200
gap> hom:=ActionHomomorphism(G,VertexNames(gamma),OnProjSubspaces);
<action homomorphism>
gap> gamma:=NewGroupGraph(Image(hom),gamma);;
gap> Size(gamma.group);
1974067200
gap> Size(AutomorphismGroup(gamma));
3948134400
gap> maximalpartialspreads:=CompleteSubgraphs(gamma,-1,2);;
gap> L:=List(maximalpartialspreads,Length);
[ 12, 13, 13, 13, 11, 17, 14, 14, 11, 12, 14, 11, 11, 13, 13, 13, 13,
  17, 13, 17 ]
gap> Collected(L);
[ [ 11, 4 ], [ 12, 2 ], [ 13, 8 ], [ 14, 3 ], [ 17, 3 ] ]
```

More sophisticated computations of maximal partial spreads using GRAPE are described in [33], where to illustrate certain theory, the maximal partial spreads in $\mathrm{PG}(3,q)$ that are invariant under a group of order 5 are classified, for $q \in \{7,8\}$.

5.2 Flag-transitive geometries and coset incidence structures

Let Γ be an incidence structure.

A **flag** of Γ is a set of pairwise incident elements, and the **type** of a flag is the set of the types of its elements. A **chamber** of Γ is a flag containing an element of each type.

An **automorphism** of Γ is a permutation of the elements of Γ preserving the type of each element and preserving incidence between elements.

The incidence structure Γ is a **flag-transitive geometry** for a group G of automorphisms of Γ if Γ is an incidence geometry and G acts transitively on the set of chambers of Γ (and so G acts transitively on the flags of Γ of any

given type).

Suppose now Γ is a flag-transitive geometry for a group G of automorphisms. Consider a chamber $\{c_1, c_2, \ldots, c_n\}$ of Γ, such that c_i has type i, and let G_i be the stabilizer in G of c_i. Now G acts transitively on the elements of type i, so these elements are in 1-to-1 correspondence with the right cosets of G_i in G. Furthermore, two elements of Γ are incident if and only if the corresponding cosets have a nonempty intersection.

Now let G be any finite group and let $G_1 \ldots, G_n$ be distinct subgroups of G. Then these **parabolic subgroups** determine an incidence structure called a **coset incidence structure** for G, with type set $\{1, \ldots, n\}$, the elements of type i being the right cosets of G_i in G, and with two elements being incident precisely when they have nonempty intersection.

A coset incidence structure need not be an incidence geometry. However, in FinInG, if the function `IsFlagTransitiveGeometry` applied to a coset incidence structure returns `true`, this guarantees that the argument is a (flag-transitive) incidence geometry.

5.3 A flag-transitive geometry for J_2 and J_2:2

We now use GRAPE and FinInG to construct and study a certain coset incidence structure which is a flag-transitive geometry for the Hall-Janko sporadic simple group J_2 and its automorphism group J_2:2. The elements of this geometry correspond to certain induced subgraphs in a vertex- and edge-transitive graph for J_2, having 280 vertices and vertex-degree 36. As far as the author is aware, this geometry has never before been published.

```
gap> n:=280;
280
gap> L:=AllPrimitiveGroups(NrMovedPoints,n,Size,604800,IsSimple,true);
[ J_2 ]
gap> J2:=L[1];
J_2
gap> H:=Stabilizer(J2,1);
<permutation group of size 2160 with 4 generators>
gap> orbs:=List(Orbits(H,[1..n]),Set);;
gap> List(orbs,Length);
[ 1, 108, 36, 135 ]
gap> orb36:=First(orbs,orb->Length(orb)=36);;
gap> edge:=[1,orb36[1]];
[ 1, 3 ]
gap> gamma:=EdgeOrbitsGraph(J2,[edge]);;
gap> GlobalParameters(gamma);
[ [ 0, 0, 36 ], [ 1, 8, 27 ], [ 4, 32, 0 ] ]
```

```
gap> AssignVertexNames(gamma,[1..n]);
gap> f:=First(orbs,orb->Length(orb)=135)[1];
4
gap> eps:=GeodesicsGraph(gamma,1,f);;
gap> # So  eps  is the induced subgraph on the vertices of the
gap> # geodesics from 1 to f, but not including 1 and f.
gap> #
gap> GlobalParameters(eps);
[ [ 0, 0, 2 ], [ 1, 0, 1 ], [ 2, 0, 0 ] ]
gap> # So eps is a 4-gon.
gap> #
gap> oct_vertices:=Union([1,f],VertexNames(eps));
[ 1, 3, 4, 118, 223, 265 ]
gap> # So the induced subgraph on  oct_vertices  is (the 1-skeleton of)
gap> # an octahedron.
gap> #
gap> oct_stab:=Stabilizer(J2,oct_vertices,OnSets);;
gap> Size(oct_stab);
96
gap> oct_graph:=InducedSubgraph(gamma,oct_vertices,oct_stab);;
gap> GlobalParameters(oct_graph);
[ [ 0, 0, 4 ], [ 1, 2, 1 ], [ 4, 0, 0 ] ]
gap> Size(oct_graph.group);
48
gap> #
gap> #  oct_graph  is a chosen special induced subgraph of  gamma,
gap> # and is isomorphic to (the 1-skeleton of) an octahedron.
gap> #
gap> K:=CompleteSubgraphsOfGivenSize(oct_graph,3,2);
[ [ 1, 2, 4 ] ]
gap> clique:=Set(K[1],x->VertexName(oct_graph,x));
[ 1, 3, 118 ]
gap> # Now  clique  is the set of vertices in  gamma  of a triangle
gap> # in  oct_graph.
gap> #
gap> parabolics:=[];
[  ]
gap> for i in [1..3] do
>    parabolics[i]:=Stabilizer(J2,clique{[1..i]},OnSets);
> od;
gap> Add(parabolics,oct_stab);
gap> List(parabolics,x->Size(J2)/Size(x));
[ 280, 5040, 10080, 6300 ]
gap> J2geom:=CosetGeometry(J2,parabolics);
CosetGeometry( J_2 )
gap> #
gap> # In FinInG,  CosetGeometry  makes a coset incidence structure
gap> # (which is not necessarily an incidence geometry).
gap> #
```

```
gap> IsFlagTransitiveGeometry(J2geom);
true
gap> #
gap> # So  J2geom  is a flag-transitive geometry for the group  J2.
gap> #
gap> # We now use FinInG to determine some properties of  J2geom.
gap> #
gap> TypesOfElementsOfIncidenceStructure(J2geom);
[ 1 .. 4 ]
gap> IsConnected(J2geom);
true
gap> IsResiduallyConnected(J2geom);
true
gap> IsThinGeometry(J2geom);
false
gap> IsThickGeometry(J2geom);
false
gap> IsFirmGeometry(J2geom);
true
gap> Size(BorelSubgroup(J2geom));
2
gap> autgrp:=AutGroupIncidenceStructureWithNauty(J2geom);;
gap> #  the automorphism group of  J2geom
gap> #
gap> DisplayCompositionSeries(autgrp);
G (6 gens, size 1209600)
 | C2
S (3 gens, size 604800)
 | J2
1 (0 gens, size 1)
gap> #
gap> # FinInG can draw the "diagram" of a flag-transitive
gap> # coset geometry, making use of the GraphViz
gap> # software (available from https://graphviz.org).
gap> #
gap> DrawDiagram(DiagramOfGeometry(J2geom),"J2geomdiagram");
```

The diagram of the constructed flag-transitive geometry, as described in the FinInG manual and produced by the code above, is shown in Figure 1.

```
gap> incidencegraph:=IncidenceGraph(J2geom);;
gap> #
gap> # Now  incidencegraph  is the incidence graph of  J2geom,
gap> # in GRAPE format.
gap> #
gap> IsGraph(incidencegraph);
true
gap> OrderGraph(incidencegraph);  # number of vertices
```

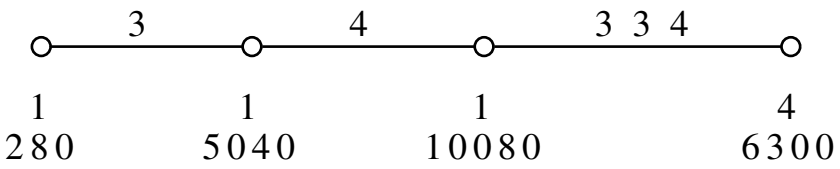

Figure 1: Diagram of the constructed flag-transitive geometry for J_2

```
21700
gap> Size(incidencegraph.group);
604800
gap> IsLoopy(incidencegraph);
true
```

We next read in a file downloadable from [32], which provides GAP functions for the computation of fundamental groups, certain quotients of fundamental groups, and covers of finite simplicial complexes, making use of the theory and algorithms of [27] and the GRAPE package. We then compute the fundamental group of the clique complex of the incidence graph of the flag-transitive geometry for J_2 we have constructed. This clique complex is the so-called flag complex of the geometry.

```
gap> Read("fundamental_v2.g");
gap> # This loads functions for fundamental groups and covers of
gap> # finite simplicial complexes.
gap> #
gap> # We now remove the loops on the incidence graph of  J2geom
gap> # to make it compatible with the fundamental group software.
gap> #
gap> for rep in incidencegraph.representatives do
>    RemoveEdgeOrbit(incidencegraph,[rep,rep]);
> od;
gap> IsSimpleGraph(incidencegraph);
true
gap> VertexDegrees(incidencegraph);
[ 11, 23, 26, 279 ]
gap> F:=FundamentalRecordSimplicialComplex(incidencegraph);;
#I  now the presentation has 1 generators, the new generator is _x1
#I  now the presentation has 2 generators, the new generator is _x2
gap> G:=F.group;
<fp group on the generators [ _x1, _x2 ]>
gap> # Now  G  is the fundamental group of the clique complex
gap> # of  incidencegraph.
```

```
gap> Size(G);
12
gap> IsAbelian(G);
false
gap> StructureDescription(G);
"C3 : C4"
```

6 Other software for graphs, designs, and finite geometries

In this article we have focussed on three GAP packages, but the reader should be aware of other software available for research in graphs, designs, and finite geometries. We now briefly discuss some of this software. Much more detailed information is available in the references cited.

While GRAPE is designed for the construction and study of usually simple and sometimes very large graphs related to groups, designs, and finite geometries, the Digraphs package [12] for GAP focuses on directed graphs (digraphs), and provides a very extensive range of constructions and graph-theoretic functionality for simple graphs, digraphs, and (at present) multidigraphs. In particular, the Digraphs package can input digraphs in GRAPE and other formats, and includes many functions analogous to those in GRAPE, as well as providing efficient functionality for digraph homomorphisms, digraph drawing, I/O facilities and efficient storage for large collections of digraphs, and more. The Digraphs package also includes an improved version of the bliss software [20] for computing automorphism groups of digraphs and testing digraph isomorphism. Some aspects of the Digraphs package were covered in the author's G2G2 lectures [36].

The GAP package AGT [13] for algebraic graph theory provides functionality for the computation of spectral properties, various bounds, and regularity properties for simple graphs given in GRAPE format. The package also provides an extensive library of strongly regular graphs and lists of "feasible" strongly regular graph parameters.

For computations with permutation groups and coherent configurations, the reader should consider the stand-alone COCO system [14], or one of its more modern descendents under development (see [21]). Andries Brouwer has made a Unix port of COCO, downloadable from [6].

Finally, Magma [5] is a large, powerful, comprehensive system for computations in algebra, number theory, algebraic geometry, and algebraic combinatorics. Magma's functionality has large overlaps with GAP and its packages.

However, Magma is not open source, and, with certain exceptions, is not free of charge.

References

[1] R.A. Bailey, P.J. Cameron, L.H. Soicher, and E.R. Williams, Substitutes for the non-existent square lattice designs for 36 varieties, *Journal of Agricultural, Biological and Environmental Statistics* **25** (2020), 487–499. Available online (open-access) at `https://doi.org/10.1007/s13253-020-00388-1`

[2] S. Ball, *Finite Geometry and Combinatorial Applications*, Cambridge University Press, Cambridge, 2015.

[3] J. Bamberg, A. Betten, Ph. Cara, J. De Beule, M. Lavrauw, and M. Neunhoeffer, The FinInG package for GAP, Version 1.5, 2022, `https://gap-packages.github.io/FinInG`

[4] J. Bamberg, A. Betten, Ph. Cara, J. De Beule, M. Neunhöffer, and M. Lavrauw, FinInG: a package for Finite Incidence Geometry, `https://arxiv.org/abs/1606.05530`, 2016.

[5] W. Bosma, J. Cannon, and C. Playoust, The Magma algebra system. I: The user language, *Journal of Symbolic Computation* **24** (1997), 235–265.

[6] A.E. Brouwer, Unix port of the COCO computer algebra system, `https://www.win.tue.nl/~aeb/ftpdocs/math/coco/coco-1.2b.tar.gz`

[7] A.E. Brouwer, A.M. Cohen, and A. Neumaier, *Distance-Regular Graphs*, Springer-Verlag, Berlin, 1989.

[8] A.E. Brouwer and W.H. Haemers, *Spectra of Graphs*, Springer, New York, 2012.

[9] P.J. Cameron, *Permutation Groups*, Cambridge University Press, Cambridge, 1999.

[10] P.J. Cameron and J.H. van Lint, *Designs, Graphs, Codes and their Links*, Cambridge University Press, Cambridge, 1991.

[11] E.R. van Dam, J.H. Koolen, and H. Tanaka, Distance-regular graphs, *Electronic Journal of Combinatorics* (2016), DS22.

[12] J. De Beule, J. Jonusas, J. Mitchell, M. Torpey, M. Tsalakou, and W.A. Wilson, The Digraphs package for GAP, Version 1.5.0, 2021, https://digraphs.github.io/Digraphs

[13] R.J. Evans, The AGT package for GAP, Version 0.2, 2020, https://gap-packages.github.io/agt

[14] I.A. Faradzev and M.H. Klin, Computer package for computations with coherent configurations, In: *ISSAC '91: Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation*, S.M. Watt (ed.), ACM Press, New York, 1991, pp. 219–223.

[15] The GAP Group, GAP — Groups, Algorithms, and Programming, Version 4.12.0, 2022, https://www.gap-system.org

[16] The GAP Group, GAP — A Tutorial, Release 4.12.0, 2022, https://www.gap-system.org/Manuals/doc/tut/manual.pdf

[17] C. Godsil and G. Royle, *Algebraic Graph Theory*, Springer-Verlag, New York, 2001.

[18] W. Haemers, A new partial geometry constructed from the Hoffman-Singleton graph, In: *Finite Geometries and Designs: Proceedings of the Second Isle of Thorns Conference 1980*, P.J. Cameron et al. (eds), LMS Lecture Note Series **49**, Cambridge University Press, Cambridge, 1981, pp. 119–127.

[19] C. Jefferson, E. Jonauskyte, M. Pfeiffer, and R. Waldecker, Minimal and canonical images, *Journal of Algebra* **521** (2019), 481–506.

[20] T. Junttila and P. Kaski, Engineering an efficient canonical labeling tool for large and sparse graphs, In: *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithmics and Combinatorics*, D. Applegate et al. (eds), SIAM, Philadelphia, 2007, pp. 135–149. bliss homepage: http://www.tcs.hut.fi/Software/bliss/

[21] M. Klin, M. Muzychuk, and S. Reichard, Proper Jordan schemes exist. First examples, computer search, patterns of reasoning. An essay, https://arxiv.org/abs/1911.06160, 2019.

[22] A. Konovalov, Programming with GAP, https://alex-konovalov.github.io/gap-lesson/

[23] R.M.R. Lewis, *A Guide to Graph Colouring: Algorithms and Applications*, Springer International Publishing, Switzerland, 2016.

[24] S. Linton, Finding the smallest image of a set, In: *ISSAC '04: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, J. Gutierrez (ed.), ACM Press, New York, 2004, pp. 229–234.

[25] B.D. McKay and A. Piperno, Practical graph isomorphism, II, *Journal of Symbolic Computation* **60** (2014), 94–112. nauty and Traces: `https://users.cecs.anu.edu.au/~bdm/nauty/`

[26] A. Pasini, *Diagram Geometries*, Oxford University Press, Oxford, 1994.

[27] S. Rees and L.H. Soicher, An algorithmic approach to fundamental groups and covers of combinatorial cell complexes, *Journal of Symbolic Computation* **29** (2000), 59–77.

[28] L.H. Soicher, Yet another distance-regular graph related to a Golay code, *Electronic Journal of Combinatorics* **2** (1995), N1.

[29] L.H. Soicher, Computation of partial spreads, 2004, `https://webspace.maths.qmul.ac.uk/l.h.soicher/partialspreads/`

[30] L.H. Soicher, Computing with graphs and groups, In: *Topics in Algebraic Graph Theory*, L.W. Beineke and R.J. Wilson (eds), Cambridge University Press, Cambridge, 2004, pp. 250–266.

[31] L.H. Soicher, Designs, groups and computing, In: *Probabilistic Group Theory, Combinatorics, and Computing: Lectures from the Fifth de Brún Workshop*, A. Detinko et al. (eds), Lecture Notes in Mathematics **2070**, Springer, London, 2013, pp. 83–107.

[32] L.H. Soicher, Functions for computing fundamental groups, certain quotients of fundamental groups, and covers of finite simplicial complexes, Version 2.0, 2015, `https://webspace.maths.qmul.ac.uk/l.h.soicher/fundamental/fundamental_v2.g`

[33] L.H. Soicher, On classifying objects with specified groups of automorphisms, friendly subgroups, and Sylow tower groups, *Portugaliae Mathematica* **74** (2017), 233–242.

[34] L.H. Soicher, The DESIGN package for GAP, Version 1.7, 2019, `https://gap-packages.github.io/design`

[35] L.H. Soicher, The GRAPE package for GAP, Version 4.8.5, 2021, https://gap-packages.github.io/grape

[36] L.H. Soicher, Using the GAP system and its packages for research in graph theory, design theory, and finite geometry, G2G2 minicourse lecture notes and exercises, 2021, https://webspace.maths.qmul.ac.uk/l.h.soicher/g2g2/