GAP 4 Package
FinInG

Finite Incidence Geometry

1.3

2015

John Bamberg, Anton Betten, Philippe Cara, Jan De Beule, Michel Lavrauw, Max Neunhoeffer.

Copyright

© 2015 by the authors

This package may be distributed under the terms and conditions of the GNU Public License Version 2 or higher.

The development group of FinInG welcomes contact with users. In case you have obtained the package as a deposited package part of archive during the installation of GAP, we call on your beneficence to register at http://cage.ugent.be/fining when you use FinInG.

Please tell us about the use of FinInG in your research or teaching. We are very interested in results obtained using FinInG and we might refer to your work in the future. If your work is published, we ask you to cite FinInG like a journal article or book. We provide the necessary BibTex and LaTeX code in Section 1.2.

Acknowledgements

The development phase of FinInG started roughly in 2005. The idea to write a package for projective geometry in GAP had emerged before, and resulted in pg, a relic that still can be found in the undeposited packages section of the GAP website. One of the main objectives was to develop the new package was to create a tighter connection between finite geometry and the group theoretical functions present in GAP.

The authors thank Michael Pauley, Maska Law and Sven Reichard, for their contributions during the early days of the project.

Jan De Beule and Michel Lavrauw have been supported by the Research Foundation Flanders – Belgium (FWO), and John Bamberg has been supported by a Marie Curie grant and an ARC grant during almost the whole development phase of FinInG. The authors are grateful for this support.

John Bamberg, Philippe Cara and Jan De Beule have spent several weeks in Vicenza while developing FinInG. We acknowledge the hospitality of Michel Lavrauw and Corrado Zanella. Our stays in Vicenza were always fruitful and very enjoyable. During or daily morning and afternoon coffee breaks, we discussed several topics, while enjoying coffee in *Caffe Pigaffeta*, http://www.caffepigafetta.it. Although one cannot acknowledge everybody, such as the bakery who provided us with the sandwiches, we acknowledge very much the hospitality of Carla and Luigi, who introduced us to many different kinds of coffees from all over the world.

Contents

Chapter 1

Introduction

1.1 Philosophy

FinInG (pronounciation: [fɪnɪŋ]) is a package for computation in Finite Incidence Geometry. It provides users with the basic tools to work in various areas of finite geometry from the realms of projective spaces to the flat lands of generalised polygons. The algebraic power of GAP is employed, particularly in its facility with matrix and permutation groups.

1.2 How to cite FinInG

The development group of FinInG welcomes contact with users. In case you have obtained the package as a deposited package part of archive during the installation of GAP, we call on your beneficence to register at http://cage.ugent.be/fining when you use FinInG.

Please tell us about the use of FinInG in your research or teaching. We are very interested in results obtained using FinInG and we might refer to your work in the future. If your work is published, we ask you to cite FinInG like a journal article or book.

If you are using BibTeX, you can use the following BibTeX entry for the current FinInG version:

```
─────── Example ───────
@manual{fining,
        Author = {Bamberg, John and Betten, Anton and Cara, Philippe and
            De Beule, Jan and Lavrauw, Michel and Neunh\"offer, Max },
        Key = {fining},
        Title = {{FinInG -- Finite Incidence Geometry, Version 1.3}},
        Url = {\verb+(http://cage.ugent.be/fining)+},
        Year = 2015}
```

Here is the bibliography entry produced by BibTeX (in bibliography style 'alpha'), to be pasted directly inside the bibliography environment of your LaTeX document:

```
─────── Example ───────
\bibitem[FinInG]{fining}
J.~Bamberg, A.~Betten, {Ph}. Cara, J.~De~Beule, M.~Lavrauw, and
  M.~Neunh\"offer.
\newblock {\em Finite Incidence Geometry}.
\newblock FInInG -- a {GAP} package, version 1.3, 2015.
```

When linking to FinInG from a web page you can use the link

```
─────────────────────────────── Example ───────────────────────────────
  <a href="http://cage.ugent.be/fining/">FinInG</a>.
```

1.3   Overview of this manual

This chapter (section 1.4) describes the installation of the package. Chapter 2 contains some extended examples to introduce the user to the basic functionality and philosophy to get started. Chapter 3 contains a rigorous discription of the basic structures. This chapter can be omitted at first reading, since the set of consequent chapters is also self contained. Chapters 4, 5 and 6 deal with projective spaces, projective semilinear groups and polarities of projective spaces, respectively. In Chapter 7 the functionality for classical polar spaces is treated and in Chapter 9 affine spaces and their groups are considered. *Geometry morphisms* between various geometries that are available in the package, are introduced and discussed in Chapter 10. The final three chapters, 11, 12, and 13 explain the basic functionality which is provided for algebraic varieties (in projective or affine spaces), generalised polygons, of which several models can be constructed, and finally coset geometries and diagrams.

1.4   Installing FinInG

This package requires the GAP packages GAPDoc, version 1.5.1 or higher, Forms, version 1.2.3 or higher, Orb, version 4.7.3 or higher, cvec, version 2.5.3 or higher, GenSS, version 1.6.2 or higher, and GRAPE, version 4.6.1 or higher. Currently, one function will use the Design package, but this package is not required to load FinInG. The package GenSS requires the package IO. The packages required by FinInG are all part of the standard GAP distribution but some of them have to be compiled. Make sure (at least) IO, orb and cvec have been compiled before trying to load FinInG. You can find instructions on how to compile packages for your operating system on the GAP webpage. Here we just show how to compile the required packages on a UNIX-like system. In case you don't have write access to the gap installation on your computer, you should ask your system administrator to do this for you. We assume your GAP installation resides in /opt/gap4r7

```
─────────────────────────────── Example ───────────────────────────────
  user@computer:~/$ cd /opt/gap4r7/pkg
  user@computer:/opt/gap4r7/pkg/$ cd orb
  user@computer:/opt/gap4r7/pkg/orb$ ./configure
  checking for a BSD-compatible install... /usr/bin/install -c
  ...
  ...lots of output...
  ...
  user@computer:/opt/gap4r7/pkg/orb$ make
  make[1]: Entering directory '/opt/gap4r7/pkg/orb'
  ...
  ...lots of output...
  ...
  user@computer:/opt/gap4r7/pkg/orb$ cd ../io
  user@computer:/opt/gap4r7/pkg/io$ ./configure
  checking for a BSD-compatible install... /usr/bin/install -c
  ...
  ...lots of output...
  ...
  user@computer:/opt/gap4r7/pkg/io$ make
```

```
make[1]: Entering directory '/opt/gap4r7/pkg/io'
...
...lots of output...
...
user@computer:/opt/gap4r7/pkg/io$ cd ../cvec
user@computer:/opt/gap4r7/pkg/cvec$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
...
...lots of output...
...
user@computer:/opt/gap4r7/pkg/cvec$ make
make[1]: Entering directory '/opt/gap4r7/pkg/cvec'
...
...lots of output...
...
user@computer:/opt/gap4r7/pkg/cvec$ cd ../grape/
user@computer:/opt/gap4r7/pkg/grape$ ./configure
user@computer:/opt/gap4r7/pkg/grape$ make
if test ! -d bin;  then mkdir bin;  chmod 755 bin;  fi
...
...lots of output...
...
```

The recent development stage of FinInG is based on GAP4r5. We have done testing using GAP4r7, and no installation differences occured. In this section, we describe in detail the installation procedure for FinInG, assuming the use of GAP4r7. We have also (succesfully) tested this procedure with some older GAP releases but these required manual installation of GenSS, IO and Orb. The installation of FinInG itself is generic for each UNIX like system, including the different flavours of MacOSX. You only need a terminal application, and you need acces to the standard unix tools gunzip and tar. The installation procedure for FinInG, a standard GAP package, does *not* require compilation. Each GAP installation has a pkg directory, containing supplemental GAP packages. If you have acces to this filesystem, you can locate it, e.g.

$$/usr/local/gap4r7/pkg/$$

Download the FinInG archive "fining-....tgz" to this location, and unpack the archive. This can be done by issuing

$$gunzip\,fining - ....tgz$$

which yields a file "fining-....tar", in the pkg directory, after which issuing the command

$$tar - xf\,fining - ....tar$$

unpacks the archive in a subdirectory fining. After successfully unpacking the archive, you can locate the directory

$$/usr/local/gap4r7/pkg/fining/.$$

This directory contains a subdirectory "./doc", containing an html and pdf version of the manual. The html version is accessible by opening the file "chap0.html" in your favorite browser. The pdf version of the manual can be found in the file "manual.pdf".

Please notice that you can unpack your archive in your favorite local "./pkg" directory, e.g. "/home/yourself/pkg/", in case you are using GAP on a server on which you have only a restricted

access. If you installed FinInG in the central GAP pkg directory, you can startup gap using the usual command. We suppose this is gap4r7. If you installed FinInG in your local pkg directory, e.g. "/home/yourself/pkg/", then move to "/home/yourself", and issue the command

$$gap - l"/usr/local/gap4r7;./"$$

This will cause gap to startup and use as pkg directory both its own central pkg directory, i.e. "/usr/local/gap4r7/pkg", as well as your local pkg directory, i.e. "/home/yourself/pkg/". You should see something like the following output. Notice that the packages GAPDoc and IO are loaded by default. This is not necessarily the case, but loading fining will force to load required packages anyway.

```
 ──────────────────────── Example ────────────────────────
  +-------+   GAP,   Version 4.7.8 of 09-Jun-2015 (free software, GPL)
  |  GAP  |   http://www.gap-system.org
  +-------+   Architecture: x86_64-unknown-linux-gnu-gcc-default64
  Libs used:  gmp
  Loading the library and packages ...
  Components: trans 1.0, prim 2.1, small* 1.0, id* 1.0
  Packages:   AClib 1.2, Alnuth 3.0.0, AtlasRep 1.5.0, AutPGrp 1.6,
              Browse 1.8.6, CRISP 1.3.8, Cryst 4.1.12, CrystCat 1.1.6,
              CTblLib 1.2.2, FactInt 1.5.3, FGA 1.2.0, GAPDoc 1.5.1, IO 4.4.4,
              IRREDSOL 1.2.4, LAGUNA 3.7.0, Polenta 1.3.2, Polycyclic 2.11,
              RadiRoot 2.7, ResClasses 3.4.0, Sophus 1.23, SpinSym 1.5,
              TomLib 1.2.5
  Try '?help' for help. See also  '?copyright' and  '?authors'
  gap>
```

To load FinInG, issue

$$LoadPackage("fining");$$

If this fails, in most cases, the reason is that either GAP does not find the directory in which FinInG is installed, or one of the required packages for FinInG is not installed or not compiled. Make sure (at least) IO, orb and cvec have been compiled before trying to load FinInG. The easiest way to find out what goes wrong is to load each required package before issuing the LoadPackage command to load FinInG. The example below shows this situation, the packages Forms and FinInG itself are installed in a local pkg directory, the other packages are installed centrally. Starting up gap not pointing to the local pkg directory, causes the locally installed packages to be unloadable.

```
 ──────────────────────── Example ────────────────────────
  gap> LoadPackage("fining");
  fail
  gap> LoadPackage("forms");
  fail
  gap> LoadPackage("grape");
  ----------------------------------------------------------------------------
  Loading  GRAPE 4.6.1 (GRaph Algorithms using PErmutation groups)
  by Leonard H. Soicher (http://www.maths.qmul.ac.uk/~leonard/).
  Homepage: http://www.maths.qmul.ac.uk/~leonard/grape/
  ----------------------------------------------------------------------------
  true
  gap> LoadPackage("orb");
  ----------------------------------------------------------------------------
  Loading  orb 4.7.3 (Methods to enumerate orbits)
  by Juergen Mueller (http://www.math.rwth-aachen.de/~Juergen.Mueller),
```

```
        Max Neunhöffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef), and
        Felix Noeske (http://www.math.rwth-aachen.de/~Felix.Noeske).
    Homepage: http://gap-system.github.io/orb/
    --------------------------------------------------------------------------------
    gap> LoadPackage("genss");
    --------------------------------------------------------------------------------
    Loading  genss 1.6.2 (Generic Schreier-Sims)
    by Max Neunhöffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef) and
        Felix Noeske (http://www.math.rwth-aachen.de/~Felix.Noeske).
    Homepage: http://gap-system.github.io/genss/
    --------------------------------------------------------------------------------
    true
```

If the installation was successful, then, the following output should be visible.

```
——————————————————————— Example ———————————————————————
gap> LoadPackage("fining");
-----------------------------------------------------------------------
Loading 'Forms' 1.2.3 (26/10/2015)
by John Bamberg (http://school.maths.uwa.edu.au/~bamberg/)
    Jan De Beule (http://www.debeule.eu)
For help, type: ?Forms
-----------------------------------------------------------------------
-----------------------------------------------------------------------
Loading  orb 4.7.3 (Methods to enumerate orbits)
by Juergen Mueller (http://www.math.rwth-aachen.de/~Juergen.Mueller),
    Max Neunhöffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef), and
    Felix Noeske (http://www.math.rwth-aachen.de/~Felix.Noeske).
Homepage: http://gap-system.github.io/orb/
-----------------------------------------------------------------------
-----------------------------------------------------------------------
Loading  cvec 2.5.3 (Compact vectors over finite fields)
by Max Neunhöffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef).
Homepage: http://gap-system.github.io/cvec/
-----------------------------------------------------------------------
-----------------------------------------------------------------------
Loading  genss 1.6.2 (Generic Schreier-Sims)
by Max Neunhöffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef) and
    Felix Noeske (http://www.math.rwth-aachen.de/~Felix.Noeske).
Homepage: http://gap-system.github.io/genss/
-----------------------------------------------------------------------
-----------------------------------------------------------------------
Loading  GRAPE 4.6.1 (GRaph Algorithms using PErmutation groups)
by Leonard H. Soicher (http://www.maths.qmul.ac.uk/~leonard/).
Homepage: http://www.maths.qmul.ac.uk/~leonard/grape/
-----------------------------------------------------------------------
loading: geometry, liegeometry, group, projectivespace, correlations, polarspace
/morphisms, enumerators, diagram, varieties, affinespace/affinegroup, gpolygons,
 orbits+stabilisers
-----------------------------------------------------------------------
        --------------     -------     --------    --------------
    ___ ____/__(_)_____ _/_____ ____/   __< /__|__ /
    __ /_   __ /__  __ \__ / __  __ \ / __    __ / ___/_ <
    _ __/   _ / _ / / /_/ /  _ / / / /_/ /    _ /_____/ /
```

```
      /_/       /_/  /_/ /_//___/  /_/ /_/\____/      /_/_(_)____/
----------------------------------------------------------------------------
Loading  FinInG 1.3 (Finite Incidence Geometry)
by John Bamberg (http://school.maths.uwa.edu.au/~bamberg/)
   Anton Betten (http://www.math.colostate.edu/~betten)
   Jan De Beule (http://www.debeule.eu)
   Philippe Cara (http://homepages.vub.ac.be/~pcara)
   Michel Lavrauw (http://cage.ugent.be/~ml)
   Max Neunhoeffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef/)
For help, type: ?FinInG
----------------------------------------------------------------
true
```

1.5 The Development Team

This is the development team (without Anton), meeting in St. Andrews in september 2008, from left to right: Philippe Cara, Michel Lavrauw, Max Neunhöffer, Jan De Beule and John Bamberg

The development team meeting again (without Anton and Max), now in Vicenza in april 2011. from left to right: Michel Lavrauw, John Bamberg, Philippe Cara, Jan De Beule.

Survivors of the first version of FinInG, enjoying a trip to Chioggia, december 2011.

The same survivors, staring at the destiny.

Anton Betten, during a milestone meeting at the finite geometries conference in Irsee, september 2014.

Chapter 2

Examples

In this chapter we provide some simple examples of the use of FinInG.

2.1 Elementary examples

2.1.1 subspaces of projective spaces

The following example shows how to create some subspaces of a projective space, test their incidence, and determine their span and intersection. Projective spaces are considered as incidence geometries too. Incidence, to be tested with `IsIncident` or equivalently `\*`, is symmetrized set-theoretic containment, the latter which can be tested through the operation `in`.

```
————————————————— Example —————————————————
gap> pg := PG(3,8);
ProjectiveSpace(3, 8)
gap> vec := [0,1,0,1]*Z(8)^0;
[ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ]
gap> point := VectorSpaceToElement(pg,vec);
<a point in ProjectiveSpace(3, 8)>
gap> mat := [[0,0,1,1],[0,1,0,0]]*Z(8)^0;
[ [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> line := VectorSpaceToElement(pg,mat);
<a line in ProjectiveSpace(3, 8)>
gap> mat2 := [[1,0,0,1],[1,0,1,0],[1,1,0,0]]*Z(8)^0;
[ [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> plane := VectorSpaceToElement(pg,mat2);
<a plane in ProjectiveSpace(3, 8)>
gap> IsIncident(point,line);
false
gap> IsIncident(line,point);
false
gap> point * line;
false
gap> line * point
> point in line;
Syntax error: ; expected
point in line;
   ^
```

```
gap> line in point;
false
gap> IsIncident(point,plane);
true
gap> IsIncident(line,plane);
false
gap> line in plane;
false
gap> plane2 := Span(point,line);
<a plane in ProjectiveSpace(3, 8)>
gap> Meet(plane,plane2);
<a line in ProjectiveSpace(3, 8)>
gap> mat3 := [[1,0,0,0],[0,0,0,1]]*Z(8)^0;
[ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> line2 := VectorSpaceToElement(pg,mat3);
<a line in ProjectiveSpace(3, 8)>
gap> Meet(line,line2);
< empty subspace >
gap> Span(plane,plane2);
ProjectiveSpace(3, 8)
```

2.1.2   Subspaces of classical polar spaces

FinInG provides classical polar spaces. Subspaces can be constructed the same way as subspaces of projective spaces. Upon construction, it is checked whether the given vector space does determine a subspace of the polar space. Subspaces of polar spaces are also subspaces of the ambient projective space. Operations like Span and Meet are naturally applicable. However, the span of two subspaces might not be a subspace of the polar space anymore, and if the two subspaces belong to two different polar spaces in the same ambient projective space, it cannot be determined in which polar space the span should be constructed. Therefore the result of Span of two subspaces of a polar space is a subspace of the ambient projective space. It can be checked whether the result belongs to a polar space using in. This illustrates very well a general philosophy: a subspace of a polar space, and more generally, an element of any incidence structure is always aware of its ambient geometry. This example also illustrates how to create an element that belongs to the polar space from the subspace of the ambient projective geometry by using ElementToElement. Finally note the behaviour of = applied on the two subspaces. Clearly, a subspace of a polar space is really also a subspace of the ambient projective space.

```
───────────────────────── Example ─────────────────────────
gap> ps := EllipticQuadric(5,7);
Q-(5, 7)
gap> vec := [1,0,0,0,0,0]*Z(7)^0;
[ Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7) ]
gap> point := VectorSpaceToElement(ps,vec);
Error, <v> does not generate an element of <geom> called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 10 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

```
gap> EquationForPolarSpace(ps);
x_1^2+x_2^2+x_3*x_4+x_5*x_6
gap> vec := [0,0,1,0,0,0]*Z(7)^0;
[ 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7) ]
gap> point := VectorSpaceToElement(ps,vec);
<a point in Q-(5, 7)>
gap> vec2 := [0,0,0,1,0,0]*Z(7)^0;
[ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7) ]
gap> point2 := VectorSpaceToElement(ps,vec2);
<a point in Q-(5, 7)>
gap> line := Span(point,point2);
<a line in ProjectiveSpace(5, 7)>
gap> mat := [[0,0,1,0,0,0],[0,0,0,0,1,0]]*Z(7)^0;
[ [ 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7) ] ]
gap> line2 := VectorSpaceToElement(ps,mat);
<a line in Q-(5, 7)>
gap> meet := Meet(line,line2);
<a point in ProjectiveSpace(5, 7)>
gap> meet in ps;
true
gap> point3 := ElementToElement(ps,meet);
<a point in Q-(5, 7)>
```

2.1.3 Underlying objects

Subspaces of projective spaces and polar spaces (and in general, elements of incidence structures), are determined by a mathematical object, called in FinInG the *underlying object*. The operation UnderlyingObject simply returns this underlying object. For elements determined by vectors or sub vector spaces, the underlying objects are a vector or a matrix. To represent these objects and to do very efficient orbit calculations under groups, we use the cvec. This can be noted when applying UnderlyingObject. The operation Unpack simply converts the cvec objects into GAP vectors and matrices. The example also illustrates how the underlying object of an element of an affine spaces looks like.

```
                                   Example
gap> pg := PG(3,169);
ProjectiveSpace(3, 169)
gap> p := Random(Points(pg));
<a point in ProjectiveSpace(3, 169)>
gap> UnderlyingObject(p);
<cvec over GF(13,2) of length 4>
gap> Unpack(last);
[ Z(13)^0, Z(13^2)^49, Z(13^2)^31, Z(13^2)^143 ]
gap> l := Random(Lines(pg));
<a line in ProjectiveSpace(3, 169)>
gap> UnderlyingObject(l);
<cmat 2x4 over GF(13,2)>
gap> Unpack(last);
[ [ Z(13)^0, 0*Z(13), 0*Z(13), Z(13^2)^96 ],
  [ 0*Z(13), Z(13)^0, Z(13^2)^113, Z(13^2)^99 ] ]
```

```
gap> quadric := EllipticQuadric(5,2);
Q-(5, 2)
gap> line := Random(Lines(quadric));
<a line in Q-(5, 2)>
gap> UnderlyingObject(line);
<cmat 2x6 over GF(2,1)>
gap> Unpack(last);
[ [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ] ]
gap> ag := AG(4,3);
AG(4, 3)
gap> plane := Random(Planes(ag));
<a plane in AG(4, 3)>
gap> UnderlyingObject(plane);
[ <cvec over GF(3,1) of length 4>, <cmat 2x4 over GF(3,1)> ]
```

2.1.4 Constructing polar spaces

FinInG provides the classical polar spaces as the geometries of which the subspaces are represented by the totally isotropic (resp. totally singular) vector subspaces with relation to a chosen sesquilinear (resp. quadratic form). The user may choose any non-degenerate (resp. non-singular) form to construct the polar space. The usage of the forms makes FinInG dependent on the package forms. Shortcuts to polar spaces in *standard* representation are included. Detailed information can be found in Section 7.2.

```
——————————— Example ———————————
gap> ps := HermitianPolarSpace(4,9);
H(4, 3^2)
gap> EquationForPolarSpace(ps);
x_1^4+x_2^4+x_3^4+x_4^4+x_5^4
gap> ps := HyperbolicQuadric(5,7);
Q+(5, 7)
gap> EquationForPolarSpace(ps);
x_1*x_2+x_3*x_4+x_5*x_6
gap> ps := SymplecticSpace(3,3);
W(3, 3)
gap> EquationForPolarSpace(ps);
x1*y2-x2*y1+x3*y4-x4*y3
gap> mat := IdentityMat(4,GF(11));
[ [ Z(11)^0, 0*Z(11), 0*Z(11), 0*Z(11) ],
  [ 0*Z(11), Z(11)^0, 0*Z(11), 0*Z(11) ],
  [ 0*Z(11), 0*Z(11), Z(11)^0, 0*Z(11) ],
  [ 0*Z(11), 0*Z(11), 0*Z(11), Z(11)^0 ] ]
gap> form := BilinearFormByMatrix(mat,GF(11));
< bilinear form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(3,GF(11)): x_1^2+x_2^2+x_3^2+x_4^2=0 >
gap> Rank(ps);
2
gap> ps;
Q+(3, 11): x_1^2+x_2^2+x_3^2+x_4^2=0
```

2.1.5 Some collineation groups

In principle, the full group of collineations of almost any incidence structure can be computed in FinInG. Mathematically, this is almost obvious for projective spaces and affine spaces. For classical polar spaces, the particular forms plays a role. The coordinate change capabilities of the package forms, together with the standard theory (see [KL90]), ensure that the full collineation group of a classical polar space can be relatively easily obtained. The computation of the full collineation group of particular incidence structures, such as generalised polygons, may rely on the computation of the automorphism group of an underlying incidence graph, which is done by using nauty through the package GRAPE. Note that the elements of a projective collineation group are semilinear maps, they consist of a matrix together with a field automorphism.

```
————— Example —————
gap> pg := PG(3,4);
ProjectiveSpace(3, 4)
gap> coll := CollineationGroup(pg);
The FinInG collineation group PGammaL(4,4)
gap> gens := GeneratorsOfGroup(coll);
[ < a collineation: <cmat 4x4 over GF(2,2)>, F^0>,
  < a collineation: <cmat 4x4 over GF(2,2)>, F^0>,
  < a collineation: <cmat 4x4 over GF(2,2)>, F^2> ]
gap> UnderlyingMatrix(gens[2]);
<cmat 4x4 over GF(2,2)>
gap> Unpack(last);
[ [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ]
gap> as := AffineSpace(3,4);
AG(3, 4)
gap> coll := CollineationGroup(as);
AGammaL(3,4)
gap> GeneratorsOfGroup(coll);
[ < a collineation: <cmat 4x4 over GF(2,2)>, F^0>,
  < a collineation: <cmat 4x4 over GF(2,2)>, F^0>,
  < a collineation: <cmat 4x4 over GF(2,2)>, F^0>,
  < a collineation: <cmat 4x4 over GF(2,2)>, F^2> ]
gap> gp := SplitCayleyHexagon(3);
H(3)
gap> coll:= CollineationGroup(gp);
#I  for Split Cayley Hexagon
#I  Computing nice monomorphism...
#I  Found permutation domain...
G_2(3)
gap> GeneratorsOfGroup(coll);
[ < a collineation: <cmat 7x7 over GF(3,1)>, F^0>,
  < a collineation: <cmat 7x7 over GF(3,1)>, F^0>,
  < a collineation: <cmat 7x7 over GF(3,1)>, F^0>,
  < a collineation: <cmat 7x7 over GF(3,1)>, F^0>,
  < a collineation: <cmat 7x7 over GF(3,1)>, F^0> ]
gap> egq := EGQByqClan(LinearqClan(3));
#I  Computed Kantor family. Now computing EGQ...
```

```
<EGQ of order [ 9, 3 ] and basepoint 0>
gap> coll := CollineationGroup(egq);
#I  Computing incidence graph of generalised polygon...
#I  Using elation of the collineation group...
<permutation group of size 26127360 with 6 generators>
```

2.2 Some objects with interesting combinatorial properties

The examples here are meant to give a flavour of how to explore geometrical objects from different point of views.

2.2.1 The Tits ovoid

In this example we consider the Tits ovoid in $PG(3,8)$. We explicitly check the intersection number of the Tits-ovoid with planes of the projective space, and compute its stabiliser group inside the homography group of $PG(3,8)$. The use of ;; after a command suppresses its output, which is particularly interesting if the output is a long list. The operation Collected is self-explanatory, and a very useful GAP command. The computed stabiliser is the Suzuki group $Sz(8)$, a finite simple group.

```
———————————————— Example ————————————————
gap> q := 8;
8
gap> pg := PG(3,q);
ProjectiveSpace(3, 8)
gap> f := GF(q);
GF(2^3)
gap> vecs := Union(List(f,x->List(f,y->[One(f),x*y+x^6+y^4,x,y])));;
gap> Add(vecs,[0,1,0,0]*Z(q)^0);
gap> ovoid := List(vecs,x->VectorSpaceToElement(pg,x));;
gap> numbers := List(Planes(pg),x->Number(ovoid,y->y in x));
[ 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 1, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 9, 1, 9, 9, 9, 9, 9, 1, 1, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 1, 9, 9, 9, 9, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1,
  9, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 9, 9, 9, 9, 1,
  9, 9, 1, 9, 9, 1, 9, 9, 9, 9, 1, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 1, 9, 9, 9, 1, 9, 9, 9, 9, 1, 9, 9, 9, 1, 9, 9,
  1, 9, 9, 9, 9, 9, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 1, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 1, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 1, 9, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 1, 9, 1, 9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 9, 1, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 9, 9, 9, 9,
  9, 1, 9, 9, 1, 9, 9, 1, 9, 9, 9, 9, 9, 1, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 1, 9, 9, 9, 9, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 9, 9, 1,
  9, 1, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 1, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 9, 9, 9, 1, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 1, 9, 9, 1, 9, 9, 9, 9, 9, 1, 9, 9, 9, 1, 9, 9, 9, 9, 9, 9,
```

```
      9, 9, 1, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 9, 9,
      9, 9, 9, 9, 9, 9, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 9, 9, 9, 9, 9, 9, 9,
      9, 1, 9, 9, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1, 9, 9, 9, 9, 9, 9, 9, 9, 9, 1,
      9, 1, 9, 9, 9, 9, 9, 9, 9, 1 ]
gap> Collected(numbers);
[ [ 1, 65 ], [ 9, 520 ] ]
gap> group := HomographyGroup(pg);
The FinInG projectivity group PGL(4,8)
gap> stab := FiningSetwiseStabiliser(group,ovoid);
#I  Computing adjusted stabilizer chain...
<projective collineation group with 5 generators>
gap> time;
55290
gap> IsSimple(stab);
true
gap> Order(stab);
29120
```

2.2.2 Lines meeting a hermitian curve

Here we see how the lines of a projective plane $PG(2, q^2)$ meet a hermitian curve. It is well known that every line meets in either 1 or $q + 1$ points. Note that the last comment takes a while to complete.

```
───────────────────────────── Example ──────────────────────────────
gap> h:=HermitianPolarSpace(2, 7^2);
H(2, 7^2)
gap> pg := AmbientSpace( h );
ProjectiveSpace(2, 49)
gap> lines := Lines( pg );
<lines of ProjectiveSpace(2, 49)>
gap> curve := AsList( Points( h ) );;
gap> Size(curve);
344
gap> Collected( List(lines, t -> Number(curve, c-> c in t)));
[ [ 1, 344 ], [ 8, 2107 ] ]
gap> time;
26412
```

2.2.3 The Patterson ovoid

In this example, we construct the unique ovoid of the parabolic quadric $Q(6,3)$, first discovered by Patterson, but for which was given a nice construction by E. E. Shult. We begin with the "sums of squares" quadratic form over $GF(3)$ and the associated polar space.

```
───────────────────────────── Example ──────────────────────────────
gap> id := IdentityMat(7, GF(3));;
gap> form := QuadraticFormByMatrix(id, GF(3));
< quadratic form >
gap> ps := PolarSpace( form );
<polar space in ProjectiveSpace(
6,GF(3)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2=0 >
```

The construction of the ovoid (a la Shult):

```
——————————————— Example ———————————————
gap> psl32 := PSL(3,2);
Group([ (4,6)(5,7), (1,2,4)(3,6,5) ])
gap> reps:=[[1,1,1,0,0,0,0], [-1,1,1,0,0,0,0],
> [1,-1,1,0,0,0,0], [1,1,-1,0,0,0,0]]*Z(3)^0;
[ [ Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ Z(3), Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ Z(3)^0, Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ Z(3)^0, Z(3)^0, Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ] ]
gap> ovoid := Union( List(reps, x-> Orbit(psl32, x, Permuted)) );;
gap> ovoid := List(ovoid, x -> VectorSpaceToElement(ps, x));;
```

We check that this is indeed an ovoid. The observant reader will notice *#I Computing collineation group of canonical polar space...* which is caused by the command AsList applied to the collection of elements `planes`. The use of AsList invokes the computation of all elements in `planes` as an orbit under the collineation group of the ambient polar space. The reader is invited to redo, in a new GAP session, the same example omitting the AsList command, just defining planes := Planes(ps);;. The result will be te same, but the computation of all elements will now be done using an enumerator, and will be slower.

```
——————————————— Example ———————————————
gap> planes := AsList(Planes( ps ));;
#I  Computing collineation group of canonical polar space...
gap> ForAll(planes, p -> Number(ovoid, x -> x * p) = 1);
true
```

The stabiliser is interesting since it yields the embedding of $Sp(6,2)$ in $PO(7,3)$. To efficiently compute the set-wise stabiliser, we refer to the induced permutation representation.

```
——————————————— Example ———————————————
gap> g := IsometryGroup( ps );
#I  Computing collineation group of canonical polar space...
<projective collineation group of size 9170703360 with 2 generators>
gap> stabovoid := FiningSetwiseStabiliser(g, ovoid);
#I  Computing adjusted stabilizer chain...
<projective collineation group with 13 generators>
gap> DisplayCompositionSeries(stabovoid);
G (size 1451520)
 | B(3,2) = O(7,2) ~ C(3,2) = S(6,2)
1 (size 1)
gap> OrbitLengths(stabovoid, ovoid);
[ 28 ]
gap> IsTransitive(stabovoid, ovoid);
true
```

2.2.4 A hyperoval

In this example, we consider a hyperoval of the projective plane $PG(2,4)$, that is, six points no three collinear. We will construct such a hyperoval by some basic explorations into particular properties of the projective plane $PG(2,4)$. The projective plane is initialised, its points are computed and listed; then a standard frame is constructed, of which we may assume that it is a subset of the hyperoval. Finally, the stabiliser group of the hyperoval is computed, and it is checked that this group is isomorphic with the symmetric group on six elements.

```
———————————— Example ————————————
gap> pg := ProjectiveSpace(2,4);
ProjectiveSpace(2, 4)
gap> points := Points(pg);
<points of ProjectiveSpace(2, 4)>
gap> pointslist := AsList(points);
[ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)> ]
gap> Display(pointslist[1]);
 . . 1
```

Now we may assume that our hyperoval contains the fundamental frame.

```
———————————— Example ————————————
gap> frame := [[1,0,0],[0,1,0],[0,0,1],[1,1,1]]*Z(2)^0;
[ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, Z(2)^0, Z(2)^0 ] ]
gap> frame := List(frame,x -> VectorSpaceToElement(pg,x));
[ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)> ]
```

Alternatively, we could use:

```
———————————— Example ————————————
gap> frame := StandardFrame( pg );
[ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
  <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)> ]
```

There are six secant lines to this frame ("four choose two"). So we put together these secant lines from the pairs of points of this frame.

```
———————————— Example ————————————
gap> pairs := Combinations(frame,2);
[ [ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)> ],
  [ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)> ],
  [ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)> ],
  [ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)> ],
  [ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)> ],
  [ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)> ] ]
gap> secants := List(pairs,p -> Span(p[1],p[2]));
[ <a line in ProjectiveSpace(2, 4)>, <a line in ProjectiveSpace(2, 4)>,
  <a line in ProjectiveSpace(2, 4)>, <a line in ProjectiveSpace(2, 4)>,
  <a line in ProjectiveSpace(2, 4)>, <a line in ProjectiveSpace(2, 4)> ]
```

By a counting argument, it is known that the frame of $PG(2,4)$ completes uniquely to a hyperoval.

```
────────── Example ──────────
 gap> leftover := Filtered(pointslist,t->not ForAny(secants,s->t in s));
 [ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)> ]
 gap> hyperoval := Union(frame,leftover);
 [ <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
   <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)>,
   <a point in ProjectiveSpace(2, 4)>, <a point in ProjectiveSpace(2, 4)> ]
```

This hyperoval has the symmetric group on six symbols as its stabiliser, which can easily be calculated:

```
────────── Example ──────────
 gap> g := CollineationGroup(pg);
 The FinInG collineation group PGammaL(3,4)
 gap> stab := Stabilizer(g,Set(hyperoval),OnSets);
 <projective collineation group of size 720>
 gap> StructureDescription(stab);
 "S6"
```

2.3 Geometry morphisms

A geometry morphism in FinInG is a map between (a subset of) the elements of one geometry to (a subset of) the elements of a second geometry, preserving the incidence. Geometry morphisms are not necessarily type preserving. This section is meant to illustrate, in a non exhaustive way the basis philisophy behind geometry morphisms in FinInG.

2.3.1 Isomorphic polar spaces

We've seen already that a polar space can be constructed from any non-degenerate sesquilinear or non-singular quadratic form. An isomorphism between polar spaces of the same type, can easily be obtained. This example illustrates `IsomorphismPolarSpaces`, which is in its basic use self-explanatory, and the use of the obtained map to compute images and pre-images of elements.

```
────────── Example ──────────
 gap> mat1 := IdentityMat(4,GF(16));
 [ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
   [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ]
 gap> form1 := HermitianFormByMatrix(mat1,GF(16));
 < hermitian form >
 gap> ps1 := PolarSpace(form1);
 <polar space in ProjectiveSpace(3,GF(2^4)): x_1^5+x_2^5+x_3^5+x_4^5=0 >
 gap> mat2 := [[0,1,0,0],[1,0,0,0],[0,0,0,1],[0,0,1,0]]*Z(16)^0;
 [ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
   [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ]
 gap> form2 := HermitianFormByMatrix(mat2,GF(16));
 < hermitian form >
 gap> ps2 := PolarSpace(form2);
 <polar space in ProjectiveSpace(
 3,GF(2^4)): x_1^4*x_2+x_1*x_2^4+x_3^4*x_4+x_3*x_4^4=0 >
 gap> map := IsomorphismPolarSpaces(ps1,ps2);
 #I  No intertwiner computed. One of the polar spaces must have a collineation group computed
 <geometry morphism from <Elements of H(3,
```

```
4^2): x_1^5+x_2^5+x_3^5+x_4^5=0> to <Elements of H(3,
4^2): x_1^4*x_2+x_1*x_2^4+x_3^4*x_4+x_3*x_4^4=0>>
gap> p := Random(Points(ps1));
<a point in H(3, 4^2): x_1^5+x_2^5+x_3^5+x_4^5=0>
gap> p^map;
<a point in H(3, 4^2): x_1^4*x_2+x_1*x_2^4+x_3^4*x_4+x_3*x_4^4=0>
gap> l := Random(Lines(ps2));
<a line in H(3, 4^2): x_1^4*x_2+x_1*x_2^4+x_3^4*x_4+x_3*x_4^4=0>
gap> PreImageElm(map,l);
<a line in H(3, 4^2): x_1^5+x_2^5+x_3^5+x_4^5=0>
```

2.3.2 Intertwiners

We reconsider the previous example. The observant reader might have noticed the message *#I No intertwiner computed....* Given a geometry morphism f from S to S', an intertwiner ϕ is a map from the automorphism group of S to the automorphism group of S', such that for every element p of S and every automorphism g of S, we have

$$f(p^g) = f(p)^{\phi(g)}.$$

─────────── Example ───────────
```
gap> mat1 := IdentityMat(4,GF(16));
[ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> form1 := HermitianFormByMatrix(mat1,GF(16));
< hermitian form >
gap> ps1 := PolarSpace(form1);
<polar space in ProjectiveSpace(3,GF(2^4)): x_1^5+x_2^5+x_3^5+x_4^5=0 >
gap> mat2 := [[0,1,0,0],[1,0,0,0],[0,0,0,1],[0,0,1,0]]*Z(16)^0;
[ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ]
gap> form2 := HermitianFormByMatrix(mat2,GF(16));
< hermitian form >
gap> ps2 := PolarSpace(form2);
<polar space in ProjectiveSpace(
3,GF(2^4)): x_1^4*x_2+x_1*x_2^4+x_3^4*x_4+x_3*x_4^4=0 >
gap> CollineationGroup(ps1);
#I  Computing collineation group of canonical polar space...
<projective collineation group of size 4073472000 with 3 generators>
gap> map := IsomorphismPolarSpaces(ps1,ps2);
<geometry morphism from <Elements of H(3,
4^2): x_1^5+x_2^5+x_3^5+x_4^5=0> to <Elements of H(3,
4^2): x_1^4*x_2+x_1*x_2^4+x_3^4*x_4+x_3*x_4^4=0>>
gap> phi := Intertwiner(map);
MappingByFunction( <projective collineation group of size 4073472000 with
3 generators>, <projective collineation group of size 4073472000 with
3 generators>, function( y ) ... end, function( x ) ... end )
gap> g := Random(CollineationGroup(ps1));
< a collineation: <cmat 4x4 over GF(2,4)>, F^4>
gap> h := g^phi;
< a collineation: <cmat 4x4 over GF(2,4)>, F^4>
gap> h in CollineationGroup(ps2);
```

```
#I  Computing collineation group of canonical polar space...
true
gap> h := Random(CollineationGroup(ps2));
< a collineation: <cmat 4x4 over GF(2,4)>, F^2>
gap> g := PreImageElm(phi,h);
< a collineation: <cmat 4x4 over GF(2,4)>, F^2>
gap> g in CollineationGroup(ps1);
true
```

2.3.3 Klein correspondence

The Klein correspondence is well known. The user may define its own hyperbolic quadric as range
for the geometry morphism in FinInG. Note that more is possible than illustrated in the elementary
example, see Section 10.3.

─────────────────── Example ───────────────────
```
gap> ps := HyperbolicQuadric(5,5);
Q+(5, 5)
gap> klein := KleinCorrespondence(ps);
<geometry morphism from <lines of ProjectiveSpace(3, 5)> to <points of Q+(5,
5)>>
gap> line1 := Random(Lines(PG(3,5)));
<a line in ProjectiveSpace(3, 5)>
gap> line2 := Random(Lines(PG(3,5)));
<a line in ProjectiveSpace(3, 5)>
gap> p := line1^klein;
<a point in Q+(5, 5)>
gap> q := line2^klein;
<a point in Q+(5, 5)>
gap> p in ps;
true
gap> q in ps;
true
gap> IsCollinear(ps,p,q);
false
gap> Meet(line1,line2);
< empty subspace >
```

2.3.4 Embedding in a subspace

A projective space can be embedded as a subspace in a higher dimensional projective space. A compa-
rable embedding is possible for polar spaces, clearly only when a given subspace intersects the polar
space of higher rank in a polar space of the same type as the polar space to be embedded.

─────────────────── Example ───────────────────
```
gap> pg2 := PG(2,5);
ProjectiveSpace(2, 5)
gap> pg3 := PG(3,5);
ProjectiveSpace(3, 5)
gap> hyp := VectorSpaceToElement(pg3,[[1,2,4,0],[0,3,2,0],[1,1,0,1]]*Z(5)^0);
<a plane in ProjectiveSpace(3, 5)>
```

```
gap> em := NaturalEmbeddingBySubspace( pg2, pg3, hyp );
<geometry morphism from <All elements of ProjectiveSpace(2,
5)> to <All elements of ProjectiveSpace(3, 5)>>
gap> l := Random(Lines(pg2));
<a line in ProjectiveSpace(2, 5)>
gap> l^em;
<a line in ProjectiveSpace(3, 5)>
gap> p := Random(Points(hyp));
<a point in ProjectiveSpace(3, 5)>
gap> PreImageElm(em,p);
<a point in ProjectiveSpace(2, 5)>
gap> mat := [[0,0,0,1],[0,0,1,0],[0,-1,0,0],[-1,0,0,0]]*Z(3);
[ [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3) ], [ 0*Z(3), 0*Z(3), Z(3), 0*Z(3) ],
  [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ], [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ] ]
gap> form := BilinearFormByMatrix(mat,GF(3));
< bilinear form >
gap> w3 := PolarSpace(form);
<polar space in ProjectiveSpace(3,GF(3)): -x1*y4-x2*y3+x3*y2+x4*y1=0 >
gap> w5 := SymplecticSpace(5, 3);
W(5, 3)
gap> pg := AmbientSpace( w5 );
ProjectiveSpace(5, 3)
gap> solid := VectorSpaceToElement(pg,[[1,0,0,0,0,0],[0,1,0,0,0,0],
> [0,0,1,0,0,0],[0,0,0,1,0,0]]*Z(3)^0);
<a solid in ProjectiveSpace(5, 3)>
gap> TypeOfSubspace(w5,solid);
"symplectic"
gap> em := NaturalEmbeddingBySubspace( w3, w5, solid );
<geometry morphism from <Elements of <polar space in ProjectiveSpace(
3,GF(3)): -x1*y4-x2*y3+x3*y2+x4*y1=0 >> to <Elements of W(5, 3)>>
gap> points := Points( w3 );
<points of W(3, 3): -x1*y4-x2*y3+x3*y2+x4*y1=0>
gap> points2 := ImagesSet(em, AsSet(points));;
#I  Computing collineation group of canonical polar space...
gap> ForAll(points2, x -> x in solid);
true
```

2.3.5 subgeometries

A projective space can be embedded as a subgeometry in a projective space of the same dimension but over a field extension. A polar space, determined by a form *f* can be embedded in a polar space considered over a field extension by interpreting the form *f* over this field extension. This is an interesting tool to construct geometrical objects in projective and polar spaces.

—————————————————— Example ——————————————————

```
gap> pgsub := PG(2,7);
ProjectiveSpace(2, 7)
gap> pg := PG(2,7^2);
ProjectiveSpace(2, 49)
gap> em := NaturalEmbeddingBySubfield(pgsub,pg);
<geometry morphism from <All elements of ProjectiveSpace(2,
7)> to <All elements of ProjectiveSpace(2, 49)>>
```

```
gap> baer := List(Points(pgsub),x->x^em);;
gap> numbers := Collected(List(Lines(pg),x->Number(baer,y->y in x)));
[ [ 1, 2394 ], [ 8, 57 ] ]
gap> mat := [[0,0,0,1],[0,0,-1,0],[0,1,0,0],[-1,0,0,0]]*Z(5)^0;
[ [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0 ], [ 0*Z(5), 0*Z(5), Z(5)^2, 0*Z(5) ],
  [ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5) ], [ Z(5)^2, 0*Z(5), 0*Z(5), 0*Z(5) ] ]
gap> form := BilinearFormByMatrix(mat,GF(5));
< bilinear form >
gap> symplecticspace := PolarSpace(form);
<polar space in ProjectiveSpace(3,GF(5)): x1*y4-x2*y3+x3*y2-x4*y1=0 >
gap> hermitianspace := HermitianPolarSpace(3,25);
H(3, 5^2)
gap> em := NaturalEmbeddingBySubfield(symplecticspace,hermitianspace);
#I  No intertwiner computed. <geom1> must have a collineation group computed
<geometry morphism from <Elements of <polar space in ProjectiveSpace(
3,GF(5)): x1*y4-x2*y3+x3*y2-x4*y1=0 >> to <Elements of H(3, 5^2)>>
gap> l := Random(Lines(symplecticspace));
<a line in W(3, 5): x1*y4-x2*y3+x3*y2-x4*y1=0>
gap> l^em;
<a line in H(3, 5^2)>
```

2.3.6 Embedding by field reduction

Field reduction is a power full tool to embedd low dimensional projective (and polar spaces) over a field *K* in to high dimensional spaces over a subfield of *K*. The mathematics behind field reduction is explained in sections 10.4.3 and 10.5.3. The examples here show the use of these embedings to construct a regular spread of a projective space and a so-called Hermitian spread of a hyperbolic quadric.

```
——————————————— Example ———————————————
gap> pg1 := PG(1,3^2);
ProjectiveSpace(1, 9)
gap> pg2 := PG(3,3);
ProjectiveSpace(3, 3)
gap> em := NaturalEmbeddingByFieldReduction(pg1,pg2);
<geometry morphism from <All elements of ProjectiveSpace(1,
9)> to <All elements of ProjectiveSpace(3, 3)>>
gap> spread := List(Points(pg1),x->x^em);
[ <a line in ProjectiveSpace(3, 3)>, <a line in ProjectiveSpace(3, 3)>,
  <a line in ProjectiveSpace(3, 3)>, <a line in ProjectiveSpace(3, 3)>,
  <a line in ProjectiveSpace(3, 3)>, <a line in ProjectiveSpace(3, 3)>,
  <a line in ProjectiveSpace(3, 3)>, <a line in ProjectiveSpace(3, 3)>,
  <a line in ProjectiveSpace(3, 3)>, <a line in ProjectiveSpace(3, 3)> ]
gap> Union(List(spread,x->List(Points(x))))=Set(Points(pg2));
true
gap> ps1 := HermitianPolarSpace(3,3^2);
H(3, 3^2)
gap> ps2 := HyperbolicQuadric(7,3);
Q+(7, 3)
gap> em := NaturalEmbeddingByFieldReduction(ps1,ps2);
#I  These polar spaces are suitable for field reduction
<geometry morphism from <Elements of H(3, 3^2)> to <Elements of Q+(7, 3)>>
```

```
gap> spread := List(Points(ps1),x->x^em);;
gap> Union(List(spread,x->List(Points(x))))=Set(Points(ps2));
true
```

2.4 Some geometrical objects

2.4.1 Spreads of $W(5,3)$

A spread of $W(5,q)$ is a set of $q^3 + 1$ planes which partition the points of $W(5,q)$. Here we enumerate all spreads of $W(5,3)$ which have a set-wise stabiliser of order a multiple of 13.

```
─────────────────────── Example ───────────────────────
gap> w := SymplecticSpace(5, 3);
W(5, 3)
gap> g := IsometryGroup(w);
PSp(6,3)
gap> syl := SylowSubgroup(g, 13);
<projective collineation group of size 13>
gap> planes := Planes( w );
<planes of W(5, 3)>
gap> points := Points( w );
<points of W(5, 3)>
gap> orbs := Orbits(syl, planes , OnProjSubspaces);;
gap> IsPartialSpread := x -> Number(points, p ->
>         ForAny(x, i-> p in i)) = Size(x)*13;;
gap> partialspreads := Filtered(orbs, IsPartialSpread);;
gap> Length(partialspreads);
8
gap> 13s := Filtered(partialspreads, i -> Size(i) = 13);;
gap> Length(13s);
6
gap> 13s[1];
[ <a plane in W(5, 3)>, <a plane in W(5, 3)>, <a plane in W(5, 3)>,
  <a plane in W(5, 3)>, <a plane in W(5, 3)>, <a plane in W(5, 3)>,
  <a plane in W(5, 3)>, <a plane in W(5, 3)>, <a plane in W(5, 3)>,
  <a plane in W(5, 3)>, <a plane in W(5, 3)>, <a plane in W(5, 3)>,
  <a plane in W(5, 3)> ]
gap> 26s := List(Combinations(13s,2), Union);;
gap> two := Union(Filtered(partialspreads, i -> Size(i) = 1));;
gap> good26s := Filtered(26s, x->IsPartialSpread(Union(x, two)));;
gap> spreads := List(good26s, x->Union(x, two));;
gap> Length(spreads);
5
```

2.4.2 Distance-6 spread of the split Cayley hexagon

A distance 6 spread of a split Cayley hexagon is a set of lines mutually at maximal distance in the incidence graph. It is well known that the lines of the hexagon contained in a hyperplane meeting the ambient polar space in an elliptic quadric, yield such a spread. This example also illustrates how an

element of a geometry *remembers* its ambient geometry, and how `ElementToElement` can be used to embed an element in another geometry, see 3.8.1.

```
——————————————————————— Example ———————————————————————
  gap> gh := SplitCayleyHexagon(3);
  H(3)
  gap> q6 := AmbientPolarSpace(gh);
  Q(6, 3): -x_1*x_5-x_2*x_6-x_3*x_7+x_4^2=0
  gap> hyp := First(Hyperplanes(PG(6,3)),x->TypeOfSubspace(q6,x)="elliptic");
  <a proj. 5-space in ProjectiveSpace(6, 3)>
  gap> q5 := EllipticQuadric(5,3);
  Q-(5, 3)
  gap> lines := AsList(Lines(q5));
  <closed orbit, 280 points>
  gap> em := NaturalEmbeddingBySubspace(q5,q6,hyp);
  <geometry morphism from <Elements of Q-(5, 3)> to <Elements of Q(6,
  3): -x_1*x_5-x_2*x_6-x_3*x_7+x_4^2=0>>
  gap> spread := Filtered(lines,x->x^em in gh);
  [ <a line in Q-(5, 3)>, <a line in Q-(5, 3)>, <a line in Q-(5, 3)>,
    <a line in Q-(5, 3)>, <a line in Q-(5, 3)>, <a line in Q-(5, 3)>,
    <a line in Q-(5, 3)>, <a line in Q-(5, 3)>, <a line in Q-(5, 3)>,
    <a line in Q-(5, 3)>, <a line in Q-(5, 3)>, <a line in Q-(5, 3)>,
    <a line in Q-(5, 3)>, <a line in Q-(5, 3)>, <a line in Q-(5, 3)>,
    <a line in Q-(5, 3)>, <a line in Q-(5, 3)>, <a line in Q-(5, 3)>,
    <a line in Q-(5, 3)>, <a line in Q-(5, 3)>, <a line in Q-(5, 3)>,
    <a line in Q-(5, 3)>, <a line in Q-(5, 3)>, <a line in Q-(5, 3)>,
    <a line in Q-(5, 3)> ]
  gap> spread := List(spread,x->ElementToElement(gh,x^em));
  [ <a line in H(3)>, <a line in H(3)>, <a line in H(3)>, <a line in H(3)>,
    <a line in H(3)>, <a line in H(3)>, <a line in H(3)>, <a line in H(3)>,
    <a line in H(3)>, <a line in H(3)>, <a line in H(3)>, <a line in H(3)>,
    <a line in H(3)>, <a line in H(3)>, <a line in H(3)>, <a line in H(3)>,
    <a line in H(3)>, <a line in H(3)>, <a line in H(3)>, <a line in H(3)>,
    <a line in H(3)>, <a line in H(3)>, <a line in H(3)>, <a line in H(3)>,
    <a line in H(3)>, <a line in H(3)>, <a line in H(3)>, <a line in H(3)> ]
  gap> Collected(Concatenation(List(spread,x->List(spread,y->DistanceBetweenElements(x,y)))));
  [ [ 0, 28 ], [ 6, 756 ] ]
```

2.5 Some particular incidence geometries

2.5.1 The split Cayley hexagon

The split Cayley hexagon is one the well known classical generalised hexagons that are obtained using a triality of the hyperbolic quadric in 7 dimensions. This example shows some basic properties of this geometry.

```
——————————————————————— Example ———————————————————————
  gap> hexagon := SplitCayleyHexagon(5);
  H(5)
  gap> Order(hexagon);
  [ 5, 5 ]
```

```
gap> g := CollineationGroup( hexagon );
#I  for Split Cayley Hexagon
#I  Computing nice monomorphism...
#I  Found permutation domain...
G_2(5)
gap> incgraph := IncidenceGraph( hexagon );;
#I  Computing incidence graph of generalised polygon...
gap> Diameter(incgraph);
6
gap> Girth(incgraph);
12
gap> points := Points(hexagon);
<points of H(5)>
gap> lines := Lines(hexagon);
<lines of H(5)>
gap> iter := Iterator(points);
<iterator>
gap> x := NextIterator(iter);
<a point in H(5)>
gap> Display(x);
[.1.....]
gap> UnderlyingObject(x);
<cvec over GF(5,1) of length 7>
gap> onx := Lines(x);
<shadow lines in H(5)>
gap> l := Random(onx);
<a line in H(5)>
gap> onl := Points(l);
<shadow points in H(5)>
gap> List(onl, t -> DistanceBetweenElements(x,t));
[ 0, 2, 2, 2, 2, 2 ]
gap> stabl := FiningStabiliser(g, l);
<projective collineation group of size 1500000 with 3 generators>
gap> gl := Action(stabl, onl);
Group([ (1,6,5,4,3), (1,4,3,6), (1,5,4,3,6,2) ])
gap> StructureDescription(gl);
"S5"
gap> Transitivity(gl);
3
```

2.5.2 An (apartment of) a building of type E_6

This example shows the constructions of an incidence geometry whose automorphism group is an exceptional group of type E_6. The construction is done as a coset geometry. This example also illustrates how to get a diagram of such a coset geometry.

```
———————————————————————— Example ————————————————————————
gap> L := SimpleLieAlgebra("E",6,Rationals);
<Lie algebra of dimension 78 over Rationals>
gap> rs := RootSystem(L);
<root system of rank 6>
gap> w := WeylGroup(rs);
```

```
<matrix group with 6 generators>
gap> gens := GeneratorsOfGroup(w);;
gap> pabs := List(gens, g -> Group(Difference(gens, [g])));
[ <matrix group with 5 generators>, <matrix group with 5 generators>,
  <matrix group with 5 generators>, <matrix group with 5 generators>,
  <matrix group with 5 generators>, <matrix group with 5 generators> ]
gap> g := Group(gens);
<matrix group with 6 generators>
gap> cg := CosetGeometry(g,pabs);;
gap> diag := DiagramOfGeometry( cg );;
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
gap> DrawDiagram(diag, "E6");
gap> #Exec("open E6.ps");
```
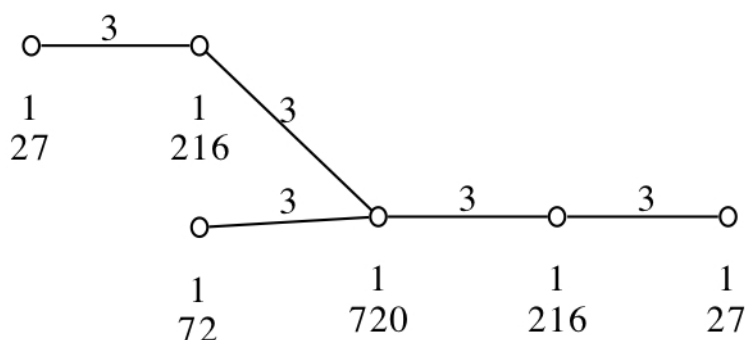

2.5.3 A rank 4 geometry for $PSL(2,11)$

Here we look at a particular flag-transitive geometry constructed from four subgroups of $PSL(2,11)$, and we construct the diagram for this geometry. To view this diagram, you need to either use a postscript viewer or a dotty viewer (such as GraphViz).

```
———————————————————— Example ————————————————————
gap> g := PSL(2,11);
Group([ (3,11,9,7,5)(4,12,10,8,6), (1,2,8)(3,7,9)(4,10,5)(6,12,11) ])
gap> g1 := Group([ (1,2,3)(4,8,12)(5,10,9)(6,11,7), (1,2)(3,4)(5,12)(6,11)(7,10)(8,9) ]);
```

```
       Group([ (1,2,3)(4,8,12)(5,10,9)(6,11,7), (1,2)(3,4)(5,12)(6,11)(7,10)(8,9) ])
gap> g2 := Group([ (1,2,7)(3,9,4)(5,11,10)(6,8,12), (1,2)(3,4)(5,12)(6,11)(7,10)(8,9) ]);
Group([ (1,2,7)(3,9,4)(5,11,10)(6,8,12), (1,2)(3,4)(5,12)(6,11)(7,10)(8,9) ])
gap> g3 := Group([ (1,2,11)(3,8,7)(4,9,5)(6,10,12), (1,2)(3,12)(4,11)(5,10)(6,9)(7,8) ]);
Group([ (1,2,11)(3,8,7)(4,9,5)(6,10,12), (1,2)(3,12)(4,11)(5,10)(6,9)(7,8) ])
gap> g4 := Group([ (1,2,11)(3,8,7)(4,9,5)(6,10,12), (1,2)(3,10)(4,9)(5,8)(6,7)(11,12) ]);
Group([ (1,2,11)(3,8,7)(4,9,5)(6,10,12), (1,2)(3,10)(4,9)(5,8)(6,7)(11,12) ])
gap> cg := CosetGeometry(g, [g1,g2,g3,g4]);
CosetGeometry( Group( [ ( 3,11, 9, 7, 5)( 4,12,10, 8, 6),
  ( 1, 2, 8)( 3, 7, 9)( 4,10, 5)( 6,12,11) ] ) )
gap> SetName(cg, "Gamma");
gap> ParabolicSubgroups(cg);
[ Group([ (1,2,3)(4,8,12)(5,10,9)(6,11,7), (1,2)(3,4)(5,12)(6,11)(7,10)
  (8,9) ]), Group([ (1,2,7)(3,9,4)(5,11,10)(6,8,12), (1,2)(3,4)(5,12)(6,11)
  (7,10)(8,9) ]), Group([ (1,2,11)(3,8,7)(4,9,5)(6,10,12), (1,2)(3,12)(4,11)
  (5,10)(6,9)(7,8) ]), Group([ (1,2,11)(3,8,7)(4,9,5)(6,10,12), (1,2)(3,10)
  (4,9)(5,8)(6,7)(11,12) ]) ]
gap> BorelSubgroup(cg);
Group(())
gap> AmbientGroup(cg);
Group([ (3,11,9,7,5)(4,12,10,8,6), (1,2,8)(3,7,9)(4,10,5)(6,12,11) ])
gap> type2 := ElementsOfIncidenceStructure( cg, 2 );
<elements of type 2 of Gamma>
gap> IsFlagTransitiveGeometry( cg );
true
gap> DrawDiagram( DiagramOfGeometry(cg), "PSL211");
```

The output of this example uses `dotty` which is a sophisticated graph drawing program. We also provide `DrawDiagramWithNeato` to make a diagram with straight lines, using `neato`. Here is what the output looks like with the standard `DrawDiagram` command:
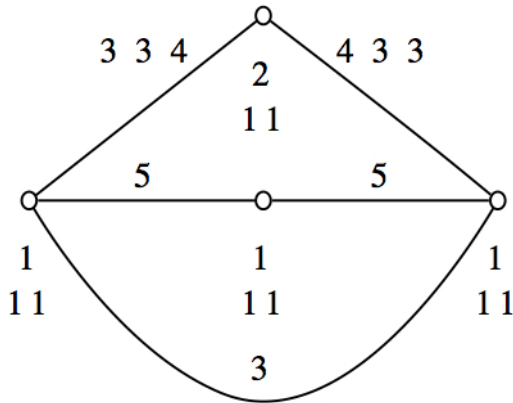
2.5.4 The Ree-Tits octagon of order $[2, 4]$ as coset geometry

In this example we construct the Ree-Tits octagon of order $[2, 4]$ as a coset geometry. From the computation of the so-called rank 2 parameters, it can be observed already that the constructed geometry must be a generalised octagon. Then the points and lines are computed explicitly, and together with

the incidence and the available group as a subgroup of the collineation group, a generalised octagon is constructed.

```
——————————————— Example ———————————————
gap> LoadPackage( "AtlasRep" );
true
gap> titsgroup:=AtlasGroup("2F4(2)'");
<permutation group of size 17971200 with 2 generators>
gap> g1:=AtlasSubgroup(titsgroup,3);
<permutation group of size 10240 with 2 generators>
gap> g2:=AtlasSubgroup(titsgroup,5);
<permutation group of size 6144 with 2 generators>
gap> conj:=ConjugacyClassSubgroups(titsgroup,g1);;
gap> # Now look for the conjugate of g1 with maximal intersection
gap> g1:=First(conj, sg -> Size(Intersection(sg,g2))=2048);
<permutation group of size 10240 with 2 generators>
gap> cg:=CosetGeometry(titsgroup,[g1,g2]);;
gap> Rank2Parameters(cg);
[ [ 8, 8, 8 ], [ 2, 1755 ], [ 4, 2925 ] ]
gap> pts := Set(ElementsOfIncidenceStructure(cg,1));;
gap> lines := Set(ElementsOfIncidenceStructure(cg,2));;
gap> gp := GeneralisedPolygonByElements(pts,lines,\*,titsgroup,OnCosetGeometryElement);
<generalised octagon of order [ 2, 4 ]>
```

2.6 Elation generalised quadrangles

In this section, we construct a classical elation generalised quadrangle from a q-clan, and we see that the associated BLT-set is a conic.

2.6.1 The classical q-clan

```
——————————————— Example ———————————————
gap> f := GF(3);
GF(3)
gap> id := IdentityMat(2, f);;
gap> clan := List( f, t -> t*id );;
gap> IsqClan( clan, f );
true
gap> clan := qClan(clan, f);
<q-clan over GF(3)>
gap> egq := EGQByqClan( clan );
#I  Computed Kantor family. Now computing EGQ...
<EGQ of order [ 9, 3 ] and basepoint 0>
gap> elations := ElationGroup( egq );
<matrix group of size 243 with 8 generators>
gap> points := Points( egq );
<points of <EGQ of order [ 9, 3 ] and basepoint 0>>
gap> p := Random(points);
<a point of class 2 of <EGQ of order [ 9, 3 ] and basepoint 0>>
gap> x := Random(elations);
[ [ Z(3)^0, 0*Z(3), 0*Z(3), Z(3)^0 ], [ 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3)^0 ], [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ]
```

```
gap> OnKantorFamily(p,x);
<a point of class 2 of <EGQ of order [ 9, 3 ] and basepoint 0>>
gap> orbs := Orbits( elations, points, OnKantorFamily);;
gap> Collected(List( orbs, Size ));
[ [ 1, 1 ], [ 9, 4 ], [ 243, 1 ] ]
gap> blt := BLTSetByqClan( clan );
[ <a point in Q(4, 3): -x_1*x_5-x_2*x_4+x_3^2=0>,
  <a point in Q(4, 3): -x_1*x_5-x_2*x_4+x_3^2=0>,
  <a point in Q(4, 3): -x_1*x_5-x_2*x_4+x_3^2=0>,
  <a point in Q(4, 3): -x_1*x_5-x_2*x_4+x_3^2=0> ]
gap> q4q := AmbientGeometry( blt[1] );
Q(4, 3): -x_1*x_5-x_2*x_4+x_3^2=0
gap> span := Span( blt );
<a plane in ProjectiveSpace(4, 3)>
gap> ProjectiveDimension( span );
2
```

2.6.2 Two ways to construct a flock generalised quadrangle from a Kantor-Knuth semifield q-clan

We will construct an elation generalised quadrangle directly from the *Kantor-Knuth semifield q-clan* and also via its corresponding BLT-set. The q-clan in question here are the set of matrices C_t of the form $\begin{pmatrix} t & 0 \\ 0 & -nt^\phi \end{pmatrix}$ where t runs over the elements of $GF(q)$, q is odd and not prime, n is a fixed nonsquare and ϕ is a nontrivial automorphism of $GF(q)$.

——————————— Example ———————————

```
gap> q := 9;
9
gap> f := GF(q);
GF(3^2)
gap> squares := AsList(Group(Z(q)^2));
[ Z(3)^0, Z(3^2)^6, Z(3), Z(3^2)^2 ]
gap> n := First(GF(q), x -> not IsZero(x) and not x in squares);
Z(3^2)
gap> sigma := FrobeniusAutomorphism( f );
FrobeniusAutomorphism( GF(3^2) )
gap> zero := Zero(f);
0*Z(3)
gap> qclan := List(GF(q), t -> [[t, zero], [zero,-n * t^sigma]] );
[ [ [ 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3) ] ],
  [ [ Z(3^2), 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ],
  [ [ Z(3^2)^5, 0*Z(3) ], [ 0*Z(3), Z(3) ] ],
  [ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3^2)^5 ] ],
  [ [ Z(3^2)^2, 0*Z(3) ], [ 0*Z(3), Z(3^2)^3 ] ],
  [ [ Z(3^2)^3, 0*Z(3) ], [ 0*Z(3), Z(3^2)^6 ] ],
  [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3^2) ] ],
  [ [ Z(3^2)^7, 0*Z(3) ], [ 0*Z(3), Z(3^2)^2 ] ],
  [ [ Z(3^2)^6, 0*Z(3) ], [ 0*Z(3), Z(3^2)^7 ] ] ]
gap> IsqClan( qclan, f );
true
gap> qclan := qClan(qclan , f);
```

```
<q-clan over GF(3^2)>
gap> egq1 := EGQByqClan( qclan);
#I  Computed Kantor family. Now computing EGQ...
<EGQ of order [ 81, 9 ] and basepoint 0>
gap> blt := BLTSetByqClan( qclan );
[ <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0> ]
gap> egq2 := EGQByBLTSet( blt );
#I  Now embedding dual BLT-set into W(5,q)...
#I  Computing elation group...
<EGQ of order [ 81, 9 ] and basepoint in W(5, 9 ) >
```

2.7 Algebraic varieties

2.7.1 A projective variety

In this example we demonstrate the construction of projective varieties.

──────── Example ────────

```
gap> pg1 := PG(1, 7);
ProjectiveSpace(1, 7)
gap> pg3 := PG(3, 7);
ProjectiveSpace(3, 7)
gap> points := Points(pg1);
<points of ProjectiveSpace(1, 7)>
gap> coords := List(points, Coordinates);
[ [ Z(7)^0, 0*Z(7) ], [ Z(7)^0, Z(7)^0 ], [ Z(7)^0, Z(7) ],
  [ Z(7)^0, Z(7)^2 ], [ Z(7)^0, Z(7)^3 ], [ Z(7)^0, Z(7)^4 ],
  [ Z(7)^0, Z(7)^5 ], [ 0*Z(7), Z(7)^0 ] ]
gap> curve := List(coords, t -> VectorSpaceToElement(pg3, [ t[1]^3, t[1]^2 * t[2], t[1] * t[2]^2
gap> pgl := ProjectivityGroup( pg3 );
The FinInG projectivity group PGL(4,7)
gap> stabcurve := FiningSetwiseStabiliser( pgl, curve );
#I  Computing adjusted stabilizer chain...
<projective collineation group with 6 generators>
gap> StructureDescription( stabcurve );
"PSL(3,2) : C2"
gap> Span( curve );
ProjectiveSpace(3, 7)
gap> pg3lines := Lines( pg3 );
<lines of ProjectiveSpace(3, 7)>
gap> orbits := FiningOrbits(stabcurve, pg3lines);
2%..3%..9%..15%..16%..21%..22%..28%..34%..40%..46%..52%..64%..75%..81%..84%..88%..94%..95%..99%.
  <closed orbit, 28 points>, <closed orbit, 168 points>,
```

```
    <closed orbit, 168 points>, <closed orbit, 28 points>,
    <closed orbit, 168 points>, <closed orbit, 28 points>,
    <closed orbit, 168 points>, <closed orbit, 168 points>,
    <closed orbit, 168 points>, <closed orbit, 168 points>,
    <closed orbit, 168 points>, <closed orbit, 336 points>,
    <closed orbit, 336 points>, <closed orbit, 168 points>,
    <closed orbit, 84 points>, <closed orbit, 112 points>,
    <closed orbit, 168 points>, <closed orbit, 21 points>,
    <closed orbit, 112 points>, <closed orbit, 21 points> ]
gap> List(orbits, Size);
[ 8, 56, 28, 168, 168, 28, 168, 28, 168, 168, 168, 168, 336, 336, 168,
  84, 112, 168, 21, 112, 21 ]
gap> pg3points := Points( pg3 );
<points of ProjectiveSpace(3, 7)>
gap> orbits := FiningOrbits(stabcurve, pg3points);
2%..16%..30%..72%..100%..[ <closed orbit, 8 points>, <closed orbit, 56 points>,
  <closed orbit, 56 points>, <closed orbit, 168 points>,
  <closed orbit, 112 points> ]
gap> List(orbits, Size);
[ 8, 56, 56, 168, 112 ]
gap> reps := List(orbits, Representative);
[ <a point in ProjectiveSpace(3, 7)>, <a point in ProjectiveSpace(3, 7)>,
  <a point in ProjectiveSpace(3, 7)>, <a point in ProjectiveSpace(3, 7)>,
  <a point in ProjectiveSpace(3, 7)> ]
gap> x := reps[2];
<a point in ProjectiveSpace(3, 7)>
gap> proj := NaturalProjectionBySubspace(pg3, x);
<geometry morphism from <All elements of ProjectiveSpace(3,
7)> to <All elements of ProjectiveSpace(2, 7)>>
gap> curveminusx := Difference(curve, [x]);
[ <a point in ProjectiveSpace(3, 7)>, <a point in ProjectiveSpace(3, 7)>,
  <a point in ProjectiveSpace(3, 7)>, <a point in ProjectiveSpace(3, 7)>,
  <a point in ProjectiveSpace(3, 7)>, <a point in ProjectiveSpace(3, 7)>,
  <a point in ProjectiveSpace(3, 7)>, <a point in ProjectiveSpace(3, 7)> ]
gap> cuspidal := ImagesSet(proj, List(curveminusx, t -> Span(x, t)));
[ <a point in ProjectiveSpace(2, 7)>, <a point in ProjectiveSpace(2, 7)>,
  <a point in ProjectiveSpace(2, 7)>, <a point in ProjectiveSpace(2, 7)>,
  <a point in ProjectiveSpace(2, 7)>, <a point in ProjectiveSpace(2, 7)>,
  <a point in ProjectiveSpace(2, 7)>, <a point in ProjectiveSpace(2, 7)> ]
gap> coords := List(cuspidal, Coordinates);
[ [ Z(7)^0, 0*Z(7), 0*Z(7) ], [ 0*Z(7), 0*Z(7), Z(7)^0 ],
  [ Z(7)^0, Z(7)^0, Z(7)^0 ], [ Z(7)^0, Z(7)^2, Z(7)^0 ],
  [ Z(7)^0, Z(7)^4, Z(7)^0 ], [ Z(7)^0, Z(7)^0, Z(7)^3 ],
  [ Z(7)^0, Z(7)^2, Z(7)^3 ], [ Z(7)^0, Z(7)^4, Z(7)^3 ] ]
gap> r := PolynomialRing(GF(7), 3);
GF(7)[x_1,x_2,x_3]
gap> indets := IndeterminatesOfPolynomialRing(r);
[ x_1, x_2, x_3 ]
gap> shapes := Filtered(Tuples([0,1,2,3], 3), t -> Sum(t) = 3);
[ [ 0, 0, 3 ], [ 0, 1, 2 ], [ 0, 2, 1 ], [ 0, 3, 0 ], [ 1, 0, 2 ],
  [ 1, 1, 1 ], [ 1, 2, 0 ], [ 2, 0, 1 ], [ 2, 1, 0 ], [ 3, 0, 0 ] ]
gap> mat := List(coords, t -> List(shapes, u -> Product([1,2,3], i -> t[i]^u[i])));
[ [ 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7),
```

```
        Z(7)^0 ],
  [ Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7),
      0*Z(7) ],
  [ Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0,
      Z(7)^0 ],
  [ Z(7)^0, Z(7)^2, Z(7)^4, Z(7)^0, Z(7)^0, Z(7)^2, Z(7)^4, Z(7)^0, Z(7)^2,
      Z(7)^0 ],
  [ Z(7)^0, Z(7)^4, Z(7)^2, Z(7)^0, Z(7)^0, Z(7)^4, Z(7)^2, Z(7)^0, Z(7)^4,
      Z(7)^0 ],
  [ Z(7)^3, Z(7)^0, Z(7)^3, Z(7)^0, Z(7)^0, Z(7)^3, Z(7)^0, Z(7)^3, Z(7)^0,
      Z(7)^0 ],
  [ Z(7)^3, Z(7)^2, Z(7), Z(7)^0, Z(7)^0, Z(7)^5, Z(7)^4, Z(7)^3, Z(7)^2,
      Z(7)^0 ],
  [ Z(7)^3, Z(7)^4, Z(7)^5, Z(7)^0, Z(7)^0, Z(7), Z(7)^2, Z(7)^3, Z(7)^4,
      Z(7)^0 ] ]
gap> mat2 := ShallowCopy(mat);
[ [ 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7),
      Z(7)^0 ],
  [ Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7),
      0*Z(7) ],
  [ Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0,
      Z(7)^0 ],
  [ Z(7)^0, Z(7)^2, Z(7)^4, Z(7)^0, Z(7)^0, Z(7)^2, Z(7)^4, Z(7)^0, Z(7)^2,
      Z(7)^0 ],
  [ Z(7)^0, Z(7)^4, Z(7)^2, Z(7)^0, Z(7)^0, Z(7)^4, Z(7)^2, Z(7)^0, Z(7)^4,
      Z(7)^0 ],
  [ Z(7)^3, Z(7)^0, Z(7)^3, Z(7)^0, Z(7)^0, Z(7)^3, Z(7)^0, Z(7)^3, Z(7)^0,
      Z(7)^0 ],
  [ Z(7)^3, Z(7)^2, Z(7), Z(7)^0, Z(7)^0, Z(7)^5, Z(7)^4, Z(7)^3, Z(7)^2,
      Z(7)^0 ],
  [ Z(7)^3, Z(7)^4, Z(7)^5, Z(7)^0, Z(7)^0, Z(7), Z(7)^2, Z(7)^3, Z(7)^4,
      Z(7)^0 ] ]
gap> sol := NullspaceMat(TransposedMat(mat2))[1];
[ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^3, Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7),
  0*Z(7) ]
gap> terms := List(shapes, u -> Product([1,2,3], i -> indets[i]^u[i]));
[ x_3^3, x_2*x_3^2, x_2^2*x_3, x_2^3, x_1*x_3^2, x_1*x_2*x_3, x_1*x_2^2,
  x_1^2*x_3, x_1^2*x_2, x_1^3 ]
gap> poly := terms * sol;
x_1*x_3^2-x_2^3
gap> pg2 := AmbientGeometry(Range(proj));
ProjectiveSpace(2, 7)
gap> variety := ProjectiveVariety(pg2, [poly]);
Projective Variety in ProjectiveSpace(2, 7)
gap> points := Points(variety);
<points of Projective Variety in ProjectiveSpace(2, 7)>
gap> Size(points);
8
```

Chapter 3

Incidence Geometry

We follow [BC13] for the definitions of incidence structure and incidence geometry. An *incidence structure* consists of a set of elements, a symmetric relation on the elements and a type function from the set of elements to an index set (i.e., every element has a "type"). It satisfies the following axiom: *(i) no two elements of the same type are incident.* An incidence structure without type function is in fact a multipartite graph where the adjacency is the incidence (so with a loop on each vertex). The term geometry, or incidence geometry, is interpreted broadly in this package. Particularly, an *incidence geometry* is an incidence structure satisfying the following axiom: *(ii) every maximal flag contains an element of each type.* In graph terminology, this means that every maximal clique contains an element of each type. Thus, a projective 5-space is an incidence geometry with five types of elements: points, lines, planes, solids, and hyperplanes. A finite classical polar space of rank 3 is an incidence geometry with three types of elements: points, lines, and planes. Depending on the viewpoint, the Grassmann variety of the lines of a projective 4-space, is an incidence structure that is not an incidence geometry.

FinInG concerns itself primarily with the most commonly studied incidence geometries of rank at least 2: projective spaces, polar spaces, and affine spaces. Throughout, no matter the geometry, we have made the convention that an element of type 1 is a "point", an element of type 2 is a "line", and so forth. The examples we use in this section use projective spaces, which have not yet been introduced to the reader in this manual. For further information on projective spaces, see Chapter 4.

In this chapter we describe functionality that is DECLARED for incidence structures, which does not imply that operations described here will work for arbitrary user-constructed incidence structures. Its aim is furthermore to allow the user to become familiar with the general philosophy of the package, using examples that are self-explanatory. Not all details of the commands used in the examples will be explained in this chapter, therefore we refer to the relevant chapter for the commands. These can easily be found using the index.

3.1   Incidence structures

Incidence structures can be more general than incidence geometries, e.g., if they do not satisfy axiom (ii) mentioned above. We allow the construction of such objects. This explains one of the top level categories in FinInG.

3.1.1 IsIncidenceStructure

▷ IsIncidenceStructure (Category)

Top level category for all objects representing an incidence structure.

3.1.2 IsIncidenceGeometry

▷ IsIncidenceGeometry (Category)

Category for all objects representing an incidence geometry. All particular geometries implemented in FinInG are incidence geometries.

3.1.3 IncidenceStructure

▷ IncidenceStructure(*eles, inc_rel, type, typeset*) (operation)

Returns: an incidence structure

eles is a set containing the elements of the incidence structure. *inc_rel* must be a function that determines whether two objects in the set *eles* are incident. *type* is a function mapping any element to its type, which is a unique element in the set *typeset*.

In the following example we define an incidence structure that is not an incidence geometry. The example used is the incidence structure with elements the subspaces contained in the line Grassmannian of PG(4,2). This example is not meant to create this incidence structure in an efficient way, but just to demonstrate the general philosophy.

```
_____ Example _____
 gap> pg := PG(4,2);
 ProjectiveSpace(4, 2)
 gap> pg2 := PG(9,2);
 ProjectiveSpace(9, 2)
 gap> points := List(Lines(pg),x->VectorSpaceToElement(pg2,GrassmannCoordinates(x)));;
 gap> flags := Concatenation(List(Points(pg),x->List(Planes(x),y->FlagOfIncidenceStructure(pg,[x,
 gap> prelines := List(flags,flag->ShadowOfFlag(pg,flag,2));;
 gap> lines := List(prelines,x->VectorSpaceToElement(pg2,List(x,y->GrassmannCoordinates(y))));;
 gap> flags2 := Concatenation(List(Points(pg),x->List(Solids(x),y->FlagOfIncidenceStructure(pg,[x
 gap> preplanes := List(flags2,flag->ShadowOfFlag(pg,flag,2));;
 gap> planes := List(preplanes,x->VectorSpaceToElement(pg2,List(x,y->GrassmannCoordinates(y))));;
 gap> maximals1 := List(Planes(pg),x->VectorSpaceToElement(pg2,List(Lines(x),y->GrassmannCoordina
 gap> maximals2 := List(Points(pg),x->VectorSpaceToElement(pg2,List(Lines(x),y->GrassmannCoordina
 gap> elements := Union(points,lines,planes,maximals1,maximals2);;
 gap> Length(elements);
 1891
 gap> type := x -> ProjectiveDimension(x)+1;
 function( x ) ... end
 gap> inc_rel := \*;
 <Operation "*">
 gap> inc := IncidenceStructure(elements,inc_rel,type,[1,2,3,4]);
 Incidence structure of rank 4
 gap> Rank(inc);
 4
```

Lie Geometries, i.e., geometries with a projective space as ambient space, affine spaces and generalised polygons have their own category, which is a subcategory of `IsIncidenceGeometry`.

3.1.4 Main categories in `IsIncidenceGeometry`

▷ `IsLieGeometry` (Category)
▷ `IsAffineSpace` (Category)
▷ `IsGeneralisedPolygon` (Category)
▷ `IsCosetGeometry` (Category)

Within each category, several subcategories are declared. Subcategories of `IsLieGeometry` are discussed in Section 3.6 and subcategories of `IsGeneralisedPolygon` are discussed in Chapter 12

3.1.5 Examples of categories of incidence geometries

```
─────────────────── Example ───────────────────
gap> CategoriesOfObject(ProjectiveSpace(5,7));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsLieGeometry",
  "IsProjectiveSpace" ]
gap> CategoriesOfObject(HermitianPolarSpace(5,9));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsLieGeometry",
  "IsClassicalPolarSpace", "IsAlgebraicVariety", "IsProjectiveVariety",
  "IsHermitianVariety" ]
gap> CategoriesOfObject(AffineSpace(3,3));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsAffineSpace" ]
gap> CategoriesOfObject(SymplecticSpace(3,11));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsLieGeometry",
  "IsClassicalPolarSpace", "IsGeneralisedPolygon", "IsGeneralisedQuadrangle",
  "IsClassicalGQ" ]
gap> CategoriesOfObject(SplitCayleyHexagon(9));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsLieGeometry",
  "IsGeneralisedPolygon", "IsGeneralisedHexagon",
  "IsClassicalGeneralisedHexagon" ]
gap> CategoriesOfObject(ParabolicQuadric(4,16));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsLieGeometry",
  "IsClassicalPolarSpace", "IsGeneralisedPolygon", "IsGeneralisedQuadrangle",
  "IsClassicalGQ", "IsAlgebraicVariety", "IsProjectiveVariety" ]
```

3.1.6 TypesOfElementsOfIncidenceStructure

▷ `TypesOfElementsOfIncidenceStructure(inc)` (attribute)
▷ `TypesOfElementsOfIncidenceStructurePlural(inc)` (attribute)

Returns: a list of strings or integers

Both attributes are declared for objects in the category `IsIncidenceStructure`. Any incidence structure has a set of types, which is usually just the list 1..*n*. If specific names are given to each type, like points, lines, etc., this attribute returns the names for the particular incidence structure *inc*. The second variant returns the list of plurals of these names. For genericly constructed incidence structures, the names of the Elements are also generic: elements of type 1, elements of type 2, etc.

```
———————————————————————— Example ————————————————————————
 gap> TypesOfElementsOfIncidenceStructure(ProjectiveSpace(5,4));
 [ "point", "line", "plane", "solid", "proj. 4-space" ]
 gap> TypesOfElementsOfIncidenceStructurePlural(AffineSpace(7,4));
 [ "points", "lines", "planes", "solids", "affine. subspaces of dim. 4",
   "affine. subspaces of dim. 5", "affine. subspaces of dim. 6" ]
```

3.1.7 Rank

▷ Rank(*inc*)   (operation)

▷ RankAttr(*inc*)   (attribute)

Returns: rank of *inc*, an object which must belong to the category IsIncidenceStructure

The operation Rank returns the rank of the incidence structure *inc*. The highest level method for Rank, applicable to objects in IsIncidenceStructure simply refers to the attribute RankAttr. In FinInG, the rank of an incidence structure is determined upon creation, when also RankAttr is set.

```
———————————————————————— Example ————————————————————————
 gap> Rank(ProjectiveSpace(5,5));
 5
 gap> Rank(AffineSpace(3,5));
 3
 gap> Rank(SymplecticSpace(5,5));
 3
```

3.1.8 IncidenceGraph

▷ IncidenceGraph(*inc*)   (attribute)

Returns: a graph

The vertices are the elements of *inc*, adjacency between different vertices is equal to incidence, and there are of course no loops. For generic incidence structures, i.e. constructed through IncidenceStructure, there is no efficient method installed, so this operation can be time consuming.

If *inc* is a generic incidence structure, i.e. created using IncidenceStructure, the vertex names of the graph are integers. It is not by default possible to use the elements of *inc* as vertex names, since it is not known in the generic case whether the elements of different type of *inc* can be ordered. For particular incidence geometries, e.g. projective spaces, etc., the vertex names will be the elements, which will be demonstrated through examples in the appropriate chapters.

In the example we consider the so-called doubling of the smallest generalised quadrangle: the points of the incidence structure are the points and the lines of the GQ, the lines of the incidence structure are all the point-line flags of the GQ. The incidence is the natural one. It is then checked that diameter and girth of the incidence graph are 8 and 16 respectively, which makes that the incidence structure is a generalised octagon.

```
———————————————————————— Example ————————————————————————
 gap> ps := SymplecticSpace(3,2);
 W(3, 2)
 gap> pts := List(Points(ps));;
 gap> lines := List(Lines(ps));;
 gap> flags := Union(List(pts,x->List(Lines(x),y->FlagOfIncidenceStructure(ps,[x,y]))));;
```

```
gap> inc := function(x,y)
> if x = y then
>      return true;
> elif IsFlagOfIncidenceStructure(x) and IsElementOfIncidenceStructure(y) then
>      return IsIncident(x,y);
> elif IsElementOfIncidenceStructure(x) and IsElementOfIncidenceStructure(y) then
>      return false;
> elif IsFlagOfIncidenceStructure(x) and IsFlagOfIncidenceStructure(y) then
>      return false;
> else
>      return inc(y,x);
> fi;
> end;
function( x, y ) ... end
gap> type := function(x)
> if IsList(Type(x)) then
>      return 2;
> else
>      return 1;
> fi;
> end;
function( x ) ... end
gap> els := Union(pts,lines,flags);;
gap> struc := IncidenceStructure(els,inc,type,[1,2]);
Incidence structure of rank 2
gap> gamma := IncidenceGraph(struc);
rec( adjacencies := [ [ 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, 56, 57 ], [ 58, 59, 60 ], [ 61, 62, 63 ],
      [ 64, 65, 66 ], [ 67, 68, 69 ], [ 70, 71, 72 ], [ 73, 74, 75 ],
      [ 31, 40, 43 ], [ 32, 52, 55 ], [ 33, 64, 67 ], [ 34, 41, 46 ],
      [ 35, 53, 58 ], [ 36, 65, 70 ], [ 37, 42, 49 ], [ 38, 54, 61 ],
      [ 39, 66, 73 ], [ 44, 59, 74 ], [ 45, 62, 71 ], [ 47, 56, 75 ],
      [ 50, 57, 72 ], [ 48, 63, 68 ], [ 51, 60, 69 ], [ 1, 16 ], [ 1, 17 ],
      [ 1, 18 ], [ 2, 19 ], [ 2, 20 ], [ 2, 21 ], [ 3, 22 ], [ 3, 23 ],
      [ 3, 24 ], [ 4, 16 ], [ 4, 19 ], [ 4, 22 ], [ 5, 16 ], [ 5, 25 ],
      [ 5, 26 ], [ 6, 19 ], [ 6, 27 ], [ 6, 29 ], [ 7, 22 ], [ 7, 28 ],
      [ 7, 30 ], [ 8, 17 ], [ 8, 20 ], [ 8, 23 ], [ 9, 17 ], [ 9, 27 ],
      [ 9, 28 ], [ 10, 20 ], [ 10, 25 ], [ 10, 30 ], [ 11, 23 ], [ 11, 26 ],
      [ 11, 29 ], [ 12, 18 ], [ 12, 21 ], [ 12, 24 ], [ 13, 18 ], [ 13, 29 ],
      [ 13, 30 ], [ 14, 21 ], [ 14, 26 ], [ 14, 28 ], [ 15, 24 ], [ 15, 25 ],
      [ 15, 27 ] ], group := Group(()), isGraph := true, names := [ 1 .. 75 ],
  order := 75,
  representatives := [ 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, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
      71, 72, 73, 74, 75 ],
  schreierVector := [ -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,
      -56, -57, -58, -59, -60, -61, -62, -63, -64, -65, -66, -67, -68, -69,
```

```
        -70, -71, -72, -73, -74, -75 ] )
gap> Diameter(gamma);
8
gap> Girth(gamma);
16
```

3.2 Elements of incidence structures

3.2.1 Main categories for individual elements of incidence structures

▷ IsElementOfIncidenceStructure   (Category)
▷ IsElementOfIncidenceGeometry   (Category)
▷ IsElementOfLieGeometry   (Category)
▷ IsElementOfAffineSpace   (Category)
▷ IsElementOfCosetGeometry   (Category)
▷ IsSubspaceOfProjectiveSpace   (Category)
▷ IsSubspaceOfClassicalPolarSpace   (Category)
▷ IsElementOfGeneralisedPolygon   (Category)

In general, elements of an incidence structure belonging to IsIncStr, are in the category IsElementOfIncStr. The inclusion for different categories of geometries is followed for their elements, with an exception for IsSubspaceOfClassicalPolarSpace, which is a subcategory of IsSubspaceOfProjectiveSpace, while IsClassicalPolarSpace is not a subcategory of IsProjectiveSpace.

```
————————————————————— Example ————————————————————————
gap> Random(Lines(SplitCayleyHexagon(3)));
#I  for Split Cayley Hexagon
#I  Computing nice monomorphism...
#I  Found permutation domain...
<a line in H(3)>
gap> CategoriesOfObject(last);
[ "IsElementOfIncidenceStructure", "IsElementOfIncidenceGeometry",
  "IsElementOfLieGeometry", "IsSubspaceOfProjectiveSpace",
  "IsSubspaceOfClassicalPolarSpace", "IsElementOfGeneralisedPolygon" ]
gap> Random(Solids(AffineSpace(7,17)));
<a solid in AG(7, 17)>
gap> CategoriesOfObject(last);
[ "IsElementOfIncidenceStructure", "IsElementOfIncidenceGeometry",
  "IsSubspaceOfAffineSpace" ]
```

3.2.2 UnderlyingObject

▷ UnderlyingObject(*el*)   (operation)

Returns: an object

An element of an incidence structure has a type and an underlying object. E.g. a line of a projective space is determined by a two dimensional sub vector space, which is determined by a basis. Elements

of incidence structure can also be objects representing elements of other incidence structures, as is e.g. the case in the example of 3.1.3. The examples shows the underlying objects of elements of three totally different incidence geometries.

```
────────── Example ──────────
gap> pg := PG(2,2);
ProjectiveSpace(2, 2)
gap> p := Random(Points(pg));
<a point in ProjectiveSpace(2, 2)>
gap> UnderlyingObject(p);
<cvec over GF(2,1) of length 3>
gap> l := Random(Lines(pg));
<a line in ProjectiveSpace(2, 2)>
gap> UnderlyingObject(l);
<cmat 2x3 over GF(2,1)>
gap> mat := [ [ 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
>    [ 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
>    [ 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0 ],
>    [ 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 ],
>    [ 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0 ],
>    [ 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0 ],
>    [ 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 ],
>    [ 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0 ],
>    [ 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0 ],
>    [ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0 ],
>    [ 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0 ],
>    [ 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1 ],
>    [ 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1 ],
>    [ 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 ],
>    [ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1 ] ];
[ [ 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
  [ 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
  [ 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0 ],
  [ 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 ],
  [ 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0 ],
  [ 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0 ],
  [ 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1 ],
  [ 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1 ],
  [ 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1 ] ]
gap> gp := GeneralisedPolygonByIncidenceMatrix(mat);
<generalised quadrangle of order [ 2, 2 ]>
gap> p := Random(Points(gp));
<a point in <generalised quadrangle of order [ 2, 2 ]>>
gap> UnderlyingObject(p);
15
gap> l := Random(Lines(gp));
<a line in <generalised quadrangle of order [ 2, 2 ]>>
gap> UnderlyingObject(l);
```

```
[ 7, 13, 15 ]
gap> egq := EGQByBLTSet(BLTSetByqClan(LinearqClan(3)));
#I  Now embedding dual BLT-set into W(5,q)...
#I  Computing elation group...
<EGQ of order [ 9, 3 ] and basepoint in W(5, 3 ) >
gap> p := Random(Points(egq));
<a point in <EGQ of order [ 9, 3 ] and basepoint in W(5, 3 ) >>
gap> UnderlyingObject(p);
<a point in W(5, 3)>
```

3.2.3 Type

▷ Type(*el*)   (operation)

    Returns: an integer

An element of an incidence structure has a type and an underlying object. Its type is always a non-negative integer. This operation returns the type of an element.

```
——————————————— Example ———————————————
gap> pg := PG(2,2);
ProjectiveSpace(2, 2)
gap> p := Random(Points(pg));
<a point in ProjectiveSpace(2, 2)>
gap> Type(p);
1
gap> l := Random(Lines(pg));
<a line in ProjectiveSpace(2, 2)>
gap> Type(l);
2
```

3.2.4 ObjectToElement

▷ ObjectToElement(*inc, t, obj*)   (operation)

    Returns: an element of the incidence structure *inc*

If *obj* represents an element of *inc* of type *t*, this operation returns the element. An error (or no method found error) is shown when *obj* does not represent an element of type *t*. Note that ObjectToElement is a generic operation. Versions with a different argument set and even alternative operations exist for some particular geometries to construct particular elements.

3.2.5 Main categories for collections of all the elements of a given type of an incidence structure

▷ IsElementsOfIncidenceStructure   (Category)

▷ IsElementsOfIncidenceGeometry   (Category)

▷ IsElementsOfLieGeometry   (Category)

▷ IsElementsOfAffineSpace   (Category)

▷ IsElementsOfCosetGeometry   (Category)

▷ IsSubspacesOfProjectiveSpace   (Category)

▷ `IsSubspacesOfClassicalPolarSpace`   (Category)

For a given incidence structure, the collection of elements of a given type can be constructed. constructed here means that an object is returned that represents all the elements of a given type, rather than listing them immediately, to avoid long computation times. Such an abstract object is e.g. used as a range for In general, the collection of elements of a given type of an incidence structure belonging to `IsIncStr`, is in the category `IsElementsOfIncStr`. The inclusion for different categories of geometries is followed for their collection of elements of a given type, with an exception for `IsSubspacesOfClassicalPolarSpace`, which is a subcategory of `IsSubspacesOfProjectiveSpace`, while `IsClassicalPolarSpace` is not a subcategory of `IsProjectiveSpace`.

The object representing the set of elements of a given type can be computed using the general operation `ElementsOfIncidenceStructure`.

3.2.6 ElementsOfIncidenceStructure

▷ `ElementsOfIncidenceStructure(inc, j)`   (operation)
▷ `ElementsOfIncidenceStructure(inc, str)`   (operation)

Returns: a collection of elements

`inc` must be an incidence structure, `j` must be a type of element of `inc`. This function returns all elements of `inc` of type `j`, and an error is displayed if `inc` has no elements of type `j`. Calling the elements (of a given type) of `inc` yields an object in the category `IsElementsOfIncidenceStructure` (or the appropriate category for projective spaces and classical polar spaces), which does not imply that all elements are computed and stored. In an alternative form of this function `str` can be one of the strings found in the list obtained by calling `TypesOfElementsOfIncidenceStructurePlural(inc)`. E.g. for projective spaces, "points", "lines", "planes" or "solids" are the names for elements of type 1,2,3 or 4, respectively, of course if `inc` has elements of the deduced type.

```
───────────────────────── Example ─────────────────────────
  gap> ps := ProjectiveSpace(3,3);
  ProjectiveSpace(3, 3)
  gap> l := ElementsOfIncidenceStructure(ps,2);
  <lines of ProjectiveSpace(3, 3)>
  gap> ps := EllipticQuadric(5,9);
  Q-(5, 9)
  gap> lines := ElementsOfIncidenceStructure(ps,2);
  <lines of Q-(5, 9)>
  gap> planes := ElementsOfIncidenceStructure(ps,3);
  Error, <geo> has no elements of type <j> called from
  <function "unknown">( <arguments> )
   called from read-eval loop at line 12 of *stdin*
  you can 'quit;' to quit to outer loop, or
  you can 'return;' to continue
  brk> quit;
  gap> as := AffineSpace(3,9);
  AG(3, 9)
  gap> lines := ElementsOfIncidenceStructure(as,"lines");
  <lines of AG(3, 9)>
```

3.2.7 ElementsOfIncidenceStructure

▷ ElementsOfIncidenceStructure(`inc`)   (operation)

    Returns: a collection of elements

    `inc` must be an incidence structure, then this operation returns the collection of all elements of `inc`. Such a collection can e.g. be the range of a geometry morphism. Note that this operation has no method for generic incidence structures constructed using `Incidence Structure`.

3.2.8 Short names for ElementsOfIncidenceStructure

▷ Points(`inc`)   (operation)
▷ Lines(`inc`)   (operation)
▷ Planes(`inc`)   (operation)
▷ Solids(`inc`)   (operation)

    Returns: The points, lines, planes, solids, respectively of `inc`

    For geometries in `IsLieGeometry`, `IsAffineSpace`, and `IsGeneralisedPolygon`, the elements of type 1,2,3,4 respectively are called usually points, lines, planes, solids, respectively. These methods are, for such geometries, are shortcuts to `ElementsOfIncidenceStructure(inc,j)`, with `j` equal to 1,2,3,4, respectively.

```
 ─────────────────────────── Example ───────────────────────────
 gap> Points(HermitianVariety(2,64));
 <points of Hermitian Variety in ProjectiveSpace(2, 64)>
 gap> Lines(EllipticQuadric(5,2));
 <lines of Q-(5, 2)>
 gap> Planes(SymplecticSpace(7,3));
 <planes of W(7, 3)>
 gap> Lines(TwistedTrialityHexagon(2^3));
 <lines of T(8, 2)>
```

3.2.9 NrElementsOfIncidenceStructure

▷ NrElementsOfIncidenceStructure(`inc`, `j`)   (operation)
▷ NrElementsOfIncidenceStructure(`inc`, `str`)   (operation)

    Returns: a positive integer

    `inc` must be an incidence structure, `j` must be a type of element of `inc`. This function returns the number of elements of `inc` of type `j`, and an error is displayed if `inc` has no elements of type `j`. In the alternative form of this function `str` can be one of "points", "lines", "planes" or "solids" and the function returns the number of elements of type 1, 2, 3 or 4 respectively, of course if `inc` has elements of the deduced type. For geometries in the category `IsLieGeometry`, `IsAffineSpace`, and `IsGeneralisedPolygon`, the number of elements of a given type is known upon construction of the geometry. As such, for these geometries, this operation requires no computing time.

```
 ─────────────────────────── Example ───────────────────────────
 gap> ps:=ProjectiveSpace(4,3);
 ProjectiveSpace(4, 3)
 gap> NrElementsOfIncidenceStructure(ps, 2);
 1210
 gap> NrElementsOfIncidenceStructure(ps, "points");
 121
```

3.2.10   Random

▷ Random(`C`)   (operation)

    Returns: an element in the collection `C`

    `C` is a collection of elements of an incidence structure, i.e., an object in the category `IsElementsOfIncidenceStructure`. `Random(C)` will return a random element in `C` provided there is a method installed. The generic method will compute all elements in `C` and return a random member from the list. For e.g. Lie geometries, more efficient methods are installed.

```
                                    Example
  gap> coll := Hyperplanes(PG(5,7));
  <proj. 4-subspaces of ProjectiveSpace(5, 7)>
  gap> Random(coll);
  <a proj. 4-space in ProjectiveSpace(5, 7)>
```

3.2.11   IsIncident

▷ IsIncident(`u, v`)   (operation)
▷ *(`u, v`)   (operation)

    Returns: true or false

    `u` and `v` must be elements of an incidence structure. This function returns true if and only if `u` is incident with `v`. Recall that `IsIncident` is a symmetric relation, while in is not. A method for the operation * is installed, applicable to objects in `IsElementOfIncidenceStructure`. It just calls `IsIncident`.

```
                                    Example
  gap> p := Random(Points(PG(5,4)));
  <a point in ProjectiveSpace(5, 4)>
  gap> l := Random(Lines(p));
  <a line in ProjectiveSpace(5, 4)>
  gap> IsIncident(p,l);
  true
  gap> IsIncident(l,p);
  true
  gap> p * l;
  true
  gap> l * p;
  true
  gap> p * p;
  true
  gap> l * l;
  true
```

3.2.12   AmbientGeometry

▷ AmbientGeometry(`v`)   (operation)

    Returns: the ambient geometry of the element `v`

    If `v` is an element of an incidence geometry currently implemented in FinInG, then this operation returns the ambient geometry of `v`, i.e., in general the geometry in which `v` was created. If an inci-

dence structure is created with elements that are a subset of elements of another incidence structure, the ambient geometry might stay unchanged.

```
───────────────────── Example ─────────────────────
gap> plane := Random(Planes(HyperbolicQuadric(5,2)));
<a plane in Q+(5, 2)>
gap> AmbientGeometry(plane);
Q+(5, 2)
gap> l := Random(Lines(SplitCayleyHexagon(3)));
#I  for Split Cayley Hexagon
#I  Computing nice monomorphism...
#I  Found permutation domain...
<a line in H(3)>
gap> Print(l);
NewMatrix(IsCMatRep,GF(3,1),7,[[ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3),
  Z(3)^0, Z(3)^0 ],[ 0*Z(3), Z(3)^0, Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3) ],])
gap> AmbientGeometry(l);
H(3)
gap> p := Random(Points(EGQByBLTSet(BLTSetByqClan(LinearqClan(3)))));
#I  Now embedding dual BLT-set into W(5,q)...
#I  Computing elation group...
<a point in <EGQ of order [ 9, 3 ] and basepoint in W(5, 3 ) >>
gap> Print(p);
NewRowVector(IsCVecRep,GF(3,1),[Z(3)^0,Z(3),Z(3),Z(3),Z(3)^0,0*Z(3),])
gap> AmbientGeometry(p);
<EGQ of order [ 9, 3 ] and basepoint in W(5, 3 ) >
```

3.3   Flags of incidence structures

A *flag* of an incidence structure S is a set F of elements of S that are two by two incident. This implies that all elements in F have a different type. A flag is maximal if it cannot be extended with more elements. FinInG provides a basic category `IsFlagOfIncidenceStructure`. For different types of incidence structures, methods to create a flag can be installed. A *chamber* is a flag of size n, where n is the rank of the incidence structure. Recall that an incidence structure is an incidence geometry if every maximal flag is a chamber.

3.3.1   FlagOfIncidenceStructure

▷ FlagOfIncidenceStructure(*inc*, *l*)   (operation)

   Returns: the flag consisting of the elements of *inc* in the list *l*

   It is checked if all elements in *l* are incident and belong to the same incidence structure. An empty list is allowed.

```
───────────────────── Example ─────────────────────
gap> ps := PG(3,7);
ProjectiveSpace(3, 7)
gap> point := VectorSpaceToElement(ps,[1,2,0,0]*Z(7)^0);
<a point in ProjectiveSpace(3, 7)>
gap> line := VectorSpaceToElement(ps,[[1,0,0,0],[0,1,0,0]]*Z(7)^0);
<a line in ProjectiveSpace(3, 7)>
gap> plane := VectorSpaceToElement(ps,[[1,0,0,0],[0,1,0,0],[0,0,0,1]]*Z(7)^0);
```

```
<a plane in ProjectiveSpace(3, 7)>
gap> flag := FlagOfIncidenceStructure(ps,[point,line,plane]);
<a flag of ProjectiveSpace(3, 7)>
```

3.3.2 IsChamberOfIncidenceStructure

▷ IsChamberOfIncidenceStructure(`flag`)   (operation)

Returns: true if and only if `flag` contains an element of each type

The incidence structure is determined by the elements.

```
_____ Example _____
gap> ps := PG(3,7);
ProjectiveSpace(3, 7)
gap> point := VectorSpaceToElement(ps,[1,2,0,0]*Z(7)^0);
<a point in ProjectiveSpace(3, 7)>
gap> line := VectorSpaceToElement(ps,[[1,0,0,0],[0,1,0,0]]*Z(7)^0);
<a line in ProjectiveSpace(3, 7)>
gap> plane := VectorSpaceToElement(ps,[[1,0,0,0],[0,1,0,0],[0,0,0,1]]*Z(7)^0);
<a plane in ProjectiveSpace(3, 7)>
gap> flag1 := FlagOfIncidenceStructure(ps,[point,plane]);
<a flag of ProjectiveSpace(3, 7)>
gap> IsChamberOfIncidenceStructure(flag1);
false
gap> flag2 := FlagOfIncidenceStructure(ps,[point,line,plane]);
<a flag of ProjectiveSpace(3, 7)>
gap> IsChamberOfIncidenceStructure(flag2);
true
```

3.3.3 IsEmptyFlag

▷ IsEmptyFlag(`flag`)   (operation)

Returns: true or false

It is possible to construct the empty flag of an incidence structure. This operation tests whether a given flag is empty.

3.3.4 ElementsOfFlag

▷ ElementsOfFlag(`flag`)   (operation)

Returns: a list of elements

This operations simply returns the list of elements that define `flag`

```
_____ Example _____
gap> gp := SplitCayleyHexagon(4);
H(4)
gap> p := Random(Points(gp));
#I  for Split Cayley Hexagon
#I  Computing nice monomorphism...
#I  Found permutation domain...
<a point in H(4)>
gap> l := Random(Lines(p));
```

```
<a line in H(4)>
gap> flag := FlagOfIncidenceStructure(gp,[l,p]);
<a flag of H(4)>
gap> ElementsOfFlag(flag);
[ <a point in H(4)>, <a line in H(4)> ]
```

3.3.5 Rank

▷ Rank(*flag*)   (attribute)

   Returns: an integer
   This operations returns the number of elements that define *flag*
—————————————————— Example ——————————————————
```
gap> ps := ParabolicQuadric(8,3);
Q(8, 3)
gap> l := Random(Lines(ps));
<a line in Q(8, 3)>
gap> plane := Random(Planes(l));
<a plane in Q(8, 3)>
gap> solid := Random(Solids(plane));
<a solid in Q(8, 3)>
gap> flag := FlagOfIncidenceStructure(ps,[l,plane,solid]);
<a flag of Q(8, 3) >
gap> Rank(flag);
3
```

3.3.6 Size

▷ Size(*flag*)   (attribute)

   Returns: an integer
   This operations returns the number of elements that define *flag*
—————————————————— Example ——————————————————
```
gap> ps := SymplecticSpace(5,7);
W(5, 7)
gap> p := Random(Points(ps));
<a point in W(5, 7)>
gap> plane := Random(Planes(p));
<a plane in W(5, 7)>
gap> flag := FlagOfIncidenceStructure(ps,[p,p,plane]);
<a flag of W(5, 7) >
gap> Size(flag);
2
gap> ElementsOfFlag(flag);
[ <a point in W(5, 7)>, <a plane in W(5, 7)> ]
```

3.3.7 AmbientGeometry

▷ AmbientGeometry(*flag*)   (attribute)

   Returns: an incidence structure

This operations returns the ambient geometry of the `flag`

```
————————————— Example —————————————
gap> ps := SymplecticSpace(5,7);
W(5, 7)
gap> p := Random(Points(ps));
<a point in W(5, 7)>
gap> plane := Random(Planes(p));
<a plane in W(5, 7)>
gap> flag := FlagOfIncidenceStructure(ps,[p,p,plane]);
<a flag of W(5, 7) >
gap> Size(flag);
2
gap> ElementsOfFlag(flag);
[ <a point in W(5, 7)>, <a plane in W(5, 7)> ]
```

3.3.8 Type

▷ Type(`flag`)   (operation)

   Returns: an list of integers
   This operations returns the list of types of the elements defining `flag`

```
————————————— Example —————————————
gap> pg := PG(8,9);
ProjectiveSpace(8, 9)
gap> l := Random(Lines(pg));
<a line in ProjectiveSpace(8, 9)>
gap> s := Random(Solids(l));
<a solid in ProjectiveSpace(8, 9)>
gap> flag := FlagOfIncidenceStructure(pg,[l,s]);
<a flag of ProjectiveSpace(8, 9)>
gap> Type(flag);
[ 2, 4 ]
gap> p := Random(Points(pg));
<a point in ProjectiveSpace(8, 9)>
gap> flag := FlagOfIncidenceStructure(pg,[p]);
<a flag of ProjectiveSpace(8, 9)>
gap> Type(flag);
[ 1 ]
```

3.3.9 IsIncident

▷ IsIncident(`el`, `flag`)   (operation)
▷ IsIncident(`flag`, `el`)   (operation)

   Returns: true or false
   An element is incident with a flag if and only if it is incident with all elements defining the flag.

```
————————————— Example —————————————
gap> pg := PG(3,5);
ProjectiveSpace(3, 5)
gap> p := Random(Points(pg));
<a point in ProjectiveSpace(3, 5)>
```

```
gap> l := Random(Lines(p));
<a line in ProjectiveSpace(3, 5)>
gap> plane := Random(Planes(l));
<a plane in ProjectiveSpace(3, 5)>
gap> flag := FlagOfIncidenceStructure(pg,[l,plane]);
<a flag of ProjectiveSpace(3, 5)>
gap> IsIncident(flag,l);
true
gap> IsIncident(l,flag);
true
```

3.4 Shadow of elements

3.4.1 ShadowOfElement

▷ ShadowOfElement(*inc*, *v*, *str*)   (operation)
▷ ShadowOfElement(*inc*, *v*, *j*)   (operation)
    Returns: The collection of elements of type *str* or type *j* incident with *v*

    inc is an incidence structure, *v* must be an element of *inc*, *str* must be a string which is
THE PLURAL of the name of one of the types of the elements of *inc*. For the second variant,
j is an integer representing one of the types of the elements of *inc*. This first variant relies on
TypesOfElementsOfIncidenceStructurePlural and on a particular method installed for the second variant for particular incidence structures. The use of the argument *inc* makes it flexible, i.e., if
the element *v* can belong to different incidence structures, its shadow can be different, as the second
example shows.

```
——————————————— Example ———————————————
gap> ps := ProjectiveSpace(3,3);
ProjectiveSpace(3, 3)
gap> pi := Random(Planes(ps));
<a plane in ProjectiveSpace(3, 3)>
gap> lines := ShadowOfElement(ps,pi,"lines");
<shadow lines in ProjectiveSpace(3, 3)>
gap> Size(lines);
13

gap> p := Random(Points(PG(3,3)));
<a point in ProjectiveSpace(3, 3)>
gap> lines1 := ShadowOfElement(SymplecticSpace(3,3),p,2);
<shadow lines in W(3, 3)>
gap> Size(lines1);
4
gap> lines2 := ShadowOfElement(PG(3,3),p,2);
<shadow lines in ProjectiveSpace(3, 3)>
gap> Size(lines2);
13
```

3.4.2 ElementsIncidentWithElementOfIncidenceStructure

▷ ElementsIncidentWithElementOfIncidenceStructure(*v, j*) (operation)

    Returns: The collection of elements of type *j* incident with *v*

    This operation is applicable for objects *v* belonging to `IsElementOfIncidenceStructure`, and is a shortcut to `ShadowOfElement(AmbientGeometry(v),v,j)`.

3.4.3 ShadowOfFlag

▷ ShadowOfFlag(*inc, flag, str*) (operation)
▷ ShadowOfFlag(*inc, list, str*) (operation)
▷ ShadowOfFlag(*inc, flag, j*) (operation)
▷ ShadowOfFlag(*inc, list, j*) (operation)

    Returns: The collection of elements of type *str* or type *j* incident with all elements of *flag*, or with all elements of *list*

    Variants 2 and 4 convert *list* to a flag of *inc*, using `FlagOfIcidenceStructure`, which performs the necessary checks. Variants 1 and 2 rely on variants 3 and 4 respectively, for which a method must be installed for the particular incidence structure *inc*.

```
———————————————————— Example ————————————————————
gap> ps := PG(3,7);
ProjectiveSpace(3, 7)
gap> point := VectorSpaceToElement(ps,[1,2,0,0]*Z(7)^0);
<a point in ProjectiveSpace(3, 7)>
gap> plane := VectorSpaceToElement(ps,[[1,0,0,0],[0,1,0,0],[0,0,0,1]]*Z(7)^0);
<a plane in ProjectiveSpace(3, 7)>
gap> flag := FlagOfIncidenceStructure(ps,[point,plane]);
<a flag of ProjectiveSpace(3, 7)>
gap> lines := ShadowOfFlag(ps,flag,"lines");
<shadow lines in ProjectiveSpace(3, 7)>
```

3.4.4 ResidueOfFlag

▷ ResidueOfFlag(*flag*) (operation)

    Returns: an incidence structure

    Consider the flag *flag*, and its ambient geometry *G*. All elements of *G* incident with all elements of *flag*, together with the incidence of *G*, determine an incidence structure. This incidence structure is returned by this operation. Note that independently of the Category of *G*, the returned incidence structure is constructed using the operation `IncidenceStructure`.

```
———————————————————— Example ————————————————————
gap> pg := PG(4,5);
ProjectiveSpace(4, 5)
gap> p := Random(Points(pg));
<a point in ProjectiveSpace(4, 5)>
gap> l := Random(Lines(p));
<a line in ProjectiveSpace(4, 5)>
gap> flag := FlagOfIncidenceStructure(pg,[p,l]);
<a flag of ProjectiveSpace(4, 5)>
gap> res := ResidueOfFlag(flag);
Incidence structure of rank 2
```

```
gap> gamma := IncidenceGraph(res);;
gap> Diameter(gamma);
3
gap> Girth(gamma);
6
```

3.4.5   Short names for ElementsIncidentWithElementOfIncidenceStructure

▷ Points(*inc, v*)   (operation)
▷ Lines(*inc, v*)   (operation)
▷ Planes(*inc, v*)   (operation)
▷ Solids(*inc, v*)   (operation)
▷ Points(*v*)   (operation)
▷ Lines(*v*)   (operation)
▷ Planes(*v*)   (operation)
▷ Solids(*v*)   (operation)

   Returns: The collections of elements of `inc` of respective type 1, 2, 3 and 4, that are incident with *v*

   If `inc`, or the ambient geometry of *v* is an incidence structure, where the elements of type 1, 2, 3 and 4 are called "points", "lines", "planes", and "solids" respectively, these operations are shortcuts to `ShadowOfElement`. If this is not the case, a method for these operations is not installed.

```
────────────────────────── Example ──────────────────────────
 gap> line := Random(Lines(AG(5,4)));
 <a line in AG(5, 4)>
 gap> Points(line);
 <shadow points in AG(5, 4)>
 gap> Planes(line);
 <shadow planes in AG(5, 4)>
```

3.5   Enumerating elements of an incidence structure

In several situations, it can be usful to compute a complete list of objects statisfying one or more conditions. To list all elements of a given type of an incidence structure, is a typical example. FinInG provides functionality that is common in GAP for this purpose.

   In FinInG, typically a list of all elements satisfying a property, e.g. all points of a projective space, are represented by a GAP object that is a *collection*. The word 'collection' is important here. E.g. subspaces of a vector space are not calculated on calling `Subspaces`, rather primitive information is stored in an `IsComponentObjectRep`. So for example

```
────────────────────────── Example ──────────────────────────
 gap> v:=GF(31)^5;
 ( GF(31)^5 )
 gap> subs:=Subspaces(v,1);
 Subspaces( ( GF(31)^5 ), 1 )
```

For a given collection *C*, one can use the GAP function `List` to compute all objects in the collection *C*. Such a list can be used to iterate over all objects in the list. However, if one needs only few random

objects from `C`, or if one needs to iterate over the list of objects until a certain condition is satisfied, it can be highly inefficient to first compute all these objects. Therefore iterators and enumerators come into the picture.

An iterator is a GAP object that gives a user friendly way to loop over all elements without repetition. Only three operations are applicable on an iterator: `NextIterator`, `IsDoneIterator` and `ShallowCopy`. Clearly, all elements of a collection can be obtained by using an available iterator.

3.5.1 Iterator

▷ Iterator(*C*)   (operation)

Returns: an iterator for the collection `C`

`C` is a collection of elements of an incidence structure. An iterator is returned. The second example demonstrates how an iterator is used by `First`. Clearly, not all points of the projective space are computed.

```
————————————————————— Example —————————————————————
 gap> ps := PG(3,7);
 ProjectiveSpace(3, 7)
 gap> planes := Planes(ps);
 <planes of ProjectiveSpace(3, 7)>
 gap> iter := Iterator(planes);
 <iterator>
 gap> NextIterator(iter);
 <a plane in ProjectiveSpace(3, 7)>
 gap> IsDoneIterator(iter);
 false

 gap> pg := PG(12,81);
 ProjectiveSpace(12, 81)
 gap> pts := Points(pg);
 <points of ProjectiveSpace(12, 81)>
 gap> Size(pts);
 80763523615333416236653
 gap> ps := ParabolicQuadric(12,81);
 Q(12, 81)
 gap> First(pts,x->x in ps);
 <a point in ProjectiveSpace(12, 81)>
 gap> time;
 23
```

In its simplest form, an enumerator is just a list containing all the elements of the collection. Given any object in the list, it is possible to retrieve its number in the list (which is then just its position). Also, given any number between 1 and the length of the list, it is possible to get the corresponding element. For some collections of elements of particular incidence structures, a more advanced version of enumerators is implemented. Such an advanced version is an object containing the two functions `ElementNumber` and `NumberElement`. Such functions are able to compute directly, without listing all elements, the element with a given number, or, conversely, compute directly the number of a given element. Clearly, using an enumerator, it is possible to obtain a list containing all elements of a collection.

3.5.2 Enumerator

▷ Enumerator(*C*)   (operation)

    Returns: an enumerator for the collection *C*

    C is a collection of elements of an incidence structure. An enumerator is returned. The second example demonstrates how an enumerator is used by Random. Clearly, not all points of the polar space are to obtain an random point.

```
 ──────────────────────────── Example ────────────────────────────
 gap> lines := Lines( ParabolicQuadric(6,3) );
 <lines of Q(6, 3)>
 gap> enum := Enumerator( lines );
 EnumeratorOfSubspacesOfClassicalPolarSpace( <lines of Q(6, 3)> )
 gap> s := Size(enum);
 3640
 gap> n := Random([1..s]);
 3081
 gap> l := enum[n];
 <a line in Q(6, 3)>
 gap> Position(enum, l);
 3081

 gap> ps := ParabolicQuadric(16,7^4);
 Q(16, 2401)
 gap> pts := Points(ps);
 <points of Q(16, 2401)>
 gap> Size(pts);
 5082335365149315417244057760679049253148397058880 16
 gap> Random(pts);
 <a point in Q(16, 2401)>
 gap> time;
 565
```

    When an iterator or enumerator is installed for a collection, List can be used to obtain all objects in that collection.

3.5.3 List

▷ List(*C*)   (operation)

    Returns: all objects in the collection *C*

```
 ──────────────────────────── Example ────────────────────────────
 gap> pg := PG(2,2);
 ProjectiveSpace(2, 2)
 gap> List(Points(pg));
 [ <a point in ProjectiveSpace(2, 2)>, <a point in ProjectiveSpace(2, 2)>,
   <a point in ProjectiveSpace(2, 2)>, <a point in ProjectiveSpace(2, 2)>,
   <a point in ProjectiveSpace(2, 2)>, <a point in ProjectiveSpace(2, 2)>,
   <a point in ProjectiveSpace(2, 2)> ]
 gap> List(Lines(pg));
 [ <a line in ProjectiveSpace(2, 2)>, <a line in ProjectiveSpace(2, 2)>,
   <a line in ProjectiveSpace(2, 2)>, <a line in ProjectiveSpace(2, 2)>,
   <a line in ProjectiveSpace(2, 2)>, <a line in ProjectiveSpace(2, 2)>,
```

```
    <a line in ProjectiveSpace(2, 2)> ]
gap> ps := ParabolicQuadric(6,2);
Q(6, 2)
gap> lines := List(Lines(ps));
[ <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
```

```
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
<a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
```

```
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)> ]
gap> time;
3661
gap> Size(lines);
315
```

From the example in 3.5.3 it can be noted that although the number of lines is only 315, already 3 seconds of computing time are needed. This is typically observed for elements of classicla polar spaces different from points or generators, and is explained by the recursive nature of the implementation of the enumerator. Computing a list of these objects as an orbit under the collineation group is, even taking into account the computation of the nice monomorphism of the collineation group, often much faster. Given a collection C, the function AsList performs this computation using the collineation group.

3.5.4 AsList

▷ AsList(*C*) (operation)

Returns: an ORB object containing all objects in C

The returned object is an ORB object. All objects can easily be obtained by applying the operation List.

——————————————— Example ———————————————
```
gap> ps := ParabolicQuadric(6,2);
Q(6, 2)
gap> lines := AsList(Lines(ps));
<closed orbit, 315 points>
gap> time;
58
gap> List(lines);
[ <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
  <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
```

```
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
    <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
```

```
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)>,
      <a line in Q(6, 2)>, <a line in Q(6, 2)>, <a line in Q(6, 2)> ]
gap> time;
1
```

3.6   Lie geometries

3.6.1   Main categories in `IsLieGeometry`

▷ `IsProjectiveSpace` (Category)
▷ `IsClassicalPolarSpace` (Category)

Lie geometries bundle projective spaces and classical polar spaces together. In the future, more subcategories could be added since the term "Lie geometry" refers to a geometry whose automorphism group lies in some group of Lie type. Both classes of geometries have their category, as a subcategory of `IsLieGeometry`.

One common fact of Lie geometries is that their elements are represented by subspaces of a vector space. In these geometries, incidence is symmetrized set-theoretic containment. In this section we describe methods that are declared in a generic way for (elements of) Lie geometries. These operations

are applicable to Lie geometries and related objects.

3.6.2 AmbientSpace

▷ AmbientSpace(`ig`)   (attribute)

    Returns: the ambient projective space of a Lie geometry `ig`

```
———————————————— Example ————————————————
gap> AmbientSpace(PG(3,4));
ProjectiveSpace(3, 4)
gap> AmbientSpace(ParabolicQuadric(4,4));
ProjectiveSpace(4, 4)
gap> AmbientSpace(SplitCayleyHexagon(3));
ProjectiveSpace(6, 3)
```

3.6.3 UnderlyingVectorSpace

▷ UnderlyingVectorSpace(`ig`)   (operation)

    Returns: the underlying vectorspace of the Lie geometry `ig`

```
———————————————— Example ————————————————
gap> UnderlyingVectorSpace(PG(5,4));
( GF(2^2)^6 )
gap> UnderlyingVectorSpace(HermitianPolarSpace(4,4));
( GF(2^2)^5 )
```

3.6.4 ProjectiveDimension

▷ ProjectiveDimension(`ig`)   (operation)

    Returns: the projective dimension of the ambient projective space of `ig`

```
———————————————— Example ————————————————
gap> ProjectiveDimension(PG(7,7));
7
gap> ProjectiveDimension(EllipticQuadric(5,2));
5
```

Mathematically, it makes sense to implement an object representing the empty subspace, since this is typically obtained as a result of a Meet operation, which computes the intersection of two or more elements. On the other hand, we do not consider the empty subspace as an element of the incidence geometry. Hence, using the empty subspace as an argument of IsIncident (and consequently of *), will result in a "no method found" error.

3.6.5 IsEmptySubspace

▷ IsEmptySubspace   (Category)

Category for objects representing the empty subspace of a particular Lie geometry. Empty subspaces of different geometries will be different objects, and have a different ambient geometry.

3.7 Elements of Lie geometries

Elements of a Lie geometry are constructed using a list of vectors. The methods installed for the particular Lie geometries check whether the subspace of the vector space represents an element of the Lie geometry.

3.7.1 VectorSpaceToElement

▷ VectorSpaceToElement(*ig, v*)   (operation)

▷ VectorSpaceToElement(*ig, l*)   (operation)

Returns: the element of *ig*, represented by the subspace spanned by *v* or *l*, or returns the empty subspace.

The first variant of this operation takes as second argument a vector of the underlying vector space of *ig*. Such a vector possibly represents a point of *ig*. The second variant takes as second argument a list of vectors in the underlying vector space of *ig*. Such a list represents a subspace of the vector space. If the dimension of the subspace generated by *l* is larger than zero and strictly less than the dimension of the vector space, it is checked if the subspace represents an element of *ig*, except when *ig* is a projective space. If *l* is a list of vectors generating the whole vector space, then *ig* is returned if and only if *ig* is a projective space, otherwise an error is produced. An empty list is not allowed as second argument.

```
———————— Example ————————
gap> v := [1,1,1,0,0,0]*Z(7)^0;
[ Z(7)^0, Z(7)^0, Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7) ]
gap> w := [0,0,0,1,1,1]*Z(7)^0;
[ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0, Z(7)^0, Z(7)^0 ]
gap> VectorSpaceToElement(PG(5,7),v);
<a point in ProjectiveSpace(5, 7)>
gap> VectorSpaceToElement(PG(5,7),[v,w]);
<a line in ProjectiveSpace(5, 7)>
gap> VectorSpaceToElement(SymplecticSpace(5,7),v);
<a point in W(5, 7)>
gap> VectorSpaceToElement(SymplecticSpace(5,7),[v,w]);
Error, <x> does not generate an element of <geom> called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 13 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> VectorSpaceToElement(HyperbolicQuadric(5,7),v);
Error, <v> does not generate an element of <geom> called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 13 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> VectorSpaceToElement(HyperbolicQuadric(5,7),0*v);
< empty subspace >
```

3.7.2 UnderlyingObject

▷ UnderlyingObject(*v*)   (operation)

Returns: a CVEC object, which is the vector or matrix representing the element *v*

The argument *v* must be an element, so it is not allowed that *v* is the empty subspace, or just a projective space. Note that Unpack can be used to convert the CVEC object into a usual GAP vector or matrix.

```
——————————————————————— Example ———————————————————————
  gap> l := Random(Lines(PG(4,3)));
  <a line in ProjectiveSpace(4, 3)>
  gap> UnderlyingObject(l);
  <cmat 2x5 over GF(3,1)>
  gap> Unpack(last);
  [ [ Z(3)^0, Z(3), 0*Z(3), 0*Z(3), Z(3) ],
    [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3)^0, 0*Z(3) ] ]
```

3.7.3 \in

▷ \in(*u*, *v*)   (operation)

Returns: true if and only if the element *u* is set-theoretically contained in the element *w*

Both arguments must be elements of the same Lie geometry. The empty subspace and a Lie geometry are also allowed as arguments. This relation is not symmetric, and the methods for IsIncident use this method to test incidence between elements.

```
——————————————————————— Example ———————————————————————
  gap> p := VectorSpaceToElement(PG(3,3),[1,0,0,0]*Z(3)^0);
  <a point in ProjectiveSpace(3, 3)>
  gap> l := VectorSpaceToElement(PG(3,3),[[1,0,0,0],[0,1,0,0]]*Z(3)^0);
  <a line in ProjectiveSpace(3, 3)>
  gap> p in l;
  true
  gap> p in p;
  true
  gap> l in p;
  false
  gap> l in PG(3,3);
  true
```

3.7.4 More short names for ElementsIncidentWithElementOfIncidenceStructure

▷ Hyperplanes(*inc*, *v*)   (operation)
▷ Hyperplanes(*v*)   (operation)

Returns: the elements of type $j - 1$ incident with *v*, which is an element of type *j*

This operation is a shortcut to the operation ShadowOfElement, where the geometry is taken from *v*, and where the elements of type one less than the type of *v* are asked. *v* is allowed to be a complete projective space here, yielding the hyperplanes of that space.

```
——————————————————————— Example ———————————————————————
  gap> pg := PG(3,7);
  ProjectiveSpace(3, 7)
```

```
gap> hyp := Random(Hyperplanes(pg));
<a plane in ProjectiveSpace(3, 7)>
gap> h1 := Random(Hyperplanes(hyp));
<a line in ProjectiveSpace(3, 7)>
gap> h2 := Random(Hyperplanes(h1));
<a point in ProjectiveSpace(3, 7)>
gap> ps := SymplecticSpace(7,3);
W(7, 3)
gap> solid := Random(Solids(ps));
<a solid in W(7, 3)>
gap> plane := Random(Hyperplanes(solid));
<a plane in W(7, 3)>
```

3.8 Changing the ambient geometry of elements of a Lie geometry

A Lie geometry, i.e., an object in the category `IsLieGeometry`, is naturally embedded in a projective space. This is of course in a mathematical sense. In FinInG, certain embeddings are implemented by providing a mapping between geometries. The Lie geometries are in some sense *hard wired* embedded, just simply because a category containing elements of a Lie geometry, is always a subcategory of `IsSubspaceOfProjectiveSpace`. As a consequence, operations applicable to objects in the category `IsSubspaceOfProjectiveSpace` are by default applicable to objects in any subcategory, so to elements of any Lie geometry. When dealing with elements of e.g. different polar spaces in the same projective space, this yields a natural way of working with them, and investigating relations between them, without bothering about all necessary mappings. On the other hand, in some situations, it is impossible to decide in which geometry an element has to be considered. An easy example is the following. Consider two different quadrics in the same projective space. The intersection of two elements, one of each quadric, is clearly an element of the ambient projective space. But also of both quadrics. Without extra input of the user, the system cannot decide in which geometry to construct the intersection. To avoid complicated methods with many arguments, in such situations, the resulting element will be constructed in the common ambient projective space. Only in clear situations, where the ambient geometry of all elements is the same, and equal to the geometry of the resulting element, the resulting element will be constructed in this common geometry. We provide however conversion operations for elements of Lie gometries.

3.8.1 ElementToElement

▷ ElementToElement(*ps, el*)   (operation)

▷ Embed(*ps, el*)   (operation)

  Returns: *el* as an element of *ps*

  Let *ps* be any Lie geometry. This method returns `VectorSpaceToElement(ps,ElementToVectorSpace(el))`, if the conversion is possible. Embed is declared as a synonym of `ElementToElement`.

```
————————————————— Example —————————————————
gap> p := VectorSpaceToElement(PG(3,7),[0,1,0,0]*Z(7)^0);
<a point in ProjectiveSpace(3, 7)>
gap> q := ElementToElement(HyperbolicQuadric(3,7),p);
<a point in Q+(3, 7)>
```

```
gap> r := VectorSpaceToElement(PG(3,7),[1,1,0,0]*Z(7)^0);
<a point in ProjectiveSpace(3, 7)>
gap> ElementToElement(HyperbolicQuadric(3,7),r);
Error, <v> does not generate an element of <geom> called from
VectorSpaceToElement( geom, Unpack( v ) ) called from
VectorSpaceToElement( ps, UnderlyingObject( el ) ) called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 11 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

Chapter 4

Projective Spaces

In this chapter we describe how to use FinInG to work with finite projective spaces.

4.1 Projective Spaces and basic operations

A *projective space* is a point-line incidence geometry, satisfying a few well known axioms. An axiomatic treatment can be found in [VY65a] and [VY65b]. In FinInG, we deal with *finite Desarguesian projective spaces*. It is well known that these geometries can be described completely using vector spaces over finite fields. The elements of the projective space are all nontrivial subspaces of the vector space. So the projective points are the one-dimensional subspaces, the projective lines are the two-dimensional subspaces, and so on. From the axiomatic point of view, a projective space is a point-line geometry, and has rank at least 2. But a projective line is obtained if we start with a two dimensional vector space. Starting with a one dimensional vector space yields a single projective point. Both examples are not a projective space in the axiomatic point of view, but in FinInG they are considered as projective spaces.

4.1.1 IsProjectiveSpace

▷ IsProjectiveSpace   (Category)

   This category is a subcategory of `IsLieGeometry`, and contains all finite Desarguesian projective spaces.

   We refer the reader to [HT91] for the necessary background theory in case it is not provided in the manual.

4.1.2 ProjectiveSpace

▷ ProjectiveSpace(*d*, *F*)   (operation)
▷ ProjectiveSpace(*d*, *q*)   (operation)
▷ PG(*d*, *q*)   (operation)
   Returns: a projective space
   d must be a positive integer. In the first form, *F* is a field and the function returns the projective space of dimension *d* over *F*. In the second form, *q* is a prime power specifying the size of the field. The user may also use an alias, namely, the common abbreviation `PG(d, q)`.

```
——————————————— Example ———————————————
gap> ProjectiveSpace(3,GF(3));
ProjectiveSpace(3, 3)
gap> ProjectiveSpace(3,3);
ProjectiveSpace(3, 3)
```

4.1.3   ProjectiveDimension

▷ ProjectiveDimension(*ps*)   (attribute)
▷ Dimension(*ps*)   (attribute)
▷ Rank(*ps*)   (attribute)
   Returns: the projective dimension of the projective space *ps*
```
——————————————— Example ———————————————
gap> ps := PG(5,8);
ProjectiveSpace(5, 8)
gap> ProjectiveDimension(ps);
5
gap> Dimension(ps);
5
gap> Rank(ps);
5
```

4.1.4   BaseField

▷ BaseField(*ps*)   (operation)
   Returns: returns the base field for the projective space *ps*
```
——————————————— Example ———————————————
gap> BaseField(ProjectiveSpace(3,81));
GF(3^4)
```

4.1.5   UnderlyingVectorSpace

▷ UnderlyingVectorSpace(*ps*)   (operation)
   Returns: a vector space
   If *ps* is a projective space of dimension n over the field of order q, then this operation simply
returns the underlying vector space, i.e. the $n + 1$ dimensional vector space over the field of order q.
```
——————————————— Example ———————————————
gap> ps := ProjectiveSpace(4,7);
ProjectiveSpace(4, 7)
gap> vs := UnderlyingVectorSpace(ps);
( GF(7)^5 )
```

4.1.6 AmbientSpace

▷ AmbientSpace(*ps*) (attribute)

    Returns: a projective space

    The ambient space of a projective space *ps* is the projective space itself. Hence, simply *ps* will be returned.

4.2 Subspaces of projective spaces

The elements of a projective space $PG(n,q)$ are the subspaces of a suitable dimension. The empty subspace, also called the trivial subspace, has dimension -1 and corresponds to the zero dimensional vector subspace of the underlying vector space of $PG(n,q)$, and is hence represented by the zero vector of length $n+1$ over the underlying field $GF(q)$. The trivial subspace and the whole projective space are mathematically considered as a subspace of the projective geometry, but not as elements of the incidence geometry, and hence do in FinInG NOT belong to the category `IsSubspaceOfProjectiveSpace`.

4.2.1 VectorSpaceToElement

▷ VectorSpaceToElement(*geo, v*) (operation)

    Returns: an element

    geo is a projective space, and *v* is either a row vector (for points) or an *mxn* matrix (for an $(m-1)$-subspace of projective space of dimension $n-1$). In the case that *v* is a matrix, the rows represent generators for the subspace. An exceptional case is when *v* is a zero-vector, in which case the trivial subspace is returned.

```
────────────────── Example ──────────────────
 gap> ps := ProjectiveSpace(6,7);
 ProjectiveSpace(6, 7)
 gap> v := [3,5,6,0,3,2,3]*Z(7)^0;
 [ Z(7), Z(7)^5, Z(7)^3, 0*Z(7), Z(7), Z(7)^2, Z(7) ]
 gap> p := VectorSpaceToElement(ps,v);
 <a point in ProjectiveSpace(6, 7)>
 gap> Display(p);
 [142.131]
 gap> ps := ProjectiveSpace(3,4);
 ProjectiveSpace(3, 4)
 gap> v := [1,1,0,1]*Z(4)^0;
 [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ]
 gap> p := VectorSpaceToElement(ps,v);
 <a point in ProjectiveSpace(3, 4)>
 gap> mat := [[1,0,0,1],[0,1,1,0]]*Z(4)^0;
 [ [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ], [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ] ]
 gap> line := VectorSpaceToElement(ps,mat);
 <a line in ProjectiveSpace(3, 4)>
 gap> e := VectorSpaceToElement(ps,[]);
 Error, <v> does not represent any element called from
 <function "unknown">( <arguments> )
  called from read-eval loop at line 17 of *stdin*
 you can 'quit;' to quit to outer loop, or
 you can 'return;' to continue
```

```
    brk> quit;
```

4.2.2 EmptySubspace

▷ EmptySubspace(*ps*)   (operation)

Returns: the trivial subspace in the projective *ps*

The object returned by this operation is contained in every projective subspace of the projective space *ps*, but is not an element of *ps*. Hence, testing incidence results in an error message.

```
─────────────────────────────── Example ───────────────────────────────
  gap> e := EmptySubspace(PG(5,9));
  < empty subspace >
  gap> p := VectorSpaceToElement(PG(5,9),[1,0,0,0,0,0]*Z(9)^0);
  <a point in ProjectiveSpace(5, 9)>
  gap> e*p;
  Error, no method found! For debugging hints type ?Recovery from NoMethodFound
  Error, no 1st choice method found for '*' on 2 arguments called from
  <function "HANDLE_METHOD_NOT_FOUND">( <arguments> )
   called from read-eval loop at line 10 of *stdin*
  you can 'quit;' to quit to outer loop, or
  you can 'return;' to continue
  brk> quit;
  gap> e in p;
  true
```

4.2.3 ProjectiveDimension

▷ ProjectiveDimension(*sub*)   (operation)

Returns: the projective dimension of a subspace of a projective space. The operation ProjectiveDimension is also applicable on the EmptySubspace.

```
─────────────────────────────── Example ───────────────────────────────
  gap> ps := PG(2,5);
  ProjectiveSpace(2, 5)
  gap> v := [[1,1,0],[0,3,2]]*Z(5)^0;
  [ [ Z(5)^0, Z(5)^0, 0*Z(5) ], [ 0*Z(5), Z(5)^3, Z(5) ] ]
  gap> line := VectorSpaceToElement(ps,v);
  <a line in ProjectiveSpace(2, 5)>
  gap> ProjectiveDimension(line);
  1
  gap> Dimension(line);
  1
  gap> p := VectorSpaceToElement(ps,[1,2,3]*Z(5)^0);
  <a point in ProjectiveSpace(2, 5)>
  gap> ProjectiveDimension(p);
  0
  gap> Dimension(p);
  0
  gap> ProjectiveDimension(EmptySubspace(ps));
  -1
```

4.2.4   ElementsOfIncidenceStructure

▷ ElementsOfIncidenceStructure(*ps, j*)   (operation)

    Returns: the collection of elements of the projective space *ps* of type *j*

    For the projective space *ps* of dimension *d* and the type *j* (where $1 \le j \le d$), this operation returns the collection of $j-1$ dimensional subspaces. An error message is produced when the projective space *ps* has no elements of the required type.

```
───────────────────────────── Example ─────────────────────────────
gap> ps := ProjectiveSpace(6,7);
ProjectiveSpace(6, 7)
gap> ElementsOfIncidenceStructure(ps,1);
<points of ProjectiveSpace(6, 7)>
gap> ElementsOfIncidenceStructure(ps,2);
<lines of ProjectiveSpace(6, 7)>
gap> ElementsOfIncidenceStructure(ps,3);
<planes of ProjectiveSpace(6, 7)>
gap> ElementsOfIncidenceStructure(ps,4);
<solids of ProjectiveSpace(6, 7)>
gap> ElementsOfIncidenceStructure(ps,5);
<proj. 4-subspaces of ProjectiveSpace(6, 7)>
gap> ElementsOfIncidenceStructure(ps,6);
<proj. 5-subspaces of ProjectiveSpace(6, 7)>
gap> ElementsOfIncidenceStructure(ps,7);
Error, <ps> has no elements of type <j> called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 15 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

4.2.5   Short names for ElementsOfIncidenceStructure

▷ Points(*ps*)   (operation)
▷ Lines(*ps*)   (operation)
▷ Planes(*ps*)   (operation)
▷ Solids(*ps*)   (operation)
▷ Hyperplanes(*ps*)   (operation)

    Returns: The elements of *ps* of respective type 1, 2, 3, 4, and the hyperplanes

    An error message is produced when the projective space *ps* has no elements of a required type.

```
───────────────────────────── Example ─────────────────────────────
gap> ps := PG(6,13);
ProjectiveSpace(6, 13)
gap> Points(ps);
<points of ProjectiveSpace(6, 13)>
gap> Lines(ps);
<lines of ProjectiveSpace(6, 13)>
gap> Planes(ps);
<planes of ProjectiveSpace(6, 13)>
gap> Solids(ps);
<solids of ProjectiveSpace(6, 13)>
```

```
gap> Hyperplanes(ps);
<proj. 5-subspaces of ProjectiveSpace(6, 13)>
gap> ps := PG(2,2);
ProjectiveSpace(2, 2)
gap> Hyperplanes(ps);
<lines of ProjectiveSpace(2, 2)>
```

4.2.6   Incidence and containment

▷ IsIncident(*el1, el2*)   (operation)
▷ *(*el1, el2*)   (operation)
▷ \in(*el1, el2*)   (operation)

Returns: true or false

Recall that for projective spaces, incidence is symmetrized containment, where the empty sub-space and the whole projective space are excluded as arguments for this operation, since they are not considered as elements of the geometry, but both the empty subspace and the whole projective space are allowed as arguments for \in.

―――――――――――― Example ――――――――――――
```
gap> ps := ProjectiveSpace(5,9);
ProjectiveSpace(5, 9)
gap> p := VectorSpaceToElement(ps,[1,1,1,1,0,0]*Z(9)^0);
<a point in ProjectiveSpace(5, 9)>
gap> l := VectorSpaceToElement(ps,[[1,1,1,1,0,0],[0,0,0,0,1,0]]*Z(9)^0);
<a line in ProjectiveSpace(5, 9)>
gap> plane := VectorSpaceToElement(ps,[[1,0,0,0,0,0],[0,1,0,0,0,0],[0,0,1,0,0,0]]*Z(9)^0);
<a plane in ProjectiveSpace(5, 9)>
gap> p * l;
true
gap> l * p;
true
gap> IsIncident(p,l);
true
gap> p in l;
true
gap> l in p;
false
gap> p * plane;
false
gap> l * plane;
false
gap> l in plane;
false
gap> e := EmptySubspace(ps);
< empty subspace >
gap> e * l;
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 1st choice method found for '*' on 2 arguments called from
<function "HANDLE_METHOD_NOT_FOUND">( <arguments> )
 called from read-eval loop at line 21 of *stdin*
you can 'quit;' to quit to outer loop, or
```

```
  you can 'return;' to continue
  brk> quit;
  gap> e in l;
  true
  gap> l in ps;
  true
```

4.2.7 StandardFrame

▷ StandardFrame(*ps*) (operation)

Returns: the standard frame of the projective space *ps*

```
                           ———————— Example ————————
  gap> StandardFrame(PG(5,4));
  [ <a point in ProjectiveSpace(5, 4)>, <a point in ProjectiveSpace(5, 4)>,
    <a point in ProjectiveSpace(5, 4)>, <a point in ProjectiveSpace(5, 4)>,
    <a point in ProjectiveSpace(5, 4)>, <a point in ProjectiveSpace(5, 4)>,
    <a point in ProjectiveSpace(5, 4)> ]
  gap> Display(last);
  [ NewRowVector(IsCVecRep,GF(2,2),[Z(2)^0,0*Z(2),0*Z(2),0*Z(2),0*Z(2),0*Z(2),])
      , NewRowVector(IsCVecRep,GF(2,2),[0*Z(2),Z(2)^0,0*Z(2),0*Z(2),0*Z(2),
      0*Z(2),]), NewRowVector(IsCVecRep,GF(2,2),[0*Z(2),0*Z(2),Z(2)^0,0*Z(2),
      0*Z(2),0*Z(2),]), NewRowVector(IsCVecRep,GF(2,2),[0*Z(2),0*Z(2),0*Z(2),
      Z(2)^0,0*Z(2),0*Z(2),]), NewRowVector(IsCVecRep,GF(2,2),[0*Z(2),0*Z(2),
      0*Z(2),0*Z(2),Z(2)^0,0*Z(2),]), NewRowVector(IsCVecRep,GF(2,2),[0*Z(2),
      0*Z(2),0*Z(2),0*Z(2),0*Z(2),Z(2)^0,]), NewRowVector(IsCVecRep,GF(2,2),[
      Z(2)^0,Z(2)^0,Z(2)^0,Z(2)^0,Z(2)^0,Z(2)^0,]) ]
```

4.2.8 Coordinates

▷ Coordinates(*p*) (operation)

Returns: the homogeneous coordinates of the projective point *p*

```
                           ———————— Example ————————
  gap> p := Random(Points(PG(5,16)));
  <a point in ProjectiveSpace(5, 16)>
  gap> Coordinates(p);
  [ Z(2)^0, Z(2^4)^13, Z(2)^0, Z(2^4)^8, Z(2^4)^3, Z(2^4)^7 ]
```

4.2.9 DualCoordinatesOfHyperplane

▷ DualCoordinatesOfHyperplane(*hyp*) (operation)

Returns: a list

The argument *hyp* is a hyperplane of a projective space. This operation returns the dual coordinates of the hyperplane *hyp*, i.e. the list with the coefficients of the equation defining the hyperplane *hyp* as an algebraic variety.

4.2.10 HyperplaneByDualCoordinates

▷ HyperplaneByDualCoordinates(*pg*, *list*)   (operation)

    Returns: a hyperplane of a projective space

    The argument *pg* is a projective space, and *list* is the coordinate vector of a point of *pg*. This operation returns the hyperplane that has *list* as the list of coefficients of the equation defining the hyperplane as an algebraic variety.

4.2.11 EquationOfHyperplane

▷ EquationOfHyperplane(*h*)   (operation)

    Returns: the equation of the hyperplane *h* of a projective space

———————————————— Example ————————————————
```
gap> hyperplane := VectorSpaceToElement(PG(3,2),[[1,1,0,0],[0,0,1,0],[0,0,0,1]]*Z(2)^0);
<a plane in ProjectiveSpace(3, 2)>
gap> EquationOfHyperplane(hyperplane);
x_1+x_2
```

4.2.12 AmbientSpace

▷ AmbientSpace(*el*)   (operation)

    Returns: returns the ambient space of an element *el* of a projective space

    This operation is also applicable on the empty subspace and the whole space.

———————————————— Example ————————————————
```
gap> ps := PG(3,27);
ProjectiveSpace(3, 27)
gap> p := VectorSpaceToElement(ps,[1,2,1,0]*Z(3)^3);
<a point in ProjectiveSpace(3, 27)>
gap> AmbientSpace(p);
ProjectiveSpace(3, 27)
```

4.2.13 BaseField

▷ BaseField(*el*)   (operation)

    Returns: returns the base field of an element *el* of a projective space

    This operation is also applicable on the trivial subspace and the whole space.

———————————————— Example ————————————————
```
gap> ps := PG(5,8);
ProjectiveSpace(5, 8)
gap> p := VectorSpaceToElement(ps,[1,1,1,0,0,1]*Z(2));
<a point in ProjectiveSpace(5, 8)>
gap> BaseField(p);
GF(2^3)
```

4.2.14 Random

▷ Random(*elements*) (operation)

    Returns: a random element from the collection *elements*

    The collection *elements* is an object in the category `IsElementsOfIncidenceStructure`, i.e. an object representing the set of elements of a certain incidence structure of a given type. The latter information can be derived e.g. using `AmbientSpace` and `Type`.

──────────── Example ────────────
```
gap> ps := PG(9,49);
ProjectiveSpace(9, 49)
gap> Random(Lines(ps));
<a line in ProjectiveSpace(9, 49)>
gap> Random(Points(ps));
<a point in ProjectiveSpace(9, 49)>
gap> Random(Solids(ps));
<a solid in ProjectiveSpace(9, 49)>
gap> Random(Hyperplanes(ps));
<a proj. 8-space in ProjectiveSpace(9, 49)>
gap> elts := ElementsOfIncidenceStructure(ps,6);
<proj. 5-subspaces of ProjectiveSpace(9, 49)>
gap> Random(elts);
<a proj. 5-space in ProjectiveSpace(9, 49)>
gap> Display(last);
[[Z(7)^0,0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7^2)^14,Z(7^2)^44,Z(7^2)^14,
Z(7)^5,]
 [0*Z(7),Z(7)^0,0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7^2)^29,Z(7^2)^13,Z(7^2)^19,
Z(7^2)^27,]
 [0*Z(7),0*Z(7),Z(7)^0,0*Z(7),0*Z(7),0*Z(7),Z(7^2)^20,Z(7^2)^10,Z(7^2)^18,
Z(7^2)^27,]
 [0*Z(7),0*Z(7),0*Z(7),Z(7)^0,0*Z(7),0*Z(7),Z(7),Z(7^2)^30,Z(7^2)^18,Z(7^2)^
14,]
 [0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7)^0,0*Z(7),Z(7^2)^10,Z(7^2)^28,Z(7^2)^47,
Z(7^2)^29,]
 [0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7)^0,Z(7^2)^9,Z(7^2)^42,Z(7^2)^34,
Z(7^2)^25,]
]
gap> RandomSubspace(ps,3);
<a solid in ProjectiveSpace(9, 49)>
gap> Display(last);
[[Z(7)^0,0*Z(7),0*Z(7),0*Z(7),Z(7^2)^17,Z(7^2)^33,Z(7^2)^4,0*Z(7),Z(7^2),
Z(7^2)^33,]
 [0*Z(7),Z(7)^0,0*Z(7),0*Z(7),Z(7^2)^30,Z(7)^2,Z(7)^3,Z(7)^0,Z(7^2)^20,Z(7^2)^
42,]
 [0*Z(7),0*Z(7),Z(7)^0,0*Z(7),Z(7^2)^20,Z(7^2)^30,Z(7^2)^11,Z(7^2)^39,Z(7)^3,
Z(7),]
 [0*Z(7),0*Z(7),0*Z(7),Z(7)^0,Z(7^2)^21,Z(7)^0,Z(7^2)^11,Z(7^2)^45,Z(7^2),
Z(7^2)^9,]
]
gap> RandomSubspace(ps,7);
<a proj. 7-space in ProjectiveSpace(9, 49)>
gap> Display(last);
[[Z(7)^0,0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7^2)^42,Z(7^2)^
35,]
```

```
 [0*Z(7),Z(7)^0,0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7^2)^43,Z(7^2),]
  [0*Z(7),0*Z(7),Z(7)^0,0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7^2)^44,Z(7)^4,]
  [0*Z(7),0*Z(7),0*Z(7),Z(7)^0,0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7^2)^41,Z(7^2)^
 10,]
  [0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7)^0,0*Z(7),0*Z(7),0*Z(7),Z(7^2)^37,Z(7^2)^
 12,]
  [0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7)^0,0*Z(7),0*Z(7),Z(7^2)^11,Z(7^2)^
 39,]
  [0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7)^0,0*Z(7),Z(7^2)^22,Z(7^2)^
 10,]
  [0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),0*Z(7),Z(7)^0,Z(7^2)^43,Z(7^2)^
 22,]
 ]
gap> RandomSubspace(ps);
<a plane in ProjectiveSpace(9, 49)>
gap> RandomSubspace(ps);
<a proj. 6-space in ProjectiveSpace(9, 49)>
```

4.2.15 RandomSubspace

▷ RandomSubspace(ps, i) (operation)
▷ RandomSubspace(ps) (operation)

Returns: the first variant returns a random element of type i of the projective space ps. The second variant returns a random element of a random type of the projective space ps

```
————————————— Example —————————————
gap> ps := PG(8,16);
ProjectiveSpace(8, 16)
gap> RandomSubspace(ps);
<a point in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a proj. 5-space in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a proj. 7-space in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a proj. 4-space in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a plane in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a plane in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a plane in ProjectiveSpace(8, 16)>
```

4.2.16 Span

▷ Span(u, v) (operation)
▷ Span(list) (operation)

Returns: an element or the empty subspace or the whole space

When `u` and `v` are elements of a projective space. This function returns the span of the two elements. When `list` is a list of elements of the same projective space, then this function returns the span of all elements in `list`. It is checked whether the elements `u` and `v` are elements of the same projective space. Although the trivial subspace and the whole projective space are not objects in the category `IsSubspaceOfProjectiveSpace`, they are allowed as arguments for this operation, and also for the second variant of this operation.

```
━━━━━━━━━━━━━━ Example ━━━━━━━━━━━━━━
gap> ps := ProjectiveSpace(3,3);
ProjectiveSpace(3, 3)
gap> p := Random(Planes(ps));
<a plane in ProjectiveSpace(3, 3)>
gap> q := Random(Planes(ps));
<a plane in ProjectiveSpace(3, 3)>
gap> s := Span(p,q);
ProjectiveSpace(3, 3)
gap> s = Span([p,q]);
true
gap> t := Span(EmptySubspace(ps),p);
<a plane in ProjectiveSpace(3, 3)>
gap> t = p;
true
gap> Span(ps,p);
ProjectiveSpace(3, 3)
```

4.2.17 Meet

▷ Meet(`u`, `v`)   (operation)
▷ Meet(`list`)   (operation)

    Returns: an element or the empty subspace or the whole space

When `u` and `v` are elements of a projective space. This function returns the intersection of the two elements. When `list` is a list of elements of the same projective space, then this function returns the intersection of all elements in `list`. It is checked whether the elements `u` and `v` are elements of the same projective space. Although the trivial subspace and the whole projective space are not objects in the category `IsSubspaceOfProjectiveSpace`, they are allowed as arguments for this operation, and also for the second variant of this operation. We remark that the result of a `Meet` operation can be the empty subspace.

```
━━━━━━━━━━━━━━ Example ━━━━━━━━━━━━━━
ProjectiveSpace(7, 8)
gap> p := Random(Solids(ps));
<a solid in ProjectiveSpace(7, 8)>
gap> q := Random(Solids(ps));
<a solid in ProjectiveSpace(7, 8)>
gap> s := Meet(p,q);
< empty subspace >
gap> Display(s);
< empty subspace >
gap> r := Random(Hyperplanes(ps));
<a proj. 6-space in ProjectiveSpace(7, 8)>
gap> Meet(p,r);
```

```
<a plane in ProjectiveSpace(7, 8)>
gap> Meet(q,r);
<a plane in ProjectiveSpace(7, 8)>
gap> Meet([p,q,r]);
< empty subspace >
```

4.2.18   FlagOfIncidenceStructure

▷ FlagOfIncidenceStructure(*ps, els*)   (operation)

   Returns: the flag of the projetcive space *ps*, determined by the subspaces of *ps* in the list *els*.
When *els* is empty, the empty flag is returned.

──────────────────────────────── Example ────────────────────────────────
```
gap> ps := ProjectiveSpace(12,11);
ProjectiveSpace(12, 11)
gap> s1 := RandomSubspace(ps,8);
<a proj. 8-space in ProjectiveSpace(12, 11)>
gap> s2 := RandomSubspace(s1,6);
<a proj. 6-space in ProjectiveSpace(12, 11)>
gap> s3 := RandomSubspace(s2,4);
<a proj. 4-space in ProjectiveSpace(12, 11)>
gap> s4 := Random(Solids(s3));
<a solid in ProjectiveSpace(12, 11)>
gap> s5 := Random(Points(s4));
<a point in ProjectiveSpace(12, 11)>
gap> flag := FlagOfIncidenceStructure(ps,[s1,s3,s2,s5,s4]);
<a flag of ProjectiveSpace(12, 11)>
gap> ps := PG(4,5);
ProjectiveSpace(4, 5)
gap> p := Random(Points(ps));
<a point in ProjectiveSpace(4, 5)>
gap> l := Random(Lines(ps));
<a line in ProjectiveSpace(4, 5)>
gap> v := Random(Solids(ps));
<a solid in ProjectiveSpace(4, 5)>
gap> flag := FlagOfIncidenceStructure(ps,[v,l,p]);
Error, <els> does not determine a flag> called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 19 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> flag := FlagOfIncidenceStructure(ps,[]);
<a flag of ProjectiveSpace(4, 5)>
```

4.2.19   IsEmptyFlag

▷ IsEmptyFlag(*flag*)   (operation)

   Returns: return true if *flag* is the empty flag

4.2.20 IsChamberOfIncidenceStructure

▷ IsChamberOfIncidenceStructure(*flag*)　　　　　　　　　　　　　　　(operation)

Returns: true if *flag* is a chamber

```
————————————————————— Example —————————————————————
gap> ps := PG(3,13);
ProjectiveSpace(3, 13)
gap> plane := Random(Planes(ps));
<a plane in ProjectiveSpace(3, 13)>
gap> line := Random(Lines(plane));
<a line in ProjectiveSpace(3, 13)>
gap> point := Random(Points(line));
<a point in ProjectiveSpace(3, 13)>
gap> flag := FlagOfIncidenceStructure(ps,[point,line,plane]);
<a flag of ProjectiveSpace(3, 13)>
gap> IsChamberOfIncidenceStructure(flag);
true
```

4.3 Shadows of Projective Subspaces

4.3.1 ShadowOfElement

▷ ShadowOfElement(*ps, el, i*)　　　　　　　　　　　　　　　　　　(operation)
▷ ShadowOfElement(*ps, el, str*)　　　　　　　　　　　　　　　　　(operation)

Returns: the elements of type *i* incident with *el*. The second variant determines the type *i* from the position of *str* in the list returned by TypesOfElementsOfIncidenceStructurePlural

```
————————————————————— Example —————————————————————
gap> ps := PG(4,3);
ProjectiveSpace(4, 3)
gap> plane := Random(Planes(ps));
<a plane in ProjectiveSpace(4, 3)>
gap> shadowpoints := ShadowOfElement(ps,plane,1);
<shadow points in ProjectiveSpace(4, 3)>
gap> List(shadowpoints);
[ <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)> ]
gap> shadowlines := ShadowOfElement(ps,plane,2);
<shadow lines in ProjectiveSpace(4, 3)>
gap> List(shadowlines);
[ <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
```

```
    <a line in ProjectiveSpace(4, 3)> ]
```

4.3.2 ShadowOfFlag

▷ ShadowOfFlag(ps, flag, i)   (operation)
▷ ShadowOfFlag(ps, flag, str)   (operation)
    Returns: the type `i` shadow elements of the flag `flag`, i.e. the elements of type `i` incident with all elements of `flag`. The second variant determines the type `i` from the position of `str` in the list returned by TypesOfElementsOfIncidenceStructurePlural.

```
 ───────────────────────── Example ─────────────────────────
 gap> ps := PG(5,7);
 ProjectiveSpace(5, 7)
 gap> p := VectorSpaceToElement(ps,[1,0,0,0,0,0]*Z(7)^0);
 <a point in ProjectiveSpace(5, 7)>
 gap> l := VectorSpaceToElement(ps,[[1,0,0,0,0,0],[0,1,0,0,0,0]]*Z(7)^0);
 <a line in ProjectiveSpace(5, 7)>
 gap> v := VectorSpaceToElement(ps,[[1,0,0,0,0,0],[0,1,0,0,0,0],[0,0,1,0,0,0]]*Z(7)^0);
 <a plane in ProjectiveSpace(5, 7)>
 gap> flag := FlagOfIncidenceStructure(ps,[v,l,p]);
 <a flag of ProjectiveSpace(5, 7)>
 gap> s := ShadowOfFlag(ps,flag,4);
 <shadow solids in ProjectiveSpace(5, 7)>
 gap> s := ShadowOfFlag(ps,flag,"solids");
 <shadow solids in ProjectiveSpace(5, 7)>
```

4.3.3 ElementsIncidentWithElementOfIncidenceStructure

▷ ElementsIncidentWithElementOfIncidenceStructure(el, i)   (operation)
    Returns: the elements of type `i` incident with `el`, in other words, the type `i` shadow of the element `el`

    Internally, the function FlagOfIncidenceStructure is used to create a flag from `list`. This function also performs the checking.

```
 ───────────────────────── Example ─────────────────────────
 gap> ps := PG(6,9);
 ProjectiveSpace(6, 9)
 gap> p := VectorSpaceToElement(ps,[1,0,1,0,0,0,0]*Z(9)^0);
 <a point in ProjectiveSpace(6, 9)>
 gap> els := ElementsIncidentWithElementOfIncidenceStructure(p,3);
 <shadow planes in ProjectiveSpace(6, 9)>
 gap> line := VectorSpaceToElement(ps,[[1,1,1,1,0,0,0],[0,0,0,0,1,1,1]]*Z(9)^0);
 <a line in ProjectiveSpace(6, 9)>
 gap> els := ElementsIncidentWithElementOfIncidenceStructure(line,1);
 <shadow points in ProjectiveSpace(6, 9)>
 gap> List(els);
 [ <a point in ProjectiveSpace(6, 9)>, <a point in ProjectiveSpace(6, 9)>,
   <a point in ProjectiveSpace(6, 9)>, <a point in ProjectiveSpace(6, 9)>,
   <a point in ProjectiveSpace(6, 9)>, <a point in ProjectiveSpace(6, 9)>,
   <a point in ProjectiveSpace(6, 9)>, <a point in ProjectiveSpace(6, 9)>,
```

```
    <a point in ProjectiveSpace(6, 9)>, <a point in ProjectiveSpace(6, 9)> ]
```

4.3.4 Short names for `ElementsIncidentWithElementOfIncidenceStructure`

▷ Points(*ps, v*)   (operation)
▷ Lines(*ps, v*)   (operation)
▷ Planes(*ps, v*)   (operation)
▷ Solids(*ps, v*)   (operation)
▷ Hyperplanes(*inc, v*)   (operation)
▷ Points(*v*)   (operation)
▷ Lines(*v*)   (operation)
▷ Planes(*v*)   (operation)
▷ Solids(*v*)   (operation)
▷ Hyperplanes(*v*)   (operation)

Returns: The elements of the incidence geometry of the according type. If `ps` is not given as an argument, it is deduced from `v` as its ambient geometry.

```
————————————————————— Example —————————————————————
  gap> ps := PG(6,13);
  ProjectiveSpace(6, 13)
  gap> plane := Random(Planes(ps));
  <a plane in ProjectiveSpace(6, 13)>
  gap> Points(plane);
  <shadow points in ProjectiveSpace(6, 13)>
  gap> Lines(plane);
  <shadow lines in ProjectiveSpace(6, 13)>
  gap> Solids(plane);
  <shadow solids in ProjectiveSpace(6, 13)>
  gap> Hyperplanes(plane);
  <shadow lines in ProjectiveSpace(6, 13)>
  gap> ElementsIncidentWithElementOfIncidenceStructure(plane,6);
  <shadow proj. 5-subspaces in ProjectiveSpace(6, 13)>
```

4.4 Enumerating subspaces of a projective space

4.4.1 Iterator

▷ Iterator(*subspaces*)   (operation)

Returns: an iterator for the collection `subspaces`

We refer to the GAP operation `Iterator` for the definition of an iterator.

```
————————————————————— Example —————————————————————
  gap> pg := PG(5,7);
  ProjectiveSpace(5, 7)
  gap> planes := Planes(pg);
  <planes of ProjectiveSpace(5, 7)>
  gap> iter := Iterator(planes);
  <iterator>
  gap> NextIterator(iter);
```

```
<a plane in ProjectiveSpace(5, 7)>
gap> NextIterator(iter);
<a plane in ProjectiveSpace(5, 7)>
gap> NextIterator(iter);
<a plane in ProjectiveSpace(5, 7)>
```

4.4.2 Enumerator

▷ Enumerator(*subspaces*) (operation)

Returns: an enumerator for the collection *subspaces*

For complete collections of subspaces of a given type of a projective space, currently, no non-trivial enumerator is installed, i.e. this operation just returns a list containing all elements of the collection *subspaces*. Such a list can, of course, be used as an enumerator, but this might be time consuming.

——————— Example ———————
```
gap> pg := PG(3,4);
ProjectiveSpace(3, 4)
gap> lines := Lines(pg);
<lines of ProjectiveSpace(3, 4)>
gap> enum := Enumerator(lines);;
gap> Length(enum);
357
```

4.4.3 List

▷ List(*subspaces*) (operation)
▷ AsList(*subspaces*) (operation)

Returns: the complete list of elements in the collection *subspaces*

The operation List will return a complete list, the operation AsList will return an orb object, representing a complete orbit, i.e. representing in this case a complete list. To obtain the elements explicitly, one has to issue the List operation with as argument the orb object again. Applying List directly to a collection of subspaces, refers to the enumerator for the collection, while using AsList uses the orb to compute all subspaces as an orbit.

——————— Example ———————
```
gap> pg := PG(3,4);
ProjectiveSpace(3, 4)
gap> lines := Lines(pg);
<lines of ProjectiveSpace(3, 4)>
gap> list := List(lines);;
gap> Length(list);
357
gap> aslist := AsList(lines);
<closed orbit, 357 points>
gap> list2 := List(aslist);;
gap> Length(list2);
357
```

Chapter 5

Projective Groups

A *collineation* of a projective space is a type preserving bijection of the elements of the projective space, that preserves incidence. The Fundamental Theorem of Projective Geometry states that every collineation of a Desarguesian projective space of dimension at least two is induced by a semilinear map of the underlying vector space. The group of all linear maps of a given $n + 1$-dimensional vector space over a given field $GF(q)$ is denoted by $GL(n + 1, q)$. This is a matrix group consisting of all non-singular square $n + 1$-dimensional matrices over $GF(q)$. The group of all semilinear maps of the vector space $V(n, q)$ is obtained as the semidirect product of $GL(n, q)$ and $Aut(GF(q))$, and is denoted by $\Gamma L(n + 1, q)$. Each semilinear map induces a collineation of $PG(n, q)$. The Fundamental theorem of Projective Geometry also guarantees that the converse holds. Note also that $\Gamma L(n + 1, q)$ does not act faithfully on the projective points, and the kernel of its action is the group of scalar matrices, $Sc(n + 1, q)$. So the group $P\Gamma L(n + 1, q)$ is defined as the group $\Gamma L(n + 1, q)/Sc(n + 1, q)$, and $PGL(n + 1, q) = GL(n + 1, q)/Sc(n + 1, q)$. An element of the group $PGL(n + 1, q)$ is also called a *projectivity* or *homography* of $PG(n, q)$, and the group $PGL(n + 1, q)$ is called the *projectivity group* or *homography group* of $PG(n, q)$. An element of $P\Gamma L(n + 1, q)$ is called a *collineation* of $PG(n, q)$ and the group $P\Gamma L(n + 1, q)$ is the *collineation group* of $PG(n, q)$.

As usual, we also consider the special linear group $SL(n + 1, q)$, which is the subgroup of $GL(n + 1, q)$ of all matrices having determinant one. Its projective variant, i.e. $PSL(n + 1, q) = SL(n + 1, q)/Sc(n + 1, q)$ is called the *special homography group* or *special projectivity group* of $PG(n, q)$.

Consider the projective space $PG(n, q)$. As described in Chapter 4, a point of $PG(n, q)$ is represented by a row vector. A k-dimensional subspace of $PG(n, q)$ is represented by a generating set of $k + 1$ points, and as such, by a $(k + 1) \times (n + 1)$ matrix. The convention in FinInG is that a collineation ϕ with underlying matrix A and field automorphism θ maps that projective point represented by row vector $(x_0, x_1, \ldots, x_n)$ to the projective point represented by row vector $(y_0, y_1, \ldots, y_n) = ((x_0, x_1, \ldots, x_n)A)^\theta$. This convention determines completely the action of collineations on all elements of a projective space, and it follows that the product of two collineations ϕ_1, ϕ_2 with respective underlying matrices A_1, A_2 and respective underlying field automorphisms θ_1, θ_2 is the collineation with underlying matrix $A_1 \cdot A_2^{\theta_1^{-1}}$ and underlying field automorphism $\theta_1 \theta_2$.

A *correlation* of the projective space $PG(n, q)$ is a collineation from $PG(n, q)$ to its dual. A projectivity from $PG(n, q)$ to its dual is sometimes called a *reciprocity*. The *standard duality* of the projective space $PG(n, q)$ maps any point v with coordinates $(x_0, x_1, \ldots, x_n)$ on the hyperplane with equation $x_0 X_0 + x_1 X_1 + \cdots + x_n X_n$. The standard duality acts as an automorphism on $P\Gamma L(n + 1, q)$ by mapping the underlying matrix of a collineation to its inverse transpose matrix. (Recall that

the frobenius automorphism and the standard duality commute.) The convention in FinInG is that a correlation ϕ with underlying matrix A and field automorphism θ maps that projective point represented by row vector $(x_0, x_1, \ldots, x_n)$ to the projective hyperplane represented by row vector $(y_0, y_1, \ldots, y_n) = ((x_0, x_1, \ldots, x_n)A)^{\theta \cdot}$, i.e. $(y_0, y_1, \ldots, y_n) = ((x_0, x_1, \ldots, x_n)A)^{\theta \cdot}$ are the dual coordinates of the hyperplane.

The product of two correlations of $PG(n, q)$ is a collineation, and the product of a collineation and a correlaton is a correlation. So the set of all collineations and correlations of $PG(n, q)$ forms a group, called the *correlation-collineation group* of $PG(n, q)$. The correlation-collineation group of $PG(n, q)$ is isomorphic to the semidirect product of $P\Gamma L(n+1, q)$ with the cyclic group of order 2 generated by the *standard duality* of the projective space $PG(n, q)$. The convention determines completely the action of correlations and collineations on all elements of a projective space, and it follows that the product of two elements of the correlation-collineation group ϕ_1, ϕ_2 with respective underlying matrices A_1, A_2, respective underlying field automorphisms θ_1, θ_2, and respective underlying projective space isomorphisms (standard duality or identity map) δ_1, δ_2, is the element of the correlation-collineation group with underlying matrix $A_1(A_2^{\theta_1^{-1}})^{\delta_1}$, underlying field automorphism $\theta_1\theta_2$, and underlying projective space automorphism $\delta_1\delta_2$. , where the action of δ_1 on the matrix A_2 is defined as the transpose inverse if δ_1 is the standard duality, and as the identity if δ_1 is the identity.

Action functions for collineations and correlations on the subspaces of a projective space are described in detail in Section 5.8

We mention that the commands PGL (and ProjectiveGeneralLinearGroup) and PSL (and ProjectiveSpecialLinearGroup) are available in GAP and return a (permutation) group isomorphic to the required group. Therefore we do not provide new methods for these commands, but assume that the user will obtain these groups as homography or special homography group of the appropriate projective space. We will follow this philosophy for the other classical groups. The terminology *projective semilinear group* will be used for a group generated by collineations of a projective space.

5.1 Projectivities, collineations and correlations of projective spaces.

These are the different type of actions on projective spaces in FinInG, and they naturally give rise to the following distinct categories and representations. Note that these categories and representations are to be considered on a non-user level. Below we describe all user constuction methods that hide nicely these technical details.

5.1.1 Categories for group elements

▷ IsProjGrpEl   (Category)
▷ IsProjGrpElWithFrob   (Category)
▷ IsProjGrpElWithFrobWithPSIsom   (Category)

IsProjGrpEl, IsProjGrpElWithFrob, and IsProjGrpElWithFrobWithPSIsom are the categories naturally induced by the notions of projectivities, collineations, and correlations of a projective space.

5.1.2 Representations for group elements

▷ IsProjGrpElRep   (Representation)
▷ IsProjGrpElWithFrobRep   (Representation)

▷ IsProjGrpElWithFrobWithPSIsomRep   (Representation)

IsProjGrpElRep is the representation naturally induced by a projectivity; IsProjGrpElWithFrobRep is the representation naturally induced by the notion of a collineation of projective space; and IsProjGrpElWithFrobWithPSIsomRep is the representation naturally induced by a correlation of a projective space. This means that an object in the representation IsProjGrpElRep has as underlying object a matrix; an object in the category IsProjGrpElWithFrobRep has as underlying object a pair consisting of a matrix and a field automorphism; and IsProjGrpElWithFrobWithPSIsomRep has as underlying object a triple consisting of a matrix, a field automorphism and an isomorphism from the projective space to its dual space. Also the basefield is stored as a component in the representation.

The above mentioned categories allow us to make a distinction between projectivities, collineations and correlations apart from their representation. However, in FinInG, a group element constructed in the categories IsProjGrpElMore is always constructed in the representation IsProjGrpElMoreRep. Furthermore, projectivities of projective spaces (and also collineations of projective spaces) will by default be constructed in the category IsProjGrpElWithFrobRep. This technical choice was made by the developers to have the projectivity groups naturally embedded in the collineation groups. Correlations of projective spaces will be constructed in the category IsProjGrpElWithFrobWithPSIsom.

5.1.3 Projectivities

▷ IsProjectivity   (Property)

IsProjectivity is a property. Projectivities are the elements of $PGL(n+1,q)$. Every element belonging to IsProjGrpEl is by construction a projectivity. If IsProjectivity is applied to a an element belonging to IsProjGrpElWithFrob, then it verifies whether the underlying field automorphism is the identity. If IsProjectivity is applied to a an element belonging to IsProjGrpElWithFrobWithPSIsom, then it verifies whether the underlying field automorphism is the identity, and whether the projective space isomorphism is the identity. This operation provides a user-friendly method to distinguish the projectivities from the projective strictly semilinear maps, and the correlations of a projective space.

```
_____ Example _____
gap> g := Random(HomographyGroup(PG(3,4)));
< a collineation: <cmat 4x4 over GF(2,2)>, F^0>
gap> IsProjectivity(g);
true
gap> g := Random(CollineationGroup(PG(3,4)));
< a collineation: <cmat 4x4 over GF(2,2)>, F^0>
gap> IsProjectivity(g);
true
gap> g := Random(CorrelationCollineationGroup(PG(3,4)));
<projective element with Frobenius with projectivespace isomorphism: <immutabl
e cmat 4x4 over GF(2,2)>, F^
2, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
3,GF(2^2)) ) ) >
gap> IsProjectivity(g);
false
```

5.1.4   Collineations of projective spaces

▷ IsCollineation   (Property)

IsCollineation is property. All elements of $P\Gamma L(n+1,q)$ are collineations, and therefore all elements belonging to IsProjGrpElWithFrob are collineations. But also a projectivity is a collineation, as well as an element belonging to IsProjGrpElWithFrobWithPSIsom with projective space isomorphism equal to the identity, is a collineation.

```
———————————————————— Example ————————————————————
gap> g := Random(HomographyGroup(PG(2,27)));
< a collineation: <cmat 3x3 over GF(3,3)>, F^0>
gap> IsCollineation(g);
true
gap> g := Random(CollineationGroup(PG(2,27)));
< a collineation: <cmat 3x3 over GF(3,3)>, F^0>
gap> IsCollineation(g);
true
gap> g := Random(CorrelationCollineationGroup(PG(2,27)));
<projective element with Frobenius with projectivespace isomorphism: <cmat 3x
3 over GF(3,3)>, F^0, IdentityMapping( <All elements of ProjectiveSpace(2,
27)> ) >
gap> IsCollineation(g);
true
```

5.1.5   Projective strictly semilinear maps

▷ IsStrictlySemilinear   (Property)

IsStrictlySemilinear is a property that checks whether a given collineation has a non-trivial underlying field automorphisms, i.e. whether the element belongs to $P\Gamma L(n+1,q)$, but not to $PGL(n+1,q)$. If IsStrictlySemilinear is applied to a an element belonging to IsProjGrpElWithFrobWithPSIsom, then it verifies whether the underlying field automorphism is different from the identity, and whether the projective space isomorphism equals the identity. This operation provides a user-friendly method to distinguish the projective strictly semilinear maps from projectivities inside the category of collineations of a projective space.

```
———————————————————— Example ————————————————————
gap> g := Random(HomographyGroup(PG(3,25)));
< a collineation: <cmat 4x4 over GF(5,2)>, F^0>
gap> IsStrictlySemilinear(g);
false
gap> g := Random(CollineationGroup(PG(3,25)));
< a collineation: <cmat 4x4 over GF(5,2)>, F^5>
gap> IsStrictlySemilinear(g);
true
gap> g := Random(CorrelationCollineationGroup(PG(3,25)));
<projective element with Frobenius with projectivespace isomorphism: <cmat 4x
4 over GF(5,2)>, F^5, IdentityMapping( <All elements of ProjectiveSpace(3,
25)> ) >
gap> IsStrictlySemilinear(g);
```

```
 true
```

5.1.6   Correlations and collineations

▷ IsProjGrpElWithFrobWithPSIsom   (Category)
▷ IsCorrelationCollineation   (Category)
▷ IsCorrelation   (Property)

The underlying objects of a correlation-collineation in FinInG are a nonsingular matrix, a field automorphism and a projective space isomorphism. `IsProjGrpElWithFrobWithPSIsom` is the category of these objects. If the projective space isomorphism is not the identity, then the element is a correlation, and `IsCorrelation` will return true. `IsCorrelationCollineation` is a synonym of `IsProjGrpElWithFrobWithPSIsom`.

```
 _____ Example _____
 gap> g := Random(CollineationGroup(PG(4,7)));
 < a collineation: <cmat 5x5 over GF(7,1)>, F^0>
 gap> IsCorrelationCollineation(g);
 false
 gap> IsCorrelation(g);
 false
 gap> g := Random(CorrelationCollineationGroup(PG(4,7)));
 <projective element with Frobenius with projectivespace isomorphism: <cmat 5x
 5 over GF(7,1)>, F^0, IdentityMapping( <All elements of ProjectiveSpace(4,
 7)> ) >
 gap> IsCorrelationCollineation(g);
 true
 gap> IsCorrelation(g);
 false
```

5.2   Construction of projectivities, collineations and correlations.

In FinInG, projectivities and collineations are both constructed in the category `IsProjGrpElWithFrob`; correlations are constructed in the category `IsProjGrpElWithFrobWithPSIsom`.

5.2.1   Projectivity

▷ Projectivity(*mat, f*)   (operation)
▷ Projectivity(*pg, mat*)   (operation)

Returns: a projectivity of a projective space

The argument *mat* must be a nonsingular matrix over the finite field *f*. In the second variant, the size of the nonsingular matrix *mat* must be one more than the dimension of the projective space *pg*. This creates an element of a projectivity group. But the returned object belongs to `IsProjGrpElWithFrob`!

```
 _____ Example _____
 gap> mat := [[1,0,0],[0,1,0],[0,0,1]]*Z(9)^0;
 [ [ Z(3)^0, 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3) ],
```

```
    [ 0*Z(3), 0*Z(3), Z(3)^0 ] ]
gap> Projectivity(mat,GF(9));
< a collineation: <cmat 3x3 over GF(3,2)>, F^0>
```

5.2.2 CollineationOfProjectiveSpace

▷ CollineationOfProjectiveSpace(*mat, frob, f*)   (operation)
▷ CollineationOfProjectiveSpace(*mat, f*)   (operation)
▷ CollineationOfProjectiveSpace(*mat, frob, f*)   (operation)
▷ CollineationOfProjectiveSpace(*mat, f*)   (operation)
▷ CollineationOfProjectiveSpace(*pg, mat*)   (operation)
▷ CollineationOfProjectiveSpace(*pg, mat, frob*)   (operation)
▷ Collineation(*pg, mat*)   (operation)
▷ Collineation(*pg, mat, frob*)   (operation)

   mat is a nonsingular matrix, *frob* is a field automorphism, *f* is a field, and *pg* is a projective space. This function (and its shorter version) returns the collineation with matrix *mat* and automorphism *frob* of the field *f*. If *frob* is not specified then the companion automorphism of the resulting group element will be the identity map. The returned object belongs to the category IsProjGrpElWithFrob. When the argument *frob* is given, it is checked whether the source of *frob* equals *f*. When the arguments *pg* and *mat* are used, then it is checked that these two arguments are compatible.

```
———————————————————————— Example ————————————————————————
gap> mat:=
> [[Z(2^3)^6,Z(2^3),Z(2^3)^3,Z(2^3)^3],[Z(2^3)^6,Z(2)^0,Z(2^3)^2,Z(2^3)^3],
> [0*Z(2),Z(2^3)^4,Z(2^3),Z(2^3)],[Z(2^3)^6,Z(2^3)^5,Z(2^3)^3,Z(2^3)^5 ]];
[ [ Z(2^3)^6, Z(2^3), Z(2^3)^3, Z(2^3)^3 ],
  [ Z(2^3)^6, Z(2)^0, Z(2^3)^2, Z(2^3)^3 ],
  [ 0*Z(2), Z(2^3)^4, Z(2^3), Z(2^3) ],
  [ Z(2^3)^6, Z(2^3)^5, Z(2^3)^3, Z(2^3)^5 ] ]
gap> frob := FrobeniusAutomorphism(GF(8));
FrobeniusAutomorphism( GF(2^3) )
gap> phi := ProjectiveSemilinearMap(mat,frob^2,GF(8));
< a collineation: <cmat 4x4 over GF(2,3)>, F^4>
gap> mat2 := [[Z(2^8)^31,Z(2^8)^182,Z(2^8)^49],[Z(2^8)^224,Z(2^8)^25,Z(2^8)^45],
> [Z(2^8)^128,Z(2^8)^165,Z(2^8)^217]];
[ [ Z(2^8)^31, Z(2^8)^182, Z(2^8)^49 ], [ Z(2^8)^224, Z(2^8)^25, Z(2^8)^45 ],
  [ Z(2^8)^128, Z(2^8)^165, Z(2^8)^217 ] ]
gap> psi := CollineationOfProjectiveSpace(mat2,GF(256));
< a collineation: <cmat 3x3 over GF(2,8)>, F^0>
```

5.2.3 ProjectiveSemilinearMap

▷ ProjectiveSemilinearMap(*mat, frob, f*)   (operation)
   Returns: a projectivity of a projective space
   mat is a nonsingular matrix, *frob* is a field automorphism, and *f* is a field. This function returns the collineation with matrix *mat* and automorphism *frob*. The returned object belongs to the category

IsProjGrpElWithFrob. When the argument `frob` is given, it is checked whether the source of `frob` equals `f`.

5.2.4 IdentityMappingOfElementsOfProjectiveSpace

▷ IdentityMappingOfElementsOfProjectiveSpace(`ps`)   (operation)

This operation returns the identity mapping on the collection of subspaces of a projective space `ps`.

5.2.5 StandardDualityOfProjectiveSpace

▷ StandardDualityOfProjectiveSpace(`ps`)   (operation)

This operation returns the standard duality of the projective space `ps`

```
———————————————————— Example ————————————————————
gap> ps := ProjectiveSpace(4,5);
ProjectiveSpace(4, 5)
gap> delta := StandardDualityOfProjectiveSpace(ps);
StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(4,GF(5)) ) )
gap> delta^2;
IdentityMapping( <All elements of ProjectiveSpace(4, 5)> )
gap> p := VectorSpaceToElement(ps,[1,2,3,0,1]*Z(5)^0);
<a point in ProjectiveSpace(4, 5)>
gap> h := p^delta;
<a solid in ProjectiveSpace(4, 5)>
gap> UnderlyingObject(h);
<cmat 4x5 over GF(5,1)>
gap> Unpack(last);
[ [ Z(5)^0, 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^2 ],
  [ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5), Z(5)^3 ],
  [ 0*Z(5), 0*Z(5), Z(5)^0, 0*Z(5), Z(5) ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0, 0*Z(5) ] ]
```

5.2.6 CorrelationOfProjectiveSpace

▷ CorrelationOfProjectiveSpace(`mat, f`)   (operation)
▷ CorrelationOfProjectiveSpace(`mat, frob, f`)   (operation)
▷ CorrelationOfProjectiveSpace(`mat, f, delta`)   (operation)
▷ CorrelationOfProjectiveSpace(`mat, frob, f, delta`)   (operation)
▷ CorrelationOfProjectiveSpace(`pg, mat, frob, delta`)   (operation)
▷ Correlation(`pg, mat, frob, delta`)   (operation)

`mat` is a nonsingular matrix, `frob` is a field automorphism, `f` is a field, and `delta` is the standard duality of the projective space $PG(n,q)$. This function returns the correlation with matrix `mat`, automorphism `frob`, and standard duality `delta`. If `frob` is not specified then the companion automorphism of the resulting group element will be the identity map. If the user specifies `delta`, then it must be the standard duality of a projective space, created using StandardDualityOfProjectiveSpace

(5.2.5), or the identity mapping on the collection of subspaces of a projective space, created using
`IdentityMappingOfElementsOfProjectiveSpace` (5.2.4). If not specified, then the companion
vector space isomorphism is the identity mapping. The returned object belongs to the category
`IsProjGrpElWithFrobWithPSIsom`

```
────────────────────────── Example ──────────────────────────
gap> mat := [[1,0,0],[3,0,2],[0,5,4]]*Z(7^3);
[ [ Z(7^3), 0*Z(7), 0*Z(7) ], [ Z(7^3)^58, 0*Z(7), Z(7^3)^115 ],
  [ 0*Z(7), Z(7^3)^286, Z(7^3)^229 ] ]
gap> phi1 := CorrelationOfProjectiveSpace(mat,GF(7^3));
<projective element with Frobenius with projectivespace isomorphism: <cmat 3x
3 over GF(7,3)>, F^0, IdentityMapping( <All elements of ProjectiveSpace(2,
343)> ) >
gap> frob := FrobeniusAutomorphism(GF(7^3));
FrobeniusAutomorphism( GF(7^3) )
gap> phi2 := CorrelationOfProjectiveSpace(mat,frob,GF(7^3));
<projective element with Frobenius with projectivespace isomorphism: <cmat 3x
3 over GF(7,3)>, F^7, IdentityMapping( <All elements of ProjectiveSpace(2,
343)> ) >
gap> delta := StandardDualityOfProjectiveSpace(ProjectiveSpace(2,GF(7^3)));
StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
2,GF(7^3)) ) )
gap> phi3 := CorrelationOfProjectiveSpace(mat,GF(7^3),delta);
<projective element with Frobenius with projectivespace isomorphism: <cmat 3x
3 over GF(7,
3)>, F^0, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
2,GF(7^3)) ) ) >
gap> phi4 := CorrelationOfProjectiveSpace(mat,frob,GF(7^3),delta);
<projective element with Frobenius with projectivespace isomorphism: <cmat 3x
3 over GF(7,3)>, F^
7, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
2,GF(7^3)) ) ) >
```

5.3 Basic operations for projectivities, collineations and correlations of projective spaces

5.3.1 Representative

▷ Representative(*g*)   (operation)

g is a projectivity, collineation or correlation of a projective space. This function returns the
rereresentative components that determine *g*, i.e. a matrix, a matrix and a field automorphism, and a
matrix, a field automorphism, and a vector space isomorphism, respectively.

```
────────────────────────── Example ──────────────────────────
gap> g:=CollineationGroup( ProjectiveSpace(2,49));
The FinInG collineation group PGammaL(3,49)
gap> x:=Random(g);;
gap> Representative(x);
[ <immutable cmat 3x3 over GF(7,2)>, FrobeniusAutomorphism( GF(7^2) ) ]
```

5.3.2 MatrixOfCollineation

▷ MatrixOfCollineation(*g*) (operation)

g is a collineation (including a projectivity) of a projective space. This function returns the matrix that was used to construct *g*.

```
———————————————————— Example ————————————————————
gap> g:=CollineationGroup( ProjectiveSpace(3,3));
The FinInG collineation group PGL(4,3)
gap> x:=Random(g);;
gap> MatrixOfCollineation(x);
<cmat 4x4 over GF(3,1)>
gap> Unpack(last);
[ [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ Z(3)^0, Z(3)^0, Z(3), 0*Z(3) ], [ Z(3), Z(3), 0*Z(3), 0*Z(3) ] ]
```

5.3.3 MatrixOfCorrelation

▷ MatrixOfCorrelation(*g*) (operation)

g is a correlation of a projective space. This function returns the matrix that was used to construct *g*.

```
———————————————————— Example ————————————————————
gap> g:=CorrelationCollineationGroup( ProjectiveSpace(4,9));
The FinInG correlation-collineation group PGammaL(5,9) : 2
gap> x:=Random(g);;
gap> MatrixOfCorrelation(x);
<cmat 5x5 over GF(3,2)>
gap> Unpack(last);
[ [ Z(3^2)^3, Z(3^2)^6, 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ Z(3^2), Z(3^2)^3, Z(3^2)^2, Z(3^2)^5, Z(3^2) ],
  [ Z(3^2)^2, Z(3^2)^3, Z(3^2), Z(3^2), Z(3^2)^3 ],
  [ Z(3^2)^2, Z(3^2), Z(3^2)^6, Z(3^2), Z(3^2)^5 ],
  [ Z(3^2), Z(3^2)^3, Z(3)^0, 0*Z(3), Z(3^2)^6 ] ]
```

5.3.4 BaseField

▷ BaseField(*g*) (operation)

Returns: a field

g is a projectivity, collineation or correlation of a projective space. This function returns the base field that was used to construct *g*.

```
———————————————————— Example ————————————————————
gap> mat := [[0,1,0],[1,0,0],[0,0,2]]*Z(3)^0;
[ [ 0*Z(3), Z(3)^0, 0*Z(3) ], [ Z(3)^0, 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), Z(3) ] ]
gap> g := Projectivity(mat,GF(3^6));
< a collineation: <cmat 3x3 over GF(3,6)>, F^0>
gap> BaseField(g);
```

```
  GF(3^6)
```

5.3.5 FieldAutomorphism

▷ FieldAutomorphism(*g*)   (operation)

g is a collineation of a projective space or a correlation of a projective space. This function returns the companion field automorphism which defines *g*. Note that in the following example, you may want to execute it several times to see the different possible results generated by the random choice of projective semilinear map here.

```
_____ Example _____
  gap> g:=CollineationGroup( ProjectiveSpace(3,9));
  The FinInG collineation group PGammaL(4,9)
  gap> x:=Random(g);;
  gap> FieldAutomorphism(x);
  IdentityMapping( GF(3^2) )
```

5.3.6 ProjectiveSpaceIsomorphism

▷ ProjectiveSpaceIsomorphism(*g*)   (operation)

g is a correlation of a projective space. This function returns the companion isomorphism of the projective space which defines *g*.

```
_____ Example _____
  gap> mat := [[1,0,0],[3,0,2],[0,5,4]]*Z(7^3);
  [ [ Z(7^3), 0*Z(7), 0*Z(7) ], [ Z(7^3)^58, 0*Z(7), Z(7^3)^115 ],
    [ 0*Z(7), Z(7^3)^286, Z(7^3)^229 ] ]
  gap> frob := FrobeniusAutomorphism(GF(7^3));
  FrobeniusAutomorphism( GF(7^3) )
  gap> delta := StandardDualityOfProjectiveSpace(ProjectiveSpace(2,GF(7^3)));
  StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
  2,GF(7^3)) ) )
  gap> phi := CorrelationOfProjectiveSpace(mat,frob,GF(7^3),delta);
  <projective element with Frobenius with projectivespace isomorphism: <cmat 3x
  3 over GF(7,3)>, F^
  7, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
  2,GF(7^3)) ) ) >
  gap> ProjectiveSpaceIsomorphism(phi);
  StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
  2,GF(7^3)) ) )
```

5.3.7 Order

▷ Order(*g*)   (operation)

g is a projectivity, collineation or correlation of a projective space. This function returns the order of *g*.

```
┌──────────────────────────── Example ────────────────────────────┐
  gap> x := Random(CollineationGroup(PG(4,9)));
  < a collineation: <cmat 5x5 over GF(3,2)>, F^3>
  gap> t := Order(x);
  32
  gap> IsOne(x^t);
  true
```

5.4 The groups *PΓL*, *PGL*, and *PSL* in **FinInG**

As mentioned before the commands PGL (and ProjectiveGeneralLinearGroup) and PSL (and ProjectiveSpecialLinearGroup) are already available in GAP and return a (permutation) group isomorphic to the required group. In FinInG, different categories are created for these groups.

5.4.1 ProjectivityGroup

▷ ProjectivityGroup(*geom*)   (operation)

▷ HomographyGroup(*geom*)   (operation)

Returns: the group of projectivities of geom

Let *geom* be the projective space $PG(n,q)$ This operation (and its synonym) returns the group of projectivities $PGL(n+1,q)$ of the projective space $PG(n,q)$. Note that although a projectivity is a collineation with the identity as associated field isomorphism, this group belongs to the category IsProjectiveGroupWithFrob, and its elements belong to IsProjGrpElWithFrob.

```
┌──────────────────────────── Example ────────────────────────────┐
  gap> ps := ProjectiveSpace(3,16);
  ProjectiveSpace(3, 16)
  gap> ProjectivityGroup(ps);
  The FinInG projectivity group PGL(4,16)
  gap> HomographyGroup(ps);
  The FinInG projectivity group PGL(4,16)
  gap> ps := ProjectiveSpace(4,81);
  ProjectiveSpace(4, 81)
  gap> ProjectivityGroup(ps);
  The FinInG projectivity group PGL(5,81)
  gap> HomographyGroup(ps);
  The FinInG projectivity group PGL(5,81)
  gap> ps := ProjectiveSpace(5,3);
  ProjectiveSpace(5, 3)
  gap> ProjectivityGroup(ps);
  The FinInG projectivity group PGL(6,3)
  gap> HomographyGroup(ps);
  The FinInG projectivity group PGL(6,3)
  gap> ps := ProjectiveSpace(2,2);
  ProjectiveSpace(2, 2)
  gap> ProjectivityGroup(ps);
  The FinInG projectivity group PGL(3,2)
  gap> HomographyGroup(ps);
  The FinInG projectivity group PGL(3,2)
```

5.4.2 CollineationGroup

▷ CollineationGroup(*geom*)   (operation)

Returns: the group of collineations of geom

Let *geom* be the projective space $PG(n,q)$. This operation returns the group of collineations $\Gamma L(n+1,q)$ of the projective space $PG(n,q)$. If $GF(q)$ has no non-trivial field automorphisms, i.e. when q is prime, the group $PGL(n+1,q)$ is the full collineation group and will be returned.

```
————————————————— Example —————————————————
gap> ps := ProjectiveSpace(3,16);
ProjectiveSpace(3, 16)
gap> CollineationGroup(ps);
The FinInG collineation group PGammaL(4,16)
gap> ps := ProjectiveSpace(4,81);
ProjectiveSpace(4, 81)
gap> CollineationGroup(ps);
The FinInG collineation group PGammaL(5,81)
gap> ps := ProjectiveSpace(5,3);
ProjectiveSpace(5, 3)
gap> CollineationGroup(ps);
The FinInG collineation group PGL(6,3)
gap> ps := ProjectiveSpace(2,2);
ProjectiveSpace(2, 2)
gap> CollineationGroup(ps);
The FinInG collineation group PGL(3,2)
```

5.4.3 SpecialProjectivityGroup

▷ SpecialProjectivityGroup(*geom*)   (operation)
▷ SpecialHomographyGroup(*geom*)   (operation)

Returns: the group of special projectivities of geom

Let *geom* be the projective space $PG(n,q)$ This operation (and its synonym) returns the group of special projectivities $PSL(n+1,q)$ of the projective space $PG(n,q)$.

```
————————————————— Example —————————————————
gap> ps := ProjectiveSpace(3,16);
ProjectiveSpace(3, 16)
gap> SpecialProjectivityGroup(ps);
The FinInG PSL group PSL(4,16)
gap> SpecialHomographyGroup(ps);
The FinInG PSL group PSL(4,16)
gap> ps := ProjectiveSpace(4,81);
ProjectiveSpace(4, 81)
gap> SpecialProjectivityGroup(ps);
The FinInG PSL group PSL(5,81)
gap> SpecialHomographyGroup(ps);
The FinInG PSL group PSL(5,81)
gap> ps := ProjectiveSpace(5,3);
ProjectiveSpace(5, 3)
gap> SpecialProjectivityGroup(ps);
The FinInG PSL group PSL(6,3)
gap> SpecialHomographyGroup(ps);
```

```
The FinInG PSL group PSL(6,3)
gap> ps := ProjectiveSpace(2,2);
ProjectiveSpace(2, 2)
gap> SpecialProjectivityGroup(ps);
The FinInG PSL group PSL(3,2)
gap> SpecialHomographyGroup(ps);
The FinInG PSL group PSL(3,2)
```

5.4.4   IsProjectivityGroup

▷ IsProjectivityGroup   (Property)

IsProjectivityGroup is a property, which subgroups of a the CollineationGroup or a CorrelationCollineationGroup of a projective space might have. It checks whether the generators are projectivities. Of course ProjectivityGroup has this property.

5.4.5   IsCollineationGroup

▷ IsCollineationGroup   (Property)

IsCollineationGroup is a property, which subgroups of a the CorrelationCollineationGroup of a projective space might have. It checks whether the generators are collineations. Of course ProjectivityGroup and CollineationGroup have this property.

5.4.6   CorrelationCollineationGroup

▷ CorrelationCollineationGroup(*geom*)   (operation)
   Returns: the group of correlations and collineations of geom
   Let *geom* be the projective space $PG(n,q)$. This operation returns the correlations and collineations of $PG(n,q)$.

—————————————— Example ——————————————
```
gap> pg := PG(4,3);
ProjectiveSpace(4, 3)
gap> group := CorrelationCollineationGroup(pg);
The FinInG correlation-collineation group PGL(5,3) : 2
gap> pg := PG(3,8);
ProjectiveSpace(3, 8)
gap> group := CorrelationCollineationGroup(pg);
The FinInG correlation-collineation group PGammaL(4,8) : 2
```

5.5   Basic operations for projective groups

5.5.1   BaseField

▷ BaseField(*g*)   (operation)
   Returns: a field

g must be a projective group. This function finds the base field of the vector space on which the group acts.

5.5.2 Dimension

▷ Dimension(*g*) (attribute)

Returns: a number

g must be a projective group. This function finds the dimension of the vector space on which the group acts.

5.6 Natural embedding of a collineation group in a correlation-collineation group

In FinInG a collineation group is not constructed as a subgroup of a correlation group. However, collineations can be multiplied with correlations (if they both belong mathematically to the same correlation group.

```
 _____ Example _____
 gap> x := Random(CollineationGroup(PG(3,4)));
 < a collineation: <cmat 4x4 over GF(2,2)>, F^2>
 gap> y := Random(CorrelationCollineationGroup(PG(3,4)));
 <projective element with Frobenius with projectivespace isomorphism: <immutabl
 e cmat 4x4 over GF(2,
 2)>, F^0, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
 3,GF(2^2)) ) ) >
 gap> x*y;
 <projective element with Frobenius with projectivespace isomorphism: <cmat 4x
 4 over GF(2,2)>, F^
 2, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
 3,GF(2^2)) ) ) >
```

5.6.1 Embedding

▷ Embedding(*coll, corr*) (function)

Let *coll* be a the full collineation group of a projective space, and *corr* its full correlation group. FinInG provides a method for this operation Embedding, returning the natural embedding from *coll* into *corr*. Remark that only an embedding of a collineation group into a correlation group with exactly the same underlying projective space is possible.

```
 _____ Example _____
 gap> coll := CollineationGroup(PG(4,8));
 The FinInG collineation group PGammaL(5,8)
 gap> corr := CorrelationCollineationGroup(PG(4,8));
 The FinInG correlation-collineation group PGammaL(5,8) : 2
 gap> phi := Embedding(coll,corr);
 MappingByFunction( The FinInG collineation group PGammaL(5,8), The FinInG corr
 elation-collineation group PGammaL(5,8) : 2, function( y ) ... end )
```

5.7 Basic action of projective group elements

5.7.1 \^

▷ \^(*x, g*) (operation)

Returns: a subspace of a projective space

This is an operation which returns the image of *x*, a subspace of a projective space, under *g*, an element of the projective group, the collineation group, or the correlation group.

5.8 Projective group actions

In this section we give more detailed about the actions that are used in FinInG for projective groups. Consider the projective space $PG(n,q)$. As described in Chapter 4, a point of $PG(n,q)$ is represented by a row vector and a *k*-dimensional subspace of $PG(n,q)$ is represented by a $(k+1) \times (n+1)$ matrix.

Consider a point *p* with row vector $(x_0, x_1, \ldots, x_n)$, and a collineation or correlation ϕ with underlying matrix *A* and field automorphism θ. Define the row vector $(y_0, y_1, \ldots, y_n) = ((x_0, x_1, \ldots, x_n)A)^{\theta}$. When ϕ is a collineation, p^{ϕ} is the point with underlying row vector $(y_0, y_1, \ldots, y_n)$. When ϕ is a correlation, is a hyperplane of $PG(n,q)$ with equation $y_0 X_0 + y_1 X_1 + \ldots + y_n X_n$. The action of collineations or correlations on points determines the action on subspaces of arbitrary dimension completely.

5.8.1 OnProjSubspaces

▷ OnProjSubspaces(*subspace, el*) (function)

Returns: a subspace of a projective space

This is a global function that returns the action of an element *el* of the collineation group on a subspace *subspace* of a projective space.

IMPORTANT: This function should only be used for objects *el* in the category IsProjGrpElWithFrob! This is because this function does not check whether *el* is a correlation or a collineation. So when *el* is a object in the category IsProjGrpElWithFrobWithPSIsom, and *el* is a correlation (i.e. the associated PSIsom is NOT the identity) then this action will not give the image of the *subspace* under the correlation *el*. For the action of an object *el* in the category IsProjGrpElWithFrobWithPSIsom, the action OnProjSubspacesExtended (8.3.1) should be used.

```
———————————————————————— Example ————————————————————————
gap> ps := ProjectiveSpace(4,27);
ProjectiveSpace(4, 27)
gap> p := VectorSpaceToElement(ps,[ Z(3^3)^22,Z(3^3)^10,Z(3^3),Z(3^3)^3,Z(3^3)^3]);
<a point in ProjectiveSpace(4, 27)>
gap> ps := ProjectiveSpace(3,27);
ProjectiveSpace(3, 27)
gap> p := VectorSpaceToElement(ps,[ Z(3^3)^22,Z(3^3)^10,Z(3^3),Z(3^3)^3]);
<a point in ProjectiveSpace(3, 27)>
gap> Display(p);
[16nh]
gap> mat := [[ Z(3^3)^25,Z(3^3)^6,Z(3^3)^7,Z(3^3)^15],
>    [Z(3^3)^9,Z(3)^0,Z(3^3)^10,Z(3^3)^18],
>    [Z(3^3)^19,0*Z(3),Z(3),Z(3^3)^12],
>    [Z(3^3)^4,Z(3^3),Z(3^3),Z(3^3)^22]];
[ [ Z(3^3)^25, Z(3^3)^6, Z(3^3)^7, Z(3^3)^15 ],
```

```
      [ Z(3^3)^9, Z(3)^0, Z(3^3)^10, Z(3^3)^18 ],
      [ Z(3^3)^19, 0*Z(3), Z(3), Z(3^3)^12 ],
      [ Z(3^3)^4, Z(3^3), Z(3^3), Z(3^3)^22 ] ]
gap> theta := FrobeniusAutomorphism(GF(27));
FrobeniusAutomorphism( GF(3^3) )
gap> phi := CollineationOfProjectiveSpace(mat,theta,GF(27));
< a collineation: <cmat 4x4 over GF(3,3)>, F^3>
gap> r := OnProjSubspaces(p,phi);
<a point in ProjectiveSpace(3, 27)>
gap> Display(r);
[1..l]
gap> vect := [[Z(3^3)^9,Z(3^3)^5,Z(3^3)^19,Z(3^3)^17],
>    [Z(3^3)^22,Z(3^3)^22,Z(3^3)^4,Z(3^3)^17],
>    [Z(3^3)^8,0*Z(3),Z(3^3)^24,Z(3^3)^21]]);
[ [ Z(3^3)^9, Z(3^3)^5, Z(3^3)^19, Z(3^3)^17 ],
  [ Z(3^3)^22, Z(3^3)^22, Z(3^3)^4, Z(3^3)^17 ],
  [ Z(3^3)^8, 0*Z(3), Z(3^3)^24, Z(3^3)^21 ] ]
gap> s := VectorSpaceToElement(ps,vect);
<a plane in ProjectiveSpace(3, 27)>
gap> r := OnProjSubspaces(s,phi);
<a plane in ProjectiveSpace(3, 27)>
gap> Display(r);
[[1..c]
 [.1.7]
 [..17]
]
```

5.8.2 ActionOnAllProjPoints

▷ ActionOnAllProjPoints(*g*)   (function)

g must be a projective group. This function returns the action homomorphism of *g* acting on its projective points. This function is used by NiceMonomorphism when the number of points is small enough for the action to be easy to calculate.

5.8.3 OnProjSubspacesExtended

▷ OnProjSubspacesExtended(*subspace, el*)   (function)

Returns: a subspace of a projective space

This should be used for the action of elements in the category IsProjGrpElWithFrobWithPSIsom where *subspace* is a subspace of a projective or polar space and *el* is an element of the correlation group of the ambient geometry of *subspace*. This function returns the image of *subspace* under *el*, which is a subspace of the same dimension as *subspace* if *el* is a collineation and an element of codimension equal to the dimension of *subspace* if *el* is a correlation.

```
  ———————————————————— Example ————————————————————
gap> ps := ProjectiveSpace(3,27);
ProjectiveSpace(3, 27)
gap> mat := IdentityMat(4,GF(27));
[ [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
```

```
   [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ], [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ]
gap> delta := StandardDualityOfProjectiveSpace(ps);
StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
3,GF(3^3)) ) )
gap> frob := FrobeniusAutomorphism(GF(27));
FrobeniusAutomorphism( GF(3^3) )
gap> phi := CorrelationOfProjectiveSpace(mat,frob,GF(27),delta);
<projective element with Frobenius with projectivespace isomorphism: <cmat 4x
4 over GF(3,3)>, F^
3, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
3,GF(3^3)) ) ) >
gap> p := Random(Points(ps));
<a point in ProjectiveSpace(3, 27)>
gap> OnProjSubspacesExtended(p,phi);
<a plane in ProjectiveSpace(3, 27)>
gap> l := Random(Lines(ps));
<a line in ProjectiveSpace(3, 27)>
gap> OnProjSubspacesExtended(p,phi);
<a plane in ProjectiveSpace(3, 27)>
gap> psi := CorrelationOfProjectiveSpace(mat,frob^2,GF(27));
<projective element with Frobenius with projectivespace isomorphism: <cmat 4x
4 over GF(3,3)>, F^9, IdentityMapping( <All elements of ProjectiveSpace(3,
27)> ) >
gap> OnProjSubspacesExtended(p,psi);
<a point in ProjectiveSpace(3, 27)>
gap> OnProjSubspacesExtended(l,psi);
<a line in ProjectiveSpace(3, 27)>
```

5.9 Special subgroups of the projectivity group

A *transvection* of the vector space $V = V(n+1,F)$ is a linear map τ from V to V with matrix M such that $rk(M-I) = 1$ and $(M-I)^2 = 0$. Different equivalent definitions are found in the literature, here we followed [Cam00a]. Choosing a basis $e_1, \ldots, e_n, e_{n+1}$ such that $e_1, \ldots, e_n$ generates the kernel of $M-I$, it follows that M equals

$$\begin{pmatrix} 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \\ x_1 & x_2 & \ldots & x_{n-1} & 1 \end{pmatrix}.$$

It is also a well known fact that all transvections generate the group $SL(n+1,F)$. A transvection gives rise to a projectivity of $PG(n,F)$, we call such an element an *elation*, and it is a projectivity ϕ fixing a hyperplane H pointwise, and such that there exists exactly one point $p \in H$ such that all hyperplanes through p are stabilized. The hyperplane H is called the *axis* of ϕ, and the point p is called the *centre* of ϕ. As a transvection is an element of $SL(n,F)$, an elation is an element of $PSL(n,F)$. An elation is completely determind by its axis and the image of one point (not contained in the axis). The group of elations with a given axis and centre, is isomorphic with the additive group of F. Finally, the group of all elations with a given axis H, acts regularly on the points of $PG(n,F) \setminus H$, and is isomorphic with

the additive group of the vectorspace $V(n, F)$.

5.9.1 ElationOfProjectiveSpace

▷ ElationOfProjectiveSpace(*sub, point1, point2*) (operation)

Returns: the unique elation with axis *sub* mapping *point1* on *point2*

It is checked whether the two points do not belong to *sub*. If *point1* equals *point2*, the identity mapping is returned.

```
─────────────────── Example ───────────────────
gap> ps := PG(3,9);
ProjectiveSpace(3, 9)
gap> sub := VectorSpaceToElement(ps,[[1,0,1,0],[0,1,0,1],[1,2,3,0]]*Z(3)^0);
<a plane in ProjectiveSpace(3, 9)>
gap> p1 := VectorSpaceToElement(ps,[1,0,1,2]*Z(3)^0);
<a point in ProjectiveSpace(3, 9)>
gap> p2 := VectorSpaceToElement(ps,[1,2,0,2]*Z(3)^0);
<a point in ProjectiveSpace(3, 9)>
gap> phi := ElationOfProjectiveSpace(sub,p1,p2);
< a collineation: <cmat 4x4 over GF(3,2)>, F^0>
```

5.9.2 ProjectiveElationGroup

▷ ProjectiveElationGroup(*axis, centre*) (operation)
▷ ProjectiveElationGroup(*axis*) (operation)

Returns: A group of elations

The first version returns the group of elations with with given axis *axis* and centre *centre*. It is checked whether *centre* belongs to *axis*. The second version returns the group of elations with given axis *axis*.

```
─────────────────── Example ───────────────────
gap> ps := PG(2,27);
ProjectiveSpace(2, 27)
gap> sub := VectorSpaceToElement(ps,[[1,0,1,],[0,1,0]]*Z(3)^0);
<a line in ProjectiveSpace(2, 27)>
gap> p := VectorSpaceToElement(ps,[1,1,1]*Z(3)^0);
<a point in ProjectiveSpace(2, 27)>
gap> g := ProjectiveElationGroup(sub,p);
<projective collineation group with 3 generators>
gap> Order(g);
27
gap> StructureDescription(g);
"C3 x C3 x C3"
gap> ps := PG(3,4);
ProjectiveSpace(3, 4)
gap> sub := Random(Hyperplanes(ps));
<a plane in ProjectiveSpace(3, 4)>
gap> g := ProjectiveElationGroup(sub);
<projective collineation group with 6 generators>
gap> Order(g);
64
gap> Transitivity(g,Difference(Points(ps),Points(sub)),OnProjSubspaces);
```

```
    1
gap> StructureDescription(g);
"C2 x C2 x C2 x C2 x C2 x C2"
```

A *homology* of the projective space $PG(n,q)$ is a collineation fixing a hyperplane H pointwise and fixing one more point $p \notin H$. It is easily seen that after a suitable choice of a basis for the space, the matrix of a homology is a diagonal matrix with all its diagonal entries except one equal to 1. We call the hyperplane the *axis* and the point the *centre* of the homology. Homologies with a common axis and centre are a group isomorphic to the multiplicative group of the field $GF(q)$.

5.9.3 HomologyOfProjectiveSpace

▷ HomologyOfProjectiveSpace(*sub, centre, point1, point2*) (operation)

  Returns: the unique homology with axis *sub* and centre *centre* that maps *point1* on *point2*

  It is checked whether the three points do not belong to *sub* and whether they are collinear. If *point1* equals *point2*, the identity mapping is returned.

```
─────────────────────── Example ───────────────────────
gap> ps := PG(3,81);
ProjectiveSpace(3, 81)
gap> sub := VectorSpaceToElement(ps,[[1,0,1,0],[0,1,0,1],[1,2,3,0]]*Z(3)^0);
<a plane in ProjectiveSpace(3, 81)>
gap> centre := VectorSpaceToElement(ps,[0*Z(3),Z(3)^0,Z(3^4)^36,0*Z(3)]);
<a point in ProjectiveSpace(3, 81)>
gap> p1 := VectorSpaceToElement(ps,[0*Z(3),Z(3)^0,Z(3^4)^51,0*Z(3)]);
<a point in ProjectiveSpace(3, 81)>
gap> p2 := VectorSpaceToElement(ps,[0*Z(3),Z(3)^0,Z(3^4)^44,0*Z(3)]);
<a point in ProjectiveSpace(3, 81)>
gap> phi := HomologyOfProjectiveSpace(sub,centre,p1,p2);
< a collineation: <cmat 4x4 over GF(3,4)>, F^0>
```

5.9.4 ProjectiveHomologyGroup

▷ ProjectiveHomologyGroup(*axis, centre*) (operation)

  Returns: the group of homologies with with given axis *axis* and centre *centre*.

  It is checked whether *centre* does not belong to *axis*.

```
─────────────────────── Example ───────────────────────
gap> ps := PG(2,27);
ProjectiveSpace(2, 27)
gap> sub := VectorSpaceToElement(ps,[[1,0,1,],[0,1,0]]*Z(3)^0);
<a line in ProjectiveSpace(2, 27)>
gap> p := VectorSpaceToElement(ps,[1,0,2]*Z(3)^0);
<a point in ProjectiveSpace(2, 27)>
gap> g := ProjectiveHomologyGroup(sub,p);
<projective collineation group with 1 generators>
gap> Order(g);
26
gap> StructureDescription(g);
"C26"
```

5.10   Nice Monomorphisms

A *nice monomorphism* of a group G is roughly just a permutation representation of G on a suitable action domain. An easy example is the permutation action of the full collineation group of a projective space on its points. FinInG provides (automatic) functionality to compute nice monomorphisms. Typically, for a geometry S with G a (subgroup of the) collineation group of S, a nice monomorphism for G is a homomorphism from G to the permutation action of S on a collection of elements of S. Thus, to obtain such a homomorphism, one has to enumerate the collection of elements. As nice monomorphisms for projective semilinear groups are often computed as a byproduct of some operations, suddenly, these operations get time consuming (when executed for the first time). In general, it is decided automatically whether a nice monomorphism is computed or not. A typical example is the following.

```
 _____ Example _____
 gap> pg := PG(4,8);
 ProjectiveSpace(4, 8)
 gap> group := CollineationGroup(pg);
 The FinInG collineation group PGammaL(5,8)
 gap> HasNiceMonomorphism(group);
 false
 gap> Random(group);
 < a collineation: <cmat 5x5 over GF(2,3)>, F^4>
 gap> time;
 1028
 gap> HasNiceMonomorphism(group);
 true
 gap> Random(group);
 < a collineation: <cmat 5x5 over GF(2,3)>, F^0>
 gap> time;
 3
```

5.10.1   NiceMonomorphism

▷ NiceMonomorphism(*g*)   (operation)

Returns: an action, i.e. a group homomorphism

g is a projective semilinear group. If *g* was constructed as a group stabilizing a geometry, the action of *g* on the points of the geometry is returned.

```
 _____ Example _____
 gap> g := HomographyGroup(PG(4,8));
 The FinInG projectivity group PGL(5,8)
 gap> NiceMonomorphism(g);
 <action isomorphism>
 gap> Image(last);
 <permutation group of size 4638226007491010887680 with 2 generators>
 gap> g := CollineationGroup(PG(4,8));
 The FinInG collineation group PGammaL(5,8)
 gap> NiceMonomorphism(g);
 <action isomorphism>
 gap> Image(last);
 <permutation group of size 13914678022473032663040 with 3 generators>
```

5.10.2 NiceObject

▷ NiceObject(*g*) (operation)

Returns: a permutation group

g is a projective semilinear group. If *g* was constructed as a group stabilizing a geometry, the permutation representation of *g* acting on the points of the geometry is returned. This is actually equivalent with Image(NiceMonomorphism(*g*)).

```
_____ Example _____
gap> g := HomographyGroup(PG(4,8));
The FinInG projectivity group PGL(5,8)
gap> NiceObject(g);
<permutation group of size 4638226007491010887680 with 2 generators>
gap> g := CollineationGroup(PG(4,8));
The FinInG collineation group PGammaL(5,8)
gap> NiceObject(g);
<permutation group of size 13914678022473032663040 with 3 generators>
```

5.10.3 FINING

▷ FINING (global variable)

The global variable FINING stores a record with two components, FINING.Fast and FINING.LimitForCanComputeActionOnPoints By default, FINING.Fast is set to true. Setting FINING.Fast to false causes the use of the generic GAP function ActionHomomorphism instead of the functions NiceMonomorphismByDomain and NiceMonomorphismByOrbit, which both rely on the packages GenSS and Orb.

5.10.4 CanComputeActionOnPoints

▷ CanComputeActionOnPoints(*g*) (operation)

Returns: true or false

g must be a projective group. This function returns true if GAP can feasibly compute the action of *g* on the points of the projective space on which it acts. This function can be used (and is, by other parts of FinInG) to determine whether it is worth trying to compute the action. This function actually checks if the number of points of the corresponding projective space is less than the constant FINING.LimitForCanComputeActionOnPoints, which is by default set to 1000000. The next example requires about 500M of memory.

```
_____ Example _____
gap> NiceMonomorphism(CollineationGroup(ProjectiveSpace(7,8)));
Error, action on projective points not feasible to calculate called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 8 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> FINING.LimitForCanComputeActionOnPoints := 3*10^6;
3000000
gap> NiceMonomorphism(CollineationGroup(ProjectiveSpace(7,8)));
<action isomorphism>
```

```
gap> time;
39619
```

5.10.5 NiceMonomorphismByDomain

▷ NiceMonomorphismByDomain(*g, dom, op*)   (operation)

Returns: an action, i.e. a group homomorphism

This operaion is not intended for the user. It relies on GenSS and Orb. The argument g is a projective group (in the category `IsProjectiveGroupWithFrob`) with a set `Size` attribute, *dom* is an orbit of g, and *op* an operation suitable for x and g.

5.10.6 NiceMonomorphismByOrbit

▷ NiceMonomorphismByOrbit(*g, dom, op, orblen*)   (operation)

Returns: an action, i.e. a group homomorphism

This operaion is not intended for the user. It relies on GenSS and Orb. The argument g is a projective group (in the category `IsProjectiveGroupWithFrob`) with a set `Size` attribute, *dom* is an orbit of g, *op* an operation suitable for x and g, and *orblen* is the length of the final orbit.

Chapter 6

Polarities of Projective Spaces

A *polarity* of a incidence structure is an incidence reversing, bijective, and involutory map on the elements of the incidence structure. It is well known that every polarity of a projective space is just an involutory correlation of the projective space. The construction of correlations of a projective space is described in Chapter 5. In this chapter we describe methods and operations dealing with the construction and use of polarities of projective spaces in FinInG.

6.1   Creating polarities of projective spaces

Since polarities of a projective space necessarily have an involutory field automorphism as companion automorphism and the standard duality of the projective space as the companion projective space isomorphism, a polarity of a projective space is determined completely by a suitable matrix A. Every polarity of a projective space $PG(n, q)$ is listed in the following table, including the conditions on the matrix A.

	q odd	q even
hermitian	$A^\theta = A^T$	$A^\theta = A^T$
symplectic	$A^T = -A$	$A^T = A$, all $a_{ii} = 0$
orthogonal	$A^T = A$	
pseudo		$A^T = A$, not all $a_{ii} = 0$

Table: polarities of a projective space

A hermitian polarity of the projective space $PG(n, q)$ exists if and only if the field $GF(q)$ admits an involutory field automorphism θ.

It is well known that there is a correspondence between polarities of projective spaces and non-degenerate sesquilinear forms on the underlying vector space. Consider a sesquilinear form f on the vector space $V(n + 1, q)$. Then f induces a map on the elements of $PG(n, q)$ as follows: every element with underlying subspace α is mapped to the element with underlying subspace $\alpha^\perp$, i.e. the subspace of $V(n + 1, q)$ orthogonal to α with respect to the form f. It is clear that this induced map is a polarity of $PG(n, q)$. Also the converse is true, with any polarity of $PG(n, q)$ corresponds a sesquilinear form on $V(n + 1, q)$. The above classification of polarities of $PG(n, q)$ follows from the classification of sesquilinear forms on $V(n + 1, q)$. For more information, we refer to [HT91] and [KL90]. We mention that the implementation of the action of correlations on projective points (see 5.8) guarantees that a sesquilinear form with matrix M and field automorphism θ corresponds to a polarity with matrix M and field automorphism θ and vice versa.

In FinInG, polarities of projective spaces are always objects in the category IsPolarityOfProjectiveSpace, which is a subcategory of the category IsProjGrpElWithFrobWithPSIsom.

6.1.1 PolarityOfProjectiveSpace

▷ PolarityOfProjectiveSpace(`mat, f`)   (operation)

   Returns: a polarity of a projective space

   The underlying correlation of the projective space is constructed using matrix `mat`, field `f`, the identity mapping as field automorphism and the standard duality of the projective space. It is checked whether the matrix `mat` satisfies the necessary conditions to induce a polarity.

```
 _____ Example _____
 gap> mat := [[0,1,0],[1,0,0],[0,0,1]]*Z(169)^0;
 [ [ 0*Z(13), Z(13)^0, 0*Z(13) ], [ Z(13)^0, 0*Z(13), 0*Z(13) ],
   [ 0*Z(13), 0*Z(13), Z(13)^0 ] ]
 gap> phi := PolarityOfProjectiveSpace(mat,GF(169));
 <polarity of PG(2, GF(13^2)) >
```

6.1.2 PolarityOfProjectiveSpace

▷ PolarityOfProjectiveSpace(`mat, frob, f`)   (operation)
▷ HermitianPolarityOfProjectiveSpace(`mat, f`)   (operation)

   Returns: a polarity of a projective space

   The underlying correlation of the projective space is constructed using matrix `mat`, field automorphism `frob`, `f` and the standard duality of the projective space. It is checked whether the `mat` satisfies the necessary conditions to induce a polarity, and whether `frob` is a non-trivial involutory field automorphism. The second operation only needs the arguments `mat` and `f` to construct a hermitian polarity of a projective space, provided the field `f` allows an involutory field automorphism and `mat` satisfies the necessary conditions. The latter is checked by constructing the underlying hermitian form.

```
 _____ Example _____
 gap> mat := [[Z(11)^0,0*Z(11),0*Z(11)],[0*Z(11),0*Z(11),Z(11)],
 >      [0*Z(11),Z(11),0*Z(11)]];
 [ [ Z(11)^0, 0*Z(11), 0*Z(11) ], [ 0*Z(11), 0*Z(11), Z(11) ],
   [ 0*Z(11), Z(11), 0*Z(11) ] ]
 gap> frob := FrobeniusAutomorphism(GF(121));
 FrobeniusAutomorphism( GF(11^2) )
 gap> phi := PolarityOfProjectiveSpace(mat,frob,GF(121));
 <polarity of PG(2, GF(11^2)) >
 gap> psi := HermitianPolarityOfProjectiveSpace(mat,GF(121));
 <polarity of PG(2, GF(11^2)) >
 gap> phi = psi;
 true
```

6.1.3 PolarityOfProjectiveSpace

▷ PolarityOfProjectiveSpace(`form`)   (operation)

   Returns: a polarity of a projective space

The polarity of the projective space is constructed using a non-degenerate sesquilinear form `form`. It is checked whether the given form is non-degenerate.

```
_____ Example _____
 gap> mat := [[0,1,0,0],[1,0,0,0],[0,0,0,1],[0,0,1,0]]*Z(16)^0;
 [ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
   [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ]
 gap> form := BilinearFormByMatrix(mat,GF(16));
 < bilinear form >
 gap> phi := PolarityOfProjectiveSpace(form);
 <polarity of PG(3, GF(2^4)) >
```

6.1.4 PolarityOfProjectiveSpace

▷ PolarityOfProjectiveSpace(*ps*)   (operation)

    Returns: a polarity of a projective space

    The polarity of the projective space is constructed using the non-degenerate sesquilinear form that defines the polar space *ps*. When *ps* is a parabolic quadric in even characteristic, no polarity of the ambient projective space can be associated to *ps*, and an error message is returned.

```
_____ Example _____
 gap> ps := HermitianPolarSpace(4,64);
 H(4, 8^2)
 gap> phi := PolarityOfProjectiveSpace(ps);
 <polarity of PG(4, GF(2^6)) >
 gap> ps := ParabolicQuadric(6,8);
 Q(6, 8)
 gap> PolarityOfProjectiveSpace(ps);
 Error, no polarity of the ambient projective space can be associated to <ps> called from
 <function "unknown">( <arguments> )
  called from read-eval loop at line 11 of *stdin*
 you can 'quit;' to quit to outer loop, or
 you can 'return;' to continue
 brk> quit;
```

6.2 Operations, attributes and properties for polarities of projective spaces

6.2.1 SesquilinearForm

▷ SesquilinearForm(*phi*)   (attribute)

    Returns: a sesquilinear form

    The sesquilinear form corresponding to the given polarity *phi* is returned.

```
_____ Example _____
 gap> mat := [[0,-2,0,1],[2,0,3,0],[0,-3,0,1],[-1,0,-1,0]]*Z(19)^0;
 [ [ 0*Z(19), Z(19)^10, 0*Z(19), Z(19)^0 ],
   [ Z(19), 0*Z(19), Z(19)^13, 0*Z(19) ],
   [ 0*Z(19), Z(19)^4, 0*Z(19), Z(19)^0 ],
   [ Z(19)^9, 0*Z(19), Z(19)^9, 0*Z(19) ] ]
 gap> phi := PolarityOfProjectiveSpace(mat,GF(19));
```

```
    <polarity of PG(3, GF(19)) >
    gap> form := SesquilinearForm(phi);
    < non-degenerate bilinear form >
```

6.2.2 BaseField

▷ BaseField(phi)   (attribute)

   Returns: a field

  The base field over which the polarity phi was constructed.

```
——————————————— Example ———————————————
gap> mat := [[1,0,0],[0,0,2],[0,2,0]]*Z(5)^0;
[ [ Z(5)^0, 0*Z(5), 0*Z(5) ], [ 0*Z(5), 0*Z(5), Z(5) ],
  [ 0*Z(5), Z(5), 0*Z(5) ] ]
gap> phi := PolarityOfProjectiveSpace(mat,GF(25));
<polarity of PG(2, GF(5^2)) >
gap> BaseField(phi);
GF(5^2)
```

6.2.3 GramMatrix

▷ GramMatrix(phi)   (attribute)

   Returns: a matrix

  The Gram matrix of the polarity phi.

```
——————————————— Example ———————————————
gap> mat := [[1,0,0],[0,0,3],[0,3,0]]*Z(11)^0;
[ [ Z(11)^0, 0*Z(11), 0*Z(11) ], [ 0*Z(11), 0*Z(11), Z(11)^8 ],
  [ 0*Z(11), Z(11)^8, 0*Z(11) ] ]
gap> phi := PolarityOfProjectiveSpace(mat,GF(11));
<polarity of PG(2, GF(11)) >
gap> GramMatrix(phi);
<immutable cmat 3x3 over GF(11,1)>
```

6.2.4 CompanionAutomorphism

▷ CompanionAutomorphism(phi)   (attribute)

   Returns: a field automorphism

  The involutory field automorphism accompanying the polarity phi.

```
——————————————— Example ———————————————
gap> mat := [[0,2,0,0],[2,0,0,0],[0,0,0,5],[0,0,5,0]]*Z(7)^0;
[ [ 0*Z(7), Z(7)^2, 0*Z(7), 0*Z(7) ], [ Z(7)^2, 0*Z(7), 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^5 ], [ 0*Z(7), 0*Z(7), Z(7)^5, 0*Z(7) ] ]
gap> phi := HermitianPolarityOfProjectiveSpace(mat,GF(49));
<polarity of PG(3, GF(7^2)) >
gap> CompanionAutomorphism(phi);
FrobeniusAutomorphism( GF(7^2) )
```

6.2.5 IsHermitianPolarityOfProjectiveSpace

▷ IsHermitianPolarityOfProjectiveSpace(phi)   (property)

Returns: true or false

The polarity phi is a hermitian polarity of a projective space if and only if the underlying matrix is hermitian.

```
_____ Example _____
 gap> mat := [[0,2,7,1],[2,0,3,0],[7,3,0,1],[1,0,1,0]]*Z(19)^0;
 [ [ 0*Z(19), Z(19), Z(19)^6, Z(19)^0 ], [ Z(19), 0*Z(19), Z(19)^13, 0*Z(19) ],
   [ Z(19)^6, Z(19)^13, 0*Z(19), Z(19)^0 ],
   [ Z(19)^0, 0*Z(19), Z(19)^0, 0*Z(19) ] ]
 gap> frob := FrobeniusAutomorphism(GF(19^4));
 FrobeniusAutomorphism( GF(19^4) )
 gap> phi := PolarityOfProjectiveSpace(mat,frob^2,GF(19^4));
 <polarity of PG(3, GF(19^4)) >
 gap> IsHermitianPolarityOfProjectiveSpace(phi);
 true
```

6.2.6 IsSymplecticPolarityOfProjectiveSpace

▷ IsSymplecticPolarityOfProjectiveSpace(phi)   (property)

Returns: true or false

The polarity phi is a symplectic polarity of a projective space if and only if the underlying matrix is symplectic.

```
_____ Example _____
 gap> mat := [[0,0,1,0],[0,0,0,1],[1,0,0,0],[0,1,0,0]]*Z(8)^0;
 [ [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
   [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ]
 gap> phi := PolarityOfProjectiveSpace(mat,GF(8));
 <polarity of PG(3, GF(2^3)) >
 gap> IsSymplecticPolarityOfProjectiveSpace(phi);
 true
```

6.2.7 IsOrthogonalPolarityOfProjectiveSpace

▷ IsOrthogonalPolarityOfProjectiveSpace(phi)   (property)

Returns: true or false

The polarity phi is an orthogonal polarity of a projective space if and only if the underlying matrix is symmetric and the characteristic of the field is odd.

```
_____ Example _____
 gap> mat := [[1,0,2,0],[0,2,0,1],[2,0,0,0],[0,1,0,0]]*Z(9)^0;
 [ [ Z(3)^0, 0*Z(3), Z(3), 0*Z(3) ], [ 0*Z(3), Z(3), 0*Z(3), Z(3)^0 ],
   [ Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ] ]
 gap> phi := PolarityOfProjectiveSpace(mat,GF(9));
 <polarity of PG(3, GF(3^2)) >
 gap> IsOrthogonalPolarityOfProjectiveSpace(phi);
 true
```

6.2.8 IsPseudoPolarityOfProjectiveSpace

▷ IsPseudoPolarityOfProjectiveSpace(*phi*) (property)

Returns: true or false

The polarity *phi* is a pseudo-polarity of a projective space if and only if the underlying matrix is symmetric, not all elements on the main diagonal are zero and the characteristic of the field is even.

```
———————————————————— Example ————————————————————
gap> mat := [[1,0,1,0],[0,1,0,1],[1,0,0,0],[0,1,0,0]]*Z(16)^0;
[ [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> phi := PolarityOfProjectiveSpace(mat,GF(16));
<polarity of PG(3, GF(2^4)) >
gap> IsPseudoPolarityOfProjectiveSpace(phi);
true
```

6.3 Polarities, absolute points, totally isotropic elements and finite classical polar spaces

We already mentioned the equivalence between polarities of $PG(n,q)$ and sesquilinear forms on $V(n+1,q)$, hence there is a relation between polarities of $PG(n,q)$ and polar spaces induced by sesquilinear forms. The following concepts express these relations geometrically.

Suppose that ϕ is a polarity of $PG(n,q)$ and that α is an element of $PG(n,q)$. We call α a *totally isotropic element* or an *absolute element* if and only if α is incident with α^ϕ. An absolute element that is a point is also called an *absolute point* or an *isotropic point*. It is clear that an element of $PG(n,q)$ is absolute if and only if the underlying vector space is totally isotropic with respect to the sesquilinear form equivalent to ϕ. Hence the absolute elements induce a *finite classical polar space*, the same that is induced by the equivalent sesquilinear form. When ϕ is a pseudo-polarity, the set of absolute elements are the elements of a hyperplane of $PG(n,q)$.

We restrict our introduction to finite classical polar spaces in this section to the following examples. Many aspects of these geometries are extensively described in Chapter 7.

6.3.1 GeometryOfAbsolutePoints

▷ GeometryOfAbsolutePoints(*f*) (operation)

Returns: a polar space or a hyperplane

When *f* is not a pseudo-polarity, this operation returns the polar space induced by *f*. When *f* is a pseudo-polarity, this operation returns the hyperplane containing all absolute elements.

```
———————————————————— Example ————————————————————
gap> mat := IdentityMat(4,GF(16));
[ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ], [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> phi := HermitianPolarityOfProjectiveSpace(mat,GF(16));
<polarity of PG(3, GF(2^4)) >
gap> geom := GeometryOfAbsolutePoints(phi);
<polar space in ProjectiveSpace(3,GF(2^4)): x_1^5+x_2^5+x_3^5+x_4^5=0 >
gap> mat := [[1,0,0,0],[0,0,1,1],[0,1,1,0],[0,1,0,0]]*Z(32)^0;
[ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ]
```

```
gap> phi := PolarityOfProjectiveSpace(mat,GF(32));
<polarity of PG(3, GF(2^5)) >
gap> geom := GeometryOfAbsolutePoints(phi);
<a plane in ProjectiveSpace(3, 32)>
```

6.3.2   AbsolutePoints

▷ AbsolutePoints(f)   (operation)

Returns: a set of points

This operation returns all points that are absolute with respect to *f*.

```
                              ──────── Example ────────
gap> mat := IdentityMat(4,GF(3));
[ [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ], [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ]
gap> phi := PolarityOfProjectiveSpace(mat,GF(3));
<polarity of PG(3, GF(3)) >
gap> points := AbsolutePoints(phi);
<points of Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>
gap> List(points);
[ <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0>,
  <a point in Q+(3, 3): x_1^2+x_2^2+x_3^2+x_4^2=0> ]
```

6.3.3   PolarSpace

▷ PolarSpace(f)   (operation)

Returns: a polar space

When *f* is not a pseudo-polarity, this operation returns the polar space induced by *f*.

```
                              ──────── Example ────────
gap> mat := [[1,0,0,0],[0,0,1,1],[0,1,1,0],[0,1,0,0]]*Z(32)^0;
[ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> phi := PolarityOfProjectiveSpace(mat,GF(32));
<polarity of PG(3, GF(2^5)) >
gap> ps := PolarSpace(phi);
Error, <polarity> is pseudo and does not induce a polar space called from
```

```
 <function "unknown">( <arguments> )
  called from read-eval loop at line 10 of *stdin*
 you can 'quit;' to quit to outer loop, or
 you can 'return;' to continue
 brk> quit;
 gap> mat := IdentityMat(5,GF(7));
 [ [ Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7) ],
   [ 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7) ],
   [ 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7) ],
   [ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7) ],
   [ 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0 ] ]
 gap> phi := PolarityOfProjectiveSpace(mat,GF(7));
 <polarity of PG(4, GF(7)) >
 gap> ps := PolarSpace(phi);
 <polar space in ProjectiveSpace(4,GF(7)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2=0 >
```

6.4   Commuting polarities

FinInG constructs polarities of projective spaces as correlations. This allows polarities to be multiplied easily, resulting in a collineation. The resulting collineation is constructed in the correlation group but can be mapped onto its unique representative in the collineation group. We provide an example with two commuting polarities.

```
_____ Example _____
 gap> mat := [[0,1,0,0],[1,0,0,0],[0,0,0,1],[0,0,1,0]]*Z(5)^0;
 [ [ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5) ], [ Z(5)^0, 0*Z(5), 0*Z(5), 0*Z(5) ],
   [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0 ], [ 0*Z(5), 0*Z(5), Z(5)^0, 0*Z(5) ] ]
 gap> phi := HermitianPolarityOfProjectiveSpace(mat,GF(25));
 <polarity of PG(3, GF(5^2)) >
 gap> mat2 := IdentityMat(4,GF(5));
 [ [ Z(5)^0, 0*Z(5), 0*Z(5), 0*Z(5) ], [ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5) ],
   [ 0*Z(5), 0*Z(5), Z(5)^0, 0*Z(5) ], [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0 ] ]
 gap> psi := PolarityOfProjectiveSpace(mat2,GF(25));
 <polarity of PG(3, GF(5^2)) >
 gap> phi*psi = psi*phi;
 true
 gap> g := CorrelationCollineationGroup(PG(3,25));
 The FinInG correlation-collineation group PGammaL(4,25) : 2
 gap> h := CollineationGroup(PG(3,25));
 The FinInG collineation group PGammaL(4,25)
 gap> hom := Embedding(h,g);
 MappingByFunction( The FinInG collineation group PGammaL(4,25), The FinInG cor
 relation-collineation group PGammaL(4,25) : 2, function( y ) ... end )
 gap> coll := PreImagesRepresentative(hom,phi*psi);
 < a collineation: <cmat 4x4 over GF(5,2)>, F^5>
```

Chapter 7

Finite Classical Polar Spaces

In this chapter we describe how to use FinInG to work with finite classical polar spaces.

7.1 Finite Classical Polar Spaces

A *polar space* is a point-line incidence geometry, satisfying the famous one-or-all axiom, i.e. for any point P, not incident with a line l, P is collinear with exactly one point of l or with all points of l. The axiomatic treatment of polar spaces has its foundations in [Vel59], [Tit74], and [BS74], the latter in which the one-or-all axiom is described. Polar spaces are axiomatically, point-line geometries, but may contain higher dimensional projective subspaces too. All maximal subspaces have the same projective dimension, and this determines the rank of the polar space.

Well known examples of *finite* polar spaces are the geometries attached to sesquilinear and quadratic forms of vector spaces over a finite field, these geometries are called the *finite classical polar spaces*. For a given sesquilinear, respectively quadratic, form f, the elements of the associated geometry are the totally isotropic, respectively totally singular, subspaces of the vectors space with relation to the form f. The treatment of the forms is done through the package Forms.

From the axiomatic point of view, a polar space is a point-line geometry, and has rank at least 2. Considering a sesquilinear or quadratic form f, of Witt index 1, the associated geometry consists only of projective points, and is then in the axiomatic treatment, not a polar space. However, as is the case for projective spaces, we will consider the rank one geometries associated to forms of Witt index 1 as examples of classical polar spaces. Even the elliptic quadric on the projective line, a *geometry* associated to an elliptic quadratic form on a two dimensional vector space over a finite field, is considered as a classical polar space, though it has no singular subspaces. The reason for this treatment is that most, if not all, methods for operations applicable on these geometries, rely on the same algebraic methodology. So, in FinInG, a classical polar space (sometimes abbreviated to polar space), is the geometry associated with a sesquilinear or quadratic form on a finite dimensional vector space over a finite field.

7.1.1 IsClassicalPolarSpace

▷ IsClassicalPolarSpace   (Category)

This category is a subcategory of IsLieGeometry, and contains all the geometries associated to a non-degenerate sesquilinear or quadratic form.

The underlying vector space and matrix group are to our advantage in the treatment of classical polar spaces. We refer the reader to [HT91] and [Cam00b] for the necessary background theory (if it is not otherwise provided), and we follow the approach of [Cam00b] to introduce all different flavours.

Consider the projective space $PG(n,q)$ with underlying vector space $V(n+1,q)$. Consider a non-degenerate sesquilinear form f. Then f is Hermitian, alternating or symmetric. When the characteristic of the field is odd, respectively even, a symmetric bilinear form is called orthogonal, respectively, pseudo. We do not consider the pseudo case, so we suppose that f is Hermitian, symplectic or orthogonal. The classical polar space associated with f is the incidence geometry whose elements are of the subspaces of $PG(n,q)$ whose underlying vector subspace is totally isotropic with relation to f. We call a polar space *Hermitian*, respectively, *symplectic*, *orthogonal*, if the underlying sesquilinear form is Hermitian, respectively, symplectic, orthogonal.

Symmetric bilinear forms have completely different geometric properties in even characteristic than in odd characteristic. On the other hand, polar spaces geometrically comparable to orthogonal polar spaces in odd characteristic, do exist in even characteristic. The algebraic background is now established by quadratic forms on a vector space instead of bilinear forms. Consider a non-singular quadratic form q on a vector space $V(n+1,q)$. The classical polar space associated with f is the incidence geometry whose elements are the subspaces of $PG(n,q)$ whose underlying vector subspace is totally singular with relation to q. The connection with orthogonal polar spaces in odd characteristic is clear, since in odd characteristic, quadratic forms and symmetric bilinear forms are equivalent. Therefore, we call polar spaces with an underlying quadratic form in even characteristic also *orthogonal* polar spaces.

7.1.2 PolarSpace

▷ PolarSpace(*form*) (operation)
▷ PolarSpace(*pol*) (operation)

Returns: a classical polar space

form must be a sesquilinear or quadratic form created by use of the GAP package Forms. In the second variant, the argument *pol* must be a polarity of a projective space. An error message will be displayed if *pol* is a pseudo polarity. We refer to Chapter 6 for more information on polarities of projective spaces, and more particularly to Section 6.3 for the connection between polarities and forms.

```
_____ Example _____
gap> mat := [[0,0,0,1],[0,0,-2,0],[0,2,0,0],[-1,0,0,0]]*Z(5)^0;
[ [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0 ], [ 0*Z(5), 0*Z(5), Z(5)^3, 0*Z(5) ],
  [ 0*Z(5), Z(5), 0*Z(5), 0*Z(5) ], [ Z(5)^2, 0*Z(5), 0*Z(5), 0*Z(5) ] ]
gap> form := BilinearFormByMatrix(mat,GF(25));
< bilinear form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(
3,GF(5^2)): x1*y4+Z(5)^3*x2*y3+Z(5)*x3*y2-x4*y1=0 >
gap> r := PolynomialRing(GF(32),4);
GF(2^5)[x_1,x_2,x_3,x_4]
gap> poly := r.3*r.2+r.1*r.4;
x_1*x_4+x_2*x_3
gap> form := QuadraticFormByPolynomial(poly,r);
< quadratic form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(3,GF(2^5)): x_1*x_4+x_2*x_3=0 >
```

```
gap> mat := IdentityMat(5,GF(7));
[ [ Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0 ] ]
gap> phi := PolarityOfProjectiveSpace(mat,GF(7));
<polarity of PG(4, GF(7)) >
gap> ps := PolarSpace(phi);
<polar space in ProjectiveSpace(4,GF(7)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2=0 >
```

FinInG relies on the package Forms for its facility with sesquilinear and quadratic forms. One can specify a polar space with a user-defined form, and we refer to the documentation for Forms for information on how one can create and use forms. Here we just display a worked example.

```
————————————————————————— Example —————————————————————————
gap> id := IdentityMat(7, GF(3));;
gap> form := QuadraticFormByMatrix(id, GF(3));
< quadratic form >
gap> ps := PolarSpace( form );
<polar space in ProjectiveSpace(
6,GF(3)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2=0 >
gap> psl32 := PSL(3,2);
Group([ (4,6)(5,7), (1,2,4)(3,6,5) ])
gap> reps:=[[1,1,1,0,0,0,0], [-1,1,1,0,0,0,0], [1,-1,1,0,0,0,0], [1,1,-1,0,0,0,0]]*Z(3)^0;;
gap> ovoid := Union( List(reps, x-> Orbit(psl32, x, Permuted)) );;
gap> ovoid := List(ovoid, x -> VectorSpaceToElement(ps, x));;
gap> planes := AsList( Planes( ps ) );;
#I  Computing collineation group of canonical polar space...
gap> ForAll(planes, p -> Number(ovoid, x -> x in p) = 1);
true
```

7.2   Canonical and standard Polar Spaces

To introduce the classification of polar spaces, we use the classification of the underlying forms in similarity classes. We follow mostly the approach and terminology of [KL90], as we did in the manual of the package Forms.

Consider a vector space $V = V(n+1, q)$ and a sesquilinear form f on V. The pair (V, f) is called a formed space. Consider now two formed spaces (V, f) and (V, f'), where f and f' are two sesquilinear forms on V. A non-singular linear map ϕ from V to itself induces a *similarity* of the formed space (V, f) to the formed space (V, f') if and only if $f(v, w) = \lambda f'(\phi(v), \phi(w))$, for all vectors v, w some non-zero . Up to similarity, there is only one class of non-degenerate Hermitian forms, and one class of non-degenerate symplectic forms on a given vector space V. For symmetric bilinear forms in odd characteristic, the number of similarity classes depends on the dimension of V. In odd dimension, there is only one similarity class, and non-degenerate forms in this class are called parabolic (bilinear) forms. In even dimension, there are two similarity classes, and non-degenerate forms are either elliptic (bilinear) forms or hyperbolic (bilinear) forms.

Consider now a vector space V and a quadratic form q on V. The pair (V, q) is called a formed space. Consider now two formed spaces (V, q) and (V, q'), where q and q' are two quadratic forms on V. A non-degenerate linear map ϕ from V to itself induces a *similarity* of the formed space (V, q) to the formed space (V, q') if and only if $q(v) = \lambda f'(\phi(v)))$, for all vectors v some non-zero . For quadratic forms in even characteristic, the number of similarity classes depends on the dimension of V. In odd dimension, there is only one similarity class, and non-degenerate forms in this class are called parabolic (bilinear) forms. In even dimension, there are two similarity classes, and non-degenerate forms are either elliptic (bilinear) forms or hyperbolic (bilinear) forms.

If ϕ induces a similarity of a formed vector space such that $\lambda = 1$, then the similarity is called an *isometry* of the formed vector space. In almost all cases, each similarity class contains exactly one isometry class. Only the orthogonal sesquilinear forms (in odd characteristic) have two isometry classes. Consequently, if an isometry exists between formed vector spaces, they are called *isometric*. Projectively, a formed vector space becomes a classical polar space embedded in a projective space. Obviously, forms in the same similarity class determine exactly the same classical polar space. Conversely, it is well known that a classical polar space determines a form up to a constant factor, i.e. it determines a similarity class of forms. In FinInG, the word *canonical* is used in the mathematical sense, i.e. a classical polar space is *canonical* if its determining form belongs to a fixed similarity class. A classical polar space is called *standard* if its determining form is the fixed representative of the canonical similarity class. Hence a *standard* classical polar space is always a *canonical* classical polar space, a canonical polar space is determined by a standard form up to a constant factor. In the following table, we summerise the above information on polar spaces, together with the standard forms that are chosen in FinInG. Note that Tr refers to the absolute Trace.

polar space	standard form	characteristic p
hermitian polar space	$X_0^{q+1} + X_1^{q+1} + \ldots + X_n^{q+1}$	odd and even
symplectic space	$X_0 Y_1 - Y_0 X_1 + \ldots + X_{n-1} Y_n - Y_{n-1} X_n$	odd and even
hyperbolic quadric	$X_0 X_1 + \ldots + X_{n-1} X_n$	$p \equiv 3 \bmod 4$ and p even
hyperbolic quadric	$2(X_0 X_1 + \ldots + X_{n-1} X_n)$	$p \equiv 1 \bmod 4$
parabolic quadric	$X_0^2 + X_1 X_2 + \ldots + X_{n-1} X_n$	$p \equiv 1, 3 \bmod 8$ and p even
parabolic quadric	$t(X_0^2 + X_1 X_2 + \ldots + X_{n-1} X_n)$, t a primitive element of $GF(p)$	$p \equiv 5, 7 \bmod 8$
elliptic quadric	$X_0^2 + X_1^2 + X_2 X_3 + \ldots + X_{n-1} X_n$	$p \equiv 3 \bmod 4$
elliptic quadric	$X_0^2 + t X_1^2 + X_2 X_3 + \ldots + X_{n-1} X_n$, t a primitive element of $GF(p)$	odd
elliptic quadric	$X_0^2 + X_0 X_1 + d X_1^2 + X_2 X_3 + \ldots + X_{n-1} X_n$, $Tr(d) = 1$	even

Table: finite classical polar spaces

We refer to Appendix B for information on the operations that construct gram matrices that are used to obtain the above standard forms.

The FinInG provides a wealth of flexibility in constructing polar spaces. The user may choose a particular quadratic or sesquilinear form, but may also chose to construct polars spaces that have one of the above mentioned forms as underlying form. Furthermore, FinInG will detect when necessary if the user-constructed polar space is canonical. This mechanism gives the user complete flexibility while avoiding unnecessary computations when, for example, constructing the collineation group of a user-defined polar space.

The following five operations always return polar spaces induced by one of the above standard forms.

7.2.1 SymplecticSpace

▷ SymplecticSpace(*d, F*)   (operation)

▷ SymplecticSpace(*d, q*)   (operation)

    Returns: a symplectic polar space

    This function returns the symplectic polar space of dimension *d* over *F* for a field *F* or over GF(*q*) for a prime power *q*.

———————————— Example ————————————
```
gap> ps := SymplecticSpace(3,4);
W(3, 4)
gap> Display(ps);
W(3, 4)
Symplectic form
Gram Matrix:
 . 1 . .
 1 . . .
 . . . 1
 . . 1 .
Witt Index: 2
```

7.2.2 HermitianPolarSpace

▷ HermitianPolarSpace(*d, F*)   (operation)

▷ HermitianPolarSpace(*d, q*)   (operation)

    Returns: a Hermitian polar space

    This function returns the Hermitian polar space of dimension *d* over *F* for a field *F* or over GF(*q*) for a prime power *q*.

———————————— Example ————————————
```
gap> ps := HermitianPolarSpace(2,25);
H(2, 5^2)
gap> Display(ps);
H(2, 25)
Hermitian form
Gram Matrix:
 1 . .
 . 1 .
 . . 1
Polynomial: [ [  x_1^6+x_2^6+x_3^6 ] ]
Witt Index: 1
```

7.2.3 ParabolicQuadric

▷ ParabolicQuadric(*d, F*)   (operation)

▷ ParabolicQuadric(*d, q*)   (operation)

    Returns: a parabolic quadric

    d must be an even positive integer. This function returns the parabolic quadric of dimension *d* over *F* for a field *F* or over GF(*q*) for a prime power *q*.

```
                            ─── Example ───
  gap> ps := ParabolicQuadric(2,9);
  Q(2, 9)
  gap> Display(ps);
  Q(2, 9)
  Parabolic bilinear form
  Gram Matrix:
   1 . .
   . . 2
   . 2 .
  Polynomial: [ [  x_1^2+x_2*x_3 ] ]
  Witt Index: 1
  gap> ps := ParabolicQuadric(4,16);
  Q(4, 16)
  gap> Display(ps);
  Q(4, 16)
  Parabolic quadratic form
  Gram Matrix:
   1 . . . .
   . . 1 . .
   . . . . .
   . . . . 1
   . . . . .
  Polynomial: [ [  x_1^2+x_2*x_3+x_4*x_5 ] ]
  Witt Index: 2
  Bilinear form
  Gram Matrix:
   . . . . .
   . . 1 . .
   . 1 . . .
   . . . . 1
   . . . 1 .
```

7.2.4 HyperbolicQuadric

▷ HyperbolicQuadric(d, F)   (operation)
▷ HyperbolicQuadric(d, q)   (operation)

Returns: a hyperbolic quadric

d must be an odd positive integer. This function returns the hyperbolic quadric of dimension d over F for a field F or over GF(q) for a prime power q.

```
                            ─── Example ───
  gap> ps := HyperbolicQuadric(5,3);
  Q+(5, 3)
  gap> Display(ps);
  Q+(5, 3)
  Hyperbolic bilinear form
  Gram Matrix:
   . 2 . . . .
   2 . . . . .
   . . . 2 . .
   . . 2 . . .
```

```
 . . . . . 2
 . . . . 2 .
Polynomial: [ [  x_1*x_2+x_3*x_4+x_5*x_6 ] ]
Witt Index: 3
gap> ps := HyperbolicQuadric(3,4);
Q+(3, 4)
gap> Display(ps);
Q+(3, 4)
Hyperbolic quadratic form
Gram Matrix:
 . 1 . .
 . . . .
 . . . 1
 . . . .
Polynomial: [ [  x_1*x_2+x_3*x_4 ] ]
Witt Index: 2
Bilinear form
Gram Matrix:
 . 1 . .
 1 . . .
 . . . 1
 . . 1 .
```

7.2.5 EllipticQuadric

▷ EllipticQuadric(*d*, *F*)   (operation)
▷ EllipticQuadric(*d*, *q*)   (operation)

   Returns: an elliptic quadric

   d must be an odd positive integer. This function returns the elliptic quadric of dimension *d* over *F* for a field *F* or over GF(*q*) for a prime power *q*.

─────────────────────────── Example ───────────────────────────
```
gap> ps := EllipticQuadric(3,27);
Q-(3, 27)
gap> Display(ps);
Q-(3, 27)
Elliptic bilinear form
Gram Matrix:
 1 . . .
 . 1 . .
 . . . 2
 . . 2 .
Polynomial: [ [  x_1^2+x_2^2+x_3*x_4 ] ]
Witt Index: 1
gap> ps := EllipticQuadric(5,8);
Q-(5, 8)
gap> Display(ps);
Q-(5, 8)
Elliptic quadratic form
Gram Matrix:
 1 1 . . . .
 . 1 . . . .
```

```
  . . . 1 . .
  . . . . . .
  . . . . . 1
  . . . . . .
Polynomial: [ [  x_1^2+x_1*x_2+x_2^2+x_3*x_4+x_5*x_6 ] ]
Witt Index: 2
Bilinear form
Gram Matrix:
  . 1 . . . .
  1 . . . . .
  . . . 1 . .
  . . 1 . . .
  . . . . . 1
  . . . . 1 .
```

The following operations are applicable on any classical polar space in FinInG.

7.2.6 IsCanonicalPolarSpace

▷ IsCanonicalPolarSpace(*ps*)   (attribute)

Returns: true or false

This attribute returns true when a polar space with a particular underlying form is canonical. The execution of this attribute on a general user constructed polar space needs to check the type of *ps*. The obtained extra information is stored automatically as attribute for *ps*, as can be noted by the different printing of *ps* before and after execution.

─────────────────────────── Example ───────────────────────────
```
gap> mat := [[0,1,0,0],[0,0,0,0],[0,0,0,1],[0,0,0,0]]*Z(5)^0;
[ [ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5) ], [ 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0 ], [ 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5) ] ]
gap> form := QuadraticFormByMatrix(mat,GF(5));
< quadratic form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(3,GF(5)): x_1*x_2+x_3*x_4=0 >
gap> IsCanonicalPolarSpace(ps);
true
gap> ps;
Q+(3, 5): x_1*x_2+x_3*x_4=0
gap> mat := [[1,0,0],[0,0,1],[0,1,0]]*Z(3)^0;
[ [ Z(3)^0, 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3), Z(3)^0 ],
  [ 0*Z(3), Z(3)^0, 0*Z(3) ] ]
gap> form := QuadraticFormByMatrix(mat,GF(3));
< quadratic form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(2,GF(3)): x_1^2-x_2*x_3=0 >
gap> IsCanonicalPolarSpace(ps);
false
gap> ps;
Q(2, 3): x_1^2-x_2*x_3=0
```

7.2.7 CanonicalPolarSpace

▷ CanonicalPolarSpace(*form*) (operation)
▷ CanonicalPolarSpace(*P*) (operation)
    Returns: a classical polar space

the canonical polar space isometric to the given polar space *P* or the classical polar space with underlying form *form*.

7.2.8 StandardPolarSpace

▷ StandardPolarSpace(*form*) (operation)
▷ StandardPolarSpace(*P*) (operation)
    Returns: a classical polar space

the polar space induced by a standard form and similar to the given polar space *P* or the classical polar space with underlying form *form*.

7.3 Basic operations for finite classical polar spaces

7.3.1 UnderlyingVectorSpace

▷ UnderlyingVectorSpace(*ps*) (operation)
    Returns: a vector space

The polar space *ps* is the geometry associated with a sesquilinear or quadratic form f. The vector space on which f is acting is returned.

```
_____ Example _____
  gap> ps := EllipticQuadric(5,4);
  Q-(5, 4)
  gap> vs := UnderlyingVectorSpace(ps);
  ( GF(2^2)^6 )
  gap> ps := SymplecticSpace(3,81);
  W(3, 81)
  gap> vs := UnderlyingVectorSpace(ps);
  ( GF(3^4)^4 )
```

7.3.2 AmbientSpace

▷ AmbientSpace(*ps*) (operation)
    Returns: the ambient projective space

When *ps* is a polar space, this operation returns the ambient projective space, i.e. the underlying projective space of the sequilinear or quadratic form that defines *ps*.

```
_____ Example _____
  gap> ps := EllipticQuadric(5,4);
  Q-(5, 4)
  gap> AmbientSpace(ps);
  ProjectiveSpace(5, 4)
  gap> ps := SymplecticSpace(3,81);
  W(3, 81)
  gap> AmbientSpace(ps);
```

```
  ProjectiveSpace(3, 81)
```

7.3.3 ProjectiveDimension

▷ ProjectiveDimension(*ps*)   (operation)
▷ Dimension(*ps*)   (operation)
    Returns: the dimension of the ambient projective space of *ps*

    When *ps* is a polar space, an ambient projective space *P* is uniquely defined and can be asked using AmbientSpace. This operation and its synomym Dimension returns the dimension of *P*.

———————————— Example ————————————
```
  gap> ps := EllipticQuadric(5,4);
  Q-(5, 4)
  gap> ProjectiveDimension(ps);
  5
  gap> ps := SymplecticSpace(3,81);
  W(3, 81)
  gap> ProjectiveDimension(ps);
  3
```

7.3.4 Rank

▷ Rank(*ps*)   (operation)
    Returns: the rank of *ps*

    When *ps* is a polar space, its rank, i.e. the number of different types, equals the Witt index of the defining sesquilinear or quadratic form.

———————————— Example ————————————
```
  gap> ps := EllipticQuadric(5,4);
  Q-(5, 4)
  gap> Rank(ps);
  2
  gap> ps := HyperbolicQuadric(5,4);
  Q+(5, 4)
  gap> Rank(ps);
  3
  gap> ps := SymplecticSpace(7,81);
  W(7, 81)
  gap> Rank(ps);
  4
```

7.3.5 BaseField

▷ BaseField(*ps*)   (operation)
    Returns: the base field of the polar space *ps*

———————————— Example ————————————
```
  gap> ps := HyperbolicQuadric(5,7);
  Q+(5, 7)
```

```
gap> BaseField(ps);
GF(7)
gap> ps := HermitianPolarSpace(2,256);
H(2, 16^2)
gap> BaseField(ps);
GF(2^8)
```

7.3.6   IsHyperbolicQuadric

▷ IsHyperbolicQuadric(*ps*)   (property)

    Returns: true or false

    returns true if and only if *ps* is a hyperbolic quadric.

—————————————— Example ——————————————
```
gap> mat := IdentityMat(6,GF(5));
< mutable compressed matrix 6x6 over GF(5) >
gap> form := BilinearFormByMatrix(mat,GF(5));
< bilinear form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(
5,GF(5)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2=0 >
gap> IsHyperbolicQuadric(ps);
true
gap> mat := IdentityMat(6,GF(7));
< mutable compressed matrix 6x6 over GF(7) >
gap> form := BilinearFormByMatrix(mat,GF(7));
< bilinear form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(
5,GF(7)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2=0 >
gap> IsHyperbolicQuadric(ps);
false
```

7.3.7   IsEllipticQuadric

▷ IsEllipticQuadric(*ps*)   (property)

    Returns: true or false

    returns true if and only if *ps* is an elliptic quadric.

—————————————— Example ——————————————
```
gap> mat := IdentityMat(6,GF(5));
< mutable compressed matrix 6x6 over GF(5) >
gap> form := BilinearFormByMatrix(mat,GF(5));
< bilinear form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(
5,GF(5)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2=0 >
gap> IsEllipticQuadric(ps);
false
gap> mat := IdentityMat(6,GF(7));
< mutable compressed matrix 6x6 over GF(7) >
```

```
gap> form := BilinearFormByMatrix(mat,GF(7));
< bilinear form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(
5,GF(7)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2=0 >
gap> IsEllipticQuadric(ps);
true
```

7.3.8 IsParabolicQuadric

▷ IsParabolicQuadric(*ps*)   (property)
   Returns: true or false
   returns true if and only if *ps* is a parabolic quadric.
―――――――――――――――――――――― Example ――――――――――――――――――――――
```
gap> mat := IdentityMat(5,GF(9));
[ [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ]
gap> form := BilinearFormByMatrix(mat,GF(9));
< bilinear form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(4,GF(3^2)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2=0 >
gap> IsParabolicQuadric(ps);
true
gap> mat := [[1,0,0,0,0],[0,0,1,0,0],[0,0,0,0,0],[0,0,0,0,1],[0,0,0,0,0]]*Z(2)^0;
[ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ] ]
gap> form := QuadraticFormByMatrix(mat,GF(8));
< quadratic form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(4,GF(2^3)): x_1^2+x_2*x_3+x_4*x_5=0 >
gap> IsParabolicQuadric(ps);
true
```

7.4 Subspaces of finite classical polar spaces

The elements of a finite classical polar space *P* are the subspaces of the ambient projective space that are totally isotropic with relation to the sesquilinear or quadratic form that defines *P*. Constructing subspaces of finite classical polar spaces is done as in the projective space case, except that additional checks are implemented in the methods to check that the subspace of the vector space is totally isotropic. The empty subspace, also called the trivial subspace, which has dimension -1, corresponds with the zero dimensional vector space of the underlying vector space of the ambient projective space

of *P*, and is of course totally isotropic. As such, is is considered as a subspace of a finite classical polar space in the mathematical sense, but not as an element of the incidence geometry, and hence do in FinInG not belong to the category `IsSubspaceOfClassicalPolarSpace`.

7.4.1 VectorSpaceToElement

▷ VectorSpaceToElement(*ps, v*) (operation)

Returns: an element of the polar space *geo*

Let *ps* be a polar space, and *v* is either a row vector (for points) or an *m*x*n* matrix (for an $(m-1)$-subspace of a polar space with an $(n-1)$-dimensional ambient projective space. In the case that *v* is a matrix, the rows represent basis vectors for the subspace. An exceptional case is when *v* is a zero-vector, whereby the trivial subspace is returned. It is checked that the subspace defined by *v* is totally isotropic with relation to the form defining *ps*.

```
————————————————————————————— Example —————————————————————————————
  gap> ps := SymplecticSpace(3,4);
  W(3, 4)
  gap> v := [1,0,1,0]*Z(4)^0;
  [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ]
  gap> p := VectorSpaceToElement(ps,v);
  <a point in W(3, 4)>
  gap> mat := [[1,1,0,1],[0,0,1,0]]*Z(4)^0;
  [ [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ]
  gap> line := VectorSpaceToElement(ps,mat);
  Error, <x> does not generate an element of <geom> called from
  <function "unknown">( <arguments> )
   called from read-eval loop at line 12 of *stdin*
  you can 'quit;' to quit to outer loop, or
  you can 'return;' to continue
  brk> quit;
  gap> mat := [[1,1,0,0],[0,0,1,0]]*Z(4)^0;
  [ [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ]
  gap> line := VectorSpaceToElement(ps,mat);
  <a line in W(3, 4)>
  gap> p := VectorSpaceToElement(ps,[0,0,0,0]*Z(4)^0);
  < empty subspace >
```

7.4.2 EmptySubspace

▷ EmptySubspace(*ps*) (operation)

Returns: the trivial subspace in the projective *ps*

The object returned by this operation is contained in every projective subspace of the projective space *ps*, but is not an element of *ps*. Hence, testing incidence results in an error message.

```
————————————————————————————— Example —————————————————————————————
  gap> ps := HermitianPolarSpace(10,49);
  H(10, 7^2)
  gap> e := EmptySubspace(ps);
  < empty subspace >
```

7.4.3 ProjectiveDimension

▷ ProjectiveDimension(*sub*) (operation)

▷ Dimension(*sub*) (operation)

Returns: the projective dimension of a subspace of a polar space. The operation ProjectiveDimension is also applicable on the EmptySubspace

```
──────────────── Example ────────────────
 gap> ps := EllipticQuadric(7,8);
 Q-(7, 8)
 gap> mat := [[0,0,1,0,0,0,0,0],[0,0,0,0,1,0,0,0]]*Z(8)^0;
 [ [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
   [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ] ]
 gap> line := VectorSpaceToElement(ps,mat);
 <a line in Q-(7, 8)>
 gap> ProjectiveDimension(line);
 1
 gap> Dimension(line);
 1
 gap> e := EmptySubspace(ps);
 < empty subspace >
 gap> ProjectiveDimension(e);
 -1
```

7.4.4 ElementsOfIncidenceStructure

▷ ElementsOfIncidenceStructure(*ps*, *j*) (operation)

Returns: the collection of elements of the projective space *ps* of type *j*

For the projective space *ps* of dimension *d* and the type *j*, $1 \leq j \leq d$ this operation returns the collection of $j - 1$ dimensional subspaces.

```
──────────────── Example ────────────────
 gap> ps := HermitianPolarSpace(8,13^2);
 H(8, 13^2)
 gap> planes := ElementsOfIncidenceStructure(ps,3);
 <planes of H(8, 13^2)>
 gap> solids := ElementsOfIncidenceStructure(ps,4);
 <solids of H(8, 13^2)>
 gap> ElementsOfIncidenceStructure(ps,5);
 Error, <geo> has no elements of type <j> called from
 <function "unknown">( <arguments> )
  called from read-eval loop at line 11 of *stdin*
 you can 'quit;' to quit to outer loop, or
 you can 'return;' to continue
 brk> quit;
```

7.4.5 AmbientSpace

▷ AmbientSpace(*el*) (operation)

Returns: returns the ambient space of an element *el* of a polar space

This operation is also applicable on the trivial subspace. For a Lie geometry, the ambient space of an element is defined as the ambient space of the Lie geometry, i.e. a projective space.

```
——————————————— Example ———————————————
gap> ps := HermitianPolarSpace(3,7^2);
H(3, 7^2)
gap> line := VectorSpaceToElement(ps,[[Z(7)^0,0*Z(7),Z(7^2)^34,Z(7^2)^44],
> [0*Z(7),Z(7)^0,Z(7^2)^2,Z(7^2)^4]]);
<a line in H(3, 7^2)>
gap> AmbientSpace(line);
ProjectiveSpace(3, 49)
```

7.4.6   Coordinates

▷ Coordinates(*p*)   (operation)

   Returns: the homogeneous coordinates of the point *p*
```
——————————————— Example ———————————————
gap> ps := ParabolicQuadric(6,5);
Q(6, 5)
gap> p := VectorSpaceToElement(ps,[0,1,0,0,0,0,0]*Z(5)^0);
<a point in Q(6, 5)>
gap> Coordinates(p);
[ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5) ]
```

7.5   Basic operations for polar spaces and subspaces of projective spaces

7.5.1   Incidence and containment

▷ IsIncident(*el1, el2*)   (operation)
▷ *(*el1, el2*)   (operation)
▷ \in(*el1, el2*)   (operation)
   Returns: true or false
   Recall that for projective spaces, incidence is symmetrized containment, where the empty subspace and the whole projective space are excluded as arguments for this operation, since they are not considered as elements of the geometry, but both the empty subspace and the whole projective space are allowed as arguments for \in.
```
——————————————— Example ———————————————
gap> ps := HyperbolicQuadric(7,7);
Q+(7, 7)
gap> p := VectorSpaceToElement(ps,[1,0,1,0,0,0,0,0]*Z(7)^0);
<a point in Q+(7, 7)>
gap> l := VectorSpaceToElement(ps,[[1,0,1,0,0,0,0,0],[0,-1,0,1,0,0,0,0]]*Z(7)^0);
<a line in Q+(7, 7)>
gap> p * l;
true
gap> l * p;
true
gap> IsIncident(p,l);
true
```

```
gap> p in l;
true
gap> l in p;
false
gap> e := EmptySubspace(ps);
< empty subspace >
gap> e * l;
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 1st choice method found for '*' on 2 arguments called from
<function "HANDLE_METHOD_NOT_FOUND">( <arguments> )
 called from read-eval loop at line 17 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> e in l;
true
gap> l in ps;
true
```

7.5.2 Span

▷ Span(*u*, *v*)   (operation)

Returns: an element

u and *v* are elements of a projective or polar space. This function returns the join of the two elements, that is, the span of the two subspaces.

─────────────────── Example ───────────────────
```
gap> ps := HyperbolicQuadric(5,2);
Q+(5, 2)
gap> p := Random(Planes(ps));
<a plane in Q+(5, 2)>
gap> q := Random(Planes(ps));
<a plane in Q+(5, 2)>
gap> s := Span(p,q);
<a proj. 4-space in ProjectiveSpace(5, 2)>
gap> s = Span([p,q]);
true
gap> t := Span(EmptySubspace(ps),p);
<a plane in Q+(5, 2)>
gap> t = p;
true
```

7.5.3 Meet

▷ Meet(*u*, *v*)   (operation)

Returns: an element

u and *v* are elements of a projective or polar space. This function returns the meet of the two elements. If two elements do not meet, then Meet returns EmptySubspace, which in FinInG, is an element with projective dimension -1. (Note that the poset of subspaces of a polar space is a meet-semilattice, but not closed under taking spans).

```
                                ─── Example ───
  gap> ps := HyperbolicQuadric(5,3);;
  gap> pi := Random( Planes(ps) );;
  gap> tau := Random( Planes(ps) );;
  gap> Meet(pi,tau);
  <a point in Q+(5, 3)>
```

Note: the above example will return different answers depending on the two planes chosen at random.

7.5.4 IsCollinear

▷ IsCollinear(*ps*, *u*, *v*)   (operation)

  Returns: Boolean

  u and *v* are points of the ambient space of the polar space *ps*. This function returns True if *u* and *v* are collinear in *ps*. Note that belonging to *ps* is a necessary condition for *u* and *v* to be collinear.

```
                                ─── Example ───
  gap> ps := ParabolicQuadric(4,9);
  Q(4, 9)
  gap> p := VectorSpaceToElement(PG(4,9),[0,1,0,0,0]*Z(9)^0);
  <a point in ProjectiveSpace(4, 9)>
  gap> q := VectorSpaceToElement(PG(4,9),[0,0,1,0,0]*Z(9)^0);
  <a point in ProjectiveSpace(4, 9)>
  gap> r := VectorSpaceToElement(PG(4,9),[0,0,0,1,0]*Z(9)^0);
  <a point in ProjectiveSpace(4, 9)>
  gap> p in ps;
  true
  gap> q in ps;
  true
  gap> r in ps;
  true
  gap> IsCollinear(ps,p,q);
  false
  gap> IsCollinear(ps,p,r);
  true
  gap> IsCollinear(ps,q,r);
  true
  gap> ps := ParabolicQuadric(4,4);
  Q(4, 4)
  gap> p := VectorSpaceToElement(PG(4,4),[1,0,0,0,0]*Z(2)^0);
  <a point in ProjectiveSpace(4, 4)>
  gap> q := VectorSpaceToElement(PG(4,4),[0,1,0,0,0]*Z(2)^0);
  <a point in ProjectiveSpace(4, 4)>
  gap> r := VectorSpaceToElement(PG(4,4),[0,0,0,1,0]*Z(2)^0);
  <a point in ProjectiveSpace(4, 4)>
  gap> p in ps;
  false
  gap> q in ps;
  true
  gap> r in ps;
  true
  gap> IsCollinear(ps,p,q);
  false
```

```
gap> IsCollinear(ps,q,r);
true
```

7.5.5 PolarityOfProjectiveSpace

▷ PolarityOfProjectiveSpace(*ps*)   (operation)

Returns: a polarity of a projective space

ps must be a polar space. This operation returns, when possible a polarity of the ambient projective space of *ps*. It is well known that except for orthogonal polar spaces in even characteristic, a classical polar space is in fact the geometry of absolute points of a polarity of a projective space, and that no polarity can be associated to parabolic quadrics in even characteristic. Polarities of projective spaces are discussed in more detail in Chapter 6.

——————————————— Example ———————————————
```
gap> ps := SymplecticSpace(5,9);
W(5, 9)
gap> phi := PolarityOfProjectiveSpace(ps);
<polarity of PG(5, GF(3^2)) >
gap> ps := EllipticQuadric(3,4);
Q-(3, 4)
gap> phi := PolarityOfProjectiveSpace(ps);
<polarity of PG(3, GF(2^2)) >
gap> ps := ParabolicQuadric(4,4);
Q(4, 4)
gap> phi := PolarityOfProjectiveSpace(ps);
Error, no polarity of the ambient projective space can be associated to <ps> called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 13 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

7.5.6 TypeOfSubspace

▷ TypeOfSubspace(*ps, v*)   (operation)

Returns: a string

This operation is a convenient way to find out the intersection type of a projective subspace with a polar space. The argument *ps* is a nondegenerate polar space, and the argument *v* is a subspace of the ambient projective space. The operation returns a string in accordance with the type of subspace: "degenerate", "symplectic", "hermitian", "elliptic", "hyperbolic" or "parabolic".

——————————————— Example ———————————————
```
gap> h1 := HermitianPolarSpace(2, 3^2);
H(2, 3^2)
gap> h2 := HermitianPolarSpace(3, 3^2);
H(3, 3^2)
gap> pg := AmbientSpace( h2 );
ProjectiveSpace(3, 9)
gap> pi := VectorSpaceToElement( pg, [[1,0,0,0],[0,1,0,0],[0,0,1,0]] * Z(9)^0 );
<a plane in ProjectiveSpace(3, 9)>
```

```
gap> TypeOfSubspace(h2, pi);
"hermitian"
gap> pi := VectorSpaceToElement( pg, [[1,0,0,0],[0,1,0,0],[0,0,1,Z(9)]] * Z(9)^0 );
<a plane in ProjectiveSpace(3, 9)>
gap> TypeOfSubspace(h2, pi);
"degenerate"
```

7.5.7 TangentSpace

▷ TangentSpace(*el*)   (operation)

▷ TangentSpace(*ps, el*)   (operation)

    Returns: A subspace of a projective space

    Let *el* be an element of a classical polar space. The first version returns the tangent space at *el* to this polar space. Let *el* be a subspace of the ambient space of the polar space *ps*. The second version checks whether *el* belongs to *ps* and returns the tangent space at *el* to *ps*. Some obvious properties are demonstrated in the example.

```
                              Example
gap> ps := HermitianPolarSpace(3,4^2);
H(3, 4^2)
gap> p := Random(Points(ps));
<a point in H(3, 4^2)>
gap> plane := TangentSpace(p);
<a plane in ProjectiveSpace(3, 16)>
gap> TypeOfSubspace(ps,plane);
"degenerate"
gap> ps := ParabolicQuadric(6,4);
Q(6, 4)
gap> p := VectorSpaceToElement(PG(6,4),[0,1,0,0,0,0,0]*Z(4)^0);
<a point in ProjectiveSpace(6, 4)>
gap> hyp := TangentSpace(ps,p);
<a proj. 5-space in ProjectiveSpace(6, 4)>
gap> NucleusOfParabolicQuadric(ps) in hyp;
true
gap> ps := EllipticQuadric(5,2);
Q-(5, 2)
gap> line := Random(Lines(ps));
<a line in Q-(5, 2)>
gap> TangentSpace(line);
<a solid in ProjectiveSpace(5, 2)>
gap> ps := HermitianPolarSpace(5,4);
H(5, 2^2)
gap> plane := Random(Planes(ps));
<a plane in H(5, 2^2)>
gap> tan := TangentSpace(plane);
<a plane in ProjectiveSpace(5, 4)>
gap> tan in ps;
true
gap> tan = plane;
true
```

7.5.8 Pole

▷ Pole(*ps, el*) (operation)

   Returns: A subspace of a projective space

   Let *el* be a subspace of the ambient space of the polar space *ps*. This operation returns the pole of *el* with relation to the polar space *ps*.

```
————————————————— Example —————————————————
gap> conic := ParabolicQuadric(2,13);
Q(2, 13)
gap> p := VectorSpaceToElement(PG(2,13),[1,0,0]*Z(13)^0);
<a point in ProjectiveSpace(2, 13)>
gap> pole := Pole(conic,p);
<a line in ProjectiveSpace(2, 13)>
gap> TypeOfSubspace(conic,pole);
"hyperbolic"
gap> tangent_points := Filtered(Points(pole),x->x in conic);
[ <a point in ProjectiveSpace(2, 13)>, <a point in ProjectiveSpace(2, 13)> ]
gap> tangent_lines_on_p := List(tangent_points,x->Span(x,p));
[ <a line in ProjectiveSpace(2, 13)>, <a line in ProjectiveSpace(2, 13)> ]
gap> List(tangent_lines_on_p,x->Number(Points(x),y->y in conic));
[ 1, 1 ]
```

7.6 Shadow of elements

The functionality in this section is comparable to the shadow functionality for elements of projective spaces, which are described in Section 4.3. The generic description of shadows of elements of incidence structures can be found in Section 3.4.

7.6.1 ShadowOfElement

▷ ShadowOfElement(*ps, el, i*) (operation)
▷ ShadowOfElement(*ps, el, str*) (operation)

   Returns: the elements of type *i* incident with *el*. The second variant determines the type *i* from the position of *str* in the list returned by TypesOfElementsOfIncidenceStructurePlural

```
————————————————— Example —————————————————
gap> id := IdentityMat(8,GF(7));
< mutable compressed matrix 8x8 over GF(7) >
gap> form := BilinearFormByMatrix(id,GF(7));
< bilinear form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(
7,GF(7)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2+x_8^2=0 >
gap> Rank(ps);
4
gap> ps;
Q+(7, 7): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2+x_8^2=0
gap> mat := [[1,0,0,0,3,2,0,0],[0,1,0,0,0,0,3,2],[0,0,1,0,5,3,0,0]]*Z(7)^0;
[ [ Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7), Z(7), Z(7)^2, 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), Z(7), Z(7)^2 ],
```

```
    [ 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7), Z(7)^5, Z(7), 0*Z(7), 0*Z(7) ] ]
gap> plane := VectorSpaceToElement(ps,mat);
<a plane in Q+(7, 7): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2+x_8^2=0>
gap> time;
1
gap> shadow := ShadowOfElement(ps,plane,4);
<shadow solids in Q+(7, 7): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2+x_8^2=0>
gap> List(shadow);
[ <a solid in Q+(7, 7): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2+x_8^2=0>,
  <a solid in Q+(7, 7): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2+x_8^2=0> ]
gap> shadow := ShadowOfElement(ps,plane,2);
<shadow lines in Q+(7, 7): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2+x_8^2=0>
```

7.6.2 ElementsIncidentWithElementOfIncidenceStructure

▷ ElementsIncidentWithElementOfIncidenceStructure(*el*, *i*)   (operation)

Returns: the elements of type *i* incident with *el*, in other words, the type *i* shadow of the element *el*

Internally, the function FlagOfIncidenceStructure is used to create a flag from *list*. This function also performs the checking.

——————————— Example ———————————

7.7 Projective Orthogonal/Unitary/Symplectic groups in FinInG

The classical groups (apart from the general lines group), are the matrix groups that *respect*, in a certain way, a sesquilinear or quadratic form. We formally recall the definitions used in FinInG. These definitions are exactly the same as in Forms.

Let (V, f) and (W, g) be two formed vector spaces over the same field F, where both f and g are sesquilinear forms. Suppose that ϕ is a linear map from V to W. The map ϕ is an *isometry* from the formed space (V, f) to the formed space (W, g) if for all v, w in V we have

$$f(v, w) = f'(\phi(v), \phi(w)).$$

The map ϕ is a *similarity* from the formed space (V, f) to the formed space (W, g) if for all v, w in V we have

$$f(v, w) = \lambda f'(\phi(v), \phi(w)).$$

for some non-zero $\lambda \in F$. Finally, the map ϕ. is a *semi-similarity* from the formed space (V, f) to the formed space (W, g) if for all v, w in V we have

$$f(v, w) = \lambda f'(\phi(v), \phi(w))^{\alpha}$$

for some non-zero $\lambda \in F$ and a field automorphism α of F.

Let (V, f) and (W, g) be two formed vector spaces over the same field F, where both f and g are quadratic forms. Suppose that ϕ is a linear map from V to W. The map ϕ is an *isometry* from the formed space (V, f) to the formed space (W, g) if for all v, w in V we have

$$f(v) = f'(\phi(v)).$$

The map ϕ is a *similarity* from the formed space (V, f) to a formed space (W, g) if for all v, w in V we have

$$f(v) = \lambda f'(\phi(v)).$$

for some non-zero $\lambda \in F$. Finally, the map ϕ. is a *semi-similarity* from the formed space (V, f) to the formed space (W, g) if for all v, w in V we have

$$f(v) = \lambda f'(\phi(v))^{\alpha}$$

for some non-zero $\lambda \in F$ and a field automorphism α of F.

Collineations of classical polar spaces are induced by semi-similarities of the underlying formed vector space, and vice versa, analoguously by factoring out scalar matrices. The only exceptions are the two-dimensional unitary groups where the the full semi-similarity group can contain elements of its centre that are not scalars. In FinInG, the subgroups corresponding with similarities and isometries are also implemented, including a *special* variant, corresponding with the matrices having determinant one. We use a consistent terminology, where isometries, similarities, respectively, of the polar space, correspond with isometries, similarities, respectively, of the underlying formed vector space. Special isometries of a polar space are induced by isometries of the formed vector space that have a matrix with determinant one. If P is a polar space with special isometry group, isometry group, similarity group, collineation group, respectively, SI, I, G, Γ, respectively, then clearly $SI \leqslant I \leqslant G \leqslant \Gamma$. Equalities can occur in certain cases, and will, as we will see in the following overview.

(sub)group	symplectic	hyperbolic	elliptic	parabolic	hermitian
special isometry	$PSp(d,q)$	$PSO(1,d,q)$	$PSO(-1,d,q)$	$PSO(0,d,q)$	$PSU(d,q^2)$
isometry	$PSp(d,q)$	$PGO(1,d,q)$	$PGO(-1,d,q)$	$PGO(0,d,q)$	$PGU(d,q^2)$
similarity	$PGSp(d,q)$	$P\Delta O^{+}(d,q)$	$P\Delta O^{-}(d,q)$	$PGO(0,d,q)$	$PGU(d,q^2)$
collineation	$P\Gamma Sp(d,q)$	$P\Gamma O^{+}(d,q)$	$P\Gamma O^{-}(d,q)$	$P\Gamma O(d,q)$	$P\Gamma U(d,q^2)$

Table: projective finite classical groups

7.7.1 SpecialIsometryGroup

▷ SpecialIsometryGroup(*ps*)   (operation)

    Returns: the special isometry group of the polar space *ps*

```
————————————————————— Example —————————————————————
gap> ps := SymplecticSpace(3,4);
W(3, 4)
gap> SpecialIsometryGroup(ps);
PSp(4,4)
gap> ps := HyperbolicQuadric(5,8);
Q+(5, 8)
gap> SpecialIsometryGroup(ps);
PSO(1,6,8)
gap> ps := EllipticQuadric(3,27);
Q-(3, 27)
gap> SpecialIsometryGroup(ps);
PSO(-1,4,27)
gap> ps := ParabolicQuadric(4,8);
Q(4, 8)
gap> SpecialIsometryGroup(ps);
PSO(0,5,8)
```

```
gap> ps := HermitianPolarSpace(4,9);
H(4, 3^2)
gap> SpecialIsometryGroup(ps);
PSU(5,3^2)
```

7.7.2 IsometryGroup

▷ IsometryGroup(ps) (operation)

   Returns: the isometry group of the polar space *ps*

─────────────── Example ───────────────
```
gap> ps := SymplecticSpace(3,4);
W(3, 4)
gap> IsometryGroup(ps);
PSp(4,4)
gap> ps := HyperbolicQuadric(5,8);
Q+(5, 8)
gap> IsometryGroup(ps);
PGO(1,6,8)
gap> ps := EllipticQuadric(3,27);
Q-(3, 27)
gap> IsometryGroup(ps);
PGO(-1,4,27)
gap> ps := ParabolicQuadric(4,8);
Q(4, 8)
gap> IsometryGroup(ps);
PGO(0,5,8)
gap> ps := HermitianPolarSpace(4,9);
H(4, 3^2)
gap> IsometryGroup(ps);
PGU(5,3^2)
```

7.7.3 SimilarityGroup

▷ SimilarityGroup(ps) (operation)

   Returns: the similarity group of the polar space *ps*

─────────────── Example ───────────────
```
gap> ps := SymplecticSpace(3,4);
W(3, 4)
gap> SimilarityGroup(ps);
PGSp(4,4)
gap> ps := HyperbolicQuadric(5,8);
Q+(5, 8)
gap> SimilarityGroup(ps);
PDeltaO+(6,8)
gap> ps := EllipticQuadric(3,27);
Q-(3, 27)
gap> SimilarityGroup(ps);
PDeltaO-(4,27)
gap> ps := ParabolicQuadric(4,8);
```

```
 Q(4, 8)
 gap> SimilarityGroup(ps);
 PGO(0,5,8)
 gap> ps := HermitianPolarSpace(4,9);
 H(4, 3^2)
 gap> SimilarityGroup(ps);
 PGU(5,3^2)
```

7.7.4 CollineationGroup

▷ CollineationGroup(*ps*)   (operation)

   Returns: the collineation group of the polar space *ps*

   In most cases, the full projective semisimilarity group is returned. For two-dimensional unitary groups, the centre may contain elements that are not scalars. In this case, we return a central extension of the projective semisimilarity group. If the basefield of *ps* is $GF(q^2)$, q prime, the similarity group is returned.

```
                                — Example —
 gap> ps := SymplecticSpace(3,4);
 W(3, 4)
 gap> CollineationGroup(ps);
 PGammaSp(4,4)
 gap> ps := HyperbolicQuadric(5,8);
 Q+(5, 8)
 gap> CollineationGroup(ps);
 PGammaO+(6,8)
 gap> ps := EllipticQuadric(3,27);
 Q-(3, 27)
 gap> CollineationGroup(ps);
 PGammaO-(4,27)
 gap> ps := ParabolicQuadric(4,8);
 Q(4, 8)
 gap> CollineationGroup(ps);
 PGammaO(5,8)
 gap> ps := HermitianPolarSpace(4,9);
 H(4, 3^2)
 gap> CollineationGroup(ps);
 PGammaU(5,3^2)
```

7.8 Enumerating subspaces of polar spaces

7.8.1 Enumerators for polar spaces

An enumerator for a collection of subspaces of a given type of a polar space is provided in FinInG. If *C* is such a collection, then List(C) will use the enumerator to compute a list with all the elements of *C*.

7.8.2 Enumerator

▷ Enumerator(*C*) (operation)

▷ List(*C*) (operation)

Returns: an enumerator for the collection *C* and a list with all elements of *C*

The argument *C* is a collection of subspaces of a polar space.

```
──────────────────── Example ────────────────────
gap> Enumerator(Points(ParabolicQuadric(6,3)));
EnumeratorOfSubspacesOfClassicalPolarSpace( <points of Q(6, 3)> )
gap> Enumerator(Lines(HermitianPolarSpace(4,4)));
EnumeratorOfSubspacesOfClassicalPolarSpace( <lines of H(4, 2^2)> )
gap> planes := List(Planes(HermitianPolarSpace(5,4)));;
gap> time;
11515
gap> Length(planes);
891
```

7.8.3 Iterators for polar spaces

For all polar spaces an iterator is constructed using IteratorList(enum), where *enum* is an appropriate enumerator.

7.8.4 Iterator

▷ Iterator(*elements*) (operation)

Returns: an iterator

C is a collection of subspaces of a polar space.

```
──────────────────── Example ────────────────────
gap> iter := Iterator(Lines(ParabolicQuadric(4,2)));
<iterator>
gap> NextIterator(iter);
<a line in Q(4, 2)>
gap> NextIterator(iter);
<a line in Q(4, 2)>
gap> NextIterator(iter);
<a line in Q(4, 2)>
gap> NextIterator(iter);
<a line in Q(4, 2)>
gap> NextIterator(iter);
<a line in Q(4, 2)>
```

7.8.5 AsList

▷ AsList(*subspaces*) (operation)

Returns: an Orb object or list

```
──────────────────── Example ────────────────────
gap> ps := HyperbolicQuadric(5,3);
Q+(5, 3)
```

```
gap> lines := AsList(Lines(ps));
<closed orbit, 520 points>
```

Chapter 8

Orbits, stabilisers and actions

8.1 Orbits

GAP provides generic functionality to compute orbits. These functions are, generally spoken, applicable to the groups implemented in FinInG, combined with the appropriate action functions. However, the generic functions applied in such situations are rather time comsuming. FinInG therefore provides alternative functions to compute orbits.

8.1.1 FiningOrbit

▷ FiningOrbit(*g, obj, act*) (operation)

Returns: The orbit of the object *obj* under the action *act* of the group *g*.

The argument *obj* is either a subspace of a projective space, then combined with the action function OnProjSubspaces, or a set of elements of a projective space, then combined with the action function OnSetsProjSubspaces. The group *g* is a subgroup of a collineation group of a projective space. In both cases the action function computes the action of *el* under the group element *g*.

```
——————————————— Example ———————————————
gap> ps := ParabolicQuadric(6,3);
Q(6, 3)
gap> g := CollineationGroup(ps);
PGammaO(7,3)
gap> pg := PG(6,3);
ProjectiveSpace(6, 3)
gap> s := First(Solids(pg),t -> TypeOfSubspace(ps,t) = "elliptic" );
<a solid in ProjectiveSpace(6, 3)>
gap> orbit := FiningOrbit(g,s,OnProjSubspaces);
<closed orbit, 265356 points>
gap> time;
33555
```

The second example shows the possible use of FiningOrbit in combination with the action function OnSetsProjSubspaces. Please note that this variant is probably not the most efficient way to compute all elliptic quadrics contained in the parabolic quadric *ps*. Experiments show that for $q = 5$ the second variant takes an unreasonable amount of time. Also note that the second argument *el* must be a set (and therefore it might be necessary to apply Set on a collection of elements).

```
gap> ps := ParabolicQuadric(4,3);
Q(4, 3)
gap> g := CollineationGroup(ps);
PGammaO(5,3)
gap> pg := PG(4,3);
ProjectiveSpace(4, 3)
gap> s := First(Solids(pg),t -> TypeOfSubspace(ps,t) = "elliptic" );
<a solid in ProjectiveSpace(4, 3)>
gap> orbit1 := FiningOrbit(g,s,OnProjSubspaces);
<closed orbit, 36 points>
gap> time;
9
gap> spts := Filtered(Points(s),s->s in ps);
[ <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)>, <a point in ProjectiveSpace(4, 3)> ]
gap> orbit2 := FiningOrbit(g,Set(spts),OnSetsProjSubspaces);
<closed orbit, 36 points>
gap> time;
18
```

8.1.2 FiningOrbits

▷ FiningOrbits(g, set, act) (operation)
▷ FiningOrbits(g, coll) (operation)

Returns: The orbits of the group g on set under the action of act.

The set is a set of elements of a projective space, the group g is a subgroup of the collineation group of a projective space, and act is the function OnProjSubspaces. If coll is a collection of elements of a projective space (i.e. not a list or set, but and object representing the collection of elements of a given type, such as Lines(PG(3,4))), then the second versions returns the orbits of g on the elements of coll under the action OnProjSubspaces.

```
gap> ps := HermitianPolarSpace(3,9);
H(3, 3^2)
gap> g := CollineationGroup(ps);
PGammaU(4,3^2)
gap> FiningOrbits(g,Lines(PG(3,9)));
75%..98%..100%..[ <closed orbit, 5670 points>, <closed orbit, 1680 points>,
  <closed orbit, 112 points> ]
gap> FiningOrbits(g,Planes(PG(3,9)));
65%..100%..[ <closed orbit, 540 points>, <closed orbit, 280 points> ]
gap> ps := ParabolicQuadric(2,5);
Q(2, 5)
gap> g := CollineationGroup(ps);
PGammaO(3,5)
gap> pts := Filtered(Points(PG(2,5)),x->not x in ps);;
gap> Length(pts);
```

```
25
gap> FiningOrbits(g,Points(PG(2,5)));
48%..67%..100%..[ <closed orbit, 15 points>, <closed orbit, 6 points>,
  <closed orbit, 10 points> ]
gap> FiningOrbits(g,pts,OnProjSubspaces);
60%..100%..[ <closed orbit, 15 points>, <closed orbit, 10 points> ]
```

8.2   Stabilisers

The GAP function `Stabilizer` is a generic function to compute stabilisers of one object (or sets or tuples etc. of objects) under a group, using a specified action function. This generic function can be used together with the in FinInG implemented groups and elements of geometries. However, computing time can be very long, already in small geometries.

```
———————————————————————— Example ————————————————————————
gap> ps := PG(3,8);
ProjectiveSpace(3, 8)
gap> g := CollineationGroup(ps);
The FinInG collineation group PGammaL(4,8)
gap> p := Random(Points(ps));
<a point in ProjectiveSpace(3, 8)>
gap> Stabilizer(g,p,OnProjSubspaces);
<projective collineation group of size 177223237632 with 2 generators>
gap> time;
10026
gap> line := Random(Lines(ps));
<a line in ProjectiveSpace(3, 8)>
gap> Stabilizer(g,line,OnProjSubspaces);
<projective collineation group of size 21849440256 with 2 generators>
gap> time;
78126
```

The packages GenSS and orb required by FinInG provide efficient operations to compute stabilisers, and FinInG provides functionality to use these operations for the particular groups and (elements) of geometries.

8.2.1   FiningStabiliser

▷ FiningStabiliser(*g*, *el*)   (operation)
    Returns: The subgroup of *g* stabilising the element *el*
    The argument *g* is a group of collineations acting on the element *el*, being a subspace of a projective space (and hence, all elements of a Lie geometry are allowed as second argument). This operation relies on the GenSS operation `Stab`.

```
———————————————————————— Example ————————————————————————
gap> ps := PG(5,4);
ProjectiveSpace(5, 4)
gap> g := SpecialHomographyGroup(ps);
The FinInG PSL group PSL(6,4)
gap> p := Random(Points(ps));
```

```
    <a point in ProjectiveSpace(5, 4)>
gap> FiningStabiliser(g,p);
<projective collineation group of size 264696069567283200 with 2 generators>
gap> line := Random(Lines(ps));
<a line in ProjectiveSpace(5, 4)>
gap> FiningStabiliser(g,line);
<projective collineation group of size 3881174040576000 with 3 generators>
gap> plane := Random(Planes(ps));
<a plane in ProjectiveSpace(5, 4)>
gap> FiningStabiliser(g,plane);
#I  Have 106048 points.
#I  Have 158748 points.
<projective collineation group of size 958878292377600 with 2 generators>
gap> ps := HyperbolicQuadric(5,5);
Q+(5, 5)
gap> g := IsometryGroup(ps);
PGO(1,6,5)
gap> p := Random(Points(ps));
<a point in Q+(5, 5)>
gap> FiningStabiliser(g,p);
<projective collineation group of size 36000000 with 3 generators>
gap> line := Random(Lines(ps));
<a line in Q+(5, 5)>
gap> FiningStabiliser(g,line);
<projective collineation group of size 6000000 with 3 generators>
gap> plane := Random(Planes(ps));
<a plane in Q+(5, 5)>
gap> FiningStabiliser(g,plane);
<projective collineation group of size 93000000 with 2 generators>
gap> h := SplitCayleyHexagon(3);
H(3)
gap> g := CollineationGroup(h);
#I  for Split Cayley Hexagon
#I  Computing nice monomorphism...
#I  Found permutation domain...
G_2(3)
gap> p := Random(Points(h));
<a point in H(3)>
gap> FiningStabiliser(g,p);
<projective collineation group of size 11664 with 2 generators>
gap> line := Random(Lines(h));
<a line in H(3)>
gap> FiningStabiliser(g,line);
<projective collineation group of size 11664 with 2 generators>
```

8.2.2 FiningStabiliserOrb

▷ FiningStabiliserOrb(g, el)   (operation)

Returns: The subgroup of g stabilising the element el

The argument g is a group of collineations acting on the element el, being a subspace of a

projective space (and hence, all elements of a Lie geometry are allowed as second argument). This operation relies on some particular orb functionality.

```
_____ Example _____
 gap> ps := PG(5,4);
 ProjectiveSpace(5, 4)
 gap> g := SpecialHomographyGroup(ps);
 The FinInG PSL group PSL(6,4)
 gap> p := Random(Points(ps));
 <a point in ProjectiveSpace(5, 4)>
 gap> FiningStabiliserOrb(g,p);
 <projective collineation group with 15 generators>
 gap> line := Random(Lines(ps));
 <a line in ProjectiveSpace(5, 4)>
 gap> FiningStabiliserOrb(g,line);
 <projective collineation group with 15 generators>
 gap> plane := Random(Planes(ps));
 <a plane in ProjectiveSpace(5, 4)>
 gap> FiningStabiliserOrb(g,plane);
 <projective collineation group with 15 generators>
 gap> ps := HyperbolicQuadric(5,5);
 Q+(5, 5)
 gap> g := IsometryGroup(ps);
 PGO(1,6,5)
 gap> p := Random(Points(ps));
 <a point in Q+(5, 5)>
 gap> FiningStabiliserOrb(g,p);
 <projective collineation group with 15 generators>
 gap> line := Random(Lines(ps));
 <a line in Q+(5, 5)>
 gap> FiningStabiliserOrb(g,line);
 <projective collineation group with 15 generators>
 gap> plane := Random(Planes(ps));
 <a plane in Q+(5, 5)>
 gap> FiningStabiliserOrb(g,plane);
 <projective collineation group with 15 generators>
 gap> h := SplitCayleyHexagon(3);
 H(3)
 gap> g := CollineationGroup(h);
 #I  for Split Cayley Hexagon
 #I  Computing nice monomorphism...
 #I  Found permutation domain...
 G_2(3)
 gap> p := Random(Points(h));
 <a point in H(3)>
 gap> FiningStabiliserOrb(g,p);
 <projective collineation group with 15 generators>
 gap> line := Random(Lines(h));
 <a line in H(3)>
 gap> FiningStabiliserOrb(g,line);
 <projective collineation group with 15 generators>
```

A small example shows the difference in computing time. Clearly the `FiningStabiliserOrb` is

the fastest way to compute stabilizers of one element.

```
————————————————— Example ——————————————————
gap> ps := PG(3,8);
ProjectiveSpace(3, 8)
gap> g := CollineationGroup(ps);
The FinInG collineation group PGammaL(4,8)
gap> p := Random(Points(ps));
<a point in ProjectiveSpace(3, 8)>
gap> g1 := Stabilizer(g,p);
<projective collineation group of size 177223237632 with 2 generators>
gap> time;
9576
gap> g2 := FiningStabiliser(g,p);
<projective collineation group of size 177223237632 with 2 generators>
gap> time;
244
gap> g3 := FiningStabiliserOrb(g,p);
<projective collineation group with 15 generators>
gap> time;
46
gap> g1=g2;
true
gap> g2=g3;
true
```

8.2.3 FiningSetwiseStabiliser

▷ FiningSetwiseStabiliser(*g*, *els*)   (operation)

Returns: The subgroup of *g* stabilising the set *els*

The argument *g* is a group of collineations acting on the element *el*, being a subspace of a projective space (and hence, all elements of a Lie geometry are allowed as second argument). The argument *els* is a set of elements of the same type of the same Lie geometry, the elements are all in the category IsSubspaceOfProjectiveSpace. The underlying action function is assumed to be OnProjSubspaces

```
————————————————— Example ——————————————————
gap> ps := HyperbolicQuadric(5,5);
Q+(5, 5)
gap> g := IsometryGroup(ps);
PGO(1,6,5)
gap> plane1 := Random(Planes(ps));
<a plane in Q+(5, 5)>
gap> plane2 := Random(Planes(ps));
<a plane in Q+(5, 5)>
gap> FiningSetwiseStabiliser(g,Set([plane1,plane2]));
#I  Computing adjusted stabilizer chain...
<projective collineation group with 4 generators>
```

Computing the setwise stabiliser under a group is also possible using Stabilizer. But, not suprisingly, the computing time can be very long.

```
                              ─── Example ───
  gap> ps := PG(3,4);
  ProjectiveSpace(3, 4)
  gap> p := Random(Points(ps));
  <a point in ProjectiveSpace(3, 4)>
  gap> q := Random(Points(ps));
  <a point in ProjectiveSpace(3, 4)>
  gap> g := CollineationGroup(ps);
  The FinInG collineation group PGammaL(4,4)
  gap> Stabilizer(g,Set([p,q]),OnSets);
  <projective collineation group of size 552960 with 5 generators>
  gap> time;
  10440
```

The package **GenSS** provides an efficient operations to compute setwise stabilisers. This is why **FinInG** provides functionality, such as `FiningSetwiseStabiliser`, to use these **GenSS** operations for the particular groups and (elements) of geometries. A small example shows the difference in computing time.

```
                              ─── Example ───
  gap> ps := ParabolicQuadric(4,4);
  Q(4, 4)
  gap> g := CollineationGroup(ps);
  PGammaO(5,4)
  gap> l1 := Random(Lines(ps));
  <a line in Q(4, 4)>
  gap> l2 := Random(Lines(ps));
  <a line in Q(4, 4)>
  gap> g1 := Stabilizer(g,Set([l1,l2]),OnSets);
  <projective collineation group of size 2304 with 6 generators>
  gap> time;
  2633
  gap> g2 := FiningSetwiseStabiliser(g,Set([l1,l2]));
  #I  Computing adjusted stabilizer chain...
  <projective collineation group with 5 generators>
  gap> time;
  70
  gap> g1=g2;
  true
```

8.3 Actions and nice monomorphisms revisited

GAP provides generic functions to compute action homomorphisms and their images for arbitrary groups. These functions are applicable on the projective groups implemented in **FinInG**.

8.3.1 Action functions

▷ OnProjSubspaces(*el, g*)   (function)
▷ OnProjSubspacesExtended(*el, g*)   (function)

▷ OnSetsProjSubspaces(*set, g*)   (function)
    Returns: a element of a Lie geometry
    Let `el` be an element of any Lie geometry, and `g` an element of a projective group acting on the elements of the ambient Lie geometry of `el`. Then then `OnProjSubspaces` will return simply the image of `el` under `g`. When `g` is an element of the correlationcollineation group of a projective space, `OnProjSubspacesExtended` returns the image of `el` under `g`. Finally, when `set` is a set of elements of a Lie geometry, `OnSetsProjSubspaces` returns the set of images under `g`. `OnProjSubspaces` is also explained in 5.8.1, `OnProjSubspacesExtended` is also explained in 5.8.3.

8.3.2 Generic GAP functions

▷ ActionHomomorphism(*g, S, act*)   (operation)
▷ Action(*g, S, act*)   (operation)

    `g` is a projective group, `S` is a set or a collection of elements, `act` is an action function. `Action` simply returns `Image(hom)`, if `hom` is the result of `ActionHomomorphism`. The examples are self-explanatory.

```
———————————————————— Example ————————————————————
  gap> pg := PG(2,3);
  ProjectiveSpace(2, 3)
  gap> conic := Set(Points(ParabolicQuadric(2,3)));;
  gap> coll := CollineationGroup(pg);
  The FinInG collineation group PGL(3,3)
  gap> orb := Orbit(coll,conic,OnSetsProjSubspaces);;
  gap> Length(orb);
  234
  gap> hom := ActionHomomorphism(coll,orb,OnSetsProjSubspaces);
  <action homomorphism>
  gap> perm := Image(hom);
  <permutation group with 2 generators>
  gap> Order(perm);
  5616
  gap> NrMovedPoints(perm);
  234
  gap> ps := SymplecticSpace(5,2);
  W(5, 2)
  gap> coll := CollineationGroup(ps);
  PGammaSp(6,2)
  gap> perm := Action(coll,Lines(ps),OnProjSubspaces);
  <permutation group with 4 generators>
  gap> NrMovedPoints(perm);
  315
```

    A nice monomorphism of a group G is roughly just a permutation representation of G on a suitable action domain. An easy example is the permutation action of the full collineation group of a projective space on its points.

8.3.3   NiceMonomorphism

▷ NiceMonomorphism(*group*) (attribute)

   Returns: A group homomorphism
   This is a generic GAP function, and returns a homorphism to a "better" representation.

8.3.4   NiceObject

▷ NiceObject(*group*) (attribute)

   Returns: A permutation group
   group is a projective group. The object this operation returns is actually equivalent with
Image(NiceMonomorphism(group)).

8.3.5   Different behaviour for different collineation groups

For the different Lie geometries implemented in FinInG, nicemonomorphisms are (necessarily) treated in a different way. As the aim of a nicemonomorphism of group *G* is to provide a permutation representation, such that efficient algorithms for permutation groups become available for certain operations applicable on *G*, clearly the efficiency will be increased if the degree of the permutation representation is as small as possible.

   For the collineation group, projectivity group and special projectivity group of a projective space, it is clear that the smallest degree permutation representation is the action of the group on the projective points. In principle, one could also consider the action on the hyerplanes. For the collineation group, similarity group and isometry group of a classical polar space, in most cases, the smallest degree permutation representation is the action on the points. A notorious exception to this is the hermitian polar space in three dimensions, of which the number of lines is smaller than the number of points, and hence of which the smallest degree permutation representation is the action of the group on the lines. When constructing a collineation group (or (special) projectiviity group) of a projective space, the nicemonomorphism is not computed. It is only computed when needed. The reason is that from the underlying field and dimension, the underlying projective space can be determined at any time, and hence the smallest degree representation can be computed. For the collineation groups (and similarity and isometry groups) of classical polar spaces, this behaviour is different. Indeed, given a group of collineations, from the underlying field and dimension, the original polar space can not be determined. Of course one could consider the action on the points of the underlying projective space, but typically the number of points of a classical polar space is much smaller than the number points of the underlying projective space. This explains why, currently, a nice monomorphism is computed at the moment a collineation group of a classical polar space is computed. As a consequence, just asking the collineation group of a polar space can be time consuming.

```
──────────── Example ────────────
gap> g := CollineationGroup(PG(5,9));
The FinInG collineation group PGammaL(6,9)
gap> time;
28
gap> HasNiceMonomorphism(g);
false
gap> h := CollineationGroup(EllipticQuadric(5,9));
PGammaO-(6,9)
gap> time;
1584
```

```
gap> HasNiceMonomorphism(h);
true
```

8.3.6 SetParent

▷ SetParent(*group*)   (operation)

Assume that G is a group of collineations. As mentioned already, from the underlying field and dimension, only the underlying projective space can be determined. An operation like Order requires a nice monomorphism, so for an arbitrary group G, the action on the points of the underlying projective space will be computed, which can be time consuming for large projective spaces. However, if it is known that G is a subgroup of another collineation group H, this group H can be set as a parent group for G. If a nice monomorphism is available for H, it will become available for G. In the example we construct the collineation group of the hermtian polar space $H(3,81)$. As explained, a nice monomorphism is computed upon construction. Then construct a group generated by two random elements of this collineation group of $H(3,81)$, and compute its order. Without further information, it will be assumed by the system that this new group is a subgroup of the collineation group of $PG(3,81)$, and a nice monomorphism will be computed through this group. In the second part we set the parent group as the collineation group of $H(3,81)$, and compute the order again. Compare the different timings.

```
———————————————————————— Example ————————————————————————
gap> ps := HermitianPolarSpace(3,81);
H(3, 9^2)
gap> group := CollineationGroup(ps);
PGammaU(4,9^2)
gap> time;
2219
gap> g := Random(group);
< a collineation: <cmat 4x4 over GF(3,4)>, F^27>
gap> h := Random(group);
< a collineation: <cmat 4x4 over GF(3,4)>, F^3>
gap> group2 := Group([g,h]);
<projective collineation group with 2 generators>
gap> HasNiceMonomorphism(group2);
false
gap> Order(group2);
407194345728000
gap> time;
371559
gap> HasNiceMonomorphism(group2);
true
gap> NrMovedPoints(NiceObject(group2));
538084
gap> Size(Points(PG(3,81)));
538084
gap> group2 := Group([g,h]);
<projective collineation group with 2 generators>
gap> SetParent(group2,group);
gap> HasNiceMonomorphism(group2);
```

```
true
gap> HasNiceObject(group2);
false
gap> Order(group2);
407194345728000
gap> time;
888
gap> HasNiceObject(group2);
true
gap> NrMovedPoints(NiceObject(group2));
7300
gap> Size(Lines(ps));
7300
```

Chapter 9

Affine Spaces

In this chapter we show how one can work with finite affine spaces in FinInG.

9.1 Affine spaces and basic operations

An *affine space* is a point-line incidence geometry, satisfying few well known axioms. An axiomatic treatment can e.g. be found in [VY65a] and [VY65b]. As is the case with projective spaces, affine spaces are axiomatically point-line geometries, but may contain higher dimensional affine subspaces too. An affine space can also be described as the "geometry you get" when you remove a hyperplane from a projective space. Conversely, each affine space can be extended to a projective space in a unique way (by "adding its hyperplane at infinity"). In FinInG, we deal with *finite Desarguesian affine spaces*, i.e. an affine space, such that its projective completion is Desarguesian. Other concepts can be easily defined using this projective completion. E.g. lines of the projective space which are concurrent in a point of the hyperplane at infinity, become now *parallel* in the affine space. In order to implement (Desarguesian) affine spaces in FinInG, we have to represent the elements of the affine space (the affine subspaces), in a standard way. By definition, the points (i.e. the elements of type 1) of the n-dimensional affine space $AG(n,q)$ are the vectors of the underlying n-dimensional vector space over the finite field $GF(q)$. The i-dimensional subspaces of $AG(n,q)$ (i.e. the elements of type $i-1$) are defined as the cosets of the i-dimensional subspaces of the underlying vector space. Hence, the common representation of such a subspace is

$$v + S,$$

where v is a vector and S is a subspace of a vector space. Equivalently one can also think of a subspace of an affine space as consisting of: (i) an affine point, representing the coset, and and (ii) a "direction", which is an element of an $n-1$-dimensional projective space, representing the hyperplane at infinity. In FinInG, we represent an i-dimensional subspace, $1 \leq i \leq n-1$ as

$$[v, mat]$$

where v is a row vector and `mat` is a matrix (representing a basis of the projective element representing the direction at infinity). For affine points, we simply use vectors.

9.1.1 IsAffineSpace

▷ IsAffineSpace   (Category)

This category is a subcategory of `IsIncidenceGeometry`, and contains all finite Desarguesian affine spaces.

9.1.2 AffineSpace

▷ AffineSpace(*d*, *F*)   (operation)
▷ AffineSpace(*d*, *q*)   (operation)
▷ AG(*d*, *F*)   (operation)
▷ AG(*d*, *q*)   (operation)

    Returns: an affine space

    d must be a positive integer. In the first form, *F* is a field and the function returns the affine space of dimension *d* over *F*. In the second form, *q* is a prime power specifying the size of the field. The user may also use an alias, namely, the common abbreviation `AG(d, q)`.

```
_____ Example _____
gap> AffineSpace(3,GF(4));
AG(3, 4)
gap> AffineSpace(3,4);
AG(3, 4)
gap> AG(3,GF(4));
AG(3, 4)
gap> AG(3,4);
AG(3, 4)
```

9.1.3 Dimension

▷ Dimension(*as*)   (attribute)
▷ Rank(*as*)   (attribute)

    Returns: the dimension of the affine space *as* (which is equal to its rank)

```
_____ Example _____
gap> Dimension(AG(5,7));
5
gap> Rank(AG(5,7));
5
```

9.1.4 BaseField

▷ BaseField(*as*)   (operation)

    Returns: returns the base field for the affine space *as*

```
_____ Example _____
gap> BaseField(AG(6,49));
GF(7^2)
```

9.1.5 UnderlyingVectorSpace

▷ UnderlyingVectorSpace(*as*)   (operation)

    Returns: a vector space

The underlying vectorspace of $AG(n,q)$ is simply $V(n,q)$.

```
─────────────────── Example ───────────────────
gap> UnderlyingVectorSpace(AG(4,5));
( GF(5)^4 )
```

9.1.6 AmbientSpace

▷ AmbientSpace(as)   (attribute)

   Returns: an affine space

   The ambient space of an affine space `as` is the affine space itself. Hence, simply `as` will be returned.

```
─────────────────── Example ───────────────────
gap> AmbientSpace(AG(4,7));
AG(4, 7)
```

9.2 Subspaces of affine spaces

9.2.1 AffineSubspace

▷ AffineSubspace(geo, v)   (operation)
▷ AffineSubspace(geo, v, M)   (operation)

   Returns: a subspace of an affine space

   `geo` is an affine space, `v` is a row vector, and `M` is a matrix. There are two representations necessary for affine subspaces in FinInG: (i) points represented as vectors and (ii) subspaces of dimension at least 1 represented as a coset of a vector subspace:

$$v + S.$$

For the former, the underlying object is just a vector, whereas the second is a pair $[v, M]$ where v is a vector and M is a matrix representing the basis of S. Now there is a canonical representative for the coset $v + S$, and the matrix M is in semi-echelon form, therefore we can easily compare two affine subspaces. If no matrix is given in the arguments, then it is assumed that the user is constructing an affine point.

```
─────────────────── Example ───────────────────
gap> ag := AffineSpace(3, 3);
AG(3, 3)
gap> x := [[1,1,0]]*Z(3)^0;
[ [ Z(3)^0, Z(3)^0, 0*Z(3) ] ]
gap> v := [0,-1,1] * Z(3)^0;
[ 0*Z(3), Z(3), Z(3)^0 ]
gap> line := AffineSubspace(ag, v, x);
<a line in AG(3, 3)>
```

9.2.2 ElementsOfIncidenceStructure

▷ ElementsOfIncidenceStructure(as, j)   (operation)

Returns: the collection of elements of the affine space as of type j

For the affine space as of dimension d and the type j, $1 \le j \le d$ this operation returns the collection of $j - 1$ dimensional subspaces. An error message is produced when the projective space ps has no elements of a required type.

```
———————————————————— Example ————————————————————
gap> ag := AffineSpace(9, 64);
AG(9, 64)
gap> ElementsOfIncidenceStructure(ag,1);
<points of AG(9, 64)>
gap> ElementsOfIncidenceStructure(ag,2);
<lines of AG(9, 64)>
gap> ElementsOfIncidenceStructure(ag,3);
<planes of AG(9, 64)>
gap> ElementsOfIncidenceStructure(ag,4);
<solids of AG(9, 64)>
gap> ElementsOfIncidenceStructure(ag,6);
<affine. subspaces of dim. 5 of AG(9, 64)>
gap> ElementsOfIncidenceStructure(ag,9);
<affine. subspaces of dim. 8 of AG(9, 64)>
gap> ElementsOfIncidenceStructure(ag,10);
Error, <as> has no elements of type <j> called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 15 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

9.2.3 Short names for ElementsOfIncidenceStructure

▷ Points(ps)   (operation)
▷ Lines(ps)   (operation)
▷ Planes(ps)   (operation)
▷ Solids(ps)   (operation)
▷ Hyperplanes(ps)   (operation)

Returns: The elements of ps of respective type 1, 2, 3, 4, and the hyperplanes

An error message is produced when the projective space ps has no elements of a required type.

```
———————————————————— Example ————————————————————
gap> as := AG(5,4);
AG(5, 4)
gap> Points(as);
<points of AG(5, 4)>
gap> Lines(as);
<lines of AG(5, 4)>
gap> Planes(as);
<planes of AG(5, 4)>
gap> Solids(as);
<solids of AG(5, 4)>
```

```
gap> Hyperplanes(as);
<affine. subspaces of dim. 4 of AG(5, 4)>
gap> as := AG(2,8);
AG(2, 8)
gap> Hyperplanes(as);
<lines of AG(2, 8)>
```

9.2.4 Incidence and containment

▷ IsIncident(*el1, el2*)   (operation)
▷ *(*el1, el2*)   (operation)
▷ \in(*el1, el2*)   (operation)

   Returns: true or false

   Recall that for affine spaces, incidence is symmetrized containment, where the whole affine space is excluded as one of the arguments for the operation IsIncident, since they it is not considered as an element of the geometry, but the whole affine space is allowed as one of the arguments for \in. The method for * is using IsIncident.

─────────────────────── Example ───────────────────────
```
gap> as := AG(3,16);
AG(3, 16)
gap> p := AffineSubspace(as,[1,0,0]*Z(16)^0);
<a point in AG(3, 16)>
gap> l := AffineSubspace(as,[1,0,0]*Z(16),[[0,1,1]]*Z(16)^0);
<a line in AG(3, 16)>
gap> plane := AffineSubspace(as,[1,0,0]*Z(16)^0,[[1,0,0],[0,1,1]]*Z(16)^0);
<a plane in AG(3, 16)>
gap> p in p;
true
gap> p in l;
false
gap> l in p;
false
gap> l in plane;
true
gap> plane in l;
false
gap> p in plane;
true
gap> p in as;
true
gap> l in as;
true
gap> plane in as;
true
gap> as in p;
false
gap> IsIncident(p,l);
false
gap> IsIncident(l,p);
false
```

```
gap> IsIncident(l,plane);
true
gap> IsIncident(plane,l);
true
gap> IsIncident(p,plane);
true
gap> IsIncident(plane,p);
true
```

9.2.5 AmbientSpace

▷ AmbientSpace(el)   (operation)

   Returns: returns the ambient space of an element el of an affine space
———————————————————————— Example ————————————————————————
```
gap> as := AG(5,7);
AG(5, 7)
gap> solid := AffineSubspace(as,[1,0,0,1,0]*Z(7)^3,[[1,0,0,0,0],[0,1,1,1,0]]*Z(7)^0);
<a plane in AG(5, 7)>
gap> AmbientSpace(solid);
AG(5, 7)
```

9.2.6 BaseField

▷ BaseField(el)   (operation)

   Returns: returns the base field of an element el of an affine space
———————————————————————— Example ————————————————————————
```
gap> as := AG(5,11);
AG(5, 11)
gap> sub := AffineSubspace(as,[1,4,3,1,0]*Z(11)^5,[[1,0,0,0,0],[0,1,1,1,0],
> [0,0,0,0,1]]*Z(11)^0);
<a solid in AG(5, 11)>
gap> BaseField(sub);
GF(11)
```

9.2.7 Span

▷ Span(u, v)   (operation)

   Returns: a subspace

   u and v are subspaces of an affine space. This function returns the span of the two subspaces.
———————————————————————— Example ————————————————————————
```
gap> ag := AffineSpace(4,5);
AG(4, 5)
gap> p := AffineSubspace(ag, [1,0,0,0] * One(GF(5)) );
<a point in AG(4, 5)>
gap> r := AffineSubspace(ag, [0,1,0,0] * One(GF(5)) );
<a point in AG(4, 5)>
gap> l := Span(p, r);
```

```
<a line in AG(4, 5)>
gap> l^_;
[ [ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5) ], [ [ Z(5)^0, Z(5)^2, 0*Z(5), 0*Z(5) ] ] ]
gap> Display(l);
Affine line:
Coset representative: [ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5) ]
Coset (direction): [ [ Z(5)^0, Z(5)^2, 0*Z(5), 0*Z(5) ] ]
```

9.2.8   Meet

▷ Meet(u, v)   (operation)

   Returns: an affine subspace or the empty list

   u and v are subspaces of an affine space. This function returns the meet of the two subspaces. If the two subspaces are disjoint, then Meet returns the empty list.

―――――――――――――――― Example ――――――――――――――――
```
gap> ag := AffineSpace(4,5);
AG(4, 5)
gap> p := AffineSubspace(ag, [1,0,0,0] * One(GF(5)),
>         [[1,0,0,-1], [0,1,0,0],[0,0,1,3]] * One(GF(5)));
<a solid in AG(4, 5)>
gap> l := AffineSubspace(ag, [0,0,0,0] * One(GF(5)), [[1,1,0,0]] * One(GF(5)) );
<a line in AG(4, 5)>
gap> x := Meet(p, l);
<a point in AG(4, 5)>
gap> x^_;
[ Z(5)^0, Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> Display(x);
Affine point:  1 1 . .
```

9.2.9   IsParallel

▷ IsParallel(u, v)   (operation)

   Returns: true or false

   The arguments u and v must be affine subspaces of a common affine space. Two subspaces are parallel if and only if the direction space of the first is contained in the direction space of the second or viceversa.

―――――――――――――――― Example ――――――――――――――――
```
gap> as := AffineSpace(3, 3);
AG(3, 3)
gap> l := AffineSubspace(as,[0,0,0]*Z(3)^0,[[1,0,0]]*Z(3)^0);
<a line in AG(3, 3)>
gap> m := AffineSubspace(as,[1,0,0]*Z(3)^0,[[1,0,0]]*Z(3)^0);
<a line in AG(3, 3)>
gap> n := AffineSubspace(as,[1,0,0]*Z(3)^0,[[0,1,0]]*Z(3)^0);
<a line in AG(3, 3)>
gap> IsParallel(l,m);
true
gap> IsParallel(m,n);
false
```

```
gap> IsParallel(l,n);
false
```

9.2.10 ParallelClass

▷ ParallelClass(*as, v*)   (operation)
▷ ParallelClass(*v*)   (operation)
  Returns: a collection of affine subspaces

The argument *v* is an affine subspace of *as*. This operation returns a collection for which an iterator is installed. The collection represents the set of elements of *as* of the same type as *v* which are parallel to *v*; they have the same direction. If *v* is a point, then this operation returns the collection of all points of *as*. If one argument is given, then it is assumed that the affine space which we are working with is the ambient space of *v*.

——————————— Example ———————————
```
gap> as := AffineSpace(3, 3);
AG(3, 3)
gap> l := Random( Lines( as ) );
<a line in AG(3, 3)>
gap> pclass := ParallelClass( l );
<parallel class of lines in AG(3, 3)>
gap> AsList(pclass);
[ <a line in AG(3, 3)>, <a line in AG(3, 3)>, <a line in AG(3, 3)>,
  <a line in AG(3, 3)>, <a line in AG(3, 3)>, <a line in AG(3, 3)>,
  <a line in AG(3, 3)>, <a line in AG(3, 3)>, <a line in AG(3, 3)> ]
```

9.3 Shadows of Affine Subspaces

9.3.1 ShadowOfElement

▷ ShadowOfElement(*as, v, type*)   (operation)
  Returns: the subspaces of the affine space *as* of dimension *type* which are incident with *v*

as is an affine space and *v* is an element of *as*. This operation computes and returns the subspaces of dimension *type* which are incident with *v*. In fact, this operation returns a collection which is only computed when iterated (e.g. when applying AsList to the collection). Some shorthand notation for ShadowOfElement is available for affine spaces: Points(as,v), Points(v), Lines(v), etc.

——————————— Example ———————————
```
gap> as := AffineSpace(3, 3);
AG(3, 3)
gap> l := Random( Lines( as ) );
<a line in AG(3, 3)>
gap> planesonl := Planes(l);
<shadow planes in AG(3, 3)>
gap> AsList(planesonl);
[ <a plane in AG(3, 3)>, <a plane in AG(3, 3)>, <a plane in AG(3, 3)>,
  <a plane in AG(3, 3)> ]
```

9.3.2 ShadowOfFlag

▷ ShadowOfFlag(`as, list, type`)   (operation)

   Returns: the subspaces of the affine space `as` of dimension `type` which are incident with each element of `list`

   `as` is an affine space and `list` is a list of pairwise incident elements of `as`. This operation computes and returns the subspaces of dimension `type` which are incident with every element of `list`. In fact, this operation returns a collection which is only computed when iterated (e.g. when applying `AsList` to the collection).

—————————————————— Example ——————————————————
```
gap> as := AffineSpace(3, 3);
AG(3, 3)
gap> l := Random( Lines( as ) );
<a line in AG(3, 3)>
gap> x := Random( Points( l ) );
<a point in AG(3, 3)>
gap> flag := FlagOfIncidenceStructure(as,[x,l]);
<a flag of AffineSpace(3, 3)>
gap> shadow := ShadowOfFlag( as, flag, 3 );
<shadow planes in AG(3, 3)>
gap> AsList(shadow);
Iterators of shadows of flags in affine spaces are not complete in this versio
n
[ <a plane in AG(3, 3)>, <a plane in AG(3, 3)>, <a plane in AG(3, 3)>,
  <a plane in AG(3, 3)> ]
```

9.4   Iterators and enumerators

Recall from Section 4.4 ("Enumerating subspaces of a projective space", Chapter 4), that an iterator allows us to obtain elements from a collection one at a time in sequence, whereas an enumerator for a collection give us a way of picking out the i-th element. In FinInG we have enumerators and iterators for subspace collections of affine spaces.

9.4.1   Iterator

▷ Iterator(`subs`)   (operation)

   Returns: an iterator for the given subspaces collection
   `subs` is a collection of subspaces of an affine space, such as `Points( AffineSpace(3, 3) )`.

—————————————————— Example ——————————————————
```
gap> ag := AffineSpace(3, 3);
AG(3, 3)
gap> lines := Lines( ag );
<lines of AG(3, 3)>
gap> iter := Iterator( lines );
<iterator>
gap> l := NextIterator( iter );
<a line in AG(3, 3)>
```

9.4.2 Enumerator

▷ Enumerator(*subs*) (operation)

    Returns: an enumerator for the given subspaces collection

    subs is a collection of subspaces of an affine space, such as Points(AffineSpace(3, 3)).

─────────────────── Example ───────────────────
```
gap> ag := AffineSpace(3, 3);
AG(3, 3)
gap> lines := Lines( ag );
<lines of AG(3, 3)>
gap> enum := Enumerator( lines );
<enumerator of <lines of AG(3, 3)>>
gap> l := enum[20];
<a line in AG(3, 3)>
gap> Display(l);
Affine line:
Coset representative: [ 0*Z(3), 0*Z(3), Z(3)^0 ]
Coset (direction): [ [ Z(3)^0, 0*Z(3), Z(3) ] ]
```

9.5 Affine groups

A *collineation* of an affine space is a permutation of the points which preserves the relation of collinearity within the affine space. The fundamental theorem of affine geometry states that the group $A\Gamma L(d,F)$ of collineations of an affine space $AG(d,F)$ is generated by the translations T, matrices of $GL(d,F)$ and the automorphisms of the field F. The translations T form a normal subgroup of $A\Gamma L(d,F)$, and $A\Gamma L(d,F)$ is the semidirect product of T and $\Gamma L(d,F)$.

Suppose we have an affine transformation of the form $x + A$ where x is a vector representing a translation, and A is a matrix in $GL(d,q)$. Then by using the natural embedding of $AGL(d,q)$ in $PGL(d+1,q)$, we can write this collineation as a matrix:
$$\left(\begin{array}{c|c} A \quad\quad 0 & \begin{matrix} 0 \\ 0 \end{matrix} \\ \hline x & 1 \end{array} \right)$$
. We can extend this idea to the full affine collineation group by adjoining the field automorphisms as we would for projective collineations. Here is an example:

─────────────────── Example ───────────────────
```
gap> ag := AffineSpace(3,3);
AG(3, 3)
gap> g := AffineGroup(ag);
AGL(3,3)
gap> x:=Random(g);;
gap> Display(x);
<a collineation , underlying matrix:
 . 1 1 .
 2 2 . .
 2 1 . .
 1 2 1 1
, F^0>
```

Here we see that this affine transformation is

$$(1,2,1) + \begin{pmatrix} 1 \\ 1 \\ 2 \\ 2 \\ 0 \\ 2 \\ 1 \\ 1 \end{pmatrix}.$$

As we have seen, in FinInG, we represent an element of an affine collineation group as a projective semilinear element, i.e. as an object in the category `ProjElsWithFrob`, so that we can use all the functionality that exists for such objects. However, an affine collineation group is not by default constructed as a subgroup of $P\Gamma L(d,F)$, but the compatibility between the elements of both groups enables testing for such relations.

```
─────── Example ───────
gap> G := CollineationGroup(AG(3,27));
AGammaL(3,27)
gap> H := CollineationGroup(PG(3,27));
The FinInG collineation group PGammaL(4,27)
gap> g := Random(G);
< a collineation: [ [ Z(3^3)^25, Z(3^3)^11, Z(3^3)^23, 0*Z(3) ],
  [ Z(3^3)^20, 0*Z(3), Z(3^3), 0*Z(3) ],
  [ Z(3^3)^16, Z(3^3)^15, Z(3^3)^21, 0*Z(3) ],
  [ Z(3^3)^20, Z(3^3)^4, 0*Z(3), Z(3)^0 ] ], F^3>
gap> g in H;
true
gap> IsSubgroup(H,G);
true
```

9.5.1 AffineGroup

▷ AffineGroup(as)   (operation)
   Returns: a group
   This operation returns the affine linear group $AGL(V)$ acting on the affine space with underlying vector space V. The elements of this group are projectivities of the associated projective space. In order to get the full group of collineations of the affine space, one needs to use the operation `CollineationGroup`.

```
─────── Example ───────
gap> as := AffineSpace(4,7);
AG(4, 7)
gap> g := AffineGroup(as);
AGL(4,7)
gap> as := AffineSpace(4,8);
AG(4, 8)
gap> g := AffineGroup(as);
AGL(4,8)
```

9.5.2 CollineationGroup

▷ CollineationGroup(*as*) (operation)

Returns: a group

If *as* is the affine space $AG(d,q)$, then this operation returns the affine semilinear group $A\Gamma L(d,q)$ The elements of this group are collineations of the associated projective space. Note that if the defining field has prime order, then $A\Gamma L(d,q) = AGL(d,q)$.

```
───────────────────── Example ─────────────────────
gap> as := AffineSpace(4,8);
AG(4, 8)
gap> g := CollineationGroup(as);
AGammaL(4,8)
gap> h := AffineGroup(as);
AGL(4,8)
gap> IsSubgroup(g,h);
true
gap> as := AffineSpace(4,7);
AG(4, 7)
gap> g := CollineationGroup(as);
AGL(4,7)
```

9.5.3 OnAffineSpaces

▷ OnAffineSpaces(*subspace, el*) (operation)
▷ \^(*subspace, el*) (operation)

Returns: an element of an affine space

subspace must be an element of an affine space and *el* a collineation of an affine space (which is in fact also a collineation of an associated projective space). This is the action one should use for collineations of affine spaces, and it acts on subspaces of all types of affine spaces: points, lines, planes, etc.

```
───────────────────── Example ─────────────────────
gap> as := AG(3,27);
AG(3, 27)
gap> p := Random(Points(as));
<a point in AG(3, 27)>
gap> g := Random(CollineationGroup(as));
< a collineation: [ [ Z(3^3)^25, Z(3^3)^11, Z(3^3)^23, 0*Z(3) ],
  [ Z(3^3)^20, 0*Z(3), Z(3^3), 0*Z(3) ],
  [ Z(3^3)^16, Z(3^3)^15, Z(3^3)^21, 0*Z(3) ],
  [ Z(3^3)^20, Z(3^3)^4, 0*Z(3), Z(3)^0 ] ], F^3>
gap> OnAffineSubspaces(p,g);
<a point in AG(3, 27)>
gap> p^g;
<a point in AG(3, 27)>
gap> l := Random(Lines(as));
<a line in AG(3, 27)>
gap> OnAffineSubspaces(l,g);
<a line in AG(3, 27)>
gap> l^g;
```

```
<a line in AG(3, 27)>
```

9.6 Low level operations

One technical aspect of the design behind affine spaces in FinInG is having canonical transversals for subspaces of vector spaces. We provide some documentation below for the interested user.

9.6.1 IsVectorSpaceTransversal

▷ IsVectorSpaceTransversal   (filter)

The category `IsVectorSpaceTransversal` represents a special object in FinInG which carries a record with two components: *space* and *subspace*. This category is a subcategory of `IsSubspacesOfVectorSpace`, however, we do not recommend the user to apply methods to objects in `IsVectorSpaceTransversal`, which are normally used for the category `IsSubspacesOfVectorSpace` (they won't work!). The objects in `IsVectorSpaceTransversal` are only used in order to facilitate computing enumerators of subspace collections.

9.6.2 VectorSpaceTransversal

▷ VectorSpaceTransversal(*space*, *mat*)   (operation)

Returns: a collection for representing a transversal of a subspaces of a vector space

space is a vector space V and `mat` is a matrix whose rows are a basis for a subspace U of V. A transversal for U in V is a set of coset representatives for the quotient V/U. This collection comes equipped with an enumerator operation.

9.6.3 VectorSpaceTransversalElement

▷ VectorSpaceTransversalElement(*space*, *mat*, *vector*)   (operation)

Returns: a canonical coset representative

space is a vector space V, `mat` is a matrix whose rows are a basis for a subspace U of V, and *vector* is a vector v of V. A canonical representative v' is returned for the coset $U + v$.

9.6.4 ComplementSpace

▷ ComplementSpace(*space*, *mat*)   (operation)

Returns: a collection for representing a transversal of a subspaces of a vector space

space is a vector space V and `mat` is a matrix whose rows are a basis for a subspace U of V. The operation is almost a complete copy of the function `BaseSteinitzVector` except that just a basis for the complement of U is returned instead of a full record.

Chapter 10

Geometry Morphisms

Here we describe what is meant by a *geometry morphism* in FinInG and the various operations and tools available to the user. When using groups in GAP, we often use homomorphisms to pass from one situation to another, even though mathematically it may appear to be unneccessary, there can be ambiguities if the functionality is too flexible. This also applies to finite geometry. Take for example the usual exercise of thinking of a hyperplane in a projective space as another projective space. To conform with similar situations in GAP, the right thing to do is to embed one projective space into another, rather than having one projective space automatically as a substructure of another. The reason for this is that there are many ways one can do this embedding, even though we may dispense with this choice when we are working mathematically. So to avoid ambiguity, we stipulate that one should construct the embedding explicitly. How this is done will be described this chapter.

Suppose that S and S' are two incidence geometries. A *geometry morphism* from S to S' is defined to be a map from the elements of S to the elements of S' which preserves incidence and induces a function from the type set of S to the type set of S'. For instance, a correlation and a collineation are examples of geometry morphisms, but they have been dealt with in more specific ways in FinInG. We will mainly be concerned with geometry morphisms where the source and range are different. Hence, the natural embedding of a projective space in a larger projective space, the mapping induced by field reduction, and e.g. the Klein correspondence are examples of such geometry morphisms.

As a geometry morphism from S to S' preserves incidence, it also preserves the symmetry, and hence it induces also a map from the collineation group of S into the collineation group of S'. Such a map will be called an *Intertwiner*, and FinInG can provide these maps for some of the geometry morphisms.

Note that quite some technicalities are needed in the implementation of some geometry morphisms. This chapters deals only with the user interface. Some low level functions for geometry morphisms are described in Appendix C.

10.1 Geometry morphisms in FinInG

10.1.1 IsGeometryMorphism

▷ IsGeometryMorphism (family)

The category `IsGeometryMorphism` represents a special object in FinInG which carries attributes and the given element map. The element map is given as a `IsGeneralMapping`, and so has a source

and range.

```
 ───────────────────────── Example ─────────────────────────
 gap> ShowImpliedFilters(IsGeometryMorphism);
 Implies:
    IsGeneralMapping
    IsTotal
    Tester(IsTotal)
    IsSingleValued
    Tester(IsSingleValued)
```

The usual operations of `ImageElm`, and `PreImageElm`, have methods installed for geometry morphisms, as well as the overload operator `\^`.

10.1.2 Intertwiner

▷ `Intertwiner(f)` (attribute)

Returns: a group homomorphism

The argument `f` is a geometry morphism. If `f` comes equipped with a natural intertwiner from an automorphism group of the source of `f` to the automorphism group to the image of `f`, then the user may is able to obtain the intertwiner by calling this operation (see the individual geometry morphism constructions). For most geometry morphisms, there is also an accompanying intertwiner for the automorphism groups of the source and range. Given a geometry morphism f from S to S', an intertwiner ϕ is a map from the automorphism group of S to the automorphism group of S', such that for every element p of S and every automorphism g of S, we have

$$f(p^g) = f(p)^{\phi(g)}.$$

There is no method to compute an intertwiner for a given geometry morphism, the attribute is or is not set during the construction of the geometry morphism, depending whether the Source and Range of the morphism have the appropriate automorphism group known as an attribute. When this condition is not satisfied, the user is expected to call the appropriate automorphism groups, so that they are computed, and to recompute the geometry morphism (which will not cost a lot of computation time then). This will make the attribute `Intertwiner` available. Here is a simple example of the intertwiner for the isomorphism of two polar spaces (see `IsomorphismPolarSpaces` (10.2.1)). The source of the homomorphism is dependent on the geometry.

```
 ───────────────────────── Example ─────────────────────────
 gap> form := BilinearFormByMatrix( IdentityMat(3,GF(3)), GF(3) );
 < bilinear form >
 gap> ps := PolarSpace(form);
 <polar space in ProjectiveSpace(2,GF(3)): x_1^2+x_2^2+x_3^2=0 >
 gap> pq := ParabolicQuadric(2,3);
 standard Q(2, 3)
 gap> iso := IsomorphismPolarSpaces(ps, pq);
 #I  Computing nice monomorphism...
 <geometry morphism from <Elements of <polar space in ProjectiveSpace(2,GF(
 3)): x_1^2+x_2^2+x_3^2=0 >> to <Elements of standard Q(2, 3)>>
 gap> KnownAttributesOfObject(iso);
 [ "Range", "Source", "Intertwiner" ]
 gap> hom := Intertwiner(iso);
```

```
MappingByFunction( <projective semilinear group with
3 generators>, PGammaO(3,3), function( y ) ... end, function( x ) ... end )
```

10.2 Type preserving bijective geometry morphisms

An important class of geometry morphisms in FinInG are the isomorphisms between polar spaces of
the same kind that are induced by coordinate transformations.

10.2.1 IsomorphismPolarSpaces

▷ IsomorphismPolarSpaces(*ps1*, *ps2*)   (operation)
▷ IsomorphismPolarSpaces(*ps1*, *ps2*, *boolean*)   (operation)
   Returns: a geometry morphism
   The arguments *ps1* and *ps2* are *equivalent* polar spaces, i.e. up to coordinate transformation,
the underlying sesquilinear or quadratic form determines the same polar space, or, *ps1* is a parabolic
quadric over a finite field f of even charateristic in dimension $2n$ and *ps2* is a symplectic space over f
in dimension $2n - 1$, then this operation returns a geometry isomorphism between them. The optional
third argument *boolean* can take either true or false as input, and then the operation will or will
not compute the intertwiner accordingly. The user may wish that the intertwiner is not computed
when working with large polar spaces. The default (when calling the operation with two arguments)
is set to true, and in this case, if at least one of *ps1* or *ps2* has a collineation group installed as
an attribute, then an intertwining homomorphism is installed as an attribute of the resulting geometry
morphism. Hence we also obtain a natural group isomorphism from the collineation group of *ps1*
onto the collineation group of *ps2* (see also Intertwiner (10.1.2)).

─────────────────────────── Example ───────────────────────────
```
gap> mat1 := IdentityMat(6,GF(5));
< mutable compressed matrix 6x6 over GF(5) >
gap> form1 := BilinearFormByMatrix(mat1,GF(5));
< bilinear form >
gap> ps1 := PolarSpace(form1);
<polar space in ProjectiveSpace(
5,GF(5)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2=0 >
gap> mat2 := [[0,0,0,0,0,1],[0,0,0,0,1,0],[0,0,0,1,0,0],
> [0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0]]*Z(5)^0;
[ [ 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0 ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0, 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5) ] ]
gap> form2 := QuadraticFormByMatrix(mat2,GF(5));
< quadratic form >
gap> ps2 := PolarSpace(form2);
<polar space in ProjectiveSpace(5,GF(5)): x_1*x_6+x_2*x_5+x_3*x_4=0 >
gap> iso := IsomorphismPolarSpaces(ps1,ps2,true);
#I  No intertwiner computed. One of the polar spaces must have a collineation group computed
<geometry morphism from <Elements of Q+(5,
5): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2=0> to <Elements of Q+(5,
```

```
5): x_1*x_6+x_2*x_5+x_3*x_4=0>>
gap> CollineationGroup(ps1);
#I  Computing collineation group of canonical polar space...
<projective collineation group of size 58032000000 with 4 generators>
gap> CollineationGroup(ps2);
#I  Computing collineation group of canonical polar space...
<projective collineation group of size 58032000000 with 4 generators>
gap> iso := IsomorphismPolarSpaces(ps1,ps2,true);
<geometry morphism from <Elements of Q+(5,
5): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2=0> to <Elements of Q+(5,
5): x_1*x_6+x_2*x_5+x_3*x_4=0>>
gap> hom := Intertwiner( iso );
MappingByFunction( <projective collineation group of size 58032000000 with
4 generators>, <projective collineation group of size 58032000000 with
4 generators>, function( y ) ... end, function( x ) ... end )
gap> ps1 := ParabolicQuadric(6,8);
Q(6, 8)
gap> ps2 := SymplecticSpace(5,8);
W(5, 8)
gap> em := IsomorphismPolarSpaces(ps1,ps2);
#I  Have 36171 points.
#I  Have 37381 points in new orbit.
#I  Have 36171 points.
#I  Have 37388 points in new orbit.
<geometry morphism from <Elements of Q(6, 8)> to <Elements of W(5, 8)>>
gap> hom := Intertwiner(em);
MappingByFunction( PGammaO(7,8), <projective collineation group of size
27231016821530296320 with
3 generators>, function( el ) ... end, function( el ) ... end )
```

10.3 Klein correspondence and derived dualities

The Klein correspondence is a well known geometry morphism from the lines of $PG(3,q)$ to the points of a hyperbolic quadric in $PG(5,q)$. This morphism and some derived morphisms are provided in FinInG. The bare essential of the Klein corresondence is the so-called Plücker map.

10.3.1 PluckerCoordinates

▷ PluckerCoordinates(*line*)   (operation)

This operation takes a line of $PG(3,q)$ as argument. It returns the plucker coordinates of the argument as list of finite field elements. The retured list can be used in operations as VectorSpaceToElement, and represents a point of the hyperbolic quadric in $PG(5,q)$ with equation $X_0X_5 + X_1X_4 + X_2X_3 = 0$

——————————————— Example ———————————————
```
gap> pg := PG(3,169);
ProjectiveSpace(3, 169)
gap> l := Random(Lines(pg));
<a line in ProjectiveSpace(3, 169)>
```

```
gap> vec := PluckerCoordinates(l);
[ Z(13)^0, Z(13^2)^138, Z(13^2)^93, Z(13^2)^53, Z(13^2)^71, Z(13^2)^106 ]
gap> mat := [[0,0,0,0,0,1],[0,0,0,0,1,0],[0,0,0,1,0,0],
> [0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0]]*Z(13)^0;
[ [ 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13), Z(13)^0 ],
  [ 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13), Z(13)^0, 0*Z(13) ],
  [ 0*Z(13), 0*Z(13), 0*Z(13), Z(13)^0, 0*Z(13), 0*Z(13) ],
  [ 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13) ],
  [ 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13) ],
  [ 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13), 0*Z(13) ] ]
gap> form := QuadraticFormByMatrix(mat,GF(169));
< quadratic form >
gap> klein := PolarSpace(form);
<polar space in ProjectiveSpace(5,GF(13^2)): x_1*x_6+x_2*x_5+x_3*x_4=0 >
gap> VectorSpaceToElement(klein,vec);
<a point in Q+(5, 169): x_1*x_6+x_2*x_5+x_3*x_4=0>
```

10.3.2 KleinCorrespondence

▷ KleinCorrespondence(f)   (operation)
▷ KleinCorrespondence(f, boolean)   (operation)
▷ KleinCorrespondence(q)   (operation)
▷ KleinCorrespondence(q, boolean)   (operation)

  Returns: a geometry morphism

The argument f is a finite field, the argument q is a prime power. The first an the third version use `true` as value for `boolean`. When using `true` as value for the boolean, the intertwiner is computed. This variant of the operation KleinCorrespondence has always as ambient geometry of its range the hyperbolic quadric $Q^+(5,q)$ with equation $X_0 X_5 + X_1 X_4 + X_2 X_3 = 0$ The returned geometry morphism has the lines of $PG(3,q)$ as source and the points of $Q^+(5,q)$ as range.

```
————————————————— Example —————————————————
gap> k := KleinCorrespondence( 9 );
<geometry morphism from <lines of ProjectiveSpace(3, 9)> to <points of Q+(5,
9): x_1*x_6+x_2*x_5+x_3*x_4=0>>
gap> Intertwiner(k);
MappingByFunction( The FinInG collineation group PGammaL(4,9), <projective col
lineation group with
3 generators>, function( g ) ... end, function( g ) ... end )
gap> pg := ProjectiveSpace(3, 9);
ProjectiveSpace(3, 9)
gap> AmbientGeometry(Range(k));
Q+(5, 9): x_1*x_6+x_2*x_5+x_3*x_4=0
gap> l := Random( Lines(pg) );
<a line in ProjectiveSpace(3, 9)>
gap> l^k;
<a point in Q+(5, 9): x_1*x_6+x_2*x_5+x_3*x_4=0>
```

10.3.3 KleinCorrespondence

▷ KleinCorrespondence(*quadric*) (operation)
▷ KleinCorrespondence(*quadric, boolean*) (operation)

Returns: a geometry morphism

The argument *quadric* is a hyperbolic quadric in a 5 dimensional projective space. If `boolean` is `true` or not given, this operation returns the geometry morphism equipped with an intertwiner. The returned geometry morphism has the lines of $PG(3, q)$ as source and the points of $Q^+(5, q)$ as range.

```
─────────────────────────── Example ───────────────────────────
gap> quadric := HyperbolicQuadric(5,3);
Q+(5, 3)
gap> k := KleinCorrespondence( quadric );
<geometry morphism from <lines of ProjectiveSpace(3, 3)> to <points of Q+(5,
3)>>
gap> pg := ProjectiveSpace(3, 3);
ProjectiveSpace(3, 3)
gap> l := Random( Lines(pg) );
<a line in ProjectiveSpace(3, 3)>
gap> l^k;
<a point in Q+(5, 3)>
gap> id := IdentityMat(6,GF(13));
< mutable compressed matrix 6x6 over GF(13) >
gap> form := QuadraticFormByMatrix(id,GF(13));
< quadratic form >
gap> quadric := PolarSpace(form);
<polar space in ProjectiveSpace(
5,GF(13)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2=0 >
gap> k := KleinCorrespondence( quadric );
<geometry morphism from <lines of ProjectiveSpace(3, 13)> to <points of Q+(5,
13): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2=0>>
gap> pg := AmbientGeometry(Source(k));
ProjectiveSpace(3, 13)
gap> l := Random(Lines(pg));
<a line in ProjectiveSpace(3, 13)>
gap> l^k;
<a point in Q+(5, 13): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2=0>
```

10.3.4 KleinCorrespondenceExtended

▷ KleinCorrespondenceExtended(*quadric*) (operation)
▷ KleinCorrespondenceExtended(*quadric, boolean*) (operation)

Returns: a geometry morphism

The argument *quadric* is a hyperbolic quadric in a 5 dimensional projective space. If `boolean` is `true` or not given, this operation returns the geometry morphism equipped with an intertwiner. The returned geometry morphism has all the elements of $PG(3, q)$ as source (not just the lines) and the elements of $Q^+(5, q)$ as range, hence this operation is a kind of extension of `KleinCorrespondence`.

```
─────────────────────────── Example ───────────────────────────
gap> ps := HyperbolicQuadric(5,7);
Q+(5, 7)
gap> em := KleinCorrespondenceExtended(ps);
```

```
 <geometry morphism from <All elements of ProjectiveSpace(3,
 7)> to <Elements of Q+(5, 7)>>
gap> hom := Intertwiner(em);
MappingByFunction( The FinInG collineation group PGL(4,7), <projective colline
ation group with 2 generators>, function( g ) ... end, function( g ) ... end )
gap> mat := [[0,0,0,0,0,1],[0,0,0,0,1,0],[0,0,0,1,0,0],
>           [0,0,1,0,0,0],[0,1,0,0,0,0],[1,0,0,0,0,0]]*Z(7)^0;
[ [ 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0 ],
  [ 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7) ],
  [ Z(7)^0, 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7) ] ]
gap> g := Projectivity(mat,GF(7));
< a collineation: <cmat 6x6 over GF(7,1)>, F^0>
gap> g in CollineationGroup(ps);
true
gap> PreImageElm(hom,g);
#I  <el> is not inducing a collineation of PG(3,q)
fail
```

It is well known that the classical generalised quadrangles $W(3,q)$ and $Q(4,q)$ are dual incidence structures, the same holds for the classical generalised quadrangles $Q^-(5,q)$ and $H(3,q^2)$. Essentially, these dual dualities are based on the Klein correspondence, and are implemented through the operation `NaturalDuality`, this operation will return a geometry morphism with `ElementsOfIncidenceStructure(gq1)` as source and `ElementsOfIncidenceStructure(gq2)` as range, in other words, it is a geometry morphism from all the elements of *gq1* onto all the elements of *gq2*, preserving the incidence, and swapping the types.

10.3.5 NaturalDuality

▷ NaturalDuality(*gq1, gq2*)   (operation)
▷ NaturalDuality(*gq1, gq2, boolean*)   (operation)
▷ NaturalDuality(*gq*)   (operation)
▷ NaturalDuality(*gq, boolean*)   (operation)
  Returns: a geometry morphism
  The operation allows the construction of the duality between $W(3,q)$ and $Q(4,q)$, respectively $Q^-(5,q)$, or $H(3,q^2)$, in two directions. It is checked if the arguments are appropriate, i.e. the right type of generalised quadrangle(s). The first version requires two arguments: either the symplectic or parabolic quadrangle, in any order, and defined by any suitable bilinear/quadratic and bilinear form; or the elliptic or hermitian quadrangle (in dimension 3), in any order, and defined by any suitable bilinear/quadratic and hermitian form. In all cases the generalised quadrangles may be the standard one provided by the package *FinInG*.
  The third version requires only one argument, either $W(3,q)$, $Q(4,q)$, $Q^-(5,q)$, or $H(3,q^2)$, standard or user specified using an appropriate bilinear, quadratic or hermitian form. The range of the returned geometry morphism will be the set of all elements of a suitable generalised quadrangle, in standard form.
  The first and third version without a boolean as argument will, if possible return a geometry

morphism equipped with an intertwiner. Using the boolean argument `false` will return a geometry morphism that is not equipped with an intertwiner.

```
────────────────────────── Example ──────────────────────────
gap> w := SymplecticSpace(3,5);
W(3, 5)
gap> lines:=AsList(Lines(w));;
gap> duality := NaturalDuality(w);
<geometry morphism from <Elements of W(3, 5)> to <Elements of Q(4, 5)>>
gap> l:=lines[1];
<a line in W(3, 5)>
gap> l^duality;
<a point in Q(4, 5)>
gap> PreImageElm(duality,last);
<a line in W(3, 5)>
gap> hom := Intertwiner(duality);
MappingByFunction( PGammaSp(4,5), <projective collineation group of size
9360000 with 4 generators>, function( g ) ... end, function( g ) ... end )
gap> q := 5;
5
gap> q5q := EllipticQuadric(5,q);
Q-(5, 5)
gap> mat := [[0,1,0,0],[1,0,0,0],[0,0,0,Z(q)],[0,0,Z(q),0]]*Z(q)^0;
[ [ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5) ], [ Z(5)^0, 0*Z(5), 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5) ], [ 0*Z(5), 0*Z(5), Z(5), 0*Z(5) ] ]
gap> hform := HermitianFormByMatrix(mat,GF(q^2));
< hermitian form >
gap> herm := PolarSpace(hform);
<polar space in ProjectiveSpace(
3,GF(5^2)): x1^5*x2+x1*x2^5+Z(5)*x3^5*x4+Z(5)*x3*x4^5=0 >
gap> duality := NaturalDuality(q5q,herm,true);
<geometry morphism from <Elements of Q-(5, 5)> to <Elements of H(3,
5^2): x1^5*x2+x1*x2^5+Z(5)*x3^5*x4+Z(5)*x3*x4^5=0>>
gap> hom := Intertwiner(duality);
MappingByFunction( PDeltaO-(6,5), <projective collineation group of size
58968000000 with 3 generators>, function( g ) ... end, function( g ) ... end )
gap> g := Random(CollineationGroup(q5q));
< a collineation: <cmat 6x6 over GF(5,1)>, F^0>
gap> g^hom;
< a collineation: <cmat 4x4 over GF(5,2)>, F^5>
```

The combination of the isomorphism of the GQs $W(3,q)$, $Q(4,q)$ when q is even and the duality between the same GQs, yields a duality from each of these GQs itself. The operation `SelfDuality` implements this combination.

10.3.6 SelfDuality

▷ SelfDuality(*gq*)   (operation)
▷ SelfDuality(*gq, boolean*)   (operation)
   Returns: a geometry morphism
   It is checked wheter the base field of *gq* is a field of charateristic 2 and whether *gq* is a symplectic generalised quadrangle in 3-dimensional projective space or a parabolic quadric in 4-dimensional

projective space. The first version will return, when possible, a geometry morphism equipped with an intertwiner. Using the boolean argument `false` will return a geometry morphism that is not equipped with an intertwiner. The example shows the use of the boolean argument.

```
————————————————————— Example —————————————————————
gap> q := 16;
16
gap> mat := [[0,1,0,0,0],[0,0,0,0,0],[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0]]*Z(q)^0;
[ [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ] ]
gap> form := QuadraticFormByMatrix(mat,GF(q));
< quadratic form >
gap> q4q := PolarSpace(form);
<polar space in ProjectiveSpace(4,GF(2^4)): x_1*x_2+x_3^2+x_4*x_5=0 >
gap> em := SelfDuality(q4q);
#I  No intertwiner computed. The polar space must have a collineation group computed
<geometry morphism from <Elements of Q(4,
16): x_1*x_2+x_3^2+x_4*x_5=0> to <Elements of Q(4,
16): x_1*x_2+x_3^2+x_4*x_5=0>>
gap> CollineationGroup(q4q);
#I  Computing collineation group of canonical polar space...
<projective collineation group of size 4380799795200 with 3 generators>
gap> em := SelfDuality(q4q);
<geometry morphism from <Elements of Q(4,
16): x_1*x_2+x_3^2+x_4*x_5=0> to <Elements of Q(4,
16): x_1*x_2+x_3^2+x_4*x_5=0>>
gap> hom := Intertwiner(em);
MappingByFunction( <projective collineation group of size 4380799795200 with
3 generators>, <projective collineation group of size 4380799795200 with
3 generators>, function( el ) ... end, function( el ) ... end )
gap> q := 16;
16
gap> w := SymplecticSpace(3,q);
W(3, 16)
gap> em := SelfDuality(w);
<geometry morphism from <Elements of W(3, 16)> to <Elements of W(3, 16)>>
```

10.4   Embeddings of projective spaces

The most natural of geometry morphisms include, for example, the embedding of a projective space into another via a subspace, the embedding of a projective space over a field into a projective space of the same dimension over an extended field, or the embedding of a projective space over a field into a projective space of higher dimension over a subfield through so-called field reduction.

10.4.1   NaturalEmbeddingBySubspace

▷ NaturalEmbeddingBySubspace(geom1, geom2, v)   (operation)

   Returns: a geometry morphism

The arguments *geom1* and *geom2* are both projective spaces, and *v* is an element of a geom2. This function returns a geometry morphism representing the natural embedding of *geom1* into *geom2* as the subspace *v*. Hence *geom1* and *v* must be equivalent as geometries. An Intertwiner is not implemented for this geometry morphism.

```
———————————————————————— Example ————————————————————————
gap> geom1 := ProjectiveSpace(2, 3);
ProjectiveSpace(2, 3)
gap> geom2 := ProjectiveSpace(3, 3);
ProjectiveSpace(3, 3)
gap> planes := Planes(geom2);
<planes of ProjectiveSpace(3, 3)>
gap> hyp := Random(planes);
<a plane in ProjectiveSpace(3, 3)>
gap> em := NaturalEmbeddingBySubspace(geom1, geom2, hyp);
<geometry morphism from <All elements of ProjectiveSpace(2,
3)> to <All elements of ProjectiveSpace(3, 3)>>
gap> points := Points(geom1);
<points of ProjectiveSpace(2, 3)>
gap> x := Random(points);
<a point in ProjectiveSpace(2, 3)>
gap> x^em;
<a point in ProjectiveSpace(3, 3)>
```

10.4.2 NaturalEmbeddingBySubField

▷ NaturalEmbeddingBySubField(*geom1, geom2*)   (operation)

▷ NaturalEmbeddingBySubField(*geom1, geom2, boolean*)   (operation)

   Returns: a geometry morphism

   The arguments *geom1* and *geom2* are projective spaces of the same dimension. This function returns a geometry morphism representing the natural embedding of *geom1* into *geom2* as a subfield geometry. The geometry morphism also comes equipped with an intertwiner (see `Intertwiner` (10.1.2)). The optional third argument *boolean* can take either `true` or `false` as input, and then our operation will or will not compute the intertwiner accordingly. The default (when calling the operation with two arguments) is set to `true`. Note that the source of the intertwiner is the projectivity group of *geom1* and its range is a subgroup of the projectivity group of *geom2*. Here is a simple example where the geometry morphism embeds $PG(2,3)$ into $PG(2,9)$.

```
———————————————————————— Example ————————————————————————
gap> pg1 := PG(2,3);
ProjectiveSpace(2, 3)
gap> pg2 := PG(2,9);
ProjectiveSpace(2, 9)
gap> em := NaturalEmbeddingBySubfield(pg1,pg2);
<geometry morphism from <All elements of ProjectiveSpace(2,
3)> to <All elements of ProjectiveSpace(2, 9)>>
gap> points := AsList(Points( pg1 ));
[ <a point in ProjectiveSpace(2, 3)>, <a point in ProjectiveSpace(2, 3)>,
  <a point in ProjectiveSpace(2, 3)>, <a point in ProjectiveSpace(2, 3)>,
  <a point in ProjectiveSpace(2, 3)>, <a point in ProjectiveSpace(2, 3)>,
  <a point in ProjectiveSpace(2, 3)>, <a point in ProjectiveSpace(2, 3)>,
  <a point in ProjectiveSpace(2, 3)>, <a point in ProjectiveSpace(2, 3)>,
```

```
     <a point in ProjectiveSpace(2, 3)>, <a point in ProjectiveSpace(2, 3)>,
     <a point in ProjectiveSpace(2, 3)> ]
gap> image := ImagesSet(em, points);
[ <a point in ProjectiveSpace(2, 9)>, <a point in ProjectiveSpace(2, 9)>,
  <a point in ProjectiveSpace(2, 9)>, <a point in ProjectiveSpace(2, 9)>,
  <a point in ProjectiveSpace(2, 9)>, <a point in ProjectiveSpace(2, 9)>,
  <a point in ProjectiveSpace(2, 9)>, <a point in ProjectiveSpace(2, 9)>,
  <a point in ProjectiveSpace(2, 9)>, <a point in ProjectiveSpace(2, 9)>,
  <a point in ProjectiveSpace(2, 9)> ]
gap> hom := Intertwiner(em);
MappingByFunction( The FinInG projectivity group PGL(3,3), <projective colline
ation group of size 5616 with
2 generators>, function( x ) ... end, function( y ) ... end )
gap> group1 := ProjectivityGroup(pg1);
The FinInG projectivity group PGL(3,3)
gap> gens := GeneratorsOfGroup(group1);
[ < a collineation: <cmat 3x3 over GF(3,1)>, F^0>,
  < a collineation: <cmat 3x3 over GF(3,1)>, F^0> ]
gap> group1_image := Group(List(gens,x->x^hom));
<projective collineation group with 2 generators>
gap> Order(group1_image);
5616
gap> group2 := ProjectivityGroup(pg2);
The FinInG projectivity group PGL(3,9)
gap> Order(group2);
42456960
gap> g := Random(group2);
< a collineation: <cmat 3x3 over GF(3,2)>, F^0>
gap> PreImageElm(hom,g);
#I  <el> is not in the range of the intertwiner
fail
```

10.4.3 Embedding of projective spaces by field reduction

We briefly describe the mathematics behind field reduction. For more details we refer to [LVdV]. Consider the fields $K = GF(q)$ and $L = GF(q^t)$. The field L is a t-dimensional vector space over K. Hence, with respect to a chosen basis B for L as a K-vectorspace, the bijection between the vectorspaces $V(n,q^t)$ and $V(tn,q)$ can be implemented. Consider the projective space $PG(n-1,q^t)$. The elements are represented by subspaces of $V(n,q^t)$. Clearly, a k dimensional subspace of $V(n,q^t)$ is also a kn-dimensional subspace of the $V(nt,q)$. This induces an embedding from $PG(n-1,q^t)$ into $PG(nt-1,q)$. The embedding will be determined by the chosen basis of L as a vector space over $GF(q)$.

10.4.4 BlownUpSubspaceOfProjectiveSpace

▷ BlownUpSubspaceOfProjectiveSpace(B, subspace) (operation)

    Returns: a subspace of a projective space

    Let B be a basis for the field $L = GF(q^t)$ as $GF(q)$ vector space, and let *subspace* be a $k-1$-

dimensional subspace of $PG(n-1, q^t)$ represented by a k-dimensional subspace S of $V(n, q^t)$. This operation returns the $kt - 1$-dimensional subspace of $PG(nt-1, q)$ represented by blowing up S with respect to the base B. This operation relies on the GAP operation `BlownUpMat`. In the example, the effect of chosen a different basis is shown.

```
———————————————— Example ————————————————
  gap> pg := PG(3,5^2);
  ProjectiveSpace(3, 25)
  gap> basis := Basis(AsVectorSpace(GF(5),GF(5^2)));
  CanonicalBasis( GF(5^2) )
  gap> line := Random(Lines(pg));
  <a line in ProjectiveSpace(3, 25)>
  gap> solid1 := BlownUpSubspaceOfProjectiveSpace(basis,line);
  <a solid in ProjectiveSpace(7, 5)>
  gap> BasisVectors(basis);
  [ Z(5)^0, Z(5^2) ]
  gap> basis := Basis(AsVectorSpace(GF(5),GF(5^2)),[Z(5),Z(5^2)^8]);
  Basis( GF(5^2), [ Z(5), Z(5^2)^8 ] )
  gap> solid2 := BlownUpSubspaceOfProjectiveSpace(basis,line);
  <a solid in ProjectiveSpace(7, 5)>
  gap> solid1 = solid2;
  false
```

10.4.5 NaturalEmbeddingByFieldReduction

▷ NaturalEmbeddingByFieldReduction(*geom1, f2, B*)   (operation)
▷ NaturalEmbeddingByFieldReduction(*geom1, f2*)   (operation)
▷ NaturalEmbeddingByFieldReduction(*geom1, geom2*)   (operation)
▷ NaturalEmbeddingByFieldReduction(*geom1, geom2, B*)   (operation)

Returns: a geometry morphism

This operation comes in four flavours. For the first flavour, the argument *geom1* is a projective space over a field $L = GF(q^t)$. The argument *f2* is a subfield $K = GF(q)$ of L. The argument B is a basis for L as a K-vectorspace. When this argument is not given, a basis for L over K is computed using `Basis(AsVectorSpace(K,L))`. It is checked whether *f2* is a subfield of the basefield of *geom1*. The third and fourth flavour are comparable, where now K is found as the basefield of *geom2*. In fact the arguments *geom1* and *geom2* are the projective spaces $PG(r-1, q^t)$ and $PG(rt-1, q)$ respectively. As in the previous flavours, the argument B is optional.

An intertwiner is always available for this geometry morphism, and has source the homography group of *geom1* and as range a subgroup of the homography group of *geom2* (or the projective space of the appropriate dimension over *f2*. Notice in the example below the difference of a factor 2 in the orders of the group, which comes of course from restricing the homomorphism to the homography group, which differes a factor 2 from the collineation group of the projective line, that has an extra automorphism of order two, corresponding with the Frobenius automorphism.

```
———————————————— Example ————————————————
  gap> pg1 := ProjectiveSpace(2,81);
  ProjectiveSpace(2, 81)
  gap> f2 := GF(9);
  GF(3^2)
  gap> em := NaturalEmbeddingByFieldReduction(pg1,f2);
  <geometry morphism from <All elements of ProjectiveSpace(2,
```

```
81)> to <All elements of ProjectiveSpace(5, 9)>>
gap> f2 := GF(3);
GF(3)
gap> em := NaturalEmbeddingByFieldReduction(pg1,f2);
<geometry morphism from <All elements of ProjectiveSpace(2,
81)> to <All elements of ProjectiveSpace(11, 3)>>
gap> pg2 := ProjectiveSpace(11,3);
ProjectiveSpace(11, 3)
gap> em := NaturalEmbeddingByFieldReduction(pg1,pg2);
<geometry morphism from <All elements of ProjectiveSpace(2,
81)> to <All elements of ProjectiveSpace(11, 3)>>
gap> pg1 := PG(1,9);
ProjectiveSpace(1, 9)
gap> em := NaturalEmbeddingByFieldReduction(pg1,GF(3));
<geometry morphism from <All elements of ProjectiveSpace(1,
9)> to <All elements of ProjectiveSpace(3, 3)>>
gap> i := Intertwiner(em);
MappingByFunction( The FinInG projectivity group PGL(2,9), <projective colline
ation group of size 720 with
2 generators>, function( m ) ... end, function( m ) ... end )
gap> spread := List(Points(pg1),x->x^em);
[ <a line in ProjectiveSpace(3, 3)>, <a line in ProjectiveSpace(3, 3)>,
  <a line in ProjectiveSpace(3, 3)>, <a line in ProjectiveSpace(3, 3)>,
  <a line in ProjectiveSpace(3, 3)>, <a line in ProjectiveSpace(3, 3)>,
  <a line in ProjectiveSpace(3, 3)>, <a line in ProjectiveSpace(3, 3)>,
  <a line in ProjectiveSpace(3, 3)>, <a line in ProjectiveSpace(3, 3)> ]
gap> stab := Stabilizer(CollineationGroup(PG(3,3)),Set(spread),OnSets);
<projective collineation group of size 5760 with 3 generators>
gap> hom := HomographyGroup(pg1);
The FinInG projectivity group PGL(2,9)
gap> gens := GeneratorsOfGroup(hom);;
gap> group := Group(List(gens,x->x^i));
<projective collineation group with 2 generators>
gap> Order(group);
2880
gap> IsSubgroup(stab,group);
true
```

10.5 Embeddings of polar spaces

10.5.1 NaturalEmbeddingBySubspace

▷ NaturalEmbeddingBySubspace(*geom1, geom2, v*)   (operation)
   Returns: a geometry morphism
   The arguments *geom1* and *geom2* both polar spaces, and *v* is an element of a projective space. This function returns a geometry morphism representing the natural embedding of *geom1* into the intersection of *geom2* and *v*. Hence the intersection of *geom2* and *v* must induce a polar space of the same type as *geom1* in *v*. This operation performs all necessary checks. An Intertwiner is not implemented for this geometry morphism.

```
                                  ─── Example ───
gap> h1 := HermitianPolarSpace(2, 3^2);
H(2, 3^2)
gap> h2 := HermitianPolarSpace(3, 3^2);
H(3, 3^2)
gap> pg := AmbientSpace( h2 );
ProjectiveSpace(3, 9)
gap> pi := VectorSpaceToElement( pg, [[1,0,0,0],[0,1,0,0],[0,0,1,0]] * Z(9)^0 );
<a plane in ProjectiveSpace(3, 9)>
gap> em := NaturalEmbeddingBySubspace( h1, h2, pi );
<geometry morphism from <Elements of H(2, 3^2)> to <Elements of H(3, 3^2)>>
gap> ps1 := ParabolicQuadric(4,4);
Q(4, 4)
gap> ps2 := ParabolicQuadric(6,4);
Q(6, 4)
gap> pg := AmbientSpace( ps2 );
ProjectiveSpace(6, 4)
gap> pi := VectorSpaceToElement( pg, [[1,0,0,0,0,0,0],[0,1,0,0,0,0,0],[0,0,1,0,0,0,0],
> [0,0,0,1,0,0,0],[0,0,0,0,1,0,0]] * Z(4)^0 );
<a proj. 4-space in ProjectiveSpace(6, 4)>
gap> em := NaturalEmbeddingBySubspace( ps1, ps2, pi );
<geometry morphism from <Elements of Q(4, 4)> to <Elements of Q(6, 4)>>
gap> List(Lines(ps1),x->x^em);
[ <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)>, <a line in Q(6, 4)>, <a line in Q(6, 4)>,
  <a line in Q(6, 4)> ]
```

10.5.2 NaturalEmbeddingBySubField

▷ NaturalEmbeddingBySubField(*geom1, geom2*)   (operation)

▷ NaturalEmbeddingBySubField(*geom1, geom2, boolean*)   (operation)

    Returns: a geometry morphism

    The arguments *geom1* and *geom2* are projective or polar spaces with an underlying vector space of the same dimension and the basefield L of *geom2* is an extension of the basefield K of *geom1*. The form f determining *geom1* also defines a form over L, and determines a polar space. By considering the underlying vectorspaces determining the elements of *geom1* over the extension field L, there is an obvious embedding of *geom1* in the polar space over the extension field. Considering f over a field extension might change its type. The possible embeddings, where the polar spaces may be chosen up to equivalent form, are listed in the table below (see [KL90]):

Polar Space 1	Polar Space 2	Conditions
$W(2n-1,q)$	$W(2n-1,q^a)$	–
$W(2n-1,q)$	$H(2n-1,q^2)$	–
$H(d,q^2)$	$H(d,q^{2r})$	r odd
$O^\varepsilon(d,q)$	$H(d,q^2)$	q odd
$O^\varepsilon(d,q)$	$O^{\varepsilon'}(d,q^r)$	$\varepsilon = (\varepsilon')^r$

Table: Subfield embeddings of polar spaces

    The geometry morphism also comes equipped with an intertwiner (see Intertwiner (10.1.2)). The optional third argument *boolean* can take either `true` or `false` as input, and then our operation will or will not compute the intertwiner accordingly. When set `true`, the intertwiner will be computed if `HasCollineationGroup(geom1)` is true. The user may wish that the intertwiner is not computed when embedding large polar spaces. The default (when calling the operation with two arguments) is set to `true`.

```
                              ────── Example ──────
  gap> w := SymplecticSpace(5, 3);
  W(5, 3)
  gap> h := HermitianPolarSpace(5, 3^2);
  H(5, 3^2)
  gap> em := NaturalEmbeddingBySubfield(w, h);
  #I  No intertwiner computed. <geom1> must have a collineation group computed
  <geometry morphism from <Elements of W(5, 3)> to <Elements of H(5, 3^2)>>
  gap> points := AsList(Points(w));;
  gap> image := ImagesSet(em, points);;
  gap> ForAll(image, x -> x in h);
  true
  gap> hq:=HyperbolicQuadric(3,4);
  Q+(3, 4)
  gap> eq:=EllipticQuadric(3,2);
  Q-(3, 2)
  gap> em:=NaturalEmbeddingBySubfield(eq,hq);
  #I  No intertwiner computed. <geom1> must have a collineation group computed
  <geometry morphism from <Elements of Q-(3, 2)> to <Elements of Q+(3, 4)>>
  gap> eqpts:=ImagesSet(em,AsList(Points(eq)));
```

```
  [ <a point in Q+(3, 4)>, <a point in Q+(3, 4)>, <a point in Q+(3, 4)>,
    <a point in Q+(3, 4)>, <a point in Q+(3, 4)> ]
```

10.5.3   Embedding of polar spaces by field reduction

Field reduction for polar spaces is somewhat more involved than for projective spaces, and we give a brief description. Let P be a polar space over a field $L = GF(q^t)$ and let $K = GF(q)$. Let f be the form on the r dimensional vector space V over L determining P. Consider the trace map $T : L = GF(q^t) \to GF(q) : x \mapsto x^{q^t} + x^{q^{t-1}} + \ldots x$. Define for any $\alpha \in L$ the map $T_\alpha : GF(q^t) \to GF(q) : T_\alpha(x) = T(\alpha x)$. Consider the rt dimensional vector space W over K. There is a bijective map Φ from V to W and $T_\alpha \circ f \circ \Phi^{-1}$ defines a quadratic or sesquilinear form (depending on α, and f being quadratic or sesquilinear) acting on W, and hence, if not singular or degenerate, inducing a polar space S over the finite field $GF(q)$. An element of P can be mapped onto an element of S by simply blowing up P using field reduction for projective spaces. So the resulting polar space W is dependent on the original form f, the parameter α and the blowing up of elements by field reduction, the latter being dependent on the basis of L as a K vector space. *FinInG* provides two approaches. The first approach starts from P and the parameters K, α and a basis for L as K vector space. Then the resulting form $T_\alpha \circ f \circ \Phi^{-1}$ is determined, and the associated polar space W will be the range of the embedding. Note that the resulting polar space will not necessarily be canonical. The second approach starts from two given polar spaces P and S. Based on this input, it is determined wheter an embedding based on the above described principle is possible, and the necessary parameters are computed. The resulting embedding is a geometry morhpism from P to S. Note that the polar spaces used as an argument may be freely chosen and are not required to be in the canonical form.

Fo the embeddings by field reduction of polar spaces, including conditions on the parameter α, we refer to [Gil08] and [LVdV]. The possible embeddings are listed in the following table.

Polar Space 1	Polar Space 2	Conditions
$W(2n-1,q^t)$	$W(2nt-1,q)$	–
$Q^+(2n-1,q^t)$	$Q^+(2nt-1,q)$	–
$Q^-(2n-1,q^t)$	$Q^-(2nt-1,q)$	–
$Q(2n,q^{2a+1})$	$Q((2a+1)(2n+1)-1,q)$	q odd
$Q(2n,q^{2a})$	$Q^-(2a(2n+1)-1,q)$	q=1 mod 4
$Q(2n,q^{4a+2})$	$Q^+((4a+2)(2n+1)-1,q)$	q=3 mod 4
$Q(2n,q^{4a})$	$Q^-(4a(2n+1)-1,q)$	q=3 mod 4
$H(n,q^{2a+1})$	$H((n+1)(2a+1)-1,q)$	q square
$H(n,q^{2a})$	$W(2a(n+1)-1,q)$	q even
$H(2n,q^{2a})$	$Q^-(2a(2n+1)-1,q)$	q odd
$H(2n+1,q^{2a})$	$Q^+(2a(2n+2)-1,q)$	q odd

Table: Field reduction of polar spaces

10.5.4   NaturalEmbeddingByFieldReduction

▷ NaturalEmbeddingByFieldReduction(*ps1, f2, alpha, basis, bool*)   (operation)
▷ NaturalEmbeddingByFieldReduction(*ps1, f2, alpha, basis*)   (operation)
▷ NaturalEmbeddingByFieldReduction(*ps1, f2, alpha, bool*)   (operation)
▷ NaturalEmbeddingByFieldReduction(*ps1, f2, alpha*)   (operation)

▷ NaturalEmbeddingByFieldReduction(*ps1, f2, bool*)   (operation)
▷ NaturalEmbeddingByFieldReduction(*ps1, f2*)   (operation)

   Returns: a geometry morphism

   ps1 is a polar space over a field extension L of *f2*, *basis* is a basis for L over *f2*, *alpha* is a non-zero element of L. The version of NaturalEmbeddingByFieldReduction implements the first approach as explained in 10.5.3. When no argument *basis* is given, a basis for L over *f2* is computed using Basis(AsVectorSpace(K,L)). When no argument *alpha* is given, One(f2) is used as value for *alpha*. When *bool* is true or not given, an intertwiner is computed, when *bool* is *false*, no intertwiner is computed. This intertwiner has as its domain the ISOMETRY GROUP of *ps1*. The user may wish that the intertwiner is not computed when embedding large polar spaces. The default (when calling the operation with two arguments) is set to true. In the first example, we construct a spread of maximal subspaces (solids) in a 7 dimensional symplectic space. We compute a subgroup of its stabilizer group using the intertwiner. In the second example, we construct a linear blocking set of the symplectic generalised quadrangle over $GF(9)$.

```
───────────────────────────── Example ─────────────────────────────
gap> ps1 := SymplecticSpace(1,3^3);
W(1, 27)
gap> em := NaturalEmbeddingByFieldReduction(ps1,GF(3),true);
<geometry morphism from <Elements of W(1,
27)> to <Elements of <polar space in ProjectiveSpace(
5,GF(3)): -x1*y6-x2*y5-x3*y4-x3*y6+x4*y3+x5*y2+x6*y1+x6*y3=0 >>>
gap> ps2 := AmbientGeometry(Range(em));
<polar space in ProjectiveSpace(
5,GF(3)): -x1*y6-x2*y5-x3*y4-x3*y6+x4*y3+x5*y2+x6*y1+x6*y3=0 >
gap> spread := List(Points(ps1),x->x^em);;
gap> i := Intertwiner(em);
MappingByFunction( PSp(2,27), <projective collineation group of size
9828 with 2 generators>, function( m ) ... end, function( m ) ... end )
gap> coll := CollineationGroup(ps2);
#I  Computing collineation group of canonical polar space...
<projective collineation group of size 9170703360 with 4 generators>
gap> stab := Group(ImagesSet(i,GeneratorsOfGroup(IsometryGroup(ps1))));
<projective collineation group with 2 generators>
gap> IsSubgroup(coll,stab);
true
gap> List(Orbit(stab,spread[1]),x->x in spread);
[ true, true, true, true, true, true, true, true, true, true, true, true,
  true, true, true, true, true, true, true, true, true, true, true,
  true, true, true, true ]

gap> ps1 := SymplecticSpace(3,9);
W(3, 9)
gap> em := NaturalEmbeddingByFieldReduction(ps1,GF(3),true);
<geometry morphism from <Elements of W(3,
9)> to <Elements of <polar space in ProjectiveSpace(
7
 ,GF(3)): -x1*y3+x1*y4+x2*y3+x3*y1-x3*y2-x4*y1-x5*y7+x5*y8+x6*y7+x7*y5-x7*y6-x8
*y5=0 >>>
gap> ps2 := AmbientGeometry(Range(em));
<polar space in ProjectiveSpace(
7
```

```
   ,GF(3)): -x1*y3+x1*y4+x2*y3+x3*y1-x3*y2-x4*y1-x5*y7+x5*y8+x6*y7+x7*y5-x7*y6-x8
*y5=0 >
gap> pg := AmbientSpace(ps2);
ProjectiveSpace(7, 3)
gap> spread := List(Points(ps1),x->x^em);;
gap> el := Random(ElementsOfIncidenceStructure(pg,5));
<a proj. 4-space in ProjectiveSpace(7, 3)>
gap> prebs := Filtered(spread,x->Meet(x,el) <> EmptySubspace(pg));;
gap> bs := List(prebs,x->PreImageElm(em,x));;
gap> Length(bs);
118
gap> lines := List(Lines(ps1));;
gap> Collected(List(lines,x->Length(Filtered(bs,y->y * x))));
[ [ 1, 702 ], [ 4, 117 ], [ 10, 1 ] ]
```

10.5.5 NaturalEmbeddingByFieldReduction

▷ NaturalEmbeddingByFieldReduction(*ps1, ps2, bool*)   (operation)
▷ NaturalEmbeddingByFieldReduction(*ps1, ps2*)   (operation)

Returns: a geometry morphism

If *ps1* and *ps2* are two polar spaces which are suitable for field reduction as listed in the table with possible embeddings in Section 10.5.3, then this operation returns the corresponding embedding. An intertwiner is computed if the third argument *bool* is true, or if there is no third argument. This intertwiner has as its domain the ISOMETRY GROUP of *ps1*. The example shows two cases where a spread is computed, including a subgroup of its stabiliser group using the intertwiner.

```
───────────────────────────── Example ─────────────────────────────
gap> ps1 := SymplecticSpace(1,5^3);
W(1, 125)
gap> ps2 := SymplecticSpace(5,5);
W(5, 5)
gap> em := NaturalEmbeddingByFieldReduction(ps1,ps2);
#I  These polar spaces are suitable for field reduction
<geometry morphism from <Elements of W(1, 125)> to <Elements of W(5, 5)>>
gap> pts := Points(ps1);
<points of W(1, 125)>
gap> spread := List(pts,x->x^em);;
gap> test := Union(List(spread,x->List(Points(x))));;
gap> Set(test)=Set(AsList(Points(ps2)));
true
gap> hom := Intertwiner(em);
MappingByFunction( PSp(2,125), <projective collineation group of size
976500 with 2 generators>, function( m ) ... end, function( m ) ... end )
gap> group := IsometryGroup(ps1);
PSp(2,125)
gap> Order(group);
976500
gap> gens := List(GeneratorsOfGroup(group),x->x^hom);
[ < a collineation: <cmat 6x6 over GF(5,1)>, F^0>,
  < a collineation: <cmat 6x6 over GF(5,1)>, F^0> ]
gap> group2 := Range(hom);
```

```
<projective collineation group of size 976500 with 2 generators>
gap> Order(group2);
976500
gap> biggroup := CollineationGroup(ps2);
PGammaSp(6,5)
gap> stab := FiningSetwiseStabiliser(biggroup,spread);
#I  Computing adjusted stabilizer chain...
<projective collineation group with 7 generators>
gap> time;
6907
gap> Order(stab);
5859000
gap> ps1 := HermitianPolarSpace(2,7^2);
H(2, 7^2)
gap> ps2 := EllipticQuadric(5,7);
Q-(5, 7)
gap> em := NaturalEmbeddingByFieldReduction(ps1,ps2);
#I  These polar spaces are suitable for field reduction
<geometry morphism from <Elements of H(2, 7^2)> to <Elements of Q-(5, 7)>>
gap> pts := Points(ps1);
<points of H(2, 7^2)>
gap> spread := List(pts,x->x^em);;
gap> test := Union(List(spread,x->List(Points(x))));;
gap> Set(test)=Set(AsList(Points(ps2)));
true
gap> hom := Intertwiner(em);
MappingByFunction( PGU(3,7^2), <projective collineation group of size
5663616 with 2 generators>, function( m ) ... end, function( m ) ... end )
gap> group := IsometryGroup(ps1);
PGU(3,7^2)
gap> Order(group);
5663616
gap> gens := List(GeneratorsOfGroup(group),x->x^hom);
[ < a collineation: <cmat 6x6 over GF(7,1)>, F^0>,
  < a collineation: <cmat 6x6 over GF(7,1)>, F^0> ]
gap> group2 := Range(hom);
<projective collineation group of size 5663616 with 2 generators>
gap> Order(group2);
5663616
gap> biggroup := CollineationGroup(ps2);
PDeltaO-(6,7)
gap> stab := FiningSetwiseStabiliser(biggroup,spread);
#I  Computing adjusted stabilizer chain...
<projective collineation group with 10 generators>
gap> time;
3438
gap> Order(stab);
90617856
```

10.6   Projections

10.6.1   NaturalProjectionBySubspace

▷ NaturalProjectionBySubspace(*ps*, *v*)   (operation)

Returns: a geometry morphism

The argument *ps* is a projective or polar space, and *v* is a subspace of *ps*. In the case that *ps* is a projective space, the geometry of subspaces containing *v* is a projective space of lower dimension over the same basefield, and this operation returns the corresponding geometry morphism. In the case that *ps* is a polar space, the geometry of elements pf *ps* containing *v* is a polar space of lower rank and of the same type over the same basefield, and this operation returns the corresponding geometry morphism. It is checked whether *v* is a subspace of *ps*, and whether the input of the function and preimage of the returned geometry morphism is valid or not. There is a shorthand for this operation which is basically an overload of the quotient operation. So, for example, ps / v achieves the same thing as AmbientGeometry(Range(NaturalProjectionBySubspace(ps, v))). An intertwiner is not available for this geometry morphism.

```
_____ Example _____
gap> ps := HyperbolicQuadric(5,3);
Q+(5, 3)
gap> x := Random(Points(ps));;
gap> planes_on_x := AsList( Planes(x) );
[ <a plane in Q+(5, 3)>, <a plane in Q+(5, 3)>, <a plane in Q+(5, 3)>,
  <a plane in Q+(5, 3)>, <a plane in Q+(5, 3)>, <a plane in Q+(5, 3)>,
  <a plane in Q+(5, 3)>, <a plane in Q+(5, 3)> ]
gap> proj := NaturalProjectionBySubspace(ps, x);
<geometry morphism from <Elements of Q+(5,
3)> to <Elements of <polar space in ProjectiveSpace(
3,GF(3)): x_1*x_2+x_3*x_4=0 >>>
gap> image := ImagesSet(proj, planes_on_x);
[ <a line in Q+(3, 3): x_1*x_2+x_3*x_4=0>,
  <a line in Q+(3, 3): x_1*x_2+x_3*x_4=0>,
  <a line in Q+(3, 3): x_1*x_2+x_3*x_4=0>,
  <a line in Q+(3, 3): x_1*x_2+x_3*x_4=0>,
  <a line in Q+(3, 3): x_1*x_2+x_3*x_4=0>,
  <a line in Q+(3, 3): x_1*x_2+x_3*x_4=0>,
  <a line in Q+(3, 3): x_1*x_2+x_3*x_4=0>,
  <a line in Q+(3, 3): x_1*x_2+x_3*x_4=0> ]
```

10.7   Projective completion

10.7.1   ProjectiveCompletion

▷ ProjectiveCompletion(*as*)   (operation)

Returns: a geometry morphism

The argument *as* is an affine space. This operation returns an embedding of *as* into the projective space *ps* of the same dimension, and over the same field. For example, the point (x, y, z) is mapped onto the projective point with homogeneous coordinates $(1, x, y, z)$. An intertwiner is unnecessary, CollineationGroup(as) is a subgroup of CollineationGroup(ps).

―――――――― Example ――――――――
```
gap> as := AffineSpace(3,5);
AG(3, 5)
gap> map := ProjectiveCompletion(as);
<geometry morphism from <Elements of AG(3,
5)> to <All elements of ProjectiveSpace(3, 5)>>
gap> p := Random( Points(as) );
<a point in AG(3, 5)>
gap> p^map;
<a point in ProjectiveSpace(3, 5)>
```

Chapter 11

Algebraic Varieties

In FinInG we provide some basic functionality for algebraic varieties defined over finite fields. The algebraic varieties in FinInG are defined by a list of multivariate polynomials over a finite field, and an ambient geometry. This ambient geometry is either a projective space, and then the algebraic variety is called a *projective variety*, or an affine geometry, and then the algebraic variety is called an *affine variety*. In this chapter we give a brief overview of the features of FinInG concerning these two types of algebraic varieties. The package FinInG also contains the Veronese varieties VeroneseVariety (11.7.1), the Segre varieties SegreVariety (11.6.1) and the Grassmann varieties GrassmannVariety (11.8.1); three classical projective varieties. These varieties have an associated *geometry map* (the VeroneseMap (11.7.3), SegreMap (11.6.3) and GrassmannMap (11.8.3)) and FinInG also provides some general functionality for these.

11.1 Algebraic Varieties

An *algebraic variety* in FinInG is an algebraic variety in a projective space or affine space, defined by a list of polynomials over a finite field.

11.1.1 AlgebraicVariety

▷ AlgebraicVariety(*space, pring, pollist*)   (operation)
▷ AlgebraicVariety(*space, pollist*)   (operation)
    Returns: an algebraic variety

The argument *space* is an affine or projective space over a finite field F, the argument *pring* is a multivariate polynomial ring defined over (a subfield of) F, and *pollist* is a list of polynomials in *pring*. If the *space* is a projective space, then *pollist* needs to be a list of homogeneous polynomials. In FinInG there are two types of projective varieties: projective varieties and affine varieties. The following operations apply to both types.

```
──────────────────── Example ────────────────────
gap> F:=GF(9);
GF(3^2)
gap> r:=PolynomialRing(F,4);
GF(3^2)[x_1,x_2,x_3,x_4]
gap> pg:=PG(3,9);
ProjectiveSpace(3, 9)
gap> f1:=r.1*r.3-r.2^2;
x_1*x_3-x_2^2
```

```
gap> f2:=r.4*r.1^2-r.4^3;
x_1^2*x_4-x_4^3
gap> var:=AlgebraicVariety(pg,[f1,f2]);
Projective Variety in ProjectiveSpace(3, 9)
gap> DefiningListOfPolynomials(var);
[ x_1*x_3-x_2^2, x_1^2*x_4-x_4^3 ]
gap> AmbientSpace(var);
ProjectiveSpace(3, 9)
```

11.1.2 DefiningListOfPolynomials

▷ DefiningListOfPolynomials(`var`)   (attribute)

   Returns: a list of polynomials

   The argument `var` is an algebraic variety. This attribute returns the list of polynomials that was used to define the variety `var`.

11.1.3 AmbientSpace

▷ AmbientSpace(`var`)   (attribute)

   Returns: an affine or projective space

   The argument `var` is an algebraic variety. This attribute returns the affine or projective space in which the variety `var` was defined.

11.1.4 PointsOfAlgebraicVariety

▷ PointsOfAlgebraicVariety(`var`)   (operation)
▷ Points(`var`)   (operation)

   Returns: a list of points

   The argument `var` is an algebraic variety. This operation returns the list of points of the AmbientSpace (11.1.3) of the algebraic variety `var` whose coordinates satify the DefiningListOfPolynomials (11.1.2) of the algebraic variety `var`.

```
────────────────────── Example ──────────────────────
gap> F:=GF(9);
GF(3^2)
gap> r:=PolynomialRing(F,4);
GF(3^2)[x_1,x_2,x_3,x_4]
gap> pg:=PG(3,9);
ProjectiveSpace(3, 9)
gap> f1:=r.1*r.3-r.2^2;
x_1*x_3-x_2^2
gap> f2:=r.4*r.1^2-r.4^3;
x_1^2*x_4-x_4^3
gap> var:=AlgebraicVariety(pg,[f1,f2]);
Projective Variety in ProjectiveSpace(3, 9)
gap> points:=Points(var);
<points of Projective Variety in ProjectiveSpace(3, 9)>
gap> Size(points);
28
gap> iter := Iterator(points);
```

```
    <iterator>
gap> for i in [1..4] do
>          x := NextIterator(iter);
>          Display(x);
> od;
[1...]
[1..1]
[1..2]
[111.]
```

11.1.5 Iterator

▷ Iterator(*pts*) (operation)

   Returns: an iterator

   The argument *pts* is the set of `PointsOfAlgebraicVariety` (11.1.4) of an algebraic variety *var*. This operation returns an iterator for the points of an algebraic variety.

11.1.6 \in

▷ \in(*x*, *var*) (operation)
▷ \in(*x*, *pts*) (operation)

   Returns: true or false

   The argument *x* is a point of the `AmbientSpace` (11.1.3) of an algebraic variety `AlgebraicVariety` (11.4.1). This operation also works for a point *x* and the collection *pts* returned by `PointsOfAlgebraicVariety` (11.1.4).

11.2 Projective Varieties

A *projective variety* in FinInG is an algebraic variety in a projective space defined by a list of homogeneous polynomials over a finite field.

11.2.1 ProjectiveVariety

▷ ProjectiveVariety(*pg*, *pring*, *pollist*) (operation)
▷ ProjectiveVariety(*pg*, *pollist*) (operation)
▷ AlgebraicVariety(*pg*, *pring*, *pollist*) (operation)
▷ AlgebraicVariety(*pg*, *pollist*) (operation)

   Returns: a projective algebraic variety

———————————————— Example ————————————————

```
gap> F:=GF(9);
GF(3^2)
gap> r:=PolynomialRing(F,4);
GF(3^2)[x_1,x_2,x_3,x_4]
gap> pg:=PG(3,9);
ProjectiveSpace(3, 9)
gap> f1:=r.1*r.3-r.2^2;
x_1*x_3-x_2^2
gap> f2:=r.4*r.1^2-r.4^3;
```

```
x_1^2*x_4-x_4^3
gap> var:=AlgebraicVariety(pg,[f1,f2]);
Projective Variety in ProjectiveSpace(3, 9)
gap> DefiningListOfPolynomials(var);
[ x_1*x_3-x_2^2, x_1^2*x_4-x_4^3 ]
gap> AmbientSpace(var);
ProjectiveSpace(3, 9)
```

11.3   Quadrics and Hermitian varieties

Quadrics (QuadraticVariety (11.3.2)) and Hermitian varieties (HermitianVariety (11.3.1)) are projective varieties that have the associated quadratic or hermitian form as an extra attribute installed. Furthermore, we provide a method for PolarSpace taking as an argument a projective algebraic variety.

11.3.1   HermitianVariety

▷ HermitianVariety(*pg*, *pring*, *pol*)   (operation)
▷ HermitianVariety(*pg*, *pol*)   (operation)
▷ HermitianVariety(*n*, *F*)   (operation)
▷ HermitianVariety(*n*, *q*)   (operation)

Returns: a hermitian variety in a projective space

For the first two methods, the argument *pg* is a projective space, *pring* is a polynomial ring, and *pol* is polynomial. For the third and fourth variations, the argument *n* is an integer, the argument *F* is a finite field, and the argument *q* is a prime power. These variations of the operation return the hermitian variety associated to the standard hermitian form in the projective space of dimension *n* over the field *F* of order *q*.

```
———————————————————— Example ————————————————————
gap> F:=GF(25);
GF(5^2)
gap> r:=PolynomialRing(F,3);
GF(5^2)[x_1,x_2,x_3]
gap> x:=IndeterminatesOfPolynomialRing(r);
[ x_1, x_2, x_3 ]
gap> pg:=PG(2,F);
ProjectiveSpace(2, 25)
gap> f:=x[1]^6+x[2]^6+x[3]^6;
x_1^6+x_2^6+x_3^6
gap> hv:=HermitianVariety(pg,f);
Hermitian Variety in ProjectiveSpace(2, 25)
gap> AsSet(List(Lines(pg),l->Size(Filtered(Points(l),x->x in hv))));
[ 1, 6 ]
gap> hv:=HermitianVariety(5,4);
Hermitian Variety in ProjectiveSpace(5, 4)
gap> hps:=PolarSpace(hv);
<polar space in ProjectiveSpace(
5,GF(2^2)): x_1^3+x_2^3+x_3^3+x_4^3+x_5^3+x_6^3=0 >
gap> hf:=SesquilinearForm(hv);
```

```
  < hermitian form >
gap> PolynomialOfForm(hf);
x_1^3+x_2^3+x_3^3+x_4^3+x_5^3+x_6^3
```

11.3.2 QuadraticVariety

▷ QuadraticVariety(*pg*, *pring*, *pol*)   (operation)
▷ QuadraticVariety(*pg*, *pol*)   (operation)
▷ QuadraticVariety(*n*, *F*, *type*)   (operation)
▷ QuadraticVariety(*n*, *q*, *type*)   (operation)
▷ QuadraticVariety(*n*, *F*)   (operation)
▷ QuadraticVariety(*n*, *q*)   (operation)

    Returns: a quadratic variety in a projective space

In the first two methods, the argument *pg* is a projective space, *pring* is a polynomial ring, and *pol* is a polynomial. The latter four return a standard non-degenerate quadric. The argument *n* is a projective dimension, *F* is a field, and *q* is a prime power that gives just the order of the defining field. If the *type* is given, then it will return a quadric of a particular type as follows:

variety	standard form	characteristic p
hyperbolic quadric	$X_0X_1 + \ldots + X_{n-1}X_n$	$p \equiv 3 \bmod 4$ and p even
hyperbolic quadric	$2(X_0X_1 + \ldots + X_{n-1}X_n)$	$p \equiv 1 \bmod 4$
parabolic quadric	$X_0^2 + X_1X_2 + \ldots + X_{n-1}X_n$	$p \equiv 1,3 \bmod 8$ and p even
parabolic quadric	$t(X_0^2 + X_1X_2 + \ldots + X_{n-1}X_n)$, t a primitive element of $GF(p)$	$p \equiv 5,7 \bmod 8$
elliptic quadric	$X_0^2 + X_1^2 + X_2X_3 + \ldots + X_{n-1}X_n$	$p \equiv 3 \bmod 4$
elliptic quadric	$X_0^2 + tX_1^2 + X_2X_3 + \ldots + X_{n-1}X_n$, t a primitive element of $GF(p)$	odd
elliptic quadric	$X_0^2 + X_0X_1 + dX_1^2 + X_2X_3 + \ldots + X_{n-1}X_n$, $Tr(d) = 1$	even

Table: standard quadratic varieties

If no type is given, and only the dimension and field/field order are given, then it is assumed that the dimension is even and the user wants a standard parabolic quadric.

```
──────────────────────────── Example ────────────────────────────
gap> F:=GF(5);
GF(5)
gap> r:=PolynomialRing(F,4);
GF(5)[x_1,x_2,x_3,x_4]
gap> x:=IndeterminatesOfPolynomialRing(r);
[ x_1, x_2, x_3, x_4 ]
gap> pg:=PG(3,F);
ProjectiveSpace(3, 5)
gap> Q:=x[2]*x[3]+x[4]^2;
x_2*x_3+x_4^2
gap> qv:=QuadraticVariety(pg,Q);
Quadratic Variety in ProjectiveSpace(3, 5)
gap> AsSet(List(Planes(pg),z->Size(Filtered(Points(z),x->x in qv))));
[ 1, 6, 11 ]
gap> qf:=QuadraticForm(qv);
< quadratic form >
gap> Display(qf);
```

```
  Quadratic form
  Gram Matrix:
   . . . .
   . . 1 .
   . . . .
   . . . 1
  Polynomial: [ [  x_2*x_3+x_4^2 ] ]
gap> IsDegenerateForm(qf);
  #I  Testing degeneracy of the *associated bilinear form*
  true
gap> qv:=QuadraticVariety(3,F,"-");
  Quadratic Variety in ProjectiveSpace(3, 5)
gap> PolarSpace(qv);
  <polar space in ProjectiveSpace(3,GF(5)): x_1^2+Z(5)*x_2^2+x_3*x_4=0 >
gap> Display(last);
  <polar space of rank 3 over GF(5)>
  Non-singular elliptic quadratic form
  Gram Matrix:
   1 . . .
   . 2 . .
   . . . 1
   . . . .
  Polynomial: [ [  x_1^2+Z(5)*x_2^2+x_3*x_4 ] ]
  Witt Index: 1
  Bilinear form
  Gram Matrix:
   2 . . .
   . 4 . .
   . . . 1
   . . 1 .
gap> qv:=QuadraticVariety(3,F,"+");
  Quadratic Variety in ProjectiveSpace(3, 5)
gap> Display(last);
  Quadratic Variety in ProjectiveSpace(3, 5)
   Polynomial: [ Z(5)*x_1*x_2+Z(5)*x_3*x_4 ]
```

11.3.3 QuadraticForm

▷ QuadraticForm(*var*)   (attribute)

Returns: a quadratic form

When the argument *var* is a QuadraticVariety (11.3.2), this returns the associated quadratic form.

11.3.4 SesquilinearForm

▷ SesquilinearForm(*var*)   (attribute)

Returns: a hermitian form

If the argument *var* is a HermitianVariety (11.3.1), this returns the associated hermitian form.

11.3.5 PolarSpace

▷ PolarSpace(var)   (operation)

the argument `var` is a projective algebraic variety. When its list of defining polynomial contains exactly one polynomial, depending on its degree, the operation `QuadraticFormByPolynomial` or `HermitianFormByPolynomial` is used to compute a quadratic form or a hermitian form. These operations check whether this is possible, and produce an error message if not. If the conversion is possible, then the appropriate polar space is returned.

```
───────────────────────────── Example ─────────────────────────────
 gap> f := GF(25);
 GF(5^2)
 gap> r := PolynomialRing(f,4);
 GF(5^2)[x_1,x_2,x_3,x_4]
 gap> ind := IndeterminatesOfPolynomialRing(r);
 [ x_1, x_2, x_3, x_4 ]
 gap> eq1 := Sum(List(ind,t->t^2));
 x_1^2+x_2^2+x_3^2+x_4^2
 gap> var := ProjectiveVariety(PG(3,f),[eq1]);
 Projective Variety in ProjectiveSpace(3, 25)
 gap> PolarSpace(var);
 <polar space in ProjectiveSpace(3,GF(5^2)): x_1^2+x_2^2+x_3^2+x_4^2=0 >
 gap> eq2 := Sum(List(ind,t->t^4));
 x_1^4+x_2^4+x_3^4+x_4^4
 gap> var := ProjectiveVariety(PG(3,f),[eq2]);
 Projective Variety in ProjectiveSpace(3, 25)
 gap> PolarSpace(var);
 Error, <poly> does not generate a Hermitian matrix called from
 GramMatrixByPolynomialForHermitianForm( pol, gf, n, vars ) called from
 HermitianFormByPolynomial( pol, pring, n ) called from
 HermitianFormByPolynomial( eq, r ) called from
 <function "unknown">( <arguments> )
  called from read-eval loop at line 16 of *stdin*
 you can 'quit;' to quit to outer loop, or
 you can 'return;' to continue
 brk> quit;
 gap> eq3 := Sum(List(ind,t->t^6));
 x_1^6+x_2^6+x_3^6+x_4^6
 gap> var := ProjectiveVariety(PG(3,f),[eq3]);
 Projective Variety in ProjectiveSpace(3, 25)
 gap> PolarSpace(var);
 <polar space in ProjectiveSpace(3,GF(5^2)): x_1^6+x_2^6+x_3^6+x_4^6=0 >
```

11.4 Affine Varieties

An *affine variety* in FinInG is an algebraic variety in an affine space defined by a list of polynomials over a finite field.

11.4.1 AffineVariety

▷ AffineVariety(*ag, pring, pollist*)   (operation)
▷ AffineVariety(*ag, pollist*)   (operation)
▷ AlgebraicVariety(*ag, pring, pollist*)   (operation)
▷ AlgebraicVariety(*ag, pollist*)   (operation)
    Returns: an affine algebraic variety

The argument `ag` is an affine space over a finite field `F`, the argument `pring` is a multivariate polynomial ring defined over (a subfield of) `F`, and `pollist` is a list of polynomials in `pring`.

11.5   Geometry maps

A `geometry map` is a map from a set of elements of a geometry to a set of elements of another geometry, which is not necessarily a geometry morphism. Examples are the SegreMap (11.6.3), the VeroneseMap (11.7.3), and the GrassmannMap (11.8.3).

11.5.1 Source

▷ Source(*gm*)   (operation)
    Returns: the source of a geometry map
    The argument `gm` is a geometry map.

11.5.2 Range

▷ Range(*gm*)   (operation)
    Returns: the range of a geometry map
    The argument `gm` is a geometry map.

11.5.3 ImageElm

▷ ImageElm(*gm, x*)   (operation)
    Returns: the image of an element under a geometry map
    The argument `gm` is a geometry map, the element `x` is an element of the Source (11.8.4) of the geometry map `gm`.

11.5.4 ImagesSet

▷ ImagesSet(*gm, elms*)   (operation)
    Returns: the image of a subset of the source under a geometry map
    The argument `gm` is a geometry map, the elements `elms` is a subset of the Source (11.8.4) of the geometry map `gm`.

11.5.5 \^

▷ \^(*x, gm*)   (operation)
    Returns: the image of an element of the source under a geometry map
    The argument `gm` is a geometry map, the element `x` is an element of the Source (11.8.4) of the geometry map `gm`.

11.6   Segre Varieties

A *Segre variety* in FinInG is a projective algebraic variety in a projective space over a finite field. The set of points that lie on this variety is the image of the *Segre map*.

11.6.1   SegreVariety

▷ SegreVariety(*listofpgs*)   (operation)
▷ SegreVariety(*listofdims, field*)   (operation)
▷ SegreVariety(*pg1, pg2*)   (operation)
▷ SegreVariety(*d1, d2, field*)   (operation)
▷ SegreVariety(*d1, d2, q*)   (operation)
    Returns: a Segre variety

The argument *listofpgs* is a list of projective spaces defined over the same finite field, say $[PG(n_1-1,q) , PG(n_2-1,q) , ..., PG(n_k-1,q)]$. The operation also takes as input the list of dimensions (*listofdims*) and a finite field *field* (e.g. $[n_1,n_2,\ldots,n_k,GF(q)]$). A Segre variety with only two factors ($k = 2$), can also be constructed using the operation with two projective spaces *pg1* and *pg2* as arguments, or with two dimensions *d1*, *d2*, and a finite field *field*(or a prime power *q*). The operation returns a projective algebraic variety in the projective space of dimension $n_1 n_2 \ldots n_k - 1$.

11.6.2   PointsOfSegreVariety

▷ PointsOfSegreVariety(*sv*)   (operation)
▷ Points(*sv*)   (operation)
    Returns: the points of a Segre variety

The argument *sv* is a Segre variety. This operation returns a set of points of the AmbientSpace (11.1.3) of the Segre variety. This set of points corresponds to the image of the SegreMap (11.6.3).

11.6.3   SegreMap

▷ SegreMap(*listofpgs*)   (operation)
▷ SegreMap(*listofdims, field*)   (operation)
▷ SegreMap(*pg1, pg2*)   (operation)
▷ SegreMap(*d1, d2, field*)   (operation)
▷ SegreMap(*d1, d2, q*)   (operation)
▷ SegreMap(*sv*)   (operation)
    Returns: a geometry map

The argument *listofpgs* is a list of projective spaces defined over the same finite field, say $[PG(n_1-1,q) , PG(n_2-1,q) , ..., PG(n_k-1,q)]$. The operation also takes as input the list of dimensions (*listofdims*) and a finite field *field* (e.g. $[n_1,n_2,\ldots,n_k,GF(q)]$). A Segre map with only two factors ($k = 2$), can also be constructed using the operation with two projective spaces *pg1* and *pg2* as arguments, or with two dimensions *d1*, *d2*, and a finite field *field*(or a prime power *q*). The operation returns a function with domain the product of the point sets of projective spaces in the list $[PG(n_1-1,q) , PG(n_2-1,q) , ..., PG(n_k-1,q)]$ and image the set of points of the Segre variety (PointsOfSegreVariety (11.6.2)) in the projective space of dimension $n_1 n_2 \ldots n_k - 1$. When a Segre variety *sv* is given as input, the operation returns the associated Segre map.

```
 ─────────────────────────── Example ───────────────────────────
  gap> sv:=SegreVariety(2,2,9);
  Segre Variety in ProjectiveSpace(8, 9)
  gap> sm:=SegreMap(sv);
  Segre Map of [ <points of ProjectiveSpace(2, 9)>,
    <points of ProjectiveSpace(2, 9)> ]
  gap> cart1:=Cartesian(Points(PG(2,9)),Points(PG(2,9)));;
  gap> im1:=ImagesSet(sm,cart1);;
  gap> Span(im1);
  ProjectiveSpace(8, 9)
  gap> l:=Random(Lines(PG(2,9)));
  <a line in ProjectiveSpace(2, 9)>
  gap> cart2:=Cartesian(Points(l),Points(PG(2,9)));;
  gap> im2:=ImagesSet(sm,cart2);;
  gap> Span(im2);
  <a proj. 5-space in ProjectiveSpace(8, 9)>
  gap> x:=Random(Points(PG(2,9)));
  <a point in ProjectiveSpace(2, 9)>
  gap> cart3:=Cartesian(Points(PG(2,9)),Points(x));;
  gap> im3:=ImagesSet(sm,cart3);;
  gap> pi:=Span(im3);
  <a plane in ProjectiveSpace(8, 9)>
  gap> AsSet(List(Points(pi),y->y in sv));
  [ true ]
```

11.6.4 Source

▷ Source(*sm*)   (operation)

Returns: the source of a Segre map

The argument *sm* is a SegreMap (11.6.3). This operation returns the cartesian product of the list consisting of the pointsets of the projective spaces that were used to construct the SegreMap (11.6.3).

11.7 Veronese Varieties

A *Veronese variety* in FinInG is a projective algebraic variety in a projective space over a finite field. The set of points that lie on this variety is the image of the *Veronese map*.

11.7.1 VeroneseVariety

▷ VeroneseVariety(*pg*)   (operation)
▷ VeroneseVariety(*n-1, field*)   (operation)
▷ VeroneseVariety(*n-1, q*)   (operation)

Returns: a Veronese variety

The argument *pg* is a projective space defined over a finite field, say $PG(n-1,q)$. The operation also takes as input the dimension and a finite field *field* (e.g. $[n-1,q]$). The operation returns a projective algebraic variety in the projective space of dimension $(n^2+n)/2-1$, known as the (quadratic) Veronese variety. It is the image of the map $(x_0,x_1,\ldots,x_n) \mapsto (x_0^2,x_0x_1,\ldots,x_0x_n,x_1^2,x_1x_2,\ldots,x_1x_n,\ldots,x_n^2)$

11.7.2 PointsOfVeroneseVariety

▷ PointsOfVeroneseVariety(*vv*)   (operation)

▷ Points(*vv*)   (operation)

    Returns: the points of a Veronese variety

    The argument *vv* is a Veronese variety. This operation returns a set of points of the `AmbientSpace` (11.1.3) of the Veronese variety. This set of points corresponds to the image of the `VeroneseMap` (11.7.3).

11.7.3 VeroneseMap

▷ VeroneseMap(*pg*)   (operation)

▷ VeroneseMap(*n-1, field*)   (operation)

▷ VeroneseMap(*n-1, q*)   (operation)

▷ VeroneseMap(*vv*)   (operation)

    Returns: a geometry map

    The argument *pg* is a projective space defined over a finite field, say $PG(n-1, q)$. The operation also takes as input the dimension and a finite field `field` (e.g. $[n-1, q]$). The operation returns a function with domain the product of the point set of the projective space $PG(n-1, q)$ and image the set of points of the Veronese variety (`PointsOfVeroneseVariety` (11.7.2)) in the projective space of dimension $(n^2+n)/2 - 1$. When a Veronese variety *vv* is given as input, the operation returns the associated Veronese map.

```
---------------------------------- Example ----------------------------------
gap> pg:=PG(2,5);
ProjectiveSpace(2, 5)
gap> vv:=VeroneseVariety(pg);
Veronese Variety in ProjectiveSpace(5, 5)
gap> Size(Points(vv))=Size(Points(pg));
true
gap> vm:=VeroneseMap(vv);
Veronese Map of <points of ProjectiveSpace(2, 5)>
gap> r:=PolynomialRing(GF(5),3);
GF(5)[x_1,x_2,x_3]
gap> f:=r.1^2-r.2*r.3;
x_1^2-x_2*x_3
gap> c:=AlgebraicVariety(pg,r,[f]);
Projective Variety in ProjectiveSpace(2, 5)
gap> pts:=List(Points(c));
[ <a point in ProjectiveSpace(2, 5)>, <a point in ProjectiveSpace(2, 5)>,
  <a point in ProjectiveSpace(2, 5)>, <a point in ProjectiveSpace(2, 5)>,
  <a point in ProjectiveSpace(2, 5)>, <a point in ProjectiveSpace(2, 5)> ]
gap> Dimension(Span(ImagesSet(vm,pts)));
4
```

11.7.4 Source

▷ Source(*vm*)   (operation)

    Returns: the source of a Veronese map

The argument *vm* is a `VeroneseMap` (11.7.3). This operation returns the pointset of the projective space that was used to construct the `VeroneseMap` (11.7.3).

11.8 Grassmann Varieties

A *Grassmann variety* in FinInG is a projective algebraic variety in a projective space over a finite field. The set of points that lie on this variety is the image of the *Grassmann map*.

11.8.1 GrassmannVariety

▷ GrassmannVariety(*k, pg*)   (operation)
▷ GrassmannVariety(*subspaces*)   (operation)
▷ GrassmannVariety(*k, n, q*)   (operation)
  Returns: a Grassmann variety
  The argument *pg* is a projective space defined over a finite field, say $PG(n,q)$, and argument *k* is an integer (*k* at least 1 and at most $n-2$) and denotes the projective dimension determining the Grassmann Variety. The operation also takes as input the set *subspaces* of subspaces of a projective space, or the dimension *k*, the dimension *n* and the size *q* of the finite field (*k* at least 1 and at most $n-2$). The operation returns a projective algebraic variety known as the Grassmann variety.

11.8.2 PointsOfGrassmannVariety

▷ PointsOfGrassmannVariety(*gv*)   (operation)
▷ Points(*gv*)   (operation)
  Returns: the points of a Grassmann variety
  The argument *gv* is a Grassmann variety. This operation returns a set of points of the `AmbientSpace` (11.1.3) of the Grassmann variety. This set of points corresponds to the image of the `GrassmannMap` (11.8.3).

11.8.3 GrassmannMap

▷ GrassmannMap(*k, pg*)   (operation)
▷ GrassmannMap(*subspaces*)   (operation)
▷ GrassmannMap(*k, n, q*)   (operation)
▷ GrassmannMap(*gv*)   (operation)
  Returns: a geometry map
  The argument *pg* is a projective space defined over a finite field, say $PG(n,q)$, and argument *k* is an integer (*k* at least 1 and at most $n-2$), and denotes the projective dimension determining the Grassmann Variety. The operation also takes as input the set *subspaces* of subspaces of a projective space, or the dimension *k*, the dimension *n* and the size *q* of the finite field (*k* at least 1 and at most $n-2$). The operation returns a function with domain the set of subspaces of dimension *k* in the *n*-dimensional projective space over $GF(q)$, and image the set of points of the Grassmann variety (`PointsOfGrassmannVariety` (11.8.2)). When a Grassmann variety *gv* is given as input, the operation returns the associated Grassmann map.

11.8.4 Source

▷ Source(*gm*) (operation)

    Returns: the source of a Grassmann map

    The argument *gm* is a `GrassmannMap` (11.8.3). This operation returns the set of subspaces of the projective space that was used to construct the `GrassmannMap` (11.8.3).

Chapter 12

Generalised Polygons

A *generalised n-gon* is a point/line geometry whose incidence graph is bipartite of diameter `n` and girth `2n`. Although these rank 2 structures are very much a subdomain of GRAPE and Design, their significance in finite geometry warrants their inclusion in FinInG. By the famous theorem of Feit and Higman, a generalised n-gon which has at least three points on every line, must have *n* in $\{2,3,4,6,8\}$. The case $n = 2$ concerns the complete multipartite graphs, which we disregard. The more interesting cases are accordingly projective planes ($n = 3$), generalised quadrangles ($n = 4$), generalised hexagons ($n = 6$), and generalised octagons ($n = 8$).

FinInG provides some basic functionality to deal with generalised polygons as incidence geometries. A lot of non-trivial interaction with the package GRAPE has been very useful and even necessary. Currently, generic functions to create generalised polygons, to create elements of generalised polygons, and to explore the elements are implemented. This generic functionality allows the user to construct generalised polygons through many different objects available in GAP and FinInG. Apart from these generic functions, some particular generalised polygons are available: the classical generalised hexagons and elation generalised quadrangles from different perspectives can be constructed.

12.1 Categories

12.1.1 IsGeneralisedPolygon

▷ IsGeneralisedPolygon   (Category)
▷ IsGeneralisedPolygonRep   (Representation)

This category is a subcategory of `IsIncidenceGeometry`, and contains all generalised polygons. Generalised polygons constructed through functions described in this chapter, all belong to `IsGeneralisedPolygonRep`.

12.1.2 Subcategories in IsGeneralisedPolygon

▷ IsProjectivePlaneCategory   (Category)
▷ IsGeneralisedQuadrangle   (Category)
▷ IsGeneralisedHexagon   (Category)
▷ IsGeneralisedOctagon   (Category)

All generalised polygons in FinInG belong to one of these four categories. It is not possible to construct generalised polygons of which the gonality is not known (or checked). Note that the classical generalised quadrangles (which are the classical polar spaces of rank 2) belong also to `IsGeneralisedQuadrangle` and that the desarguesian projective planes (which are the projective spaces of dimension 2) also belong to `IsProjectivePlaneCategory`, but both do not belong to `IsGeneralisedPolygonRep`.

12.1.3 **IsWeakGeneralisedPolygon**

▷ IsWeakGeneralisedPolygon (Category)

IsWeakGeneralisedPolygon is the category for weak generalised polygons.

12.1.4 **Subcategories in** `IsProjectivePlaneCategory`

▷ IsDesarguesianPlane (Category)

IsDesarguesianPlane is declared as a subcategory of `IsProjectivePlaneCategory` and `IsProjecticeSpace`. Projective spaces of dimension 2 constructed using `ProjectiveSpace` belong to IsDesarguesianPlane.

12.1.5 **Subcategories in** `IsGeneralisedQuadrangle`

▷ IsClassicalGQ (Category)
▷ IsElationGQ (Category)

IsClassicalGQ is declared as a subcategory of `IsGeneralisedQuadrangle` and `IsClassicalPolarSpace`. All classical polar spaces of rank 2 belong to `IsClassicalGQ`. IsElationGQ is declared as subcategory of `IsGeneralisedQuadrangle`. Elation GQs will be discusedd in detail in Section 12.6

```
——————————————— Example ———————————————
  gap> gp := SymplecticSpace(3,2);
  W(3, 2)
  gap> IsGeneralisedPolygon(gp);
  true
  gap> IsGeneralisedQuadrangle(gp);
  true
  gap> IsClassicalGQ(gp);
  true
  gap> IsGeneralisedPolygonRep(gp);
  false
```

12.1.6 **IsClassicalGeneralisedHexagon**

▷ IsClassicalGeneralisedHexagon (Category)

IsClassicalGeneralisedHexagon is declared as subcategory of `IsGeneralisedHexagon` and `IsLieGeometry`. The so called classical generalised hexagons are the hexagons that come from the

triality of the hyperbolic quadric $Q^+(7,q)$. The Split Cayley hexagon is embedded in the parabolic quadric $Q(6,q)$, or $W(5,q)$ in even characteristic. The twisted triality hexagon is embedded in the hyperbolic quadric $Q^+(7,q)$. The construction of these hexagons in a subcategory of `IsLieGeometry` means that the usual operations for Lie geometries become applicable. The classical generalised hexagons are in detail discussed in Section 12.5

```
_____ Example _____
gap> gp := SplitCayleyHexagon(3);
H(3)
gap> IsGeneralisedHexagon(gp);
true
gap> IsClassicalGeneralisedHexagon(gp);
true
gap> IsLieGeometry(gp);
true
gap> IsGeneralisedPolygonRep(gp);
true
```

12.2 Generic functions to create generalised polygons

12.2.1 GeneralisedPolygonByBlocks

▷ GeneralisedPolygonByBlocks(*l*)   (operation)

Returns: a generalised polygon

The argument `l` is a finite homogeneous list consisting of ordered sets of a common size $n+1$. This operation will assume that each element of `l` represents a line of the generalised polygon. Its points are assumed to be the union of all elements of `l`. The incidence is assumed to be symmetrised containment. From this information, an incidence graph is computed using GRAPE. If this graph has diameter d and girth $2d$, a generalised polygon is returned. The thickness condition is not checked. If $d \in \{3,4,6,8\}$, a projective plane, a generalised quadrangle, a generalised hexagon, a generalised octagon respectively, is returned. Note that for large input, this operation can be time consuming.

```
_____ Example _____
gap> blocks := [
>    [ 1, 2, 3, 4, 5 ], [ 1, 6, 7, 8, 9 ], [ 1, 10, 11, 12, 13 ],
>    [ 1, 14, 15, 16, 17 ], [ 1, 18, 19, 20, 21 ], [ 2, 6, 10, 14, 18 ],
>    [ 2, 7, 11, 15, 19 ], [ 2, 8, 12, 16, 20 ], [ 2, 9, 13, 17, 21 ],
>    [ 3, 6, 11, 16, 21 ], [ 3, 7, 10, 17, 20 ], [ 3, 8, 13, 14, 19 ],
>    [ 3, 9, 12, 15, 18 ], [ 4, 6, 12, 17, 19 ], [ 4, 7, 13, 16, 18 ],
>    [ 4, 8, 10, 15, 21 ], [ 4, 9, 11, 14, 20 ], [ 5, 6, 13, 15, 20 ],
>    [ 5, 7, 12, 14, 21 ], [ 5, 8, 11, 17, 18 ], [ 5, 9, 10, 16, 19 ] ];;
gap> gp := GeneralisedPolygonByBlocks( blocks );
<projective plane order 4>
```

12.2.2 GeneralisedPolygonByIncidenceMatrix

▷ GeneralisedPolygonByIncidenceMatrix(*incmat*)   (operation)

Returns: a generalised polygon

The argument *incmat* is a matrix representing the incidence matrix of a point line geometry. The points are represented by the columns, the rows represent the lines. From `incmat` a homogeneous list of sets of column entries is derived, which is then passed to GeneralisedPolygonByBlocks. When `incmat` indeed represents a generalised polygon, it is returned. The checks are performed by GeneralisedPolygonByBlocks.

```
───────────────────────── Example ─────────────────────────
 gap> incmat := [
 >    [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
 >    [ 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
 >    [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
 >    [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0 ],
 >    [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1 ],
 >    [ 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0 ],
 >    [ 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0 ],
 >    [ 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0 ],
 >    [ 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1 ],
 >    [ 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 ],
 >    [ 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0 ],
 >    [ 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0 ],
 >    [ 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0 ],
 >    [ 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0 ],
 >    [ 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0 ],
 >    [ 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1 ],
 >    [ 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0 ],
 >    [ 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0 ],
 >    [ 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1 ],
 >    [ 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0 ],
 >    [ 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0 ] ];;
 gap> pp := GeneralisedPolygonByIncidenceMatrix( incmat );
 <projective plane order 4>
```

12.2.3 GeneralisedPolygonByElements

▷ GeneralisedPolygonByElements(*pts*, *lns*, *inc*)   (operation)
▷ GeneralisedPolygonByElements(*pts*, *lns*, *inc*, *grp*, *act*)   (operation)
   Returns: a generalised polygon
   The argument `pts`, `lns` and `inc` are respectively a set of objects, a set of objects and a function. The function `inc` must represent an incidence relation between objects of `pts` and `lns`. The first version of GeneralisedPolygonByElements will construct an incidence graph, and if this graph has diameter d and girth $2d$, a generalised polygon is returned. The thickness condition is not checked. If $d \in \{3,4,6,8\}$, a projective plane, a generalised quadrangle, a generalised hexagon, a generalised octagon respectively, is returned. The argument `grp` is a group, and `act` a function, representing an action of the elements of `grp` on the objects in the lists `pts` and `lns`, preserving the incidence. The second version of GeneralisedPolygonByElements acts as the first version, but uses `grp` and `act` to construct the incidence graph in a more efficient way, so if `grp` is a non trivial group, the construction of the graph will be faster. This operation can typically be used to construct generalised polygons from objects that are available in FinInG. This difference in time is shown in the first two examples. The third examples shows the construction of the generalised quadrangle $T_2(O)$.

—————— Example ——————

```
gap> pg := PG(2,25);
ProjectiveSpace(2, 25)
gap> pts := Set(Points(pg));;
gap> lns := Set(Lines(pg));;
gap> inc := \*;
<Operation "*">
gap> gp := GeneralisedPolygonByElements(pts,lns,inc);
<projective plane order 25>
gap> time;
26427
gap> grp := CollineationGroup(pg);
The FinInG collineation group PGammaL(3,25)
gap> act := OnProjSubspaces;
function( var, el ) ... end
gap> gp := GeneralisedPolygonByElements(pts,lns,inc,grp,act);
<projective plane order 25>
gap> time;
127
gap> q := 4;
4
gap> conic := Set(Points(ParabolicQuadric(2,q)));
[ <a point in Q(2, 4)>, <a point in Q(2, 4)>, <a point in Q(2, 4)>,
  <a point in Q(2, 4)>, <a point in Q(2, 4)> ]
gap> pg := PG(3,q);
ProjectiveSpace(3, 4)
gap> hyp :=  HyperplaneByDualCoordinates(pg,[1,0,0,0]*Z(q)^0);
<a plane in ProjectiveSpace(3, 4)>
gap> em := NaturalEmbeddingBySubspace(PG(2,q),pg,hyp);
<geometry morphism from <All elements of ProjectiveSpace(2,
4)> to <All elements of ProjectiveSpace(3, 4)>>
gap> O := List(conic,x->x^em);;
gap> group := CollineationGroup(pg);
The FinInG collineation group PGammaL(4,4)
gap> stab := FiningSetwiseStabiliser(group,O);
#I  Computing adjusted stabilizer chain...
<projective collineation group with 6 generators>
gap> points1 := Set(Filtered(Points(pg),x->not x in hyp));;
gap> tangents := List(conic,x->TangentSpace(x)^em);
[ <a line in ProjectiveSpace(3, 4)>, <a line in ProjectiveSpace(3, 4)>,
  <a line in ProjectiveSpace(3, 4)>, <a line in ProjectiveSpace(3, 4)>,
  <a line in ProjectiveSpace(3, 4)> ]
gap> planes := List(tangents,x->Filtered(Planes(x),y->not y in hyp));;
gap> points2 := Union(planes);;
gap> points3 := [hyp];
[ <a plane in ProjectiveSpace(3, 4)> ]
gap> linesa := Union(List(O,x->Filtered(Lines(x),y->not y in hyp)));;
gap> linesb := Set(O);;
gap> pts := Union(points1,points2,points3);;
gap> lns := Union(linesa,linesb);;
gap> inc := \*;
<Operation "*">
gap> gp := GeneralisedPolygonByElements(pts,lns,inc,stab,\^);
```

```
<generalised quadrangle of order [ 4, 4 ]>
gap> time;
50
```

12.3 Attributes and operations for generalised polygons

All operations described in this section are applicable on objects in the category `IsGeneralisedPolygon`.

12.3.1 Order

▷ Order(*gp*)   (attribute)

Returns: a pair of positive integers

This method returns the parameters (s,t) of the generalised polygon *gp*. That is, $s + 1$ is the number of points on any line of *gp*, and $t + 1$ is the number of lines incident with any point of *gp*.

```
───────────────────────── Example ─────────────────────────
gap> gp := TwistedTrialityHexagon(2^3);
T(8, 2)
gap> Order(gp);
[ 8, 2 ]
gap> gp := HermitianPolarSpace(4,25);
H(4, 5^2)
gap> Order(gp);
[ 25, 125 ]
gap> gp := EGQByqClan(LinearqClan(3));
#I  Computed Kantor family. Now computing EGQ...
<EGQ of order [ 9, 3 ] and basepoint 0>
gap> Order(gp);
[ 9, 3 ]
```

12.3.2 IncidenceGraphAttr

▷ IncidenceGraphAttr(*gp*)   (attribute)

This attribute is declared for objects in `IsGeneralisedPolygon`. It is a mutable attribute and can be accessed by the operation `IncidenceGraph`.

12.3.3 IncidenceGraph

▷ IncidenceGraph(*gp*)   (operation)

Returns: a graph

The argument *gp* is a generalised polygon. This operation returns the incidence graph of *gp*. If *gp* is constructed using `GeneralisedPolygonByBlocks`, `GeneralisedPolygonByElements` or `GeneralisedPolygonByIncidenceMatrix`, an incidence graph is computed to check the input, and is stored as an attribute. For the particular generalised polygons available in FinInG, there is no

precomputed incidence graph. Note that computing an incidence graph may require some time, especially when the *gp* has no collineation group computed. Therefore, this operation will return an error when *gp* has no collineation group computed. As `CollineationGroup` is an attribute for objects in `IsGeneralisedPolygon`, the user should compute the collineation group and then reissue the command to compute the incidence graph.

We should also point out that this method returns a *mutable* attribute of *gp*, so that acquired information about the incidence graph can be added. For example, the automorphism group of the incidence graph may be computed and stored as a record component after the incidence graph is stored as an attribute of *gp*. Normally, attributes of GAP objects are immutable.

Note that the factor 2 as difference in the order of the collineation group of $Q(4,4)$ and the order of the automorphism group of its incidence graph is easily explained by the fact that the $Q(4,4)$ is self dual.

```
─────────────────── Example ───────────────────
gap> blocks := [
>    [ 1, 2, 3, 4, 5 ], [ 1, 6, 7, 8, 9 ], [ 1, 10, 11, 12, 13 ],
>    [ 1, 14, 15, 16, 17 ], [ 1, 18, 19, 20, 21 ], [ 2, 6, 10, 14, 18 ],
>    [ 2, 7, 11, 15, 19 ], [ 2, 8, 12, 16, 20 ], [ 2, 9, 13, 17, 21 ],
>    [ 3, 6, 11, 16, 21 ], [ 3, 7, 10, 17, 20 ], [ 3, 8, 13, 14, 19 ],
>    [ 3, 9, 12, 15, 18 ], [ 4, 6, 12, 17, 19 ], [ 4, 7, 13, 16, 18 ],
>    [ 4, 8, 10, 15, 21 ], [ 4, 9, 11, 14, 20 ], [ 5, 6, 13, 15, 20 ],
>    [ 5, 7, 12, 14, 21 ], [ 5, 8, 11, 17, 18 ], [ 5, 9, 10, 16, 19 ] ];;
gap> gp := GeneralisedPolygonByBlocks( blocks );
<projective plane order 4>
gap> incgraph := IncidenceGraph( gp );;
gap> Diameter( incgraph );
3
gap> Girth( incgraph );
6
gap> VertexDegrees( incgraph );
[ 5 ]
gap> aut := AutGroupGraph( incgraph );
<permutation group with 9 generators>
gap> DisplayCompositionSeries(aut);
G (9 gens, size 241920)
 | Z(2)
S (5 gens, size 120960)
 | Z(2)
S (5 gens, size 60480)
 | Z(3)
S (4 gens, size 20160)
 | A(2,4) = L(3,4)
1 (0 gens, size 1)
gap> gp := ParabolicQuadric(4,4);
Q(4, 4)
gap> incgraph := IncidenceGraph( gp );;
Error, No collineation group computed. Please compute collineation group before compu
ting incidence graph,n called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 24 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

```
gap> CollineationGroup(gp);
PGammaO(5,4)
gap> Order(last);
1958400
gap> incgraph := IncidenceGraph( gp );;
#I  Computing incidence graph of generalised polygon...
gap> aut := AutGroupGraph( incgraph );
<permutation group with 10 generators>
gap> Order(aut);
3916800
```

12.3.4   IncidenceMatrixOfGeneralisedPolygon

▷ IncidenceMatrixOfGeneralisedPolygon(*gp*)   (attribute)

Returns: a matrix

This method returns the incidence matrix of the generalised polygon via the operation `CollapsedAdjacencyMat` in the **GRAPE** package. The rows of the matrix correspond to the points of *gp*, and the columns correspond to the lines. Note that since this operation relies on `IncidenceGraph`, for some generalised polygons, it is necessary to compute a collineation group first.

——————— Example ———————
```
gap> gp := SymplecticSpace(3,2);
W(3, 2)
gap> CollineationGroup(gp);
PGammaSp(4,2)
gap> mat := IncidenceMatrixOfGeneralisedPolygon(gp);
#I  Computing incidence graph of generalised polygon...
[ [ 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
  [ 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
  [ 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0 ],
  [ 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 ],
  [ 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0 ],
  [ 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0 ],
  [ 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1 ],
  [ 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1 ],
  [ 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1 ] ]
```

12.3.5   CollineationGroup

▷ CollineationGroup(*gp*)   (attribute)

Returns: a group

This attribute returns the full collineation group of the generalised polygon *gp*. For some particular generalised polygons, a (subgroup) of the full collineation group can be computed efficiently without computing the incidence graph of *gp*: the full collineation group of classical generalised quadrangles and classical generalised hexagons; and an elation group with relation to a base-point of an elation generalised quadrangle. For generalised polygons constructed by the operations GeneralisedPolygonByBlocks, GeneralisedPolygonByElements or GeneralisedPolygonByIncidenceMatrix, the full collineation group is computed using the full automorphism group of the underlying incidence graph, the latter being computed by the package GRAPE.

The collineation groups computed for classical generalised quadrangles and classical generalised hexagons are collineation groups in the sense of FinInG, and come equipped with a NiceMonomorphism. The collineation groups computed in all other cases are permutations groups, acting on the vertices of the underlying incidence graph.

Note that the computation of the automorphism group of the underlying graph can be time consuming, also if the complete collineation group of the generalised polygon has been used as an argument in e.g. GeneralisedPolygonByElements.

The first example illustrates that CollineationGroup is naturally applicable to all classical generalised Polygons.

```
———————————————————— Example ————————————————————
gap> gp := PG(2,2);
ProjectiveSpace(2, 2)
gap> CollineationGroup(gp);
The FinInG collineation group PGL(3,2)
gap> gp := EllipticQuadric(5,4);
Q-(5, 4)
gap> CollineationGroup(gp);
PGammaO-(6,4)
gap> gp := TwistedTrialityHexagon(3^3);
T(27, 3)
gap> CollineationGroup(gp);
#I  Computing nice monomorphism...
#I  Found permutation domain...
3D_4(27)
gap> time;
40691
```

The second example illustrates the computation of collineation groups of generalised polygons constructed using different objects.

```
———————————————————— Example ————————————————————
gap> mat := [ [ 1, 1, 0, 0, 0, 1, 0 ], [ 1, 0, 0, 1, 1, 0, 0 ],
>          [ 1, 0, 1, 0, 0, 0, 1 ], [ 0, 1, 1, 1, 0, 0, 0 ],
>          [ 0, 1, 0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0, 1, 1 ],
>          [ 0, 0, 1, 0, 1, 1, 0 ] ];
[ [ 1, 1, 0, 0, 0, 1, 0 ], [ 1, 0, 0, 1, 1, 0, 0 ], [ 1, 0, 1, 0, 0, 0, 1 ],
  [ 0, 1, 1, 1, 0, 0, 0 ], [ 0, 1, 0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0, 1, 1 ],
  [ 0, 0, 1, 0, 1, 1, 0 ] ]
gap> gp := GeneralisedPolygonByIncidenceMatrix(mat);
<projective plane order 2>
gap> group := CollineationGroup(gp);
```

```
Group([ (3,4)(5,7)(9,10)(13,14), (3,7)(4,5)(11,12)(13,14), (2,3)(6,7)(8,9)
(12,13), (2,6)(4,5)(11,13)(12,14), (1,2)(4,7)(9,11)(10,12) ])
gap> gp := EGQByqClan(FisherqClan(3));
#I  Computed Kantor family. Now computing EGQ...
<EGQ of order [ 9, 3 ] and basepoint 0>
gap> group := CollineationGroup(gp);
#I  Computing incidence graph of generalised polygon...
#I  Using elation of the collineation group...
<permutation group of size 26127360 with 8 generators>
gap> Order(group);
26127360
gap> Random(group);
(1,75,27,191,96,50,9,110,88,53,63,154,115,213,229,19,236,226,49,143,16,266,58,
245,11,270,57,44)(2,181,116,225,262,223,17)(3,33,187,149,108,120,177,164,167,
261,198,26,196,276,52,73,94,222,101,176,32,39,43,89,31,280,65,71)(4,250,173,
112,246,38,142,138,54,208,69,243,197,42,269,242,125,8,134,265,67,206,20,13,29,
182,205,36)(5,109,129,82,210,277,185,56,104,114,90,68,61,228,132,235,78,257,
10,238,145,184,241,170,153,263,45,179)(6,159,230,106,147,91,22,137,256,113,
117,180,7,133,279,100,55,156,168,86,122,131,12,35,273,264,254,152)(14,62,66,
268,51,233,253,218,172,130,144,25,169,83,234,127,171,221,34,190,21,46,272,224,
239,267,60,98)(15,40,278,128,160,215,87,178,203,166,247,119,209,84,255,271,
232,81,193,252,92,95,111,201,107,140,135,258)( [...] )
gap> q := 4;
4
gap> conic := ParabolicQuadric(2,q);
Q(2, 4)
gap> nucleus := NucleusOfParabolicQuadric(conic);
<a point in ProjectiveSpace(2, 4)>
gap> conic := ParabolicQuadric(2,q);
Q(2, 4)
gap> nucleus := NucleusOfParabolicQuadric(conic);
<a point in ProjectiveSpace(2, 4)>
gap> hyperoval := Union(List(Points(conic)),[nucleus]);
[ <a point in ProjectiveSpace(2, 4)>, <a point in Q(2, 4)>,
  <a point in Q(2, 4)>, <a point in Q(2, 4)>, <a point in Q(2, 4)>,
  <a point in Q(2, 4)> ]
gap> pg := PG(3,q);
ProjectiveSpace(3, 4)
gap> hyp :=  HyperplaneByDualCoordinates(pg,[1,0,0,0]*Z(q)^0);
<a plane in ProjectiveSpace(3, 4)>
gap> em := NaturalEmbeddingBySubspace(PG(2,q),pg,hyp);
<geometry morphism from <All elements of ProjectiveSpace(2,
4)> to <All elements of ProjectiveSpace(3, 4)>>
gap> O := List(hyperoval,x->x^em);
[ <a point in ProjectiveSpace(3, 4)>, <a point in ProjectiveSpace(3, 4)>,
  <a point in ProjectiveSpace(3, 4)>, <a point in ProjectiveSpace(3, 4)>,
  <a point in ProjectiveSpace(3, 4)>, <a point in ProjectiveSpace(3, 4)> ]
gap> points := Set(Filtered(Points(pg),x->not x in hyp));;
gap> lines := Union(List(O,x->Filtered(Lines(x),y->not y in hyp)));;
gap> inc := \*;
<Operation "*">
gap> gp := GeneralisedPolygonByElements(points,lines,inc);
<generalised quadrangle of order [ 3, 5 ]>
```

```
gap> coll := CollineationGroup(gp);
<permutation group of size 138240 with 8 generators>
gap> Order(coll);
138240
gap> Random(coll);
(1,29,60,40)(2,42,4,10,3,61,59,19,57,51,58,8)(5,21,17,25,52,13,64,48,44,36,9,
56)(6,34,41,55,50,45,63,27,20,14,11,24)(7,53,18,46,12,35,62,16,43,23,49,
26)(15,32,47,31,28,39,54,37,22,38,33,30)(65,74,83,111,66,117,149,104,70,151,
142,78)(67,135,139,136,68,109,98,125,69,95,120,137)(71,92,73,128,77,106,141,
105,145,150,88,155)(72,121,158,160,76,143,119,103,138,152,134,84)(75,153,133,
107,115,122,118,85,154,116,147,91)(79,110,101,159,126,90,157,81,112,100,89,
108)(80,99,97,86,156,129,144,94,127,114,148,82)(87,132,102,131,123,130,124,96,
93,113,146,140)
```

In the third example, the use of an precomputed automorphism group is illustrated. It speeds up the construction of the underlying graph and the computation of the automorphism group of the underlying graph. However, as is also illustrated in the example, despite that the precomputed automorphism group of the generalised polygon is actually the full collineation group, still some time is needed to compute the automorphism group of the underlying graph. The timings after both `CollineationGroup` commands are wrong. This is because GRAPE relies on an external binary to computed the automorphism group of a graph. The generalised quadrangle in this example is known as $T_2(O)$.

```
——————————— Example ———————————
gap> q := 8;
8
gap> conic := ParabolicQuadric(2,q);
Q(2, 8)
gap> nucleus := NucleusOfParabolicQuadric(conic);
<a point in ProjectiveSpace(2, 8)>
gap> hyperoval := Union(List(Points(conic)),[nucleus]);
[ <a point in ProjectiveSpace(2, 8)>, <a point in Q(2, 8)>,
  <a point in Q(2, 8)>, <a point in Q(2, 8)>, <a point in Q(2, 8)>,
  <a point in Q(2, 8)>, <a point in Q(2, 8)>, <a point in Q(2, 8)>,
  <a point in Q(2, 8)>, <a point in Q(2, 8)> ]
gap> pg := PG(3,q);
ProjectiveSpace(3, 8)
gap> hyp :=  HyperplaneByDualCoordinates(pg,[1,0,0,0]*Z(q)^0);
<a plane in ProjectiveSpace(3, 8)>
gap> em := NaturalEmbeddingBySubspace(PG(2,q),pg,hyp);
<geometry morphism from <All elements of ProjectiveSpace(2,
8)> to <All elements of ProjectiveSpace(3, 8)>>
gap> O := List(hyperoval,x->x^em);
[ <a point in ProjectiveSpace(3, 8)>, <a point in ProjectiveSpace(3, 8)>,
  <a point in ProjectiveSpace(3, 8)>, <a point in ProjectiveSpace(3, 8)>,
  <a point in ProjectiveSpace(3, 8)>, <a point in ProjectiveSpace(3, 8)>,
  <a point in ProjectiveSpace(3, 8)>, <a point in ProjectiveSpace(3, 8)>,
  <a point in ProjectiveSpace(3, 8)>, <a point in ProjectiveSpace(3, 8)> ]
gap> points := Set(Filtered(Points(pg),x->not x in hyp));;
gap> lines := Union(List(O,x->Filtered(Lines(x),y->not y in hyp)));;
gap> inc := \*;
<Operation "*">
```

```
gap> gp := GeneralisedPolygonByElements(points,lines,inc);
<generalised quadrangle of order [ 7, 9 ]>
gap> time;
17466
gap> coll := CollineationGroup(gp);
<permutation group of size 5419008 with 9 generators>
gap> time;
69
gap> group := CollineationGroup(pg);
The FinInG collineation group PGammaL(4,8)
gap> stab := FiningSetwiseStabiliser(group,O);
#I  Computing adjusted stabilizer chain...
<projective collineation group with 11 generators>
gap> time;
2045
gap> gp := GeneralisedPolygonByElements(points,lines,inc,stab,\^);
<generalised quadrangle of order [ 7, 9 ]>
gap> time;
394
gap> coll := CollineationGroup(gp);
<permutation group of size 5419008 with 9 generators>
gap> time;
62
gap> Order(coll);
5419008
gap> Order(stab);
5419008
```

12.3.6   CollineationAction

▷ CollineationAction(*group*)   (attribute)

Returns: a function

group is a collineation group of a generalised polygon, computed using `CollineationGroup`. The collineation group of classical generalised polygons will be a collineation group in the sense of FinInG. The natural action is `OnProjectiveSubspaces`. The collineation group of any other generalised polygons will be a permutation group. The result of `CollineationAction` for such a group is a function with input a pair *(x,g)* where *x* is an element of the generalised polygon, and *g* is a collineation of the generalised polygon, so an element of *group*. The example illustrates the use in the generalised quadrangle.

—————————————————— Example ——————————————————

```
gap> q := 4;
4
gap> conic := ParabolicQuadric(2,q);
Q(2, 4)
gap> nucleus := NucleusOfParabolicQuadric(conic);
<a point in ProjectiveSpace(2, 4)>
gap> hyperoval := Union(List(Points(conic)),[nucleus]);
[ <a point in ProjectiveSpace(2, 4)>, <a point in Q(2, 4)>,
  <a point in Q(2, 4)>, <a point in Q(2, 4)>, <a point in Q(2, 4)>,
  <a point in Q(2, 4)> ]
```

```
gap> pg := PG(3,q);
ProjectiveSpace(3, 4)
gap> hyp :=  HyperplaneByDualCoordinates(pg,[1,0,0,0]*Z(q)^0);
<a plane in ProjectiveSpace(3, 4)>
gap> em := NaturalEmbeddingBySubspace(PG(2,q),pg,hyp);
<geometry morphism from <All elements of ProjectiveSpace(2,
4)> to <All elements of ProjectiveSpace(3, 4)>>
gap> O := List(hyperoval,x->x^em);
[ <a point in ProjectiveSpace(3, 4)>, <a point in ProjectiveSpace(3, 4)>,
  <a point in ProjectiveSpace(3, 4)>, <a point in ProjectiveSpace(3, 4)>,
  <a point in ProjectiveSpace(3, 4)>, <a point in ProjectiveSpace(3, 4)> ]
gap> points := Set(Filtered(Points(pg),x->not x in hyp));;
gap> lines := Union(List(O,x->Filtered(Lines(x),y->not y in hyp)));;
gap> inc := \*;
<Operation "*">
gap> gp := GeneralisedPolygonByElements(points,lines,inc);
<generalised quadrangle of order [ 3, 5 ]>
gap> coll := CollineationGroup(gp);
<permutation group of size 138240 with 8 generators>
gap> act := CollineationAction(coll);
function( el, g ) ... end
gap> g := Random(coll);
(1,37,45,63,27,19)(2,53,13,64,11,51)(3,33,38,61,31,28)(4,49,6,62,15,60)(5,46,
47,59,20,17)(7,42,40,57,24,26)(8,58)(9,55)(10,39,41,56,25,23)(12,35,34,54,29,
32)(14,48,43,52,18,21)(16,44,36,50,22,30)(65,132,90,157,89,105)(66,68,131,143,
119,103)(67,135,76,123,130,106)(69,133,112,100,81,107)(70,134,150,88,155,
104)(71,99,79,144,93,149)(72,153,95,120,73,122)(74,125,115,128,140,87)(75,121,
136,117,113,91)(77,124,98,83,147,146)(78,145,84,118,85,142)(80,92,137,141,108,
97)(82,86,116,111,138,101)(94,127,126,102,109,96)(110,152,151,154,156,
129)(114,160,139,158,148,159)
gap> l := Random(Lines(gp));
<a line in <generalised quadrangle of order [ 3, 5 ]>>
gap> act(l,g);
<a line in <generalised quadrangle of order [ 3, 5 ]>>
gap> p := Random(Points(gp));
<a point in <generalised quadrangle of order [ 3, 5 ]>>
gap> act(p,g);
<a point in <generalised quadrangle of order [ 3, 5 ]>>
gap> stab := Stabilizer(coll,p,act);
<permutation group of size 2160 with 3 generators>
gap> List(Orbits(stab,List(Points(gp)),act),x->Length(x));
[ 45, 18, 1 ]
gap> List(Orbits(stab,List(Lines(gp)),act),x->Length(x));
[ 90, 6 ]
```

12.3.7   BlockDesignOfGeneralisedPolygon

▷ BlockDesignOfGeneralisedPolygon(*gp*)   (attribute)

   Returns: a block design

   This method allows one to use the GAP package DESIGN to analyse a generalised polygon, so the

user must first load this package. The argument *gp* is a generalised polygon, and if it has a collineation group, the block design is computed with this extra information and thus the resulting design is easier to work with. Likewise, if *gp* is an elation generalised quadrangle and it has an elation group, then we use the elation group's action to efficiently compute the block design. We should also point out that this method returns a *mutable* attribute of *gp*, so that accquired information about the block design can be added. For example, the automorphism group of the block design may be computed after the design is stored as an attribute of *gp*. Normally, attributes of GAP objects are immutable.

```
——————————— Example ———————————
gap> LoadPackage("design");
#W BIND_GLOBAL: variable 'BlockDesign' already has a value
------------------------------------------------------------------------------
Loading  DESIGN 1.6 (The Design Package for GAP)
by Leonard H. Soicher (http://www.maths.qmul.ac.uk/~leonard/).
Homepage: http://www.designtheory.org/software/gap_design/
------------------------------------------------------------------------------
true
gap> gh := SplitCayleyHexagon(2);
H(2)
gap> CollineationGroup(gh);
#I  for Split Cayley Hexagon
#I  Computing nice monomorphism...
#I  Found permutation domain...
G_2(2)
gap> des := BlockDesignOfGeneralisedPolygon(gh);
rec( autSubgroup := <permutation group with 3 generators>,
  blocks := [ [ 1, 29, 52 ], [ 1, 34, 36 ], [ 1, 37, 48 ], [ 2, 13, 60 ],
      [ 2, 44, 53 ], [ 2, 45, 52 ], [ 3, 17, 35 ], [ 3, 22, 51 ],
      [ 3, 23, 48 ], [ 4, 16, 57 ], [ 4, 19, 36 ], [ 4, 54, 56 ],
      [ 5, 22, 63 ], [ 5, 31, 57 ], [ 5, 49, 52 ], [ 6, 7, 60 ],
      [ 6, 28, 57 ], [ 6, 35, 43 ], [ 7, 26, 27 ], [ 7, 33, 34 ],
      [ 8, 9, 53 ], [ 8, 22, 33 ], [ 8, 38, 56 ], [ 9, 25, 61 ],
      [ 9, 28, 37 ], [ 10, 18, 53 ], [ 10, 32, 35 ], [ 10, 36, 62 ],
      [ 11, 12, 63 ], [ 11, 26, 54 ], [ 11, 37, 42 ], [ 12, 41, 43 ],
      [ 12, 44, 50 ], [ 13, 15, 42 ], [ 13, 19, 51 ], [ 14, 15, 31 ],
      [ 14, 17, 61 ], [ 14, 34, 50 ], [ 15, 20, 38 ], [ 16, 23, 44 ],
      [ 16, 40, 59 ], [ 17, 45, 54 ], [ 18, 24, 26 ], [ 18, 30, 31 ],
      [ 19, 25, 41 ], [ 20, 21, 62 ], [ 20, 23, 27 ], [ 21, 28, 55 ],
      [ 21, 39, 45 ], [ 24, 29, 59 ], [ 24, 51, 55 ], [ 25, 27, 49 ],
      [ 29, 38, 43 ], [ 30, 39, 41 ], [ 30, 46, 48 ], [ 32, 40, 42 ],
      [ 32, 47, 49 ], [ 33, 39, 40 ], [ 46, 47, 56 ], [ 46, 58, 60 ],
      [ 47, 50, 55 ], [ 58, 59, 61 ], [ 58, 62, 63 ] ], isBlockDesign := true,
  v := 63 )
gap> f := GF(3);
GF(3)
gap> id := IdentityMat(2, f);;
gap> clan := List( f, t -> t*id );;
gap> clan := qClan(clan,f);
<q-clan over GF(3)>
gap> egq := EGQByqClan( clan );
#I  Computed Kantor family. Now computing EGQ...
<EGQ of order [ 9, 3 ] and basepoint 0>
gap> HasElationGroup( egq );
```

```
true
gap> design := BlockDesignOfGeneralisedPolygon( egq );;
#I  Computing orbits on lines of gen. polygon...
#I  Computing block design of generalised polygon...
gap> aut := AutGroupBlockDesign( design );
<permutation group with 6 generators>
gap> NrBlockDesignPoints( design );
280
gap> NrBlockDesignBlocks( design );
112
gap> DisplayCompositionSeries(aut);
G (6 gens, size 26127360)
 | Z(2)
S (5 gens, size 13063680)
 | Z(2)
S (5 gens, size 6531840)
 | Z(2)
S (4 gens, size 3265920)
 | 2A(3,3) = U(4,3) ~ 2D(3,3) = O-(6,3)
1 (0 gens, size 1)
```

12.4   Elements of generalised polygons

12.4.1   Collections of elements of generalised polygons

▷ ElementsOfIncidencStructure(*gp*, *i*)   (attribute)
▷ Points(*gp*)   (attribute)
▷ Lines(*gp*)   (attribute)
   Returns: a collection of elements of a generalised polygon
   gp is any generalised polygon, *i* is a natural number, necessarily 1 or 2. ElementsOfIncidenceStructure returns the elements of type *i* of *gp*, Points and Lines are the usual shortcuts.

12.4.2   Size

▷ Size(*els*)   (operation)
   Returns: a number
   els is a collection of elements of a generalised polygon. This operation returns the number of element in *els*.

12.4.3   Creating elements from objects and retrieving objects from elements

▷ ObjectToElement(*gp*, *obj*)   (operation)
▷ ObjectToElement(*gp*, *type*, *obj*)   (operation)
▷ UnderlyingObject(*el*)   (operation)
   Returns: a collection of elements of a generalised polygon
   To create elements in *gp* (of type *type*), one of the versions of ObjectToElement can be used. It is checked whether *obj* represents an element (of type *type*). To retrieve an underlying object of

an element `el`, `UnderlyingObject` can be used.

```
_____ Example _____
gap> mat := [ [ 1, 1, 0, 0, 0, 1, 0 ], [ 1, 0, 0, 1, 1, 0, 0 ],
>            [ 1, 0, 1, 0, 0, 0, 1 ], [ 0, 1, 1, 1, 0, 0, 0 ],
>            [ 0, 1, 0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0, 1, 1 ],
>            [ 0, 0, 1, 0, 1, 1, 0 ] ];
[ [ 1, 1, 0, 0, 0, 1, 0 ], [ 1, 0, 0, 1, 1, 0, 0 ], [ 1, 0, 1, 0, 0, 0, 1 ],
  [ 0, 1, 1, 1, 0, 0, 0 ], [ 0, 1, 0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0, 1, 1 ],
  [ 0, 0, 1, 0, 1, 1, 0 ] ]
gap> gp := GeneralisedPolygonByIncidenceMatrix(mat);
<projective plane order 2>
gap> p := Random(Points(gp));
<a point in <projective plane order 2>>
gap> UnderlyingObject(p);
7
gap> l := Random(Lines(gp));
<a line in <projective plane order 2>>
gap> UnderlyingObject(l);
[ 4, 6, 7 ]
gap> ObjectToElement(gp,1,4);
<a point in <projective plane order 2>>
gap> ObjectToElement(gp,2,5);
Error, <obj> does not represent a line of <gp> called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 18 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> ObjectToElement(gp,2,[1,2,3]);
Error, <obj> does not represent a line of <gp> called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 18 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> ObjectToElement(gp,[1,2,6]);
<a line in <projective plane order 2>>
```

12.4.4 Incidence

▷ IsIncident(*v*, *w*)   (operation)

▷ *(*v*, *w*)   (operation)

   Returns: true or false

Let *v* and *w* be two elements of a generalised polygon. It is checked if the ambient geometry of the two elements are identical, and true is returned if and only if the two elements are incident in their ambient geometry.

12.4.5 Span

▷ Span(*v, w*) (operation)

Returns: a line of a generalised polygon or fail

Let *v* and *w* be two elements of a generalised polygon. It is checked if the ambient geometries of the two elements are identical, and if the two elements are points. If *v* and *w* are incidence with a common line, this line is returned. Otherwise fail is returned. For generalised polygons constructed with GeneralisedPolygonByBlocks, GeneralisedPolygonByElements an GeneralisedPolygonByInidenceMatrix, the underlying graph is used. Note that the behaviour of Span is different for elements of generalised polygons that belong to IsLieGeometry, see 4.2.16.

```
———————————————————— Example ————————————————————
gap> mat := [ [ 1, 1, 0, 0, 0, 1, 0 ], [ 1, 0, 0, 1, 1, 0, 0 ],
>         [ 1, 0, 1, 0, 0, 0, 1 ], [ 0, 1, 1, 1, 0, 0, 0 ],
>         [ 0, 1, 0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0, 1, 1 ],
>         [ 0, 0, 1, 0, 1, 1, 0 ] ];
[ [ 1, 1, 0, 0, 0, 1, 0 ], [ 1, 0, 0, 1, 1, 0, 0 ], [ 1, 0, 1, 0, 0, 0, 1 ],
  [ 0, 1, 1, 1, 0, 0, 0 ], [ 0, 1, 0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0, 1, 1 ],
  [ 0, 0, 1, 0, 1, 1, 0 ] ]
gap> gp := GeneralisedPolygonByIncidenceMatrix(mat);
<projective plane order 2>
gap> p := Random(Points(gp));
<a point in <projective plane order 2>>
gap> q := Random(Points(gp));
<a point in <projective plane order 2>>
gap> Span(p,q);
<a line in <projective plane order 2>>
gap> ps := ParabolicQuadric(4,3);
Q(4, 3)
gap> gp := GeneralisedPolygonByElements(Set(Points(ps)),Set(Lines(ps)),\*);
<generalised quadrangle of order [ 3, 3 ]>
gap> p := Random(Points(gp));
<a point in <generalised quadrangle of order [ 3, 3 ]>>
gap> q := Random(Points(gp));
<a point in <generalised quadrangle of order [ 3, 3 ]>>
gap> Span(p,q);
#I  <x> and <y> do not span a line of gp
fail
```

12.4.6 Meet

▷ Meet(*v, w*) (operation)

Returns: a point of a generalised polygon or fail

Let *v* and *w* be two elements of a generalised polygon. It is checked if the ambient geometries of the two elements are identical, and if the two elements are lines. If *v* and *w* are incidence with a common point, this point is returned. Otherwise fail is returned. For generalised polygons constructed with GeneralisedPolygonByBlocks, GeneralisedPolygonByElements an GeneralisedPolygonByInidenceMatrix, the underlying graph is used. Note that the behavior of Meet is different for elements of generalised polygongs that belong to IsLieGeometry, see 4.2.17

```
———————————————————— Example ————————————————————
gap> mat := [ [ 1, 1, 0, 0, 0, 1, 0 ], [ 1, 0, 0, 1, 1, 0, 0 ],
```

```
>          [ 1, 0, 1, 0, 0, 0, 1 ], [ 0, 1, 1, 1, 0, 0, 0 ],
>          [ 0, 1, 0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0, 1, 1 ],
>          [ 0, 0, 1, 0, 1, 1, 0 ] ];
[ [ 1, 1, 0, 0, 0, 1, 0 ], [ 1, 0, 0, 1, 1, 0, 0 ], [ 1, 0, 1, 0, 0, 0, 1 ],
  [ 0, 1, 1, 1, 0, 0, 0 ], [ 0, 1, 0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0, 1, 1 ],
  [ 0, 0, 1, 0, 1, 1, 0 ] ]
gap> gp := GeneralisedPolygonByIncidenceMatrix(mat);
<projective plane order 2>
gap> l := Random(Lines(gp));
<a line in <projective plane order 2>>
gap> m := Random(Lines(gp));
<a line in <projective plane order 2>>
gap> Meet(l,m);
<a point in <projective plane order 2>>
gap> ps := ParabolicQuadric(4,3);
Q(4, 3)
gap> gp := GeneralisedPolygonByElements(Set(Points(ps)),Set(Lines(ps)),\*);
<generalised quadrangle of order [ 3, 3 ]>
gap> l := Random(Lines(gp));
<a line in <generalised quadrangle of order [ 3, 3 ]>>
gap> m := Random(Lines(gp));
<a line in <generalised quadrangle of order [ 3, 3 ]>>
gap> Meet(l,m);
#I  <x> and <y> do meet in a common point of gp
fail
```

12.4.7 Shadow elements

▷ ShadowOfElement(*geo*, *v*, *j*)   (operation)
▷ Points(*el*)   (operation)
▷ Lines(*el*)   (operation)
▷ ElementsIncidentWithElementOfIncidenceStructure(*el*, *i*)   (operation)

Returns: A collection of elements

geo is a generalised polygon, *v* must be an element of *geo*, *j* is an integer equal to 1 or 2, since *geo* is a rank two geometry. The operation ShadowOfElement returns the collection of elements of *geo* of type *j*, incident with the element *v*. The operations Points and Lines with argument are the usual shortcuts to ShadowOfElement with *j* respectively equal to 1, 2. The operation ElementsIncidentWithElementOfIncidenceStructure is the usual shortcut to ShadowOfElement.

```
                          ─── Example ───
gap> blocks := [
>   [ 1, 2, 3, 4, 5 ], [ 1, 6, 7, 8, 9 ], [ 1, 10, 11, 12, 13 ],
>   [ 1, 14, 15, 16, 17 ], [ 1, 18, 19, 20, 21 ], [ 2, 6, 10, 14, 18 ],
>   [ 2, 7, 11, 15, 19 ], [ 2, 8, 12, 16, 20 ], [ 2, 9, 13, 17, 21 ],
>   [ 3, 6, 11, 16, 21 ], [ 3, 7, 10, 17, 20 ], [ 3, 8, 13, 14, 19 ],
>   [ 3, 9, 12, 15, 18 ], [ 4, 6, 12, 17, 19 ], [ 4, 7, 13, 16, 18 ],
>   [ 4, 8, 10, 15, 21 ], [ 4, 9, 11, 14, 20 ], [ 5, 6, 13, 15, 20 ],
>   [ 5, 7, 12, 14, 21 ], [ 5, 8, 11, 17, 18 ], [ 5, 9, 10, 16, 19 ] ];;
gap> gp := GeneralisedPolygonByBlocks( blocks );
<projective plane order 4>
```

```
gap> l := Random(Lines(gp));
<a line in <projective plane order 4>>
gap> pts := ShadowOfElement(gp,l,1);
<shadow points in <projective plane order 4>>
gap> List(pts);
[ <a point in <projective plane order 4>>,
  <a point in <projective plane order 4>>,
  <a point in <projective plane order 4>>,
  <a point in <projective plane order 4>>,
  <a point in <projective plane order 4>> ]
gap> p := Random(Points(gp));
<a point in <projective plane order 4>>
gap> lines := Lines(p);
<shadow lines in <projective plane order 4>>
gap> List(lines);
[ <a line in <projective plane order 4>>, <a line in <projective plane order
    4>>, <a line in <projective plane order 4>>,
  <a line in <projective plane order 4>>, <a line in <projective plane order
    4>> ]
```

12.4.8 DistanceBetweenElements

▷ DistanceBetweenElements(*v*, *w*)   (operation)

    Returns: a number

Let *v* and *w* be two elements of a generalised polygon. It is checked if the ambient geometry of the two elements are identical, and the distance between the two elements in the incidence graph of their ambient geometry is returned.

```
──────────────────────────── Example ────────────────────────────
gap> g := ElementaryAbelianGroup(27);
<pc group of size 27 with 3 generators>
gap> flist1 := [ Group(g.1), Group(g.2), Group(g.3), Group(g.1*g.2*g.3) ];;
gap> flist2 := [ Group([g.1, g.2^2*g.3]), Group([g.2, g.1^2*g.3 ]),
>              Group([g.3, g.1^2*g.2]), Group([g.1^2*g.2, g.1^2*g.3 ]) ];;
gap> egq := EGQByKantorFamily(g, flist1, flist2);
<EGQ of order [ 3, 3 ] and basepoint 0>
gap> p := Random(Points(egq));
<a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>
gap> q := Random(Points(egq));
<a point of class 3 of <EGQ of order [ 3, 3 ] and basepoint 0>>
gap> DistanceBetweenElements(p,q);
2
gap> gh := SplitCayleyHexagon(3);
H(3)
gap> l := Random(Lines(gh));
#I  for Split Cayley Hexagon
#I  Computing nice monomorphism...
#I  Found permutation domain...
<a line in H(3)>
gap> m := First(Lines(gh),x->DistanceBetweenElements(l,x)=6);
<a line in H(3)>
```

12.5 The classical generalised hexagons

12.5.1 Trialities of the hyperbolic quadric and generalised hexagons

Consider the hyperbolic quadric $Q+(7,q)$. This is a polar space of rank 4. It is well known that its generators fall into two systems. Each system contains exactly $(q^3+q^2+q+1)(q^2+q+1)$ generators, which is equal to the number of points of $Q+(7,q)$. Generators from the same system meet each other in an empty subspace or in a line, Generators from a different system meet each other in a point of a plane. One defines the rank 4 geometry $\Omega(7,q)$ as follows. The 0-points are the points of $Q+(7,q)$, the 1-points are the generators of the first system, the 2-points are the generators of the second system, and the lines are the lines of $Q+(7,q)$. The incidence is the natural incidence of the underlying projective space. Denote the set of i-points as $P^{(i)}$, $i=0,1,2$.

A triality of $\Omega(7,q)$ is a map $\tau : P^{(i)} \to P^{(i+1)}$ (where $i+1$ is computed modulo 3) preserving the incidence and for which $\tau^3 = 1$. Note that the image of a line under τ is determined by the image of the points incident with the line.

An i-point is absolute with respect to a fixed triality if it is incident with its image under the triality. Consequently, a line is absolute with if it is fixed by the triality.

A generalised hexagon might be obtained as geometry of absolute points of one kind and absolute lines with relation to a fixed triality. Note that not all trialities yield (thick) generalised quadrangles. There are different types of trialities, for some of them the absolute geometry is degenerate.

The triality used in *FinInG* to construct the classical generalised hexagons is fixed. It is described explicitly in [VM98]. To describe the triality, a trilinear form expressing the incidence between i-points of $\Omega(7,q)$ is used. Given the fact that, because of the existence of a triality, the role of the 0, 1 and 2 points are the same, each of the 1 and 2 points can be labelled the same way as the 0-points, which are effectively labelled by 8-tuples $(x_0,\ldots,x_7) \in V(8,q) = V$ where each 8-tuples represents a projective point of $Q+(7,q)$.

Consider the hyperbolic quadric determined by the quadratic form $X_0X_4 + X_1X_5 + X_2X_6 + X_3X_7$. Consider the trilinear map T,

$$T(x,y,z) = \begin{vmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ z_0 & z_1 & z_2 \end{vmatrix} + \begin{vmatrix} x_4 & x_5 & x_6 \\ y_4 & y_5 & y_6 \\ z_4 & z_5 & z_6 \end{vmatrix} + x_3(z_0y_4 + z_1y_5 + z_2y_6) + x_7(y_0z_4 + y_1z_5 + y_2z_6) +$$

$y_3(x_0z_4 + x_1z_5 + x_2z_6) + y_7(z_0x_4 + z_1x_5 + z_2x_6) + z_3(y_0x_4 + y_1x_5 + y_2x_6) + z_7(x_0y_4 + x_1y_5 + x_2y_6) - x_3y_3z_3 - x_7y_7z_7$.

Now a pair $(x,y) \in V \times V$ represents an incident 0-1 pair of points if and only if $T(x,y,z)$ vanishes in the variable z, and similarly for any cyclic permutation of the letters x,y,z. So given a 1-point y, $T(x,y,z) = 0$, where z is a variable, and x is an unknown, gives a set of equations representing a generator of $Q+(7,q)$, this the generator of $Q+(7,q)$ represented by y as label of a 1-point.

Let σ be an automorphism of $GF(q)$ of order 3, or the identity. Consider the map

$\tau_\sigma : P^{(i)} \to P^{(i+1)}$

$(x_j) \mapsto (x_j^\sigma)$, $j = 0\ldots7$.

This map clearly preserves $T(x,y,z)$, so preserves the incidence, and has order three, so it is a triality of $\Omega(7,q)$. We call an element p absolute with respect to a triality τ if and only if pIp^τ. Consequently, a line is absolute if and only if it is fixed by the triality. Denote the set of i-points that are absolute with respect to the triality as $P^{(i)}_{\text{abs}}$, and set of absolute lines with respect to the triality as L_{abs}.

Then a famous theorem of Tits ([Tit59]) says that for the triality τ_σ, the point-line geometry $\Gamma^{(i)} = (P_{\text{abs}}^{(i)}, L_{\text{abs}}), I)$ is a generalised hexagon of order $(|K|, |L|)$, $K = GF(q)$ and L the subfield of invariant elements of K under the field automorphism σ. Note that a finite field has a field automorphism of order three if and only if its order equals q^3. So, for $K = GF(q)$ and $\sigma = 1$, $\Gamma^{(i)}$ is a generalised hexagon of order q, which is called the *split Cayley hexagon of order q*, denoted $H(q)$. For $K = GF(q^3)$, and σ a non-trival field automorphism of order 3, $\Gamma^{(i)}$ is a generalised hexagon of order (q^3, q), which is called the *twisted triality hexagon of order (q^3, q)*, denoted $T(q^3, q)$. Note that for a given triality, the hexagons $\Gamma^{(i)}$, $i = 0, 1, 2$ are isomorphic. Consequently, $\Gamma^{(0)}$ is a point-line geometry of which the point set, line set respectively, is a subset of the points, lines respectively of $Q + (7, q)$. Finally, we mention the following important theorem, which was shown by Tits ([Tit59]): the split Cayley hexagon, obtained by the triliaty with $\sigma = 1$, is contained in the hyperplane with equation $X_3 + X_7 = 0$, which intersects the hyperbolic quadric in the parabolic quadric $Q(6, q)$. The points of the split Cayley hexagon are the points of $Q(6, q)$.

This above description of the triality and the associated generalised hexagons, contains sufficient analytical information to implement the split Cayley hexagon and the twisted triality hexagon in an efficient way. The user is allowed to choose a representation for the ambient polar space. For $q = 2^h$ the polar spaces $Q(6, q)$ and $W(5, q)$ are isomorphic. Consequently, the user may choose $W(5, q)$ as ambient polar space for the split Cayley hexagon of even order. This embedding in $W(5, q)$ is called the perfect symplectic embedding of the split Cayley hexagon. Finally, [VM98] contains an explicit description of the generators of the collineation groups of both generalised hexagons.

12.5.2 IsLieGeometry

▷ IsLieGeometry (Category)

Recall that the classical generalised hexagons are constructed as an object in `IsLieGeometry`. This makes most operations described in the appropriate chapters on Lie geometries, projective spaces and polar spaces applicable.

12.5.3 SplitCayleyHexagon

▷ SplitCayleyHexagon(*q*) (operation)
▷ SplitCayleyHexagon(*f*) (operation)
▷ SplitCayleyHexagon(*ps*) (operation)
  Returns: a generalised hexagon

```
———————————————————— Example ————————————————————
gap> hexagon := SplitCayleyHexagon( 3 );
H(3)
gap> AmbientPolarSpace(hexagon);
Q(6, 3): -x_1*x_5-x_2*x_6-x_3*x_7+x_4^2=0
gap> ps := ParabolicQuadric(6,3);
Q(6, 3)
gap> hexagon := SplitCayleyHexagon( ps );
H(3) in Q(6, 3)
gap> AmbientPolarSpace(hexagon);
Q(6, 3)
gap> hexagon := SplitCayleyHexagon( 4 );
H(4)
gap> AmbientPolarSpace(hexagon);
```

```
W(5, 4): x1*y4+x2*y5+x3*y6+x4*y1+x5*y2+x6*y3=0
gap> ps := ParabolicQuadric(6,4);
Q(6, 4)
gap> hexagon := SplitCayleyHexagon( ps );
H(4) in Q(6, 4)
gap> AmbientPolarSpace(hexagon);
Q(6, 4)
```

12.5.4 TwistedTrialityHexagon

▷ TwistedTrialityHexagon(*q*)   (operation)
▷ TwistedTrialityHexagon(*f*)   (operation)
▷ TwistedTrialityHexagon(*ps*)   (operation)
  Returns: a generalised hexagon

—————————————————————— Example ——————————————————————
```
gap> hexagon := TwistedTrialityHexagon(2^3);
T(8, 2)
gap> AmbientPolarSpace(hexagon);
<polar space in ProjectiveSpace(
7,GF(2^3)): x_1*x_5+x_2*x_6+x_3*x_7+x_4*x_8=0 >
gap> ps := HyperbolicQuadric(7,2^3);
Q+(7, 8)
gap> hexagon := TwistedTrialityHexagon(ps);
T(8, 2) in Q+(7, 8)
gap> AmbientPolarSpace(hexagon);
Q+(7, 8)
```

12.5.5 VectorSpaceToElement

▷ VectorSpaceToElement(*gh, vec*)   (operation)
  Returns: an element of a classical generlised hexagon

  The argument *vec* is one vector or a list of vectors from the underlying vectorspace of *gh*. This operation checks whether *vec* represents a point or a line of *gh*. Note that vectors and matrices in different representations are allowed as argument.

—————————————————————— Example ——————————————————————
```
gap> ps := ParabolicQuadric(6,9);
Q(6, 9)
gap> gh := SplitCayleyHexagon(ps);
H(9) in Q(6, 9)
gap> vec := [ Z(3)^0, Z(3^2), 0*Z(3), Z(3^2), Z(3^2)^3, Z(3^2)^5, 0*Z(3) ];
[ Z(3)^0, Z(3^2), 0*Z(3), Z(3^2), Z(3^2)^3, Z(3^2)^5, 0*Z(3) ]
gap> p := VectorSpaceToElement(gh,vec);
<a point in H(9) in Q(6, 9)>
gap> vec := [ [ Z(3)^0, 0*Z(3), Z(3^2)^7, 0*Z(3), Z(3)^0, Z(3^2)^2, Z(3^2)^2 ],
>    [ 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), Z(3^2)^3, 0*Z(3) ] ];
[ [ Z(3)^0, 0*Z(3), Z(3^2)^7, 0*Z(3), Z(3)^0, Z(3^2)^2, Z(3^2)^2 ],
  [ 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), Z(3^2)^3, 0*Z(3) ] ]
gap> line := VectorSpaceToElement(gh,vec);
```

```
Error, <x> does not generate an element of <geom> called from
<function "unknown">( <arguments> )
 called from read-eval loop at line 14 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

12.5.6 ObjectToElement

▷ ObjectToElement(gh, obj) (operation)

Returns: an element of a classical generlised hexagon

The argument obj is one vector or a list of vectors from the underlying vectorspace of gh. This operation checks whether obj represents a point or a line of gh. Note that vectors and matrices in different representations are allowed as argument.

─────────── Example ───────────
```
gap> mat := IdentityMat(8,GF(5^3));
< mutable compressed matrix 8x8 over GF(125) >
gap> form := BilinearFormByMatrix(mat,GF(5^3));
< bilinear form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(
7,GF(5^3)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2+x_8^2=0 >
gap> gh := TwistedTrialityHexagon(ps);
T(125, 5) in Q+(7, 125): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2+x_8^2
gap> vec := [ Z(5)^0, Z(5^3)^55, Z(5^3)^99, Z(5^3)^107, Z(5^3)^8, Z(5^3)^35, Z(5^3)^73,
>    Z(5^3)^115 ];
[ Z(5)^0, Z(5^3)^55, Z(5^3)^99, Z(5^3)^107, Z(5^3)^8, Z(5^3)^35, Z(5^3)^73,
  Z(5^3)^115 ]
gap> p := ObjectToElement(gh,vec);
<a point in T(125, 5) in Q+(7, 125): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2
+x_8^2>
gap> vec := [ [ Z(5)^0, 0*Z(5), Z(5^3)^76, Z(5^3)^117, Z(5^3)^80, Z(5^3)^19, Z(5^3)^48,
>      Z(5^3)^100 ],
>    [ 0*Z(5), Z(5)^0, Z(5^3)^115, Z(5^3)^14, Z(5^3)^40, Z(5^3)^67, Z(5^3)^123,
>      Z(5^3)^3 ] ];
[ [ Z(5)^0, 0*Z(5), Z(5^3)^76, Z(5^3)^117, Z(5^3)^80, Z(5^3)^19, Z(5^3)^48,
      Z(5^3)^100 ],
  [ 0*Z(5), Z(5)^0, Z(5^3)^115, Z(5^3)^14, Z(5^3)^40, Z(5^3)^67, Z(5^3)^123,
      Z(5^3)^3 ] ]
gap> line := ObjectToElement(gh,vec);
<a line in T(125, 5) in Q+(7, 125): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2+
x_8^2>
```

12.5.7 UnderlyingObject

▷ UnderlyingObject(gh, obj) (operation)

Returns: a vector or a matrix

12.5.8 \in

▷ \in(*x, gh*)   (operation)
   Returns: true or false

```
                            ———— Example ————
gap> ps := HyperbolicQuadric(7,5^3);
Q+(7, 125)
gap> gh := TwistedTrialityHexagon(ps);
T(125, 5) in Q+(7, 125)
gap> repeat
> p := Random(Points(ps));
> until p in gh;
gap> time;
18399
gap> p in gh;
true
gap> q := ElementToElement(gh,p);
<a point in T(125, 5) in Q+(7, 125)>
gap> l := Random(Lines(p));
<a line in Q+(7, 125)>
gap> l in gh;
false
gap> List(Lines(q),x->x in gh);
[ true, true, true, true, true, true ]
```

12.5.9 Span and meet of elements

▷ Span(*x, y*)   (operation)
▷ Meet(*x, y*)   (operation)
   Returns: a subspace of a projective space
   x and *y* are two elements of a classical generalised hexagon. The operation Span returns the
projective line spanned by *x* and *y*. The operation Meet returns the intersection of the elements *x*
and *y*. Note that the classical generalised hexagons are Lie geometries, so their elements belong to a
subcategory of IsSubspaceOfProjectiveSpace. Therefore, the operations Span and Meet behave
as described in 7.5.2 and 7.5.3.

```
                            ———— Example ————
gap> ps := SymplecticSpace(5,8);
W(5, 8)
gap> gh := SplitCayleyHexagon(ps);
H(8) in W(5, 8)
gap> vec := [ Z(2)^0, Z(2^3)^6, Z(2^3)^5, Z(2^3)^6, Z(2)^0, Z(2^3) ];
[ Z(2)^0, Z(2^3)^6, Z(2^3)^5, Z(2^3)^6, Z(2)^0, Z(2^3) ]
gap> p := VectorSpaceToElement(gh,vec);
<a point in H(8) in W(5, 8)>
gap> vec := [ Z(2)^0, Z(2^3)^2, Z(2^3), Z(2^3)^3, Z(2^3)^5, Z(2^3)^5 ];
[ Z(2)^0, Z(2^3)^2, Z(2^3), Z(2^3)^3, Z(2^3)^5, Z(2^3)^5 ]
gap> q := VectorSpaceToElement(gh,vec);
<a point in H(8) in W(5, 8)>
gap> span := Span(p,q);
<a line in ProjectiveSpace(5, 8)>
gap> ElementToElement(gh,span);
```

```
<a line in H(8) in W(5, 8)>
gap> vec := [ [ Z(2)^0, 0*Z(2), Z(2^3)^6, Z(2)^0, 0*Z(2), Z(2^3) ],
>    [ 0*Z(2), Z(2)^0, Z(2^3)^6, Z(2^3)^4, Z(2^3)^4, 0*Z(2) ] ];
[ [ Z(2)^0, 0*Z(2), Z(2^3)^6, Z(2)^0, 0*Z(2), Z(2^3) ],
  [ 0*Z(2), Z(2)^0, Z(2^3)^6, Z(2^3)^4, Z(2^3)^4, 0*Z(2) ] ]
gap> l := VectorSpaceToElement(gh,vec);
<a line in H(8) in W(5, 8)>
gap> vec := [ [ Z(2)^0, 0*Z(2), Z(2)^0, Z(2^3), 0*Z(2), Z(2^3) ],
>    [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2^3)^2, Z(2^3)^4, Z(2^3)^4 ] ];
[ [ Z(2)^0, 0*Z(2), Z(2)^0, Z(2^3), 0*Z(2), Z(2^3) ],
  [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2^3)^2, Z(2^3)^4, Z(2^3)^4 ] ]
gap> m := VectorSpaceToElement(gh,vec);
<a line in H(8) in W(5, 8)>
gap> Meet(l,m);
< empty subspace >
gap> DistanceBetweenElements(l,m);
6
```

12.5.10 CollineationGroup

▷ CollineationGroup(*gh*)   (attribute)

  Returns: a group of collineations

  gh is a classical generalised hexagon. This attribute returns the full collineation group, equipped with a nice monomorphism.

———————————— Example ————————————
```
gap> mat := IdentityMat(7,GF(9));
< mutable compressed matrix 7x7 over GF(9) >
gap> form := BilinearFormByMatrix(mat,GF(9));
< bilinear form >
gap> ps := PolarSpace(form);
<polar space in ProjectiveSpace(
6,GF(3^2)): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2=0 >
gap> gh := SplitCayleyHexagon(ps);
H(9) in Q(6, 9): x_1^2+x_2^2+x_3^2+x_4^2+x_5^2+x_6^2+x_7^2
gap> group := CollineationGroup(gh);
#I  for Split Cayley Hexagon
#I  Computing nice monomorphism...
#I  Found permutation domain...
<projective collineation group with 18 generators>
gap> time;
19602
gap> HasNiceMonomorphism(group);
true
gap> gh := TwistedTrialityHexagon(2^3);
T(8, 2)
gap> group := CollineationGroup(gh);
#I  Computing nice monomorphism...
#I  Found permutation domain...
3D_4(8)
```

12.6   Elation generalised quadrangles

12.6.1   Elation generalised quadrangles and Kantor families

Suppose $S = (P, B, I)$ is a generalised quadrangle of order (s, t) for which there exists a point p and a group of collineations G fixing p and each line through p, with the extra property that G acts regularly on the points not collinear with p. Then S is called an *elation generalised quadrangle* with base-point p and elation group G, and G has order $s^2 t$. Let y be a fixed point of S, not collinear with p. Denote the $t + 1$ lines incident with p as $L_i, i = 0 \ldots t$. Define for each line L_i the unique point-line pair (z_i, M_i) such that $L_i I z_i I M_i I y$. Define the groups S_i as the subgroups of G fixing the lines M_i, and define the groups S_i^* as the subgroups of G fixing the point z_i. Define the set $J = \{S_i : i = 0 \ldots t\}$, and the set $J^* = \{S_i : i = 0 \ldots t\}$. Since S is an elation generalised quadrangle, J is a collection of $t + 1$ subgroups of G of order s, and each S_i^* contains has order st and contains S_i as a subgroup. Furthermore, the following two conditions are satisfied.

(K1) $S_i S_j \cap S_k = \{1\}$, for distinct i, j, k.

(K2) $S_i \cap S_j^* = \{1\}$, for distinct i, j.

The pair (J, J^*) is called a *4-gonal family* or *Kantor family* in G.

Remarkably, each Kantor family in a group of order $s^2 t$ gives rise to an elation generalised quadrangle. Kantor families and elation generalised quadrangles are equivalent objects, and one of the motivations to study Kantor families in groups was to find examples of non-classical elation generalised quadrangles.

Given a group G, together with a Kantor family (J, J^*), a generalised quadrangle is defined as follows.

The points are of three types:

(i) points of type 1 are the elements of G;

(ii) points of of type 2 are the right cosets $S^* g, s^* \in J^*$

(iii) the unique point of type (iii) is the symbol (∞).

The lines are of two types:

(a) lines of type (a) are the right cosets $Sg, S \in J$;

(b) Lines of type (b) are the symbols $[S], S \in J$.

Incidence is defined as follows. A point g of type (i) is incident with each line $Sg, S \in J$ of type (a). A point of type (ii) $S^* g$ is incident the line $[S]$ of type (b) and the t lines of type (a) for which $Sh \subset S^* g$. Finally, the unique point of type (iii) is incident with the lines of type (b), and there are no further incidences.

It is shown, see e.g. the standard work in this field of Payne and Thas [PT84], that this point-line geoemtry is a generalised quadrangle of order (s, t).

FinInG provides functions to construct elation generalised quadrangles directly from a Kantor family. The constructed generalised quadrangles are generalised polygons in the sense of FinInG, i.e. all generic operations described in Sections 12.3 and 12.4.

12.6.2   Categories

▷ IsEGQByKantorFamily   (Category)

▷ IsElementOfKantorFamily   (Category)

    IsEGQByKantorFamily is a subcategory of IsElationGQ. It contains all elations generalised quadrangles that are constructed from a Kantor family. IsElementOfKantorFamily is a subcategory

of `IsElementOfGeneralisedPolygon`. It contains the elements from generalised quadrangles in the category `IsEGQByKantorFamily`.

12.6.3 Kantor families

▷ IsKantorFamily(*g, f, fstar*)   (operation)

> **Returns:** true or false

There is no specific way to construct a Kantor family in FinInG. However, given a group *G* and two collections of subgroups, `IsKantorFamily` will check whether the input satisfies the conditions of a Kantor family. If so, the input can be used directly for the operation `EGQByKantorFamily`.

12.6.4 EGQByKantorFamily

▷ EGQByKantorFamily(*g, f, fstar*)   (operation)

> **Returns:** a generalised quadrangle

Let *g* be a group and *f* and *fstar* two collections of subgroups, satisfying the conditions of a Kantor family. This operation returns the corresponding elation generalised quadrangle. Note that this operation *does not* check if the input satisfies the conditions to be a Kantor family, it only checks whether the group `f[i]` is a subgroup of the group `fstar[i]`. In the example below, the use of `IsKantorFamily` is also demonstrated, and some categories are displayed.

```
———————————————————————— Example ————————————————————————
gap> g := ElementaryAbelianGroup(27);
<pc group of size 27 with 3 generators>
gap> flist1 := [ Group(g.1), Group(g.2), Group(g.3), Group(g.1*g.2*g.3) ];;
gap> flist2 := [ Group([g.1, g.2^2*g.3]), Group([g.2, g.1^2*g.3 ]),
>            Group([g.3, g.1^2*g.2]), Group([g.1^2*g.2, g.1^2*g.3 ]) ];;
gap> IsKantorFamily( g, flist1, flist2 );
#I  Checking tangency condition...
#I  Checking triple condition...
true
gap> egq := EGQByKantorFamily(g, flist1, flist2);
<EGQ of order [ 3, 3 ] and basepoint 0>
gap> CategoriesOfObject(egq);
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsGeneralisedPolygon",
  "IsGeneralisedQuadrangle", "IsElationGQ", "IsElationGQByKantorFamily" ]
gap> p := Random(Points(egq));
<a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>
gap> CategoriesOfObject(p);
[ "IsElementOfIncidenceStructure", "IsElementOfIncidenceGeometry",
  "IsElementOfGeneralisedPolygon", "IsElementOfKantorFamily" ]
```

12.6.5 Representation of elements and underlying objects

▷ ObjectToElement(*egq, t, obj*)   (operation)
▷ ObjectToElement(*egq, obj*)   (operation)
▷ BasePointOfEGQ(*egq*)   (operation)
▷ UnderlyingObject(*el*)   (operation)

For technical reasons, the underlying objects of the elements of an elation generalised quadrangle constructed from a Kantor family, are not exactly the mathematical objects from the definition. However, these technicalities are almost completely hidden for the user, except for the representation of lines of type (b), which are represented in FinInG by the elements of the collection J^* (instead of the elements of the collection J). This change from the original definition has no mathematical implications, since there is a bijective correspondence between the elements of J^* and J. Notice also that it is only possible to obtain the base-point of an elation generalised quadrangle constructed from a Kantor family through calling the attribute `BasePointOfEGQ`.

```
————————————————— Example —————————————————
gap> g := ElementaryAbelianGroup(27);
<pc group of size 27 with 3 generators>
gap> flist1 := [ Group(g.1), Group(g.2), Group(g.3), Group(g.1*g.2*g.3) ];;
gap> flist2 := [ Group([g.1, g.2^2*g.3]), Group([g.2, g.1^2*g.3 ]),
>             Group([g.3, g.1^2*g.2]), Group([g.1^2*g.2, g.1^2*g.3 ]) ];;
gap> egq := EGQByKantorFamily(g, flist1, flist2);
<EGQ of order [ 3, 3 ] and basepoint 0>
gap> h := Random(g);
f1*f2^2
gap> p := ObjectToElement(egq,h);
<a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>
gap> coset := RightCoset(flist1[1],h);
RightCoset(Group( [ f1 ] ),f1*f2^2)
gap> l := ObjectToElement(egq,coset);
<a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>
gap> p * l;
true
gap> S := flist2[2];
<pc group of size 9 with 2 generators>
gap> m := ObjectToElement(egq,S);
<a line of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>
gap> q := BasePointOfEGQ(egq);
<a point of class 3 of <EGQ of order [ 3, 3 ] and basepoint 0>>
gap> m * q;
true
gap> lines := List(Lines(p));
[ <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>> ]
gap> pts1 := List(Points(m));
[ <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 3 of <EGQ of order [ 3, 3 ] and basepoint 0>> ]
gap> pts2 := List(Points(l));
[ <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>> ]
gap> List(pts2,x->UnderlyingObject(x));
[ f2^2, f1*f2^2, f1^2*f2^2, RightCoset(Group( [ f1, f2^2*f3 ] ),f3^2) ]
gap> UnderlyingObject(q);
```

```
0
```

12.6.6 Elation group and natural action on elements

▷ ElationGroup(*egq*) (attribute)
▷ OnKantorFamily(*g, el*) (function)
▷ CollineationAction(*g*) (attribute)

The attribute `ElationGroup` is naturally set upon creation of an elation generalised quadrangle from a Kantor family. The elements of the elation group act "naturally" on the elements of the elation generalised quadrangle. This natural action is implemented in the action function `OnKantorFamily`. When *g* is the elation group of an elation generalised quadrangle constructed from a Kantor family, the attribute `CollineationAction` will return the action function `OnKantorFamily`. The action function makes use of generic GAP operations possible, as is demonstrated in the example.

```
──────────────── Example ────────────────
gap> g := ElementaryAbelianGroup(27);
<pc group of size 27 with 3 generators>
gap> flist1 := [ Group(g.1), Group(g.2), Group(g.3), Group(g.1*g.2*g.3) ];;
gap> flist2 := [ Group([g.1, g.2^2*g.3]), Group([g.2, g.1^2*g.3 ]),
>           Group([g.3, g.1^2*g.2]), Group([g.1^2*g.2, g.1^2*g.3 ]) ];;
gap> egq := EGQByKantorFamily(g, flist1, flist2);
<EGQ of order [ 3, 3 ] and basepoint 0>
gap> group := ElationGroup(egq);
<pc group of size 27 with 3 generators>
gap> CollineationAction(group) = OnKantorFamily;
true
gap> l := ObjectToElement(egq,RightCoset(flist1[1],One(g)));
<a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>
gap> stab := Stabilizer(group,l,OnKantorFamily);
Group([ f1 ])
gap> pts := List(Points(egq));
[ <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
```

```
          <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 3 of <EGQ of order [ 3, 3 ] and basepoint 0>> ]
gap> Orbits(group,pts,OnKantorFamily);
[ [ <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
  [ <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
  [ <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
```

```
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
      [ <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
      [ <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
          <a point of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
      [ <a point of class 3 of <EGQ of order [ 3, 3 ] and basepoint 0>> ] ]
gap> lines := List(Lines(egq));
[ <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
  <a line of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>> ]
gap> Orbits(group,lines,OnKantorFamily);
[ [ <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
      <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
```

```
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
    [ <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
    [ <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
    [ <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>>,
        <a line of class 1 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
    [ <a line of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
    [ <a line of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
    [ <a line of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>> ],
    [ <a line of class 2 of <EGQ of order [ 3, 3 ] and basepoint 0>> ] ]
```

12.6.7 Kantor families, q-clans, and elation generalised quadrangles

Let $C = \{A_i : i = 1 \ldots q\}$ be a set of q distinct upper triangle 2×2 matrices over the finite field $GF(q)$. Then C is called a *q-clan* if $A_r - A_t$ is anistropic for $r \neq t$.

Define $G = \{(\alpha, c, \beta) : \alpha, \beta \in GF(q)^2, c \in GF(q)\}$, and define the binary operator $\times$ as $(\alpha, c, \beta) \times (\alpha', c', \beta') = (\alpha + \alpha', c + c' + \beta \cdot \alpha'^T, \beta + \beta')$. Then $G, \times$ is a group with center $Z(G) = \{(0, c, 0) : c \in GF(q)\}$. Consider a q-clan $C = \{A_i : i = 1 \ldots q\}$, define $K_i = A_i + A_i^T$. Now define the following subgroups of G

$A(i) = \{(\alpha, \alpha A_t, \alpha K_t) : \alpha \in GF(q)^2, i = 1 \ldots q\}$ and $A(\infty) = 0, 0, \gamma) : \gamma \in GF(q)^2\}$, and

$A^*(i) = \{(\alpha, \underline{\ }, \alpha K_t) : \alpha \in GF(q)^2, b \in GF(q), i = 1 \ldots q\}$ and $A^*(\infty) = \{(0, b, \gamma) : \gamma \in GF(q)^2, b \in GF(q)\}$

Define $J = \{A(i) : i = 1 \dots q\} \cup \{A(\infty)\}$ and $J^* = \{A^*(i) : i = 1 \dots q\} \cup \{A^*(\infty)\}$

A combination of results of Payne and Kantor yield the famous theorem that (J,J^*) is a Kantor family in G if and only if C is a q-clan. FinInG provides functionality to construct q-clans and to construct the corresponding Kantor family. As such, elation generalised quadrangles can directly constructed from q-clans.

12.6.8 qClan

▷ qClan(`list, f`)   (operation)

Returns: a q-clan.

Given a list `list` of $2\times$ matrices over the field `f`, it is checked if the matrices in the list satisfy the condition to constitute a q-clan over f. If so, the q-clan is returned.

12.6.9 Particular q-clans

▷ LinearqClan(`q`)   (operation)
▷ FisherThasWalkerKantorBettenqClan(`q`)   (operation)
▷ KantorMonomialqClan(`q`)   (operation)
▷ KantorKnuthqClan(`q`)   (operation)
▷ FisherqClan(`q`)   (operation)

Returns: a q-clan

Some famous q-clans are built in. We refer to ... for more information on these.

12.6.10 KantorFamilyByqClan

▷ KantorFamilyByqClan(`clan`)   (operation)

Returns: A Kantor family.

The Kantor family constructed from a q-clan will be a matrix group together with the corresponding collections J and J^* .

12.6.11 EGQByqClan

▷ EGQByqClan(`clan`)   (operation)

Returns: An elation generalised quadrangle constructed from a q-clan.

Given a q-clan `clan`, the operation KantorFamilyByqClan will be used to construct the Kantor family from `clan`, followed by the construction of the elation generalised quadrangle using the operation EGQByKantorFamily. The first example shows also the use of qClan, and shows that a linear q-clan yields a classical generalised quadrangle.

```
───────── Example ─────────
gap> f := GF(3);
GF(3)
gap> id := IdentityMat(2, f);;
gap> list := List( f, t -> t * id );;
gap> clan := qClan(list,f);
<q-clan over GF(3)>
gap> egq := EGQByqClan(clan);
#I  Computed Kantor family. Now computing EGQ...
<EGQ of order [ 9, 3 ] and basepoint 0>
gap> incgraph := IncidenceGraph(egq);;
```

```
#I  Computing incidence graph of generalised polygon...
#I  Using elation of the collineation group...
gap> group := AutomorphismGroup(incgraph);
<permutation group with 6 generators>
gap> Order(group);
26127360
gap> Order(CollineationGroup(HermitianPolarSpace(3,9)));
26127360
gap> clan := KantorKnuthqClan(9);
<q-clan over GF(3^2)>
gap> egq := EGQByqClan(clan);
#I  Computed Kantor family. Now computing EGQ...
<EGQ of order [ 81, 9 ] and basepoint 0>
gap> clan := FisherThasWalkerKantorBettenqClan(11);
<q-clan over GF(11)>
```

12.6.12   BLT-sets, flocks, q-clans, and elation generalised quadrangles

A *flock* is a partition of the points of a quadratic cone in $PG(3,q)$ minus its vertex into conics. Each conic is determined by a plane, and each plane is determined uniquely by a triple of elements of the field $GF(q)$. So a flock is determined by q such triples. Remarkably, as was shown by J.A. Thas, the conditions for these triples to constitute a flock, are exactly the same conditions for these triples to constitute q upper triangle matrices making a q-clan over $GF(q)$. Hence, q-clans and flocks, and thus flocks and elation generalised quadrangles, are equivalent objects.

The quadratic cone can be embedded as a hyperplane section into the parabolic quadric $Q(4,q)$. L. Bader, G. Lunardon and J.A. Thas observed that a set of q points of $Q(4,q)$ can be constructed from the q planes determining the flock, and this set of points satisfies certain geometric conditions. Such a set is called a *BLT-set*. Dually, a BLT-set corresponds to a set of lines of $W(3,q)$. Furthermore, from this BLT-set of lines, it is possible to construct directly an elation generalised quadrangle from carefully selecting points and lines from the symplectic space $W(5,q)$. This construction is called the Knarr construction.

Consider the symplectic polar space $W(5,q)$ and choose a point $P \in W(5,q)$. Its tangent space is a 4-dimensional space F. Embed $W(3,q)$ in a solid of F not on the point p, and let L be the set of BLT lines. Each line spans with p a plane of $W(5,q)$, call these q planes the *BLT-planes*. Now we can define the points an lines of the elation generalised quadrangle as follows.

Points of type (i) are the points of $W(5,q) \setminus F$, points of type (ii) are the lines of each BLT-plane not on p and the unique point of type (iii) is the point p. Lines of type (a) are the planes of $W(5,q)$ meeting a BLT-plane in a line. Note that no plane of $W(5,q)$ meeting two BLT planes in a line can exist. Lines of type (b) are the BLT-planes. Incidence is the natural incidence (so the incidence inherited) from the polar space $W(5,q)$, and this geometry is a elation generalised quadrangle with base-point p and of order (q^2,q).

FinInG provides functions to construct elation generalised quadrangles using this model from BLT-sets, and provides a function to compute a BLT set from a q-clan directly. The advantage of constructing a elation generalised from elements of Lie geometries is the availability of the underlying projective groups and their action on elements of Lie geometries.

12.6.13   IsEGQByBLTSet

▷ IsEGQByBLTSet   (Category)

   IsEGQByBLTSet is a subcategory of IsElationGQ. It contains all elations generalised quadrangles that are constructed from a BLT set.

12.6.14   BLTSetByqClan

▷ BLTSetByqClan(*clan*)   (operation)
   Returns: A BLT-set.
   The BLT-set is a set of points of the parabolic quadric in $PG(4, q)$ with particular equation $2X_1x_5 + 2X_2X_4 + w^{(q+1)/2} = 0$, where w is a primitive element of the underlying field of *clan*.

```
―――――――――――――――――――――――――――― Example ――――――――――――――――――――――――――――
gap> clan := KantorKnuthqClan(9);
<q-clan over GF(3^2)>
gap> blt := BLTSetByqClan(clan);
[ <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
  <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0> ]
gap> clan := FisherThasWalkerKantorBettenqClan(11);
<q-clan over GF(11)>
gap> blt := BLTSetByqClan(clan);
[ <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
  <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0> ]
```

12.6.15   EGQByBLTSet

▷ EGQByBLTSet(*blt*)   (operation)
   Returns: An elation generalised quadrangle.
   blt is a BLT-set, this operation returns an elation generalised quadrangle constructed as described above consisting of elements of $W(5, q)$. Notice in the example that computing the full collineation

group of the GQ constructed directly from the q-clan (hence a group coset geometry) is substantially slower than computing the full collineation group of the GQ constructed from the BLT-set.

```
──────────────── Example ────────────────
gap> clan := LinearqClan(3);
<q-clan over GF(3)>
gap> bltset := BLTSetByqClan( clan);
[ <a point in Q(4, 3): -x_1*x_5-x_2*x_4+x_3^2=0>,
  <a point in Q(4, 3): -x_1*x_5-x_2*x_4+x_3^2=0>,
  <a point in Q(4, 3): -x_1*x_5-x_2*x_4+x_3^2=0>,
  <a point in Q(4, 3): -x_1*x_5-x_2*x_4+x_3^2=0> ]
gap> egq := EGQByBLTSet( bltset );
#I  Now embedding dual BLT-set into W(5,q)...
#I  Computing elation group...
<EGQ of order [ 9, 3 ] and basepoint in W(5, 3 ) >
gap> p := BasePointOfEGQ(egq);
<a point in <EGQ of order [ 9, 3 ] and basepoint in W(5, 3 ) >>
gap> UnderlyingObject(p);
<a point in W(5, 3)>
gap> l := Random(Lines(egq));
<a line in <EGQ of order [ 9, 3 ] and basepoint in W(5, 3 ) >>
gap> UnderlyingObject(l);
<a plane in W(5, 3)>
gap> group := ElationGroup(egq);
<projective collineation group with 5 generators>
gap> Order(group);
243
gap> CollineationGroup(egq);
#I  Using elation group to enumerate elements
#I  Using elation group to enumerate elements
#I  Computing incidence graph of generalised polygon...
#I  Using elation of the collineation group...
#I  Using elation group to enumerate elements
<permutation group of size 26127360 with 7 generators>
gap> time;
147
gap> egq := EGQByqClan(clan);
#I  Computed Kantor family. Now computing EGQ...
<EGQ of order [ 9, 3 ] and basepoint 0>
gap> CollineationGroup(egq);
#I  Computing incidence graph of generalised polygon...
#I  Using elation of the collineation group...
<permutation group of size 26127360 with 6 generators>
gap> time;
1139
```

12.6.16 DefiningPlanesOfEGQByBLTSet

▷ DefiningPlanesOfEGQByBLTSet(*egq*)   (attribute)

For an elation generalised quadrangle in the category IsEGQByBLTSet (constructed from a BLT-set), the planes of the polar space $W(5,q)$, as described in the introduction, determine the generalised

quadrangle completely. This attribute returns these q planes of $W(5, q)$.

```
——————————————— Example ———————————————
  gap> clan := KantorKnuthqClan(9);
  <q-clan over GF(3^2)>
  gap> blt := BLTSetByqClan(clan);
  [ <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
    <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
    <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
    <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
    <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
    <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
    <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
    <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
    <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0>,
    <a point in Q(4, 9): -x_1*x_5-x_2*x_4+Z(3^2)^5*x_3^2=0> ]
  gap> egq := EGQByBLTSet(blt);
  #I  Now embedding dual BLT-set into W(5,q)...
  #I  Computing elation group...
  <EGQ of order [ 81, 9 ] and basepoint in W(5, 9 ) >
  gap> DefiningPlanesOfEGQByBLTSet(egq);
  [ <a plane in W(5, 9)>, <a plane in W(5, 9)>, <a plane in W(5, 9)>,
    <a plane in W(5, 9)>, <a plane in W(5, 9)>, <a plane in W(5, 9)>,
    <a plane in W(5, 9)>, <a plane in W(5, 9)>, <a plane in W(5, 9)>,
    <a plane in W(5, 9)> ]
```

12.6.17 Representation of elements and underlying objects

▷ ObjectToElement(*egq, t, obj*)   (operation)
▷ ObjectToElement(*egq, obj*)   (operation)
▷ UnderlyingObject(*el*)   (operation)

The underlying objects of the elements of an elation generalised quadrangle in the category
IsEGQByBLTSet are elements of the polar space $W(5, q)$ in its standard representation. These elements can be used as underlying object to construct elements of *egq*. The example also demonstrates the use of DistanceBetweenElements.

```
——————————————— Example ———————————————
  gap> clan := FisherThasWalkerKantorBettenqClan(11);
  <q-clan over GF(11)>
  gap> blt := BLTSetByqClan(clan);
  [ <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0>,
    <a point in Q(4, 11): Z(11)*x_1*x_5+Z(11)*x_2*x_4+Z(11)^6*x_3^2=0> ]
```

```
gap> egq := EGQByBLTSet(blt);
#I  Now embedding dual BLT-set into W(5,q)...
#I  Computing elation group...
<EGQ of order [ 121, 11 ] and basepoint in W(5, 11 ) >
gap> planes := DefiningPlanesOfEGQByBLTSet(egq);
[ <a plane in W(5, 11)>, <a plane in W(5, 11)>, <a plane in W(5, 11)>,
  <a plane in W(5, 11)>, <a plane in W(5, 11)>, <a plane in W(5, 11)>,
  <a plane in W(5, 11)>, <a plane in W(5, 11)>, <a plane in W(5, 11)>,
  <a plane in W(5, 11)>, <a plane in W(5, 11)>, <a plane in W(5, 11)> ]
gap> p := BasePointOfEGQ(egq);
<a point in <EGQ of order [ 121, 11 ] and basepoint in W(5, 11 ) >>
gap> up := UnderlyingObject(p);
<a point in W(5, 11)>
gap> ps := SymplecticSpace(5,11);
W(5, 11)
gap> uq := VectorSpaceToElement(ps,[1,1,0,0,0,0]*Z(11)^0);
<a point in W(5, 11)>
gap> q := ObjectToElement(egq,1,uq);
<a point in <EGQ of order [ 121, 11 ] and basepoint in W(5, 11 ) >>
gap> DistanceBetweenElements(p,q);
4
gap> l := ObjectToElement(egq,2,planes[1]);
<a line in <EGQ of order [ 121, 11 ] and basepoint in W(5, 11 ) >>
gap> DistanceBetweenElements(p,l);
1
gap> DistanceBetweenElements(q,l);
3
gap> um := VectorSpaceToElement(ps,[[1,0,0,0,1,1],[0,1,0,9,1,0],[0,0,1,9,9,9]]*Z(11)^0);
<a plane in W(5, 11)>
gap> m := ObjectToElement(egq,2,um);
<a line in <EGQ of order [ 121, 11 ] and basepoint in W(5, 11 ) >>
gap> DistanceBetweenElements(p,m);
3
gap> DistanceBetweenElements(q,m);
3
gap> DistanceBetweenElements(l,m);
2
```

12.6.18 CollineationSubgroup

▷ CollineationSubgroup(*egq*)   (attribute)

For an elation generalised quadrangle in the category IsEGQByBLTSet (constructed from a BLT set), the planes of the polar space $W(5, q)$, as described in the introduction, determine the generalised quadrangle completely. The setwise stabiliser of these planes in the collineation group of $W(5, q)$ is a subgroup of the completely collineation group of the elation generalised quadrangle, and can be computed much faster than the complete collineation group. This attribute returns this setwise stabiliser. The returned group is equipped with the CollineationACtion attribute. If CollineationSubgroup is computed, this group will be used instead of the elation group to compute the incidence graph.

---------- Example ----------
```
gap> clan := FisherThasWalkerKantorBettenqClan(5);
<q-clan over GF(5)>
gap> blt := BLTSetByqClan(clan);
[ <a point in Q(4, 5): Z(5)*x_1*x_5+Z(5)*x_2*x_4+Z(5)^3*x_3^2=0>,
  <a point in Q(4, 5): Z(5)*x_1*x_5+Z(5)*x_2*x_4+Z(5)^3*x_3^2=0>,
  <a point in Q(4, 5): Z(5)*x_1*x_5+Z(5)*x_2*x_4+Z(5)^3*x_3^2=0>,
  <a point in Q(4, 5): Z(5)*x_1*x_5+Z(5)*x_2*x_4+Z(5)^3*x_3^2=0>,
  <a point in Q(4, 5): Z(5)*x_1*x_5+Z(5)*x_2*x_4+Z(5)^3*x_3^2=0>,
  <a point in Q(4, 5): Z(5)*x_1*x_5+Z(5)*x_2*x_4+Z(5)^3*x_3^2=0> ]
gap> egq := EGQByBLTSet(blt);
#I  Now embedding dual BLT-set into W(5,q)...
#I  Computing elation group...
<EGQ of order [ 25, 5 ] and basepoint in W(5, 5 ) >
gap> coll := CollineationSubgroup(egq);
#I  Computing adjusted stabilizer chain...
<projective collineation group with 13 generators>
gap> Order(coll);
9000000
gap> act := CollineationAction(coll);
function( el, x ) ... end
gap> orbs := Orbits(coll,Points(egq),act);;
#I  Using elation group to enumerate elements
gap> List(orbs,x->Length(x));
[ 1, 3125, 150 ]
gap> el := ElationGroup(egq);
<projective collineation group with 5 generators>
gap> orbs := Orbits(el,Points(egq),act);;
#I  Using elation group to enumerate elements
gap> List(orbs,x->Length(x));
[ 1, 3125, 25, 25, 25, 25, 25, 25 ]
```

Chapter 13

Coset Geometries and Diagrams

This part of FinInG depends on GRAPE.

13.1 Coset Geometries

Suppose we have an *incidence geometry* Γ (as defined in chapter 3), together with a group G of automorphisms of Γ such that G is transitive on the set of *chambers* of Γ (also defined in chapter 3). This implies that G is also transitive on the set of all elements of any chosen type i. If we consider a chamber $\{c_1, c_2, ..., c_n\}$ such that c_i is of type i, we can look at the stabilizer G_i of c_i in G. The subgroups G_i are called *parabolic subgroups* of Γ. For a type i, transitivity of G on the elements of type i gives a correspondence between the cosets of the stabilizer G_i and the elements of type i in Γ. Two elements of Γ are incident if and only if the corresponding cosets have a nonempty intersection.

We now use the above observation to define an incidence structure from a group G together with a set of subgroups $\{G_1, G_2, ..., G_n\}$. The type set is $\{1, 2, ..., n\}$. By definition the elements of type i are the (right) cosets of the subgroup G_i. Two cosets are incident if and only if their intersection is not empty. This is an incidence structure which is not necessarily a geometry (see Chapter 3 for definitions). In order to check whether a coset incidence structure is indeed a geometry you can use the command `IsFlagTransitiveGeometry` which (in case it returns `true`) guarantees that the argument is a geometry.

13.1.1 IsCosetGeometry

▷ IsCosetGeometry (Category)

This category is a subcategory of `IsIncidenceGeometry`, and contains all coset geometries.

13.1.2 CosetGeometry

▷ CosetGeometry(`G`, `l`) (operation)

Returns: the coset incidence structure defined by the list `l` of subgroups of the group `G`.

`G` must be a group and `l` is a list of subgroups of `G`. The subgroups in `l` will be the *parabolic subgroups* of the *coset incidence structure* whose rank equals the length of `l`.

```
_____ Example _____
gap> g:=SymmetricGroup(5);
Sym( [ 1 .. 5 ] )
```

```
gap> g1:=Stabilizer(g,[1,2],OnSets);
Group([ (4,5), (3,5), (1,2)(4,5) ])
gap> g2:=Stabilizer(g,[1,2,3],OnSets);
Group([ (4,5), (2,3), (1,2,3) ])
gap> cg:=CosetGeometry(g,[g1,g2]);
CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) )
gap> p:=Random(ElementsOfIncidenceStructure(cg,1));
<element of type 1 of CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) )>
gap> q:=Random(ElementsOfIncidenceStructure(cg,2));
<element of type 2 of CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) )>
gap> IsIncident(p,q);
false
gap> IsIncident(p,p);
true
gap> ParabolicSubgroups(cg);
[ Group([ (4,5), (3,5), (1,2)(4,5) ]), Group([ (4,5), (2,3), (1,2,3) ]) ]
gap> Rank(cg) = Size(last);
true
gap> BorelSubgroup(cg);
Group([ (1,2), (4,5) ])
gap> AmbientGroup(cg);
Sym( [ 1 .. 5 ] )
```

13.1.3 IsIncident

▷ IsIncident(*ele1, ele2*)   (operation)

   Returns: true if and only if *ele1* and *ele2* are incident

   ele1 and *ele2* must be two elements in the same coset geometry.

13.1.4 ParabolicSubgroups

▷ ParabolicSubgroups(*cg*)   (operation)

   Returns: the list of parabolic subgroups defining the coset geometry *cg*

13.1.5 AmbientGroup

▷ AmbientGroup(*cg*)   (operation)

   Returns: the group used to define the coset geometry *cg*

   cg must be a coset geometry.

13.1.6 Borelsubgroup

▷ Borelsubgroup(*cg*)   (operation)

   Returns: the Borel subgroup of de geometry *cg*

   The Borel subgroup is equal to the stabilizer of a chamber. It corresponds to the intersection of all parabolic subgroups.

13.1.7 RandomElement

▷ RandomElement(*cg*)   (operation)

   Returns: a random element of *cg*

  cg must be a coset geometry.

13.1.8 RandomFlag

▷ RandomFlag(*cg*)   (operation)

   Returns: a random flag of *cg*

  cg must be a coset geometry.

13.1.9 RandomChamber

▷ RandomChamber(*cg*)   (operation)

   Returns: a random chamber of *cg*

  cg must be a coset geometry.

———————————————— Example ————————————————

```
gap> g:=SymmetricGroup(5);
Sym( [ 1 .. 5 ] )
gap> g1:=Stabilizer(g,[1,2],OnSets);
Group([ (4,5), (3,5), (1,2)(4,5) ])
gap> g2:=Stabilizer(g,[[1,2],[3,4]],OnSetsSets);
Group([ (1,2), (3,4), (1,3)(2,4) ])
gap> cg:=CosetGeometry(g,[g1,g2]);
CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) )
gap> RandomElement(cg);
<element of type 1 of CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) )>
gap> Display(last);
RightCoset(Group( [ (4,5), (3,5), (1,2)(4,5) ] ),(1,4,2,5,3))
gap> RandomFlag(cg);
<Flag of coset geometry < CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) ,
[ Group( [ (4,5), (3,5), (1,2)(4,5) ] ),
  Group( [ (1,2), (3,4), (1,3)(2,4) ] ) ] ) >>
gap> flg:=RandomFlag(cg);
<Flag of coset geometry < CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) ,
[ Group( [ (4,5), (3,5), (1,2)(4,5) ] ),
  Group( [ (1,2), (3,4), (1,3)(2,4) ] ) ] ) >>
gap> Type(flg);
[ 1 ]
gap> flg2:=RandomFlag(cg);
<Flag of coset geometry < CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) ,
[ Group( [ (4,5), (3,5), (1,2)(4,5) ] ),
  Group( [ (1,2), (3,4), (1,3)(2,4) ] ) ] ) >>
gap> Type(flg2);
[ 2 ]
gap> IsChamberOfIncidenceStructure(flg2);
false
gap> IsChamberOfIncidenceStructure(flg);
false
gap> Display(flg2);
Flag of coset geometry CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) ,
```

```
   [ Group( [ (4,5), (3,5), (1,2)(4,5) ] ),
     Group( [ (1,2), (3,4), (1,3)(2,4) ] ) ] ) with elements
   [ RightCoset(Group( [ (1,2), (3,4), (1,3)(2,4) ] ),(2,3,5)) ]
   gap> cham:=RandomChamber(cg);
   <Flag of coset geometry < CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) ,
   [ Group( [ (4,5), (3,5), (1,2)(4,5) ] ),
     Group( [ (1,2), (3,4), (1,3)(2,4) ] ) ] ) >>
   gap> IsChamberOfIncidenceStructure(cham);
   true
   gap> ElementsOfFlag(cham);
   [ <element of type 1 of CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) )>,
     <element of type 2 of CosetGeometry( SymmetricGroup( [ 1 .. 5 ] ) )> ]
   gap> IsIncident(last[1],last[2]);
   true
```

13.1.10 IsFlagTransitiveGeometry

▷ IsFlagTransitiveGeometry(*cg*)   (operation)

   Returns: true if and only if the group *G* defining *cg* acts flag-transitively.

   cg must be a coset geometry.

   The group *G* used to define *cg* acts naturally on the elements of *cg* by right translation: a coset $G_i g$ is mapped to $G_i(gx)$ by an element $x \in G$. This test can be quite time consuming. You can bind the attribute IsFlagTransitiveGeometry if you are sure the coset geometry is indeed flag-transitive.

```
───────────────────────────── Example ─────────────────────────────
   gap> g:=SymmetricGroup(4);
   Sym( [ 1 .. 4 ] )
   gap> g1:=Subgroup(g,[(1,2,3)]);
   Group([ (1,2,3) ])
   gap> g2:=Subgroup(g,[(1,4)]);
   Group([ (1,4) ])
   gap> g3:=Subgroup(g,[(1,2,3,4)]);
   Group([ (1,2,3,4) ])
   gap> cg:=CosetGeometry(g,[g1,g2,g3]);
   CosetGeometry( SymmetricGroup( [ 1 .. 4 ] ) )
   gap> IsFlagTransitiveGeometry(cg);
   false
   gap> cg2:=CosetGeometry(g,[g1,g2]);
   CosetGeometry( SymmetricGroup( [ 1 .. 4 ] ) )
   gap> IsFlagTransitiveGeometry(cg2);
   true
```

13.1.11 OnCosetGeometryElement

▷ OnCosetGeometryElement(*ele, g*)   (operation)

   Returns: the image of the CosetGeometryElement *ele* under the action of *g*

   The group element *g* must belong to AmbientGroup(AmbientGeometry(*ele*)).

13.1.12 \^

▷ \^(*ele, g*) (operation)

Returns: an element of a coset geometry

This is an operation which returns the image of *ele*, an element of a coset incidence structure, under *g*, an element of `AmbientGroup(AmbientGeometry(ele))`.

13.1.13 \^

▷ \^(*flg, g*) (operation)

Returns: a flag of a coset geometry

This is an operation which returns the image of *flg*, a flag of a coset incidence structure, under *g*, an element of `AmbientGroup(AmbientGeometry(flg))`.

13.1.14 IsFirmGeometry

▷ IsFirmGeometry(*cg*) (operation)

Returns: true if and only if *cg* is firm.

An incidence geometry is said to be *firm* if every nonmaximal flag is contained in at least two chambers. *cg* must be a coset geometry.

13.1.15 IsThickGeometry

▷ IsThickGeometry(*cg*) (operation)

Returns: true if and only if *cg* is thick.

An incidence geometry is said to be *thick* if every nonmaximal flag is contained in at least three chambers. *cg* must be a coset geometry.

13.1.16 IsThinGeometry

▷ IsThinGeometry(*cg*) (operation)

Returns: true if and only if *cg* is thin.

An incidence geometry is said to be *thin* if every rank one residue contains exactly 2 elements. This means that every comaximal flag is contained in exactly 2 chambers. *cg* must be a coset geometry.

```
_____ Example _____
gap> g:=SymmetricGroup(8);;
gap> pabs:=[];;
gap> pabs[1]:=Stabilizer(g,1);; pabs[2]:=Stabilizer(g,2);;
gap> pabs[3]:=Stabilizer(g,3);;
gap> pabs[4]:=Stabilizer(g,[1,2,3,4],OnSets);;
gap> pabs[5]:=Stabilizer(g,[1,2,3,4,5],OnSets);;
gap> pabs[6]:=Stabilizer(g,6);; pabs[7]:=Stabilizer(g,7);;
gap> cg:=CosetGeometry(g,pabs);
CosetGeometry( SymmetricGroup( [ 1 .. 8 ] ) )
gap> IsFirmGeometry(cg);
true
gap> IsThinGeometry(cg);
true
gap> IsThickGeometry(cg);
```

```
false
gap> truncation:=CosetGeometry(g,pabs{[1..5]});
CosetGeometry( SymmetricGroup( [ 1 .. 8 ] ) )
gap> IsFirmGeometry(truncation);
true
gap> IsThinGeometry(truncation);
false
gap> IsThickGeometry(truncation);
false
gap> truncation2:=CosetGeometry(g,pabs{[4,5]});
CosetGeometry( SymmetricGroup( [ 1 .. 8 ] ) )
gap> IsFirmGeometry(truncation2);
true
gap> IsThinGeometry(truncation2);
false
gap> IsThickGeometry(truncation2);
true
```

13.1.17 IsConnected

▷ IsConnected(*cg*) (operation)

Returns: true if and only if *cg* is connected.

A geometry is *connected* if and only if its incidence graph is connected. *cg* must be a coset geometry.

13.1.18 IsResiduallyConnected

▷ IsResiduallyConnected(*cg*) (operation)

Returns: true if and only if *cg* is residually connected.

A geometry is *residually connected* if the incidence graphs of all its residues of rank at least 2 are connected. *cg* must be a coset geometry.

This test is quite time consuming. You can bind the attribute IsResiduallyConnected if you are sure the coset geometry is indeed residually connected.

```
——————————————— Example ———————————————
gap> ps:=HyperbolicQuadric(7,2);
Q+(7, 2)
gap> g:=IsometryGroup(ps);;
gap> reps:=RepresentativesOfElements(ps);
[ <a point in Q+(7, 2)>, <a line in Q+(7, 2)>, <a plane in Q+(7, 2)>,
  <a solid in Q+(7, 2)> ]
gap> solids:=Orbit(g,reps[4]);;
gap> ps:=HyperbolicQuadric(7,2);
Q+(7, 2)
gap> g:=IsometryGroup(ps);;
gap> reps:=RepresentativesOfElements(ps);
[ <a point in Q+(7, 2)>, <a line in Q+(7, 2)>, <a plane in Q+(7, 2)>,
  <a solid in Q+(7, 2)> ]
gap> h:=DerivedSubgroup(g);; # to get greek and latin solids
gap> orbs:=FiningOrbits(h,Solids(ps));;
50%..100%..gap> List(orbs, Size);
```

```
[ 135, 135 ]
gap> Filtered(orbs[2], s -> ProjectiveDimension(Meet(orbs[1][1],s))=2); # to
[ <a solid in Q+(7, 2)>, <a solid in Q+(7, 2)>, <a solid in Q+(7, 2)>,
  <a solid in Q+(7, 2)>, <a solid in Q+(7, 2)>, <a solid in Q+(7, 2)>,
  <a solid in Q+(7, 2)>, <a solid in Q+(7, 2)>, <a solid in Q+(7, 2)>,
  <a solid in Q+(7, 2)>, <a solid in Q+(7, 2)>, <a solid in Q+(7, 2)>,
  <a solid in Q+(7, 2)>, <a solid in Q+(7, 2)>, <a solid in Q+(7, 2)> ]
gap> #find a latin incident with the greek which is orbs[1][1]
gap> # Now we have a chamber
gap> goodreps:=[reps[1],reps[2],orbs[1][1],last[1]];
[ <a point in Q+(7, 2)>, <a line in Q+(7, 2)>, <a solid in Q+(7, 2)>,
  <a solid in Q+(7, 2)> ]
gap> pabs:=List(goodreps, r -> FiningStabiliser(h,r));
[ <projective collineation group of size 1290240 with 2 generators>,
  <projective collineation group of size 110592 with 4 generators>,
  <projective collineation group of size 1290240 with 2 generators>,
  <projective collineation group of size 1290240 with 4 generators> ]
gap> cos:=CosetGeometry(h,pabs);
CosetGeometry( Group(
[ ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),8,[
    [ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ],
    [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ],
    [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2)
    ],]),IdentityMapping( GF(2) )), ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,
    1),8,[[ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2)
    ],]),IdentityMapping( GF(2) )), ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,
    1),8,[[ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
    [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0
    ],]),IdentityMapping( GF(2) )) ] ) )
gap> IsConnected(cos);
true
gap> IsResiduallyConnected(cos);
true
gap> time;
419960
```

13.1.19 StandardFlagOfCosetGeometry

▷ StandardFlagOfCosetGeometry(*cg*) (operation)

Returns: standard chamber of *cg*

The standard chamber just consists of all parabolic subgroups (i.e. the trivial cosets of these subgroups). The object returned is a FlagOfIncidenceStructure. *cg* must be a coset geometry.

13.1.20 FlagToStandardFlag

▷ FlagToStandardFlag(*cg*, *fl*) (operation)

Returns: element of the defining group of *cg* which maps *fl* to the standard chamber of *cg*.

fl must be a chamber given as a list of cosets of the parabolic subgroups of *cg*.

```
────────────────── Example ──────────────────
gap> L:=SimpleLieAlgebra("D",8,Rationals);
<Lie algebra of dimension 120 over Rationals>
gap> rs:=RootSystem(L);
<root system of rank 8>
gap> w:=WeylGroup(rs);
<matrix group with 8 generators>
gap> gens:=GeneratorsOfGroup(w);;
gap> pabs:=List(gens, g -> Group(Difference(gens, [g])));
[ <matrix group with 7 generators>, <matrix group with 7 generators>,
  <matrix group with 7 generators>, <matrix group with 7 generators>,
  <matrix group with 7 generators>, <matrix group with 7 generators>,
  <matrix group with 7 generators>, <matrix group with 7 generators> ]
gap> g:=Group(gens);
<matrix group with 8 generators>
gap> cg:=CosetGeometry(g,pabs);;
gap> cham:=RandomChamber(cg);; # Time of last command: 23945 ms
gap> FlagToStandardFlag(cg,cham); # Time of last command: 1720 ms
[ [ 0, 0, 0, 0, 1, -1, 0, 0 ], [ 0, 0, 0, 1, 0, -1, 0, 0 ],
  [ 0, 0, 0, 1, 0, 0, -1, -1 ], [ 1, -1, 0, 1, 0, 0, -1, -1 ],
  [ 0, -1, 0, 1, 0, 0, -1, -1 ], [ 0, -1, 0, 1, 0, 0, 0, -2 ],
  [ 0, -1, 1, 0, 0, 0, 0, -1 ], [ 0, -1, 0, 1, 0, 0, 0, -1 ] ]
gap> cham^last = StandardFlagOfCosetGeometry(cg); # Time of last command:1005 ms
true
```

13.1.21 CanonicalResidueOfFlag

▷ CanonicalResidueOfFlag(*cg*, *fl*) (operation)

Returns: coset geometry isomorphic to residue of *fl* in *cg*

cg must be a coset incidence structure and *fl* must be a flag in that incidence structure. The returned coset incidence structure for a flag $\{G_{i_1}g_{i_1}, G_{i_2}g_{i_2}, \ldots, G_{i_k}g_{i_k}\}$ is the coset incidence structure defined by the group $H := \cap_{j=1}^{k} G_{i_j}$ and parabolic subgroups $G_j \cap H$ for j not in the type set $\{i_1, i_2, \ldots, i_k\}$ of *fl*.

13.1.22 ResidueOfFlag

▷ ResidueOfFlag(*fl*) (operation)

   Returns: the residue of *fl* in AmbientGeometry(fl).

   This is a CosetGeometry method for the ResidueOfFlag operation given in Chapter 3. Note that the related operation CanonicalResidueOfFlag takes *two* arguments.

─────── Example ───────
```
gap> pg:=SymplecticSpace(5,2);
W(5, 2)
gap> pi:=Random(Planes(pg));
<a plane in W(5, 2)>
gap> l:=Random(Lines(pi));
<a line in W(5, 2)>
gap> p:=Random(Points(l));
<a point in W(5, 2)>
gap> g:=CollineationGroup(pg);
PGammaSp(6,2)
gap> g1:=Stabilizer(g,p);
<projective collineation group of size 23040 with 3 generators>
gap> g2:=Stabilizer(g,l);
<projective collineation group of size 4608 with 4 generators>
gap> g3:=Stabilizer(g,pi);
<projective collineation group of size 10752 with 3 generators>
gap> cg:=CosetGeometry(g, [g1,g2,g3]);
CosetGeometry( PGammaSp(6,2) )
gap> RandomFlag(cg); # Time of last command: 10745 ms
<Flag of coset geometry < CosetGeometry( PGammaSp(6,2) ,
[
  Group(
    [ ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
        [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
        [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
        [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2)
         ],]),IdentityMapping( GF(2) )),
      ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
        [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
        [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ],
        [ Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
        [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
        [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2)
         ],]),IdentityMapping( GF(2) )),
      ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
        [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0 ],
        [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ],
        [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
        [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2)
         ],]),IdentityMapping( GF(2) )) ] ),
```

```
Group(
  [ ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2)
      ],]),IdentityMapping( GF(2) )),
    ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
      [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ],
      [ Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ],
      [ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0
      ],]),IdentityMapping( GF(2) )),
    ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
      [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
      [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
      [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0
      ],]),IdentityMapping( GF(2) )),
    ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0 ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2)
      ],]),IdentityMapping( GF(2) )) ] ),
Group(
  [ ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2)
      ],]),IdentityMapping( GF(2) )),
    ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
      [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ],
      [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0
      ],]),IdentityMapping( GF(2) )),
    ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
      [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ],
```

```
        [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
        [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
        [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
        [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2)
          ],]),IdentityMapping( GF(2) )) ] ) ] ) >>
gap> Type(last);
[ ]
gap> ResidueOfFlag(last2);
CosetGeometry( PGammaSp(6,2) )
gap> Rank(last);
3
gap> NrElementsOfIncidenceStructure(last2,1);
63
gap> flg:=RandomFlag(cg);;
gap> can:=CanonicalResidueOfFlag(cg,flg);
CosetGeometry( Group( ... ) )
gap> Type(flg);
[ 1, 2 ]
gap> Rank(can);
1
gap> res:=ResidueOfFlag(flg);
CosetGeometry( Group(
[ ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,1),6,[
    [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0 ],
    [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
    [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ],
    [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2)
     ],]),IdentityMapping( GF(2) )), ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,
    1),6,[[ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
    [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0
     ],]),IdentityMapping( GF(2) )), ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,
    1),6,[[ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
    [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
    [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ],
    [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
    [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ],
    [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0
     ],]),IdentityMapping( GF(2) )), ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,
    1),6,[[ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
    [ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0 ],
    [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2) ],
    [ Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
    [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ],
    [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2)
     ],]),IdentityMapping( GF(2) )), ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,
    1),6,[[ 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ],
    [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ],
```

```
         [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ],
         [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
         [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
         [ Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2)
          ],]),IdentityMapping( GF(2) )), ProjElWithFrob(NewMatrix(IsCMatRep,GF(2,
         1),6,[[ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ],
         [ Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ],
         [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ],
         [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ],
         [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2), 0*Z(2) ],
         [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2)
          ],]),IdentityMapping( GF(2) )) ] ) )
gap> IsIsomorphicIncidenceStructureWithNauty(res,can);
#I Using NiceMonomorphism...
#I Using NiceMonomorphism...
true
```

13.1.23 IncidenceGraph

▷ IncidenceGraph(*cg*)   (operation)

   Returns: incidence graph of *cg*.

   cg must be a coset geometry. The graph returned is a **GRAPE** object. All **GRAPE** functionality can now be used to analyse *cg* via its incidence graph.

13.1.24 Rk2GeoGonality

▷ Rk2GeoGonality(*cg*)   (operation)

   Returns: the gonality (i.e. half the girth) of the incidence graph of *cg*.

   cg must be a coset geometry of rank 2.

13.1.25 Rk2GeoDiameter

▷ Rk2GeoDiameter(*cg, type*)   (operation)

   Returns: the point (or line) diameter.

   cg must be a coset geometry of rank 2. *type* must be either 1 or 2. This function computes the point diameter of *cg* when *type* is 1 and the line diameter when *type* is 2.

13.1.26 GeometryOfRank2Residue

▷ GeometryOfRank2Residue(*resi*)   (operation)

   Returns: the geometry of the Rank2Residue object *resi*.

   The rank 2 residues of a geometry are fundamental when dealing with diagrams. Therefore they are kept in an attribute as (a list of) objects of type Rank2Residue. The present operation just extracts the residue as a coset geometry from such a Rank2Residue object.

13.1.27 Rank2Parameters

▷ Rank2Parameters(*cg*)   (operation)

Returns: a list of length 3.

cg must be a coset geometry of rank 2. This function computes the gonality, point and line diameter of *cg*. These appear as a list in the first entry of the returned list. The second entry contains a list of length 2 with the point order and the total number of points (i.e. elements of type 1) in the geometry. The last entry contains the line order and the number of lines (i.e. elements of type 2).

The following example illustrates Rank2Parameters. It uses **AtlasRep** to fetch the second Janko group, also known as the Hall-Janko group. Beware that **AtlasRep** needs special write permissions on some systems. The constructed geometry has gonality 6 and both diameters equal to 8. It is known as the *Cohen-Tits near octagon*.

```
───────────────────────────── Example ─────────────────────────────
gap> LoadPackage("atlasrep");
true
gap> j2:=AtlasGroup("J2"); #Uses AtlasRep package
<permutation group of size 604800 with 2 generators>
gap> max3:=AtlasSubgroup(j2,3); #member of 3rd ATLAS class of max. subgps
<permutation group of size 1920 with 2 generators>
gap> max4:=AtlasSubgroup(j2,4); #member of 4th ATLAS class of max. subgps
<permutation group of size 1152 with 2 generators>
gap> conj3:=ConjugacyClassSubgroups(j2,max3);;
gap> g1:=First(conj3, c -> Size(Intersection(c,max4))=384);;
gap> g2:=max4;;
gap> cg:=CosetGeometry(j2,[g1,g2]);;
gap> Rank2Parameters(cg);
[ [ 6, 8, 8 ], [ 2, 315 ], [ 4, 525 ] ]
```

13.2 Automorphisms, Correlations and Isomorphisms

An *automorphism* of an incidence structure Γ is a permutation of the element set of Γ such that incidence is preserved and types are fixed (i.e. the type of the image of an element coincides with the type of that element). One way to compute the (full) automorphism group of Γ is to compute its incidence graph and then use the available nauty machinery to obtain the group.

13.2.1 AutGroupIncidenceStructureWithNauty

▷ AutGroupIncidenceStructureWithNauty(*cg*)   (operation)

Returns: permutation group isomorphic to the full automorphism group of *cg*.

The group is computed with nauty, which is part of **GRAPE** but has to be compiled on your system before use. The group returned is a permutation group acting on the set [1..Sum(TypesOfElementsOfIncidenceStructure(cg), t -> NrElementsOfIncidenceStructure(cg,t))], which is exactly the vertex set of IncidenceGraph(cg). At the moment the action of the automorphism group on cg is not provided but it can be recovered from the knowledge that the vertex set of IncidenceGraph(cg) first contains all elements of type 1 in cg, then all elements of type 2, etc. or, better still, with the **GRAPE** command VertexNames (see example below).

13.2.2   CorGroupIncidenceStructureWithNauty

▷ CorGroupIncidenceStructureWithNauty(*cg*)   (operation)

   Returns: permutation group isomorphic to the full automorphism group of *cg*.

   The group is computed with nauty, which is part of GRAPE but has to be compiled on your system before use. The group returned is a permutation group acting on the set [1..Sum(TypesOfElementsOfIncidenceStructure(cg), t -> NrElementsOfIncidenceStructure(cg,t))], which is exactly the vertex set of IncidenceGraph(cg). At the moment the action of the automorphism group on cg is not provided but it can be recovered from the knowledge that the vertex set of IncidenceGraph(cg) first contains all elements of type 1 in cg, then all elements of type 2, etc. or with the GRAPE command VertexNames.

```
                             ———— Example ————
  gap> g := PSL(2,11);;
  gap> g1 := Group([ (1,2,3)(4,8,12)(5,10,9)(6,11,7),
  > (1,2)(3,4)(5,12)(6,11)(7,10)(8,9) ]);;
  gap> g2 := Group([ (1,2,7)(3,9,4)(5,11,10)(6,8,12),
  > (1,2)(3,4)(5,12)(6,11)(7,10)(8,9) ]);;
  gap> g3 := Group([ (1,2,11)(3,8,7)(4,9,5)(6,10,12),
  > (1,2)(3,12)(4,11)(5,10)(6,9)(7,8) ]);;
  gap> g4 := Group([ (1,2,11)(3,8,7)(4,9,5)(6,10,12),
  > (1,2)(3,10)(4,9)(5,8)(6,7)(11,12) ]);;
  gap> cg:=CosetGeometry(g,[g1,g2,g3,g4]);
  CosetGeometry( Group( [ ( 3,11, 9, 7, 5)( 4,12,10, 8, 6),
    ( 1, 2, 8)( 3, 7, 9)( 4,10, 5)( 6,12,11) ] ) )
  gap> aut:=AutGroupIncidenceStructureWithNauty(cg);
  <permutation group with 4 generators>
  gap> StructureDescription(aut);
  "PSL(2,11)"
  gap> cor:=CorGroupIncidenceStructureWithNauty(cg);
  <permutation group with 5 generators>
  gap> StructureDescription(cor);
  "C2 x PSL(2,11)"
  gap> incgrph:=IncidenceGraph(cg);;
  gap> names:=VertexNames(incgrph);;
  gap> g:=Random(aut);
  (1,9,7,6,2,3,5,11,4,8,10)(12,13,15,17,14,19,22,16,18,21,20)(23,28,33,25,29,31,
  32,26,27,24,30)(34,44,38,41,42,35,43,39,40,36,37)
  gap> e:=RandomElement(cg);
  <element of type 3 of CosetGeometry( Group(
  [ ( 3,11, 9, 7, 5)( 4,12,10, 8, 6), ( 1, 2, 8)( 3, 7, 9)( 4,10, 5)( 6,12,11)
   ] ) )>
  gap> pos:=Position(names, e);
  26
  gap> names[pos^g];
  <element of type 3 of CosetGeometry( Group(
  [ ( 3,11, 9, 7, 5)( 4,12,10, 8, 6), ( 1, 2, 8)( 3, 7, 9)( 4,10, 5)( 6,12,11)
   ] ) )>
```

13.2.3 IsIsomorphicIncidenceStructureWithNauty

▷ IsIsomorphicIncidenceStructureWithNauty(*cg1*, *cg2*) (operation)

Returns: true iff cg1 and cg2 are isomorphic.

We use `nauty`, which is part of GRAPE but has to be compiled on your system before use. Isomorphism is tested (with `nauty`) after converting the coset geometries *cg1* and *cg2* to coloured graphs.

```
———————————————— Example ————————————————
gap> g:=SymmetricGroup(4); g1:=Subgroup(g,[(1,2,3)]);
Sym( [ 1 .. 4 ] )
Group([ (1,2,3) ])
gap> g2:=Subgroup(g,[(1,4)]); g3:=Subgroup(g,[(1,2,3,4)]);
Group([ (1,4) ])
Group([ (1,2,3,4) ])
gap> cg:=CosetGeometry(g,[g1,g2,g3]);
CosetGeometry( SymmetricGroup( [ 1 .. 4 ] ) )
gap> IsFlagTransitiveGeometry(cg);
false
gap> aut:=AutGroupIncidenceStructureWithNauty(cg);
<permutation group with 4 generators>
gap> Size(aut);
48
gap> Size(g);
24
gap> newg1:=Stabilizer(aut, 1);
Group([ (5,7)(6,8)(10,15)(11,12)(13,16)(14,18)(17,19)(21,25)(23,26), (3,6)
(4,5)(9,18)(10,16)(12,20)(13,17)(15,19)(21,22)(24,26) ])
gap> newg2:=Stabilizer(aut, NrElementsOfIncidenceStructure(cg,1) + 1);
Group([ (5,7)(6,8)(10,15)(11,12)(13,16)(14,18)(17,19)(21,25)(23,26), (1,3)
(2,4)(5,6)(7,8)(10,11)(12,15)(13,16)(14,17)(18,19)(21,25)(22,24)(23,26) ])
gap> newg3:=Stabilizer(aut, NrElementsOfIncidenceStructure(cg,1) +
>                           NrElementsOfIncidenceStructure(cg,2) + 1);
Group([ (1,3)(2,4)(5,8)(6,7)(10,12)(11,15)(14,19)(17,18)(22,24), (3,8)(4,7)
(9,14)(10,17)(11,20)(13,15)(16,19)(22,25)(23,24) ])
gap> newcg:=CosetGeometry(aut, [newg1, newg2, newg3]);
CosetGeometry( Group(
[ ( 5, 7)( 6, 8)(10,15)(11,12)(13,16)(14,18)(17,19)(21,25)(23,26),
  ( 3, 6)( 4, 5)( 9,18)(10,16)(12,20)(13,17)(15,19)(21,22)(24,26),
  ( 1, 2)( 3, 4)( 5, 6)( 7, 8)( 9,20)(10,17)(11,14)(12,18)(13,16)(15,19)
    (21,26)(22,24)(23,25), ( 1, 3)( 2, 4)( 5, 6)( 7, 8)(10,11)(12,15)(13,16)
    (14,17)(18,19)(21,25)(22,24)(23,26) ] ) )
gap> IsFlagTransitiveGeometry(newcg);
true
gap> IsIsomorphicIncidenceStructureWithNauty(cg, newcg);
true
```

13.3 Diagrams

The *diagram* of a flag-transitive incidence geometry is a schematic description of the structure of the geometry. It is based on the collection of rank 2 residues of the geometry.

Technically, the diagram is added to a `CosetGeometry` object as a mutable attribute. Also the list of rank 2 residues of the geometry is added as an attribute once these have been computed. This is done with the operations `Rank2Residues` (to add the attribute) and `MakeRank2Residue` (to actually compute the resiudes). These operations are not for everyday use and hence remain undocumented.

Since the geometry is flag-transitive, all chambers are equivalent. Let's fix a chamber $C = \{c_1, c_2, \ldots, c_n\}$, with c_i of type i. For each subset $\{i, j\}$ of size two in $I = \{1, 2, \ldots, n\}$ we take the residue of the flag $C \setminus \{c_i, c_j\}$. Flag transitivity ensures that *all* residues of type $\{i, j\}$ are isomorphic to each other. For each such residue, the structure is described by some parameters: the gonality and the point and line diameters. For each type i, we also define the i-order to be one less than the number of elements of type i in the residue of a(ny) flag of type $I \setminus \{i\}$. All this information is depicted in a *diagram* which is bascically a labelled graph with vertex set I and edges whenever the point diamater, the line diameter and the gonality are all greater than 2.

13.3.1 DiagramOfGeometry

▷ DiagramOfGeometry(*Gamma*) (operation)

Returns: the diagram of the geometry *Gamma*

Gamma must be a flag-transitive coset geometry.

The flag-transitivity is not tested by this operation because such a test can be time consuming. The command `IsFlagTransitiveGeometry` can be used to check flag-transitivity if needed.

13.3.2 GeometryOfDiagram

▷ GeometryOfDiagram(*diag*) (operation)

Returns: the geometry of which *diag* is the diagram

diag must be a diagram object.

13.3.3 DrawDiagram

▷ DrawDiagram(*diag, filename*) (operation)
▷ DrawDiagram(*diag, filename, vertexverbosity*) (operation)
▷ DrawDiagram(*diag, filename, vertexverbosity, edgeverbosity*) (operation)

Returns: does not return anything but writes a file *filename*.ps

diag must be a diagram. Writes a file *filename*.ps in the current directory with a pictorial version of the diagram. This command uses the graphviz package which is available from http://www.graphviz.org.

In case graphviz is not available on your system, you will get an friendly error message and a file *filename*.dot will be written. You can then compile this file later or ask a friend to help you. By default the diagram provides for each type i the i-order and the number of elements of type i. This behaviour can be changed by providing a *vertexverbosity* level. A value *2* results in no label under the vertices and a value *1* gives only the i-order. Any other potitive integer value yields the default behaviour. The default labels for the edges of the diagram use the standard convention that a $[g, dp, dl]$-gon with all three parameters equal is labelled only with the number g. Putting *edgeverbosity* equal to 2 puts no labels at all. This yields the so called "basic diagram" of the geometry. With *edgeverbosity* equal to any integer greater than 2 all labels contain girth and both diameters.

We illustrate the diagram feature with Neumaier's A_8-geometry. The affine space of dimension 3 over the field with two elements is denoted by $AG(3,2)$. If we fix a plane Π in $PG(3,2)$, the structure induced on the 8 points not in Π by the lines and planes of $PG(3,2)$ is isomorphic to $AG(3,2)$. Since every two points of $AG(3,2)$ define a line, the collinearity graph of $AG(3,2)$ (that is the graph whose vertices are the points of $AG(3,2)$ and in which two vertices are adjacent whenever they are collinear) is the complete graph K_8 on 8 vertices. Given two copies of the complete graph on 8 vertices, one can label the vertices of each of them with the numbers from 1 to 8. These labelings are always equivalent when the two copies are seen as graphs, but not if they are understood as models of the affine space. The reason is that an affine space has parallel lines and to be affinely equivalent, the labelings must be such that edges which were parallel in the first labeling remain parallel in the second labeling. In fact there are 15 affinely nonequivalent ways to label the vertices of K_8. The affine space has 14 planes of 4 points and there are 70 subsets of 4 elements in the vertex set of K_8. Each time we label K_8, there are 14 of the 70 sets of 4 elements which become planes of $AG(3,2)$. The remaining 4-subsets will be called *nonplanes* for that labeling. A well-known rank 4 geometry discovered by Neumaier in 1984 can be described using these concepts. This geometry is quite important since its residue of cotype 1 is the famous A_7-geometry which is known to be the only flag-transitive locally classical C_3-geometry which is not a polar space (see Aschbacher1984 for details). The Neumaier geometry can be constructed as follows. The elements of types 1 and 2 are the vertices and edges of the complete graph K_8, the elements of type 3 are the 4-subsets of the vertex set of K_8 and the elements of type 4 are the 15 nonequivalent labelings of K_8. Incidences are mostly the natural ones. A 4-subset is incident with a labeling of K_8 if it is the set of points of a nonplane in the model of $AG(3,2)$ defined by the labeling.

––––––––– Example –––––––––

```
 Alt( [ 1 .. 8 ] )
gap> pabs:= [
>   Group([ (2,4,6), (1,3,2)(4,8)(6,7) ]),
>   Group([ (1,6,7,8,4), (2,5)(3,4) ]),
>   Group([ (3,6)(7,8), (2,4,5), (1,5)(2,4), (2,4)(6,7), (6,8,7),
> (1,2)(4,5), (3,7)(6,8) ]),
>   Group([ (1,7,8,4)(2,5,3,6), (1,3)(2,6)(4,8)(5,7),
>        (1,8)(2,7)(3,4)(5,6), (1,3)(2,6)(4,7)(5,8) ]) ];
[ Group([ (2,4,6), (1,3,2)(4,8)(6,7) ]), Group([ (1,6,7,8,4), (2,5)(3,4) ]),
  Group([ (3,6)(7,8), (2,4,5), (1,5)(2,4), (2,4)(6,7), (6,8,7), (1,2)
  (4,5), (3,7)(6,8) ]), Group([ (1,7,8,4)(2,5,3,6), (1,3)(2,6)(4,8)
  (5,7), (1,5)(2,4)(3,7)(6,8), (1,8)(2,7)(3,4)(5,6), (1,3)(2,6)(4,7)(5,8) ]) ]
gap> cg:=CosetGeometry(g,pabs);
CosetGeometry( AlternatingGroup( [ 1 .. 8 ] ) )
gap> diag:=DiagramOfGeometry(cg);
< Diagram of CosetGeometry( AlternatingGroup( [ 1 .. 8 ] ) ,
[ Group( [ (2,4,6), (1,3,2)(4,8)(6,7) ] ),
  Group( [ (1,6,7,8,4), (2,5)(3,4) ] ),
  Group( [ (3,6)(7,8), (2,4,5), (1,5)(2,4), (2,4)(6,7), (6,8,7), (1,2)(4,5),
      (3,7)(6,8) ] ),
  Group( [ (1,7,8,4)(2,5,3,6), (1,3)(2,6)(4,8)(5,7), (1,5)(2,4)(3,7)(6,8),
      (1,8)(2,7)(3,4)(5,6), (1,3)(2,6)(4,7)(5,8) ] ) ] ) >
gap> DrawDiagram(diag, "neuma8");
gap> #Exec("gv neuma8.ps");
gap> point:=Random(ElementsOfIncidenceStructure(cg,1));
<element of type 1 of CosetGeometry( AlternatingGroup( [ 1 .. 8 ] ) )>
gap> residue:=ResidueOfFlag(FlagOfIncidenceStructure(cg,[point]));
```

```
  CosetGeometry( Group( [ (3,5,7), (1,7)(2,4,3)(5,8) ] ) )
gap> diagc3:=DiagramOfGeometry(residue);
< Diagram of CosetGeometry( Group( [ (3,5,7), (1,7)(2,4,3)(5,8) ] ) ,
[ Group( [ (4,5,8), (1,4,5), (1,7,8), (1,8,4,2,7) ] ),
  Group( [ (1,8)(4,7), (2,5,3), (1,7)(2,3), (1,7,8), (1,4)(7,8) ] ),
  Group( [ (1,5,4,3)(7,8), (2,4)(5,8) ] ) ] ) >
gap> DrawDiagram(diagc3, "a7geo");
gap> #Exec("gv a7geo.ps");
```

The produced diagrams are included here: Neumaier's A_8

```
      3 3 4          3          4
  o————————o————————o————————o

  1          2          2          2
  8          28         70         15
```
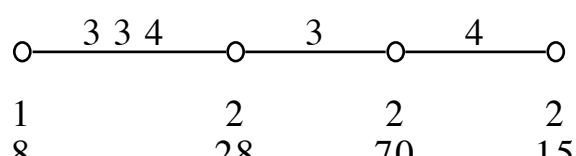
The A_7 geometry:
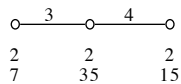
On a UNIX system we can start an external viewer ("gv" or ghostview in this case) from within GAP with the `Exec` command.

13.3.4 DrawDiagramWithNeato

▷ DrawDiagramWithNeato(*diag, filename*) (operation)

Returns: does not return anything but writes a file *filename*.ps

diag must be a diagram. Writes a file *filename*.ps in the current directory with a pictorial version of the diagram.

This command uses a "spring tension" algorithm to draw the diagram *diag* with straight edges. For some diagrams this looks better than the result of `DrawDiagram`. However this algorithm does not print the vertex labels.

This command uses the graphviz package which is available from http://www.graphviz.org. In case graphviz is not available on your system, you will get an friendly error message and a file *filename*.dot will be written. You can then compile this file later or ask a friend to help you. An E_6 geometry for comparison: on the left hand side we have the output of DrawDiagram and on the right

hand side we see the result of DrawDiagramWithNeato
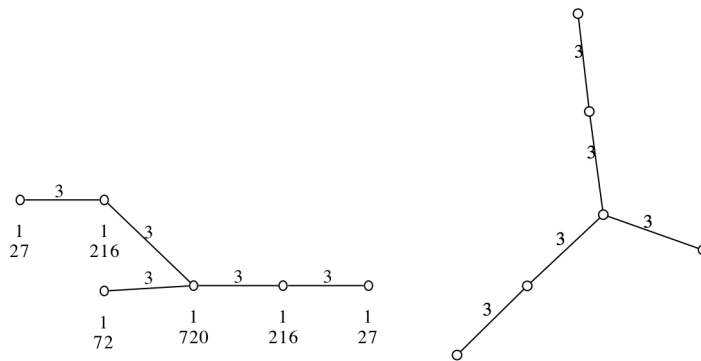
Appendix A

The structure of **FinInG**

A.1 The different components

Loading FinInG shows the following message:

```
――――――――――――――――――― Example ―――――――――――――――――――
 ----------------------------------------------------------------
  loading: geometry, liegeometry, group, projectivespace, correlations,
  polarspace/morphisms, enumerators, diagram, varieties, affinespace/affinegroup,
  gpolygons
```

The different components are listed and refer to the corresponding filenames. So *component* refers to *component.gd* and *component.gi*. When When *component1/component2* is displayed, Both *component1.gi* and *component2.gi* depend on the declarations in both *component1.gd* and *component2.gd*. In other cases, *component_n* is only dependent on its own declarations and the ones before.

A.2 The complete inventory

A.2.1 Declarations

```
――――――――――――――――――― Example ―――――――――――――――――――
  Operations

  geometry.gd: operations

  O: IncidenceStructure: [IsList, IsFunction, IsFunction, IsList]
  O: ResidueOfFlag: [IsFlagOfIncidenceStructure]
  O: ElementsOfIncidenceStructure: [IsIncidenceStructure]
  O: ElementsOfIncidenceStructure: [IsIncidenceStructure, IsPosInt]
  O: ElementsOfIncidenceStructure: [IsIncidenceStructure, IsString]
  O: NrElementsOfIncidenceStructure: [IsIncidenceStructure, IsPosInt]
  O: NrElementsOfIncidenceStructure: [IsIncidenceStructure, IsString]
  O: IncidenceGraph: [IsIncidenceStructure]
  O: Points: [IsIncidenceStructure]
  O: Lines: [IsIncidenceStructure]
  O: Planes: [IsIncidenceStructure]
  O: Solids: [IsIncidenceStructure]
  O: FlagOfIncidenceStructure: [IsIncidenceStructure, IsElementOfIncidenceStructureCollection]
  O: FlagOfIncidenceStructure: [IsIncidenceStructure, IsListandIsEmpty]
```

```
O: ChamberOfIncidenceStructure: [IsElementOfIncidenceStructureCollection]
O: ElementsOfFlag: [IsFlagOfIncidenceStructure]
O: IsIncident: [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure]
O: IsIncident: [IsElementOfIncidenceStructure, IsFlagOfIncidenceStructure]
O: IsIncident: [IsFlagOfIncidenceStructure, IsElementOfIncidenceStructure]
O: ShadowOfElement: [IsElementOfIncidenceStructure, IsPosInt]
O: IsCollinear: [IsIncidenceStructure, IsElementOfIncidenceStructure, IsElementOfIncidenceStruct
O: Span: [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure]
O: Meet: [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure]
O: Type: [IsElementOfIncidenceStructureandIsElementOfIncidenceStructureRep]
O: Type: [IsElementsOfIncidenceStructureandIsElementsOfIncidenceStructureRep]
O: Type: [IsFlagOfIncidenceStructureandIsFlagOfIncidenceStructureRep]
O: Wrap: [IsIncidenceStructure, IsPosInt, IsObject]
O: Unwrap: [IsElementOfIncidenceStructure]
O: ObjectToElement: [IsIncidenceStructure, IsPosInt, IsObject]
O: ObjectToElement: [IsIncidenceStructure, IsObject]
O: UnderlyingObject: [IsElementOfIncidenceStructure]
O: ShadowOfElement: [IsIncidenceStructure, IsElementOfIncidenceStructure, IsPosInt]
O: ShadowOfElement: [IsIncidenceStructure, IsElementOfIncidenceStructure, IsString]
O: ShadowOfFlag: [IsIncidenceStructure, IsFlagOfIncidenceStructure, IsPosInt]
O: ShadowOfFlag: [IsIncidenceStructure, IsFlagOfIncidenceStructure, IsString]
O: ShadowOfFlag: [IsIncidenceStructure, IsList, IsPosInt]
O: ShadowOfFlag: [IsIncidenceStructure, IsList, IsString]
O: ElementsIncidentWithElementOfIncidenceStructure: [IsElementOfIncidenceStructure, IsPosInt]
O: Points: [IsElementOfIncidenceStructure]
O: Lines: [IsElementOfIncidenceStructure]
O: Planes: [IsElementOfIncidenceStructure]
O: Solids: [IsElementOfIncidenceStructure]
O: Hyperplanes: [IsElementOfIncidenceStructure]
O: Points: [IsIncidenceStructure, IsElementOfIncidenceStructure]
O: Lines: [IsIncidenceStructure, IsElementOfIncidenceStructure]
O: Planes: [IsIncidenceStructure, IsElementOfIncidenceStructure]
O: Solids: [IsIncidenceStructure, IsElementOfIncidenceStructure]
O: Hyperplanes: [IsIncidenceStructure, IsElementOfIncidenceStructure]


liegeometry.gd: operations

O: UnderlyingVectorSpace: [IsLieGeometry]
O: UnderlyingVectorSpace: [IsElementOfLieGeometry]
O: UnderlyingVectorSpace: [IsFlagOfLieGeometry]
O: VectorSpaceToElement: [IsLieGeometry, IsRowVector]
O: VectorSpaceToElement: [IsLieGeometry, Is8BitVectorRep]
O: VectorSpaceToElement: [IsLieGeometry, IsPlistRep]
O: VectorSpaceToElement: [IsLieGeometry, Is8BitMatrixRep]
O: VectorSpaceToElement: [IsLieGeometry, IsGF2MatrixRep]
O: VectorSpaceToElement: [IsLieGeometry, IsCVecRep]
O: VectorSpaceToElement: [IsLieGeometry, IsCMatRep]
O: EmptySubspace: [IsLieGeometry]
O: RandomSubspace: [IsVectorSpace, IsInt]
O: IsIncident: [IsEmptySubspace, IsElementOfLieGeometry]
O: IsIncident: [IsElementOfLieGeometry, IsEmptySubspace]
O: IsIncident: [IsEmptySubspace, IsLieGeometry]
O: IsIncident: [IsLieGeometry, IsEmptySubspace]
```

```
O: IsIncident: [IsEmptySubspace, IsEmptySubspace]
O: Span: [IsEmptySubspace, IsElementOfLieGeometry]
O: Span: [IsElementOfLieGeometry, IsEmptySubspace]
O: Span: [IsEmptySubspace, IsLieGeometry]
O: Span: [IsLieGeometry, IsEmptySubspace]
O: Span: [IsEmptySubspace, IsEmptySubspace]
O: Meet: [IsEmptySubspace, IsElementOfLieGeometry]
O: Meet: [IsElementOfLieGeometry, IsEmptySubspace]
O: Meet: [IsEmptySubspace, IsLieGeometry]
O: Meet: [IsLieGeometry, IsEmptySubspace]
O: Meet: [IsEmptySubspace, IsEmptySubspace]
O: ElementToElement: [IsLieGeometry, IsElementOfLieGeometry]
O: ConvertElement: [IsLieGeometry, IsElementOfLieGeometry]
O: ConvertElementNC: [IsLieGeometry, IsElementOfLieGeometry]


group.gd: operations


O: FindBasePointCandidates: [IsGroup, IsRecord, IsInt]
O: FindBasePointCandidates: [IsGroup, IsRecord, IsInt, IsObject]
O: ProjEl: [IsMatrixandIsFFECollColl]
O: ProjEls: [IsList]
O: Projectivity: [IsList, IsField]
O: Projectivity: [IsProjectiveSpace, IsMatrix]
O: ProjElWithFrob: [IsMatrixandIsFFECollColl, IsMapping]
O: ProjElWithFrob: [IsMatrixandIsFFECollColl, IsMapping, IsField]
O: ProjElsWithFrob: [IsList]
O: ProjElsWithFrob: [IsList, IsField]
O: CollineationOfProjectiveSpace: [IsList, IsField]
O: CollineationOfProjectiveSpace: [IsList, IsMapping, IsField]
O: CollineationOfProjectiveSpace: [IsProjectiveSpace, IsMatrix]
O: CollineationOfProjectiveSpace: [IsProjectiveSpace, IsMatrix, IsMapping]
O: Collineation: [IsProjectiveSpace, IsMatrix]
O: Collineation: [IsProjectiveSpace, IsMatrix, IsMapping]
O: ProjectiveSemilinearMap: [IsList, IsMapping, IsField]
O: ProjectivityByImageOfStandardFrameNC: [IsProjectiveSpace, IsList]
O: MatrixOfCollineation: [IsProjGrpElWithFrobandIsProjGrpElWithFrobRep]
O: MatrixOfCollineation: [IsProjGrpElandIsProjGrpElRep]
O: FieldAutomorphism: [IsProjGrpElWithFrobandIsProjGrpElWithFrobRep]
O: ActionOnAllProjPoints: [IsProjectiveGroupWithFrob]
O: SetAsNiceMono: [IsProjectiveGroupWithFrob, IsGroupHomomorphism]
O: CanonicalGramMatrix: [IsString, IsPosInt, IsField]
O: CanonicalQuadraticForm: [IsString, IsPosInt, IsField]
O: SOdesargues: [IsInt, IsPosInt, IsFieldandIsFinite]
O: GOdesargues: [IsInt, IsPosInt, IsFieldandIsFinite]
O: SUdesargues: [IsPosInt, IsFieldandIsFinite]
O: GUdesargues: [IsPosInt, IsFieldandIsFinite]
O: Spdesargues: [IsPosInt, IsFieldandIsFinite]
O: GeneralSymplecticGroup: [IsPosInt, IsFieldandIsFinite]
O: GSpdesargues: [IsPosInt, IsFieldandIsFinite]
O: DeltaOminus: [IsPosInt, IsFieldandIsFinite]
O: DeltaOplus: [IsPosInt, IsFieldandIsFinite]
O: GammaOminus: [IsPosInt, IsFieldandIsFinite]
O: GammaO: [IsPosInt, IsFieldandIsFinite]
```

```
O: GammaOplus: [IsPosInt, IsFieldandIsFinite]
O: GammaU: [IsPosInt, IsFieldandIsFinite]
O: GammaSp: [IsPosInt, IsFieldandIsFinite]


projectivespace.gd: operations


O: ProjectiveSpace: [IsInt, IsField]
O: ProjectiveSpace: [IsInt, IsPosInt]
O: IsIncident: [IsSubspaceOfProjectiveSpace, IsProjectiveSpace]
O: IsIncident: [IsProjectiveSpace, IsSubspaceOfProjectiveSpace]
O: IsIncident: [IsProjectiveSpace, IsProjectiveSpace]
O: Hyperplanes: [IsProjectiveSpace]
O: BaerSublineOnThreePoints:  [ IsSubspaceOfProjectiveSpace,  IsSubspaceOfProjectiveSpace,  IsS
O: BaerSubplaneOnQuadrangle:  [ IsSubspaceOfProjectiveSpace,  IsSubspaceOfProjectiveSpace,
O: RandomSubspace: [IsProjectiveSpace, IsInt]
O: RandomSubspace: [IsSubspaceOfProjectiveSpace, IsInt]
O: RandomSubspace: [IsProjectiveSpace]
O: Span: [IsProjectiveSpace, IsSubspaceOfProjectiveSpace]
O: Span: [IsSubspaceOfProjectiveSpace, IsProjectiveSpace]
O: Span: [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace, IsBool]
O: Span: [IsList]
O: Span: [IsList, IsBool]
O: Meet: [IsSubspaceOfProjectiveSpace, IsProjectiveSpace]
O: Meet: [IsProjectiveSpace, IsSubspaceOfProjectiveSpace]
O: Meet: [IsList]
O: DualCoordinatesOfHyperplane: [IsSubspaceOfProjectiveSpace]
O: HyperplaneByDualCoordinates: [IsProjectiveSpace, IsList]
O: ComplementSpace: [IsVectorSpace, IsFFECollColl]
O: ElationOfProjectiveSpace: [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace, IsSubspa
O: ProjectiveElationGroup: [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace]
O: ProjectiveElationGroup: [IsSubspaceOfProjectiveSpace]
O: HomologyOfProjectiveSpace: [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace, IsSubsp
O: ProjectiveHomologyGroup: [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace]


correlations.gd: operations


O: StandardDualityOfProjectiveSpace: [IsProjectiveSpace]
O: IdentityMappingOfElementsOfProjectiveSpace: [IsProjectiveSpace]
O: ActionOnAllPointsHyperplanes: [IsProjGroupWithFrobWithPSIsom]
O: ProjElWithFrobWithPSIsom:   [IsMatrix and IsFFECollColl, IsMapping, IsField]
O: ProjElWithFrobWithPSIsom:   [IsMatrix and IsFFECollColl, IsMapping, IsField,   IsStandard
O: ProjElWithFrobWithPSIsom:   [IsMatrix and IsFFECollColl, IsMapping, IsField,   IsGeneralM
O: ProjElsWithFrobWithPSIsom: [IsList, IsField]
O: SetAsNiceMono:                [IsProjGroupWithFrobWithPSIsom, IsGroupHomomorphism]
O: CorrelationOfProjectiveSpace: [IsList, IsField]
O: CorrelationOfProjectiveSpace: [IsList, IsMapping, IsField]
O: CorrelationOfProjectiveSpace: [IsList, IsField, IsStandardDualityOfProjectiveSpace]
O: CorrelationOfProjectiveSpace: [IsList, IsField, IsIdentityMappingOfElementsOfProjectiveSpace]
O: CorrelationOfProjectiveSpace: [IsList, IsMapping, IsField, IsStandardDualityOfProjectiveSpace
O: CorrelationOfProjectiveSpace: [IsList, IsMapping, IsField, IsIdentityMappingOfElementsOfProje
O: CorrelationOfProjectiveSpace: [IsProjectiveSpace, IsMatrix, IsMapping, IsStandardDualityOfPro
O: CorrelationOfProjectiveSpace: [IsProjectiveSpace, IsMatrix, IsMapping, IsIdentityMappingOfEle
O: Correlation: [IsProjectiveSpace, IsMatrix, IsMapping, IsStandardDualityOfProjectiveSpace]
```

```
O: Correlation: [IsProjectiveSpace, IsMatrix, IsMapping, IsIdentityMappingOfElementsOfProjective
O: MatrixOfCorrelation: [IsProjGrpElWithFrobWithPSIsomandIsProjGrpElWithFrobWithPSIsomRep]
O: FieldAutomorphism: [IsProjGrpElWithFrobWithPSIsomandIsProjGrpElWithFrobWithPSIsomRep]
O: ProjectiveSpaceIsomorphism: [IsProjGrpElWithFrobWithPSIsomandIsProjGrpElWithFrobWithPSIsomRep
O: PolarityOfProjectiveSpaceOp: [IsForm]
O: PolarityOfProjectiveSpace: [IsForm]
O: PolarityOfProjectiveSpace: [IsMatrix, IsFieldandIsFinite]
O: PolarityOfProjectiveSpace: [IsMatrix, IsFrobeniusAutomorphism, IsFieldandIsFinite]
O: HermitianPolarityOfProjectiveSpace: [IsMatrix, IsFieldandIsFinite]
O: PolarityOfProjectiveSpace: [IsClassicalPolarSpace]
O: BaseField: [IsPolarityOfProjectiveSpace]
O: IsAbsoluteElement: [IsElementOfIncidenceStructure, IsPolarityOfProjectiveSpace]
O: GeometryOfAbsolutePoints: [IsPolarityOfProjectiveSpace]
O: AbsolutePoints: [IsPolarityOfProjectiveSpace]
O: PolarSpace: [IsPolarityOfProjectiveSpace]


polarspace.gd: operations


O: PolarSpaceStandard: [IsForm, IsBool]
O: PolarSpace: [IsForm, IsField, IsGroup, IsFunction]
O: PolarSpace: [IsForm]
O: PolarMap: [IsClassicalPolarSpace]
O: TangentSpace: [IsSubspaceOfClassicalPolarSpace]
O: TangentSpace: [IsClassicalPolarSpace, IsSubspaceOfProjectiveSpace]
O: Pole: [IsClassicalPolarSpace, IsSubspaceOfProjectiveSpace]
O: TypeOfSubspace: [IsClassicalPolarSpace, IsSubspaceOfProjectiveSpace]
O: CanonicalOrbitRepresentativeForSubspaces: [IsString, IsPosInt, IsField]
O: RandomSubspace: [IsClassicalPolarSpace, IsPosInt]
O: NumberOfTotallySingularSubspaces: [IsClassicalPolarSpace, IsPosInt]
O: EllipticQuadric: [IsPosInt, IsField]
O: EllipticQuadric: [IsPosInt, IsPosInt]
O: SymplecticSpace: [IsPosInt, IsField]
O: SymplecticSpace: [IsPosInt, IsPosInt]
O: ParabolicQuadric: [IsPosInt, IsField]
O: ParabolicQuadric: [IsPosInt, IsPosInt]
O: HyperbolicQuadric: [IsPosInt, IsField]
O: HyperbolicQuadric: [IsPosInt, IsPosInt]
O: HermitianPolarSpace: [IsPosInt, IsField]
O: HermitianPolarSpace: [IsPosInt, IsPosInt]
O: CanonicalPolarSpace: [IsClassicalPolarSpace]
O: StandardPolarSpace: [IsClassicalPolarSpace]
O: Span: [IsSubspaceOfClassicalPolarSpace, IsSubspaceOfClassicalPolarSpace, IsBool]


morphisms.gd: operations


O: GeometryMorphismByFunction:    [ IsAnyElementsOfIncidenceStructure,  IsAnyElementsOfIncidenceS
O: GeometryMorphismByFunction:    [ IsAnyElementsOfIncidenceStructure,  IsAnyElementsOfIncidenceS
O: GeometryMorphismByFunction:    [ IsAnyElementsOfIncidenceStructure,  IsAnyElementsOfIncidenceS
O: IsomorphismPolarSpacesProjectionFromNucleus: [IsClassicalPolarSpace, IsClassicalPolarSpace, I
O: IsomorphismPolarSpacesNC:  [ IsClassicalPolarSpace,  IsClassicalPolarSpace,  IsBool ]
O: IsomorphismPolarSpacesNC:                       [ IsClassicalPolarSpace,  IsClassicalPolarSpac
O: IsomorphismPolarSpaces:                         [ IsClassicalPolarSpace,  IsClassicalPolarSpace,
O: IsomorphismPolarSpaces:                         [ IsClassicalPolarSpace,  IsClassicalPolarSpace
```

```
O: NaturalEmbeddingBySubspace:                          [ IsLieGeometry, IsLieGeometry, IsSubspace
O: NaturalEmbeddingBySubspaceNC:                         [ IsLieGeometry, IsLieGeometry, IsSubspa
O: NaturalProjectionBySubspace:                         [ IsClassicalPolarSpace, IsSubspaceOfClass
O: NaturalProjectionBySubspace:                         [ IsProjectiveSpace, IsSubspaceOfProjectiv
O: NaturalProjectionBySubspaceNC:                        [ IsClassicalPolarSpace, IsSubspaceOfCla
O: NaturalProjectionBySubspaceNC:                        [ IsProjectiveSpace, IsSubspaceOfProject
O: ShrinkMat: [IsBasis, IsMatrix]
O: ShrinkMat: [IsField, IsField, IsVector]
O: ShrinkVec: [IsField, IsField, IsVector]
O: ShrinkVec: [IsField, IsField, IsVector, IsBasis]
O: BlownUpProjectiveSpace: [IsBasis, IsProjectiveSpace]
O: BlownUpProjectiveSpaceBySubfield: [IsField, IsProjectiveSpace]
O: BlownUpSubspaceOfProjectiveSpace: [IsBasis, IsSubspaceOfProjectiveSpace]
O: BlownUpSubspaceOfProjectiveSpaceBySubfield: [IsField, IsSubspaceOfProjectiveSpace]
O: IsDesarguesianSpreadElement: [IsBasis, IsSubspaceOfProjectiveSpace]
O: IsBlownUpSubspaceOfProjectiveSpace: [IsBasis, IsSubspaceOfProjectiveSpace]
O: NaturalEmbeddingByFieldReduction:                     [ IsProjectiveSpace, IsField, IsBasi
O: NaturalEmbeddingByFieldReduction:                     [ IsProjectiveSpace, IsField ]
O: NaturalEmbeddingByFieldReduction:                     [ IsProjectiveSpace, IsProjectiveSpac
O: NaturalEmbeddingByFieldReduction:                     [ IsProjectiveSpace, IsProjectiveSpac
O: BilinearFormFieldReduction: [IsBilinearForm, IsField, IsFFE, IsBasis]
O: QuadraticFormFieldReduction: [IsQuadraticForm, IsField, IsFFE, IsBasis]
O: HermitianFormFieldReduction: [IsHermitianForm, IsField, IsFFE, IsBasis]
O: BilinearFormFieldReduction: [IsBilinearForm, IsField, IsFFE]
O: QuadraticFormFieldReduction: [IsQuadraticForm, IsField, IsFFE]
O: HermitianFormFieldReduction: [IsHermitianForm, IsField, IsFFE]
O: NaturalEmbeddingByFieldReduction: [IsClassicalPolarSpace, IsField, IsFFE, IsBasis, IsBool]
O: NaturalEmbeddingByFieldReduction: [IsClassicalPolarSpace, IsField, IsFFE, IsBasis]
O: NaturalEmbeddingByFieldReduction: [IsClassicalPolarSpace, IsField, IsFFE, IsBool]
O: NaturalEmbeddingByFieldReduction: [IsClassicalPolarSpace, IsField, IsFFE]
O: NaturalEmbeddingByFieldReduction: [IsClassicalPolarSpace, IsField, IsBool]
O: NaturalEmbeddingByFieldReduction: [IsClassicalPolarSpace, IsField]
O: NaturalEmbeddingByFieldReduction: [IsClassicalPolarSpace, IsClassicalPolarSpace, IsBool]
O: NaturalEmbeddingByFieldReduction: [IsClassicalPolarSpace, IsClassicalPolarSpace]
O: CanonicalEmbeddingByFieldReduction: [ IsClassicalPolarSpace, IsField, IsBool ]
O: CanonicalEmbeddingByFieldReduction: [ IsClassicalPolarSpace, IsClassicalPolarSpace, IsBool
O: NaturalEmbeddingBySubfield:                          [ IsProjectiveSpace, IsProjectiveSpace ]
O: NaturalEmbeddingBySubfield:  [ IsClassicalPolarSpace, IsClassicalPolarSpace, IsBool ]
O: NaturalEmbeddingBySubfield:                          [ IsClassicalPolarSpace, IsClassicalPolarSp
O: PluckerCoordinates: [IsMatrix]
O: InversePluckerCoordinates: [IsVector]
O: PluckerCoordinates: [IsSubspaceOfProjectiveSpace]
O: KleinCorrespondence: [IsField, IsBool]
O: KleinCorrespondence: [IsField]
O: KleinCorrespondence: [IsPosInt, IsBool]
O: KleinCorrespondence: [IsPosInt]
O: KleinCorrespondence: [IsClassicalPolarSpace, IsBool]
O: KleinCorrespondence: [IsClassicalPolarSpace]
O: KleinCorrespondenceExtended: [IsField, IsBool]
O: KleinCorrespondenceExtended: [IsField]
O: KleinCorrespondenceExtended: [IsPosInt, IsBool]
O: KleinCorrespondenceExtended: [IsPosInt]
O: KleinCorrespondenceExtended: [IsClassicalPolarSpace, IsBool]
```

```
O: KleinCorrespondenceExtended: [IsClassicalPolarSpace]
O: NaturalDualitySymplectic: [IsClassicalGQ, IsClassicalGQ, IsBool, IsBool]
O: NaturalDualityHermitian: [IsClassicalGQ, IsClassicalGQ, IsBool, IsBool]
O: SelfDualitySymplectic: [IsClassicalGQ, IsBool]
O: SelfDualityParabolic: [IsClassicalGQ, IsBool]
O: NaturalDuality: [IsClassicalGQ, IsClassicalGQ, IsBool]
O: NaturalDuality: [IsClassicalGQ, IsClassicalGQ]
O: NaturalDuality: [IsClassicalGQ, IsBool]
O: NaturalDuality: [IsClassicalGQ]
O: SelfDuality: [IsClassicalGQ, IsBool]
O: SelfDuality: [IsClassicalGQ]
O: ProjectiveCompletion: [IsAffineSpace]


enumerators.gd: operations


O: AntonEnumerator: [IsSubspacesOfClassicalPolarSpace]
O: EnumeratorByOrbit: [IsSubspacesOfClassicalPolarSpace]


diagram.gd: operations


O: CosetGeometry: [IsGroup, IsHomogeneousList]
O: ParabolicSubgroups: [IsCosetGeometry]
O: AmbientGroup: [IsCosetGeometry]
O: FlagToStandardFlag: [IsCosetGeometry, IsFlagOfCosetGeometry]
O: ResidueOfFlag: [IsFlagOfCosetGeometry]
O: CanonicalResidueOfFlag: [IsCosetGeometry, IsFlagOfCosetGeometry]
O: RandomElement: [IsCosetGeometry]
O: RandomFlag: [IsCosetGeometry]
O: RandomChamber: [IsCosetGeometry]
O: AutGroupIncidenceStructureWithNauty: [IsCosetGeometry]
O: CorGroupIncidenceStructureWithNauty: [IsCosetGeometry]
O: IsIsomorphicIncidenceStructureWithNauty: [IsCosetGeometry, IsCosetGeometry]
O: Rk2GeoDiameter: [IsCosetGeometry, IsPosInt]
O: Rk2GeoGonality: [IsCosetGeometry]
O: GeometryOfRank2Residue: [IsRank2Residue]
O: GeometryFromLabelledGraph: [IsObjectandIS_REC]
O: Rank2Residues: [IsIncidenceGeometry]
O: MakeRank2Residue: [IsRank2Residue]


varieties.gd: operations


O: AlgebraicVariety: [IsProjectiveSpace, IsList]
O: AlgebraicVariety: [IsAffineSpace, IsList]
O: AlgebraicVariety: [IsProjectiveSpace, IsPolynomialRing, IsList]
O: AlgebraicVariety: [IsAffineSpace, IsPolynomialRing, IsList]
O: PointsOfAlgebraicVariety: [IsAlgebraicVariety]
O: Points: [IsAlgebraicVariety]
O: ProjectiveVariety: [IsProjectiveSpace, IsPolynomialRing, IsList]
O: ProjectiveVariety: [IsProjectiveSpace, IsList]
O: HermitianVariety: [IsPosInt, IsField]
O: HermitianVariety: [IsPosInt, IsPosInt]
O: HermitianVariety: [IsProjectiveSpace, IsPolynomialRing, IsPolynomial]
O: HermitianVariety: [IsProjectiveSpace, IsPolynomial]
```

```
O: QuadraticVariety: [IsPosInt, IsField]
O: QuadraticVariety: [IsPosInt, IsField, IsString]
O: QuadraticVariety: [IsPosInt, IsPosInt]
O: QuadraticVariety: [IsPosInt, IsPosInt, IsString]
O: QuadraticVariety: [IsProjectiveSpace, IsPolynomialRing, IsPolynomial]
O: QuadraticVariety: [IsProjectiveSpace, IsPolynomial]
O: PolarSpace: [IsProjectiveVariety]
O: AffineVariety: [IsAffineSpace, IsPolynomialRing, IsList]
O: AffineVariety: [IsAffineSpace, IsList]
O: SegreMap: [IsHomogeneousList]
O: SegreMap: [IsHomogeneousList, IsField]
O: SegreVariety: [IsHomogeneousList]
O: SegreVariety: [IsHomogeneousList, IsField]
O: PointsOfSegreVariety: [IsSegreVariety]
O: SegreMap: [IsSegreVariety]
O: SegreMap: [IsProjectiveSpace, IsProjectiveSpace]
O: SegreMap: [IsPosInt, IsPosInt, IsField]
O: SegreMap: [IsPosInt, IsPosInt, IsPosInt]
O: SegreVariety: [IsProjectiveSpace, IsProjectiveSpace]
O: SegreVariety: [IsPosInt, IsPosInt, IsField]
O: SegreVariety: [IsPosInt, IsPosInt, IsPosInt]
O: VeroneseMap: [IsProjectiveSpace]
O: VeroneseMap: [IsPosInt, IsField]
O: VeroneseMap: [IsPosInt, IsPosInt]
O: VeroneseVariety: [IsProjectiveSpace]
O: VeroneseVariety: [IsPosInt, IsField]
O: VeroneseVariety: [IsPosInt, IsPosInt]
O: PointsOfVeroneseVariety: [IsVeroneseVariety]
O: VeroneseMap: [IsVeroneseVariety]
O: GrassmannCoordinates: [IsSubspaceOfProjectiveSpace]
O: GrassmannMap: [IsPosInt, IsProjectiveSpace]
O: GrassmannMap: [IsPosInt, IsPosInt, IsPosInt]
O: GrassmannMap: [IsSubspacesOfProjectiveSpace]
O: GrassmannMap: [IsGrassmannVariety]
O: GrassmannVariety: [IsPosInt, IsProjectiveSpace]
O: GrassmannVariety: [IsPosInt, IsPosInt, IsField]
O: GrassmannVariety: [IsPosInt, IsPosInt, IsPosInt]
O: GrassmannVariety: [IsSubspacesOfProjectiveSpace]
O: PointsOfGrassmannVariety: [IsGrassmannVariety]
O: ConicOnFivePoints: [IsHomogeneousListand                    IsSubspaceOfProjectiveS

affinespace.gd: operations

O: VectorSpaceTransversal: [IsVectorSpace, IsFFECollColl]
O: VectorSpaceTransversalElement: [IsVectorSpace, IsFFECollColl, IsVector]
O: AffineSpace: [IsPosInt, IsField]
O: AffineSpace: [IsPosInt, IsPosInt]
O: Hyperplanes: [IsAffineSpace]
O: AffineSubspace: [IsAffineSpace, IsRowVector]
O: AffineSubspace: [IsAffineSpace, IsCVecRep]
O: AffineSubspace: [IsAffineSpace, IsRowVector, IsPlistRep]
O: AffineSubspace: [IsAffineSpace, IsRowVector, Is8BitMatrixRep]
O: AffineSubspace: [IsAffineSpace, IsRowVector, IsGF2MatrixRep]
```

```
O: AffineSubspace: [IsAffineSpace, IsCVecRep, IsCMatRep]
O: RandomSubspace: [IsAffineSpace, IsInt]
O: IsParallel: [IsSubspaceOfAffineSpace, IsSubspaceOfAffineSpace]
O: UnderlyingVectorSpace: [IsAffineSpace]
O: ParallelClass: [IsAffineSpace, IsSubspaceOfAffineSpace]
O: ParallelClass: [IsSubspaceOfAffineSpace]


affinegroup.gd: operations


gpolygons.gd: operations

O: GeneralisedPolygonByBlocks: [IsHomogeneousList]
O: GeneralisedPolygonByIncidenceMatrix: [IsMatrix]
O: GeneralisedPolygonByElements: [IsSet, IsSet, IsFunction]
O: GeneralisedPolygonByElements: [IsSet, IsSet, IsFunction, IsGroup, IsFunction]
O: DistanceBetweenElements: [IsElementOfGeneralisedPolygon, IsElementOfGeneralisedPolygon]
O: DistanceBetweenElements: [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace]
O: BlockDesignOfGeneralisedPolygon: [IsGeneralisedPolygon]
O: SplitCayleyHexagon: [IsFieldandIsFinite]
O: SplitCayleyHexagon: [IsPosInt]
O: SplitCayleyHexagon: [IsClassicalPolarSpace]
O: TwistedTrialityHexagon: [IsFieldandIsFinite]
O: TwistedTrialityHexagon: [IsPosInt]
O: TwistedTrialityHexagon: [IsClassicalPolarSpace]
O: G2fining: [IsPosInt, IsFieldandIsFinite]
O: 3D4fining: [IsFieldandIsFinite]
O: IsKantorFamily: [IsGroup, IsList, IsList]
O: EGQByKantorFamily: [IsGroup, IsList, IsList]
O: Wrap: [IsElationGQByKantorFamily, IsPosInt, IsPosInt, IsObject]
O: IsAnisotropic: [IsFFECollColl, IsFieldandIsFinite]
O: IsqClan: [IsFFECollCollColl, IsFieldandIsFinite]
O: qClan: [IsFFECollCollColl, IsField]
O: LinearqClan: [IsPosInt]
O: FisherThasWalkerKantorBettenqClan: [IsPosInt]
O: KantorMonomialqClan: [IsPosInt]
O: KantorKnuthqClan: [IsPosInt]
O: FisherqClan: [IsPosInt]
O: BLTSetByqClan: [IsqClanObjandIsqClanRep]
O: KantorFamilyByqClan: [IsqClanObjandIsqClanRep]
O: EGQByqClan: [IsqClanObjandIsqClanRep]
O: EGQByBLTSet: [IsList, IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace]
O: EGQByBLTSet: [IsList]
O: FlockGQByqClan: [IsqClanObj]
```

───────────────── Example ─────────────────

```
Attributes


geometry.gd: attributes


A: IsChamberOfIncidenceStructure: IsFlagOfIncidenceStructure
A: IsEmptyFlag: IsFlagOfIncidenceStructure
```

```
A: RankAttr: IsIncidenceStructure
A: RankAttr: IsFlagOfIncidenceStructure
A: TypesOfElementsOfIncidenceStructure: IsIncidenceStructure
A: TypesOfElementsOfIncidenceStructurePlural: IsIncidenceStructure
A: CollineationGroup: IsIncidenceStructure
A: CorrelationCollineationGroup: IsIncidenceStructure
A: CollineationAction: IsIncidenceStructure
A: CorrelationAction: IsIncidenceStructure
A: RepresentativesOfElements: IsIncidenceStructure
A: AmbientGeometry: IsIncidenceStructure
A: AmbientGeometry: IsFlagOfIncidenceStructure
A: Size: IsFlagOfIncidenceStructure
A: AmbientGeometry: IsElementOfIncidenceStructureandIsElementOfIncidenceStructureRep
A: AmbientGeometry: IsElementsOfIncidenceStructureandIsElementsOfIncidenceStructureRep
A: AmbientGeometry: IsAllElementsOfIncidenceStructure
A: CollineationAction: IsGroup


liegeometry.gd: attributes


A: AmbientSpace: IsLieGeometry
A: AmbientSpace: IsElementOfLieGeometry
A: ProjectiveDimension: IsLieGeometry
A: ProjectiveDimension: IsElementOfLieGeometry
A: ProjectiveDimension: IsEmptySubspace
A: Dimension: IsLieGeometry


group.gd: attributes


A: Dimension: IsProjectiveGroupWithFrob


projectivespace.gd: attributes


A: ProjectivityGroup: IsProjectiveSpace
A: SpecialProjectivityGroup: IsProjectiveSpace
A: Dimension: IsSubspaceOfProjectiveSpace
A: Dimension: IsEmpty
A: Coordinates: IsSubspaceOfProjectiveSpace
A: CoordinatesOfHyperplane: IsSubspaceOfProjectiveSpace
A: EquationOfHyperplane: IsSubspaceOfProjectiveSpace
A: StandardFrame: IsProjectiveSpace
A: StandardFrame: IsSubspaceOfProjectiveSpace


correlations.gd: attributes


A: Dimension: IsProjGroupWithFrobWithPSIsom
A: GramMatrix: IsPolarityOfProjectiveSpace
A: CompanionAutomorphism: IsPolarityOfProjectiveSpace
A: SesquilinearForm: IsPolarityOfProjectiveSpace


polarspace.gd: attributes


A: SesquilinearForm: IsClassicalPolarSpace
A: QuadraticForm: IsClassicalPolarSpace
```

```
A: AmbientSpace: IsClassicalPolarSpace
A: SimilarityGroup: IsClassicalPolarSpace
A: IsometryGroup: IsClassicalPolarSpace
A: SpecialIsometryGroup: IsClassicalPolarSpace
A: IsomorphismCanonicalPolarSpace: IsClassicalPolarSpace
A: IsomorphismCanonicalPolarSpaceWithIntertwiner: IsClassicalPolarSpace
A: IsCanonicalPolarSpace: IsClassicalPolarSpace
A: PolarSpaceType: IsClassicalPolarSpace
A: CompanionAutomorphism: IsClassicalPolarSpace
A: ClassicalGroupInfo: IsClassicalPolarSpace
A: EquationForPolarSpace: IsClassicalPolarSpace
A: NucleusOfParabolicQuadric: IsClassicalPolarSpace


morphisms.gd: attributes


A: Intertwiner: IsGeometryMorphism


enumerators.gd: attributes



diagram.gd: attributes


A: DiagramOfGeometry: IsIncidenceGeometry
A: IsFlagTransitiveGeometry: IsIncidenceGeometry
A: IsResiduallyConnected: IsIncidenceGeometry
A: IsConnected: IsIncidenceGeometry
A: IsFirmGeometry: IsIncidenceGeometry
A: IsThinGeometry: IsIncidenceGeometry
A: IsThickGeometry: IsIncidenceGeometry
A: BorelSubgroup: IsCosetGeometry
A: StandardFlagOfCosetGeometry: IsCosetGeometry
A: Rank2Parameters: IsCosetGeometry
A: OrderVertex: IsVertexOfDiagram
A: NrElementsVertex: IsVertexOfDiagram
A: StabiliserVertex: IsVertexOfDiagram
A: ResidueLabelForEdge: IsEdgeOfDiagram
A: GirthEdge: IsEdgeOfDiagram
A: PointDiamEdge: IsEdgeOfDiagram
A: LineDiamEdge: IsEdgeOfDiagram
A: ParametersEdge: IsEdgeOfDiagram
A: GeometryOfDiagram: IsDiagram


varieties.gd: attributes


A: DefiningListOfPolynomials: IsAlgebraicVariety
A: AmbientSpace: IsAlgebraicVariety
A: SesquilinearForm: IsHermitianVariety
A: QuadraticForm: IsQuadraticVariety
A: Source: IsGeometryMap
A: Range: IsGeometryMap


affinespace.gd: attributes
```

```
A: Dimension: IsAffineSpace
A: AmbientSpace: IsAffineSpace
A: AmbientSpace: IsSubspaceOfAffineSpace


affinegroup.gd: attributes


A: AffineGroup: IsAffineSpace

gpolygons.gd: attributes


A: Order: IsGeneralisedPolygon
A: IncidenceMatrixOfGeneralisedPolygon: IsGeneralisedPolygon
A: AmbientPolarSpace: IsGeneralisedHexagon
A: ElationGroup: IsElationGQ
A: BasePointOfEGQ: IsElationGQ
A: IsLinearqClan: IsqClanObj
A: DefiningPlanesOfEGQByBLTSet: IsElationGQByBLTSet
A: CollineationSubgroup: IsElationGQByBLTSet
```

──────── Example ────────
```
Properties

geometry.gd: properties

P: IsConfiguration: IsIncidenceStructure
P: IsConstellation: IsIncidenceStructure


liegeometry.gd: properties


group.gd: properties

P: IsProjectivity: IsProjGrpEl
P: IsProjectivity: IsProjGrpElWithFrob
P: IsStrictlySemilinear: IsProjGrpEl
P: IsStrictlySemilinear: IsProjGrpElWithFrob
P: IsCollineation: IsProjGrpEl
P: IsCollineation: IsProjGrpElWithFrob
P: IsProjectivityGroup: IsProjectiveGroupWithFrob
P: IsCollineationGroup: IsProjectiveGroupWithFrob
P: CanComputeActionOnPoints: IsProjectiveGroupWithFrob

projectivespace.gd: properties


correlations.gd: properties

P: IsCorrelation: IsProjGrpElWithFrobWithPSIsom
P: IsCorrelation: IsProjGrpElWithFrob
P: IsCorrelation: IsProjGrpEl
P: CanComputeActionOnPoints: IsProjGroupWithFrobWithPSIsom
P: IsProjectivity: IsProjGrpElWithFrobWithPSIsom
```

```
P: IsStrictlySemilinear: IsProjGrpElWithFrobWithPSIsom
P: IsCollineation: IsProjGrpElWithFrobWithPSIsom
P: IsProjectivityGroup: IsProjGroupWithFrobWithPSIsom
P: IsCollineationGroup: IsProjGroupWithFrobWithPSIsom
P: IsHermitianPolarityOfProjectiveSpace: IsPolarityOfProjectiveSpace
P: IsSymplecticPolarityOfProjectiveSpace: IsPolarityOfProjectiveSpace
P: IsOrthogonalPolarityOfProjectiveSpace: IsPolarityOfProjectiveSpace
P: IsPseudoPolarityOfProjectiveSpace: IsPolarityOfProjectiveSpace


polarspace.gd: properties


P: IsEllipticQuadric: IsClassicalPolarSpace
P: IsSymplecticSpace: IsClassicalPolarSpace
P: IsParabolicQuadric: IsClassicalPolarSpace
P: IsHyperbolicQuadric: IsClassicalPolarSpace
P: IsHermitianPolarSpace: IsClassicalPolarSpace
P: IsStandardPolarSpace: IsClassicalPolarSpace


morphisms.gd: properties



enumerators.gd: properties



diagram.gd: properties



varieties.gd: properties


P: IsStandardHermitianVariety: IsHermitianVariety
P: IsStandardQuadraticVariety: IsQuadraticVariety


affinespace.gd: properties



affinegroup.gd: properties



gpolygons.gd: properties


P: HasGraphWithUnderlyingObjectsAsVertices: IsGeneralisedPolygon
```

A.2.2 Functions/Methods

─── Example ───

```
Functions


geometry.gi: global functions


F: HashFuncForElements
F: HashFuncForSetElements
```

```
liegeometry.gi: global functions


group.gi: global functions

F: MakeAllProjectivePoints
F: IsFiningScalarMatrix
F: OnProjPoints
F: OnProjPointsWithFrob
F: OnProjSubspacesNoFrob
F: OnProjSubspacesWithFrob
F: NiceMonomorphismByOrbit
F: NiceMonomorphismByDomain

projectivespace.gi: global functions

F: OnProjSubspaces
F: OnSetsProjSubspaces

correlations.gi: global functions

F: OnProjPointsWithFrobWithPSIsom
F: OnProjSubspacesWithFrobWithPSIsom
F: OnProjSubspacesExtended

polarspace.gi: global functions


morphisms.gi: global functions


enumerators.gi: global functions

F: PositionNonZeroFromRight
F: FG_pos
F: FG_div
F: FG_ffenumber
F: FG_alpha_power
F: FG_log_alpha
F: FG_beta_power
F: FG_log_beta
F: FG_norm_one_element
F: FG_index_of_norm_one_element
F: PG_element_normalize
F: FG_evaluate_hyperbolic_quadratic_form
F: FG_evaluate_hermitian_form
F: FG_nb_pts_Nbar
F: FG_nb_pts_S
F: FG_nb_pts_N
F: FG_nb_pts_N1
F: FG_nb_pts_Sbar
F: FG_herm_nb_pts_N
```

```
F: FG_herm_nb_pts_S
F: FG_herm_nb_pts_N1
F: FG_herm_nb_pts_Sbar
F: FG_N1_unrank
F: FG_S_unrank
F: FG_Sbar_unrank
F: FG_Nbar_unrank
F: FG_N_unrank
F: FG_herm_N_unrank
F: FG_herm_N_rank
F: FG_herm_S_unrank
F: FG_herm_S_rank
F: FG_herm_N1_unrank
F: FG_herm_N1_rank
F: FG_herm_Sbar_unrank
F: FG_herm_Sbar_rank
F: FG_S_rank
F: FG_N_rank
F: FG_N1_rank
F: FG_Sbar_rank
F: FG_Nbar_rank
F: QElementNumber
F: QplusElementNumber
F: QminusElementNumber
F: QNumberElement
F: QplusNumberElement
F: QminusNumberElement
F: HermElementNumber
F: HermNumberElement
F: FG_specialresidual
F: FG_enum_orthogonal
F: FG_enum_hermitian
F: FG_enum_symplectic


diagram.gi: global functions


F: OnCosetGeometryElement
F: DrawDiagram
F: DrawDiagramWithNeato
F: Drawing_Diagram

varieties.gi: global functions



affinespace.gi: global functions



affinegroup.gi: global functions


F: OnAffinePoints
F: OnAffineNotPoints
F: OnAffineSubspaces
```

```
gpolygons.gi: global functions


F: SplitCayleyPointToPlane5
F: SplitCayleyPointToPlane
F: ZeroPointToOnePointsSpaceByTriality
F: TwistedTrialityHexagonPointToPlaneByTwoTimesTriality
F: OnKantorFamily


orbits-stabilisers.gi: global functions
```

─────────────────────────────────── Example ───────────────────────────────────

```
Methods


geometry.gi: methods


M: IncidenceStructure, [ IsList, IsFunction, IsFunction, IsList ],
M: Rank, [IsIncidenceStructure],
M: IncidenceGraph, [ IsIncidenceStructure ],
M: ElementsOfIncidenceStructure, [IsIncidenceStructure, IsPosInt],
M: ElementsOfIncidenceStructure, [IsIncidenceStructure, IsString],
M: Iterator, [ IsElementsOfIncidenceStructure ],
M: Enumerator, [ IsElementsOfIncidenceStructure ],
M: NrElementsOfIncidenceStructure, [IsIncidenceStructure, IsString],
M: NrElementsOfIncidenceStructure, [IsIncidenceStructure, IsPosInt],
M: ChooseHashFunction, [ IsElementOfIncidenceStructure, IsPosInt ],
M: ChooseHashFunction, [ CategoryCollections(IsElementOfIncidenceStructure), IsPosInt ],
M: AmbientGeometry, [ IsElementsOfIncidenceStructure and IsElementsOfIncidenceStructureRep ],
M: AmbientGeometry, [ IsAllElementsOfIncidenceStructure ],
M: Type, [IsElementsOfIncidenceStructure and IsElementsOfIncidenceStructureRep],
M: Wrap, [IsIncidenceStructure, IsPosInt, IsObject],
M: Unwrap, [IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep],
M: UnderlyingObject, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: ObjectToElement, [ IsIncidenceStructure, IsPosInt, IsObject ],
M: AmbientGeometry, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: Intersection2, [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure],
M: Type, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: \=, [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure],
M: \<, [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure],
M: \*, [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure],
M: IsIncident, [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure],
M: FlagOfIncidenceStructure, [ IsIncidenceStructure, IsElementOfIncidenceStructureCollection ],
M: FlagOfIncidenceStructure, [ IsIncidenceStructure, IsList and IsEmpty ],
M: IsChamberOfIncidenceStructure, [ IsFlagOfIncidenceStructure and IsFlagOfIncidenceStructureRep
M: AmbientGeometry, [ IsFlagOfIncidenceStructure and IsFlagOfIncidenceStructureRep],
M: ElementsOfFlag, [ IsFlagOfIncidenceStructure and IsFlagOfIncidenceStructureRep ],
M: Size, [ IsFlagOfIncidenceStructure and IsFlagOfIncidenceStructureRep ],
M: Rank, [ IsFlagOfIncidenceStructure ],
M: Type, [ IsFlagOfIncidenceStructure and IsFlagOfIncidenceStructureRep ],
M: ResidueOfFlag, [ IsFlagOfIncidenceStructure ],
M: \=, [ IsFlagOfIncidenceStructure, IsFlagOfIncidenceStructure ],
M: \<, [ IsFlagOfIncidenceStructure, IsFlagOfIncidenceStructure ],
```

```
M: \<, [ IsFlagOfIncidenceStructure, IsElementOfIncidenceStructure ],
M: \<, [ IsElementOfIncidenceStructure, IsFlagOfIncidenceStructure ],
M: IsIncident, [ IsElementOfIncidenceStructure, IsFlagOfIncidenceStructure ],
M: IsIncident, [IsFlagOfIncidenceStructure, IsElementOfIncidenceStructure],
M: \in, [ IsElementOfIncidenceStructure, IsFlagOfIncidenceStructure ],
M: ShadowOfElement, [IsIncidenceStructure, IsElementOfIncidenceStructure, IsPosInt],
M: ShadowOfElement, [IsIncidenceStructure, IsElementOfIncidenceStructure, IsString],
M: ElementsIncidentWithElementOfIncidenceStructure, [ IsElementOfIncidenceStructure, IsPosInt],
M: ShadowOfFlag, [IsIncidenceStructure, IsFlagOfIncidenceStructure, IsPosInt],
M: ShadowOfFlag, [IsIncidenceStructure, IsFlagOfIncidenceStructure, IsString],
M: ShadowOfFlag, [IsIncidenceStructure, IsList, IsPosInt],
M: ShadowOfFlag, [IsIncidenceStructure, IsList, IsString],
M: Iterator, [ IsShadowElementsOfIncidenceStructure ],
M: ViewObj, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: ViewObj, [ IsFlagOfIncidenceStructure and IsFlagOfIncidenceStructureRep ],
M: PrintObj, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: Display, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: ViewObj, [ IsAllElementsOfIncidenceStructure ],
M: PrintObj, [ IsAllElementsOfIncidenceStructure ],
M: ViewObj, [ IsShadowElementsOfIncidenceStructure ],
M: ViewObj, [ IsElementsOfIncidenceStructure ],
M: PrintObj, [ IsElementsOfIncidenceStructure ],
M: ViewObj, [ IsIncidenceStructure ],
M: PrintObj, [ IsIncidenceStructure ],
M: Display, [ IsIncidenceStructure ],
M: IsConfiguration, [ IsIncidenceStructure],
M: IsConstellation, [ IsIncidenceStructure],


liegeometry.gi: methods

M: UnderlyingVectorSpace, [ IsLieGeometry],
M: ProjectiveDimension, [ IsLieGeometry ],
M: Dimension, [ IsLieGeometry ],
M: BaseField, [ IsLieGeometry ],
M: Wrap, [IsLieGeometry, IsPosInt, IsObject],
M: UnderlyingObject, [IsElementOfLieGeometry],
M: AmbientSpace, [IsElementOfLieGeometry],
M: ViewObj, [ IsAllElementsOfLieGeometry and IsAllElementsOfLieGeometryRep ],
M: PrintObj, [ IsAllElementsOfLieGeometry and IsAllElementsOfLieGeometryRep ],
M: ViewObj, [ IsElementsOfLieGeometry and IsElementsOfLieGeometryRep ],
M: PrintObj, [ IsElementsOfLieGeometry and IsElementsOfLieGeometryRep ],
M: Points, [IsLieGeometry],
M: Lines, [IsLieGeometry],
M: Planes, [IsLieGeometry],
M: Solids, [IsLieGeometry],
M: EmptySubspace, [IsLieGeometry],
M: BaseField, [IsEmptySubspace and IsEmptySubspaceRep],
M: ViewObj, InstallMethod(ViewObj,[IsEmptySubspace],
M: PrintObj, InstallMethod(PrintObj,[IsEmptySubspace],
M: Display, InstallMethod(Display,[IsEmptySubspace],
M: \=, [IsEmptySubspace, IsEmptySubspace],
M: \in, [ IsEmptySubspace, IsEmptySubspace ],
M: \in, [ IsEmptySubspace, IsElementOfLieGeometry ],
```

```
M: \in, [ IsElementOfLieGeometry, IsEmptySubspace ],
M: \in, [ IsEmptySubspace, IsLieGeometry ],
M: Span, [ IsEmptySubspace, IsElementOfLieGeometry ],
M: Span, [ IsElementOfLieGeometry, IsEmptySubspace ],
M: Span, [IsEmptySubspace, IsEmptySubspace],
M: Meet, [ IsEmptySubspace, IsElementOfLieGeometry ],
M: Meet, [ IsElementOfLieGeometry, IsEmptySubspace ],
M: Meet, [IsEmptySubspace, IsEmptySubspace],
M: Points, [ IsElementOfLieGeometry ],
M: Points, [ IsLieGeometry, IsElementOfLieGeometry ],
M: Lines, [ IsElementOfLieGeometry ],
M: Lines, [ IsLieGeometry, IsElementOfLieGeometry ],
M: Planes, [ IsElementOfLieGeometry ],
M: Planes, [ IsLieGeometry, IsElementOfLieGeometry ],
M: Solids, InstallMethod(Solids,[IsElementOfLieGeometry],
M: Solids, [ IsLieGeometry, IsElementOfLieGeometry ],
M: Hyperplanes, [ IsElementOfLieGeometry ],
M: Hyperplanes, [ IsLieGeometry, IsElementOfLieGeometry ],
M: ViewObj, [ IsShadowElementsOfLieGeometry and IsShadowElementsOfLieGeometryRep ],
M: \in, [IsElementOfLieGeometry, IsElementOfLieGeometry],
M: Random, [ IsSubspacesVectorSpace ],
M: RandomSubspace, [IsVectorSpace,IsInt],
M: ElementToElement, [IsLieGeometry, IsElementOfLieGeometry],
M: ObjectToElement, [IsLieGeometry, IsPosInt, IsObject],
M: ObjectToElement, [IsLieGeometry, IsObject],


group.gi: methods


M: ProjEl, [IsMatrix and IsFFECollColl],
M: ProjEls, [IsList],
M: Projectivity, InstallMethod(Projectivity,[IsMatrixandIsFFECollColl,IsField],
M: Projectivity, InstallMethod(Projectivity,[IsCMatRepandIsFFECollColl,IsField],
M: Projectivity, InstallMethod(Projectivity,[IsProjectiveSpace,IsMatrix],
M: Projectivity, InstallMethod(Projectivity,[IsProjectiveSpace,IsCMatRep],
M: IsProjectivity, InstallMethod(IsProjectivity,[IsProjGrpEl],
M: IsProjectivity, InstallMethod(IsProjectivity,[IsProjGrpElWithFrob],
M: IsStrictlySemilinear, InstallMethod(IsStrictlySemilinear,[IsProjGrpEl],
M: IsStrictlySemilinear, InstallMethod(IsStrictlySemilinear,[IsProjGrpElWithFrob],
M: IsCollineation, InstallMethod(IsCollineation,[IsProjGrpEl],
M: IsCollineation, InstallMethod(IsCollineation,[IsProjGrpElWithFrob],
M: IsProjectivityGroup, InstallMethod(IsProjectivityGroup,[IsProjectiveGroupWithFrob],
M: IsCollineationGroup, InstallMethod(IsCollineationGroup,[IsProjectiveGroupWithFrob],
M: ProjElWithFrob, [IsCMatRep and IsFFECollColl, #changed 19/3/14 to cmat. IsRingHomomorphism an
M: ProjElWithFrob, [IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativeElementWi
M: ProjElWithFrob, [IsCMatRep and IsFFECollColl, #changed 19/3/14. IsRingHomomorphism and IsMult
M: ProjElWithFrob, [IsCMatRep and IsFFECollColl, IsRingHomomorphism and IsMultiplicativeElementW
M: ProjElsWithFrob, [IsList, IsField],
M: ProjElsWithFrob, [IsList],
M: CollineationOfProjectiveSpace, [ IsMatrix and IsFFECollColl, IsField],
M: CollineationOfProjectiveSpace, InstallMethod(CollineationOfProjectiveSpace,[IsProjectiveSpace
M: CollineationOfProjectiveSpace, InstallMethod(CollineationOfProjectiveSpace,[IsProjectiveSpace
M: Collineation, InstallMethod(Collineation,[IsProjectiveSpace,IsMatrix],
M: Collineation, InstallMethod(Collineation,[IsProjectiveSpace,IsMatrix,IsMapping],
```

```
M: CollineationOfProjectiveSpace, [ IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultipl
M: ProjectiveSemilinearMap, [ IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativ
M: ProjectivityByImageOfStandardFrameNC, InstallMethod(ProjectivityByImageOfStandardFrameNC,[IsP
M: MatrixOfCollineation, InstallMethod(MatrixOfCollineation,[IsProjGrpElandIsProjGrpElRep],
M: MatrixOfCollineation, InstallMethod(MatrixOfCollineation,[IsProjGrpElWithFrobandIsProjGrpElWi
M: FieldAutomorphism, InstallMethod(FieldAutomorphism,[IsProjGrpElWithFrobandIsProjGrpElWithFrob
M: Representative, [IsProjGrpEl and IsProjGrpElRep],
M: BaseField, [IsProjGrpEl and IsProjGrpElRep],
M: Representative, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: BaseField, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: ViewObj, [IsProjGrpEl and IsProjGrpElRep],
M: Display, [IsProjGrpEl and IsProjGrpElRep],
M: PrintObj, [IsProjGrpEl and IsProjGrpElRep],
M: ViewObj, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: Display, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: PrintObj, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: \=, [IsProjGrpEl and IsProjGrpElRep, IsProjGrpEl and IsProjGrpElRep],
M: \<, [IsProjGrpEl, IsProjGrpEl],
M: \=, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep, IsProjGrpElWithFrob and IsProjGrpElWithF
M: \<, [IsProjGrpElWithFrob, IsProjGrpElWithFrob],
M: Order, [IsProjGrpEl and IsProjGrpElRep],
M: Order, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: IsOne, [IsProjGrpEl and IsProjGrpElRep],
M: IsOne, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: DegreeFFE, [IsProjGrpEl and IsProjGrpElRep],
M: DegreeFFE, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: Characteristic, [IsProjGrpEl and IsProjGrpElRep],
M: Characteristic, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: \*, [IsProjGrpEl and IsProjGrpElRep, IsProjGrpEl and IsProjGrpElRep],
M: InverseSameMutability, [IsProjGrpEl and IsProjGrpElRep],
M: InverseMutable, [IsProjGrpEl and IsProjGrpElRep],
M: OneImmutable, [IsProjGrpEl and IsProjGrpElRep],
M: OneSameMutability, [IsProjGrpEl and IsProjGrpElRep],
M: \^, [ IsVector and IsFFECollection and IsMutable, IsFrobeniusAutomorphism ],
M: \^, [ IsCVecRep and IsFFECollection and IsMutable, IsFrobeniusAutomorphism ],
M: \^, [ IsVector and IsFFECollection, IsFrobeniusAutomorphism ],
M: \^, [ IsCVecRep and IsFFECollection, IsFrobeniusAutomorphism ],
M: \^, [ IsVector and IsFFECollection and IsMutable, IsMapping and IsOne ],
M: \^, [ IsCVecRep and IsFFECollection and IsMutable, IsMapping and IsOne ],
M: \^, [ IsVector and IsFFECollection and IsGF2VectorRep, IsFrobeniusAutomorphism ],
M: \^, [ IsVector and IsFFECollection and IsGF2VectorRep and IsMutable, IsFrobeniusAutomorphism
M: \^, [ IsVector and IsFFECollection and IsGF2VectorRep, IsMapping and IsOne ],
M: \^, [ IsVector and IsFFECollection and IsGF2VectorRep and IsMutable, IsMapping and IsOne ],
M: \^, [ IsVector and IsFFECollection and Is8BitVectorRep, IsFrobeniusAutomorphism ],
M: \^, [ IsVector and IsFFECollection and Is8BitVectorRep and IsMutable, IsFrobeniusAutomorphism
M: \^, [ IsVector and IsFFECollection and Is8BitVectorRep, IsMapping and IsOne ],
M: \^, [ IsVector and IsFFECollection and Is8BitVectorRep and IsMutable, IsMapping and IsOne ],
M: \^, [ IsMatrix and IsFFECollColl, IsFrobeniusAutomorphism ],
M: \^, [ IsCMatRep and IsFFECollColl, IsFrobeniusAutomorphism ],
M: \^, [ IsMatrix and IsFFECollColl and IsMutable, IsFrobeniusAutomorphism ],
M: \^, [ IsCMatRep and IsFFECollColl and IsMutable, IsFrobeniusAutomorphism ],
M: \^, [ IsMatrix and IsFFECollColl, IsMapping and IsOne ],
M: \^, [ IsCMatRep and IsFFECollColl and IsMutable, IsMapping and IsOne ],
```

```
M: \^, [ IsMatrix and IsFFECollColl, IsMapping and IsOne ],
M: \^, [ IsCMatRep and IsFFECollColl , IsMapping and IsOne ],
M: \^, [ IsMatrix and IsFFECollColl and IsGF2MatrixRep, IsFrobeniusAutomorphism ],
M: \^, [ IsMatrix and IsFFECollColl and IsGF2MatrixRep and IsMutable, IsFrobeniusAutomorphism ],
M: \^, [ IsMatrix and IsFFECollColl and IsGF2MatrixRep, IsMapping and IsOne ],
M: \^, [ IsMatrix and IsFFECollColl and IsGF2MatrixRep and IsMutable, IsMapping and IsOne ],
M: \^, [ IsMatrix and IsFFECollColl and Is8BitMatrixRep, IsFrobeniusAutomorphism ],
M: \^, [ IsMatrix and IsFFECollColl and Is8BitMatrixRep and IsMutable, IsFrobeniusAutomorphism ]
M: \^, [ IsMatrix and IsFFECollColl and Is8BitMatrixRep, IsMapping and IsOne ],
M: \^, [ IsMatrix and IsFFECollColl and Is8BitMatrixRep and IsMutable, IsMapping and IsOne ],
M: \*, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep, IsProjGrpElWithFrob and IsProjGrpElWithF
M: InverseSameMutability, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: InverseMutable, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: OneImmutable, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: OneSameMutability, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep],
M: ViewObj, [IsProjectiveGroupWithFrob],
M: ViewObj, [IsProjectiveGroupWithFrob and IsTrivial],
M: ViewObj, [IsProjectiveGroupWithFrob and HasGeneratorsOfGroup],
M: ViewObj, [IsProjectiveGroupWithFrob and HasSize],
M: ViewObj, [IsProjectiveGroupWithFrob and HasGeneratorsOfGroup and HasSize],
M: BaseField, [IsProjectiveGroupWithFrob],
M: Dimension, [IsProjectiveGroupWithFrob],
M: OneImmutable, # was [IsGroup and IsProjectiveGroupWithFrob], I think might be
M: CanComputeActionOnPoints, [IsProjectiveGroupWithFrob],
M: ActionOnAllProjPoints, [ IsProjectiveGroupWithFrob ],
M: SetAsNiceMono, [IsProjectiveGroupWithFrob, IsGroupHomomorphism and IsInjective],
M: NiceMonomorphism, [IsProjectivityGroup and CanComputeActionOnPoints and IsHandledByNiceMonomo
M: NiceMonomorphism, [IsProjectiveGroupWithFrob and IsHandledByNiceMonomorphism],
M: NiceMonomorphism, [IsProjectiveGroupWithFrob and CanComputeActionOnPoints and IsHandledByNice
M: NiceMonomorphism, [IsProjectiveGroupWithFrob and IsHandledByNiceMonomorphism], 50,
M: FindBasePointCandidates, [IsProjectivityGroup,IsRecord,IsInt],
M: FindBasePointCandidates, [IsProjectiveGroupWithFrob,IsRecord,IsInt],
M: FindBasePointCandidates, [IsProjectiveGroupWithFrob,IsRecord,IsInt,IsObject],
M: CanonicalGramMatrix, [IsString, IsPosInt, IsField],
M: CanonicalQuadraticForm, [IsString, IsPosInt, IsField],
M: SOdesargues, [IsInt, IsPosInt, IsField and IsFinite],
M: GOdesargues, InstallMethod(GOdesargues,[IsInt,IsPosInt,IsFieldandIsFinite],
M: SUdesargues, InstallMethod(SUdesargues,[IsPosInt,IsFieldandIsFinite],
M: GUdesargues, InstallMethod(GUdesargues,[IsPosInt,IsFieldandIsFinite],
M: Spdesargues, InstallMethod(Spdesargues,[IsPosInt,IsFieldandIsFinite],
M: GeneralSymplecticGroup, InstallMethod(GeneralSymplecticGroup,[IsPosInt,IsFieldandIsFinite],
M: GSpdesargues, InstallMethod(GSpdesargues,[IsPosInt,IsFieldandIsFinite],
M: GammaSp, InstallMethod(GammaSp,[IsPosInt,IsFieldandIsFinite],
M: DeltaOminus, InstallMethod(DeltaOminus,[IsPosInt,IsFieldandIsFinite],
M: GammaOminus, InstallMethod(GammaOminus,[IsPosInt,IsFieldandIsFinite],
M: GammaO, InstallMethod(GammaO,[IsPosInt,IsFieldandIsFinite],
M: DeltaOplus, InstallMethod(DeltaOplus,[IsPosInt,IsFieldandIsFinite],
M: GammaOplus, InstallMethod(GammaOplus,[IsPosInt,IsFieldandIsFinite],
M: GammaU, InstallMethod(GammaU,[IsPosInt,IsFieldandIsFinite],


projectivespace.gi: methods


M: Wrap, [IsProjectiveSpace, IsPosInt, IsObject],
```

```
M: ProjectiveSpace, [ IsInt, IsField ],
M: ProjectiveSpace, [ IsInt, IsPosInt ],
M: ViewObj, InstallMethod(ViewObj,[IsProjectiveSpaceandIsProjectiveSpaceRep],
M: ViewString, [ IsProjectiveSpace and IsProjectiveSpaceRep ],
M: PrintObj, InstallMethod(PrintObj,[IsProjectiveSpaceandIsProjectiveSpaceRep],
M: Display, InstallMethod(Display,[IsProjectiveSpaceandIsProjectiveSpaceRep],
M: \=, [IsProjectiveSpace, IsProjectiveSpace],
M: Rank, [ IsProjectiveSpace and IsProjectiveSpaceRep ],
M: BaseField, [IsSubspaceOfProjectiveSpace],
M: StandardFrame, [IsProjectiveSpace],
M: RepresentativesOfElements, "for a projective space", [IsProjectiveSpace],
M: Hyperplanes, [ IsProjectiveSpace ],
M: TypesOfElementsOfIncidenceStructure, "for a projective space", [IsProjectiveSpace],
M: TypesOfElementsOfIncidenceStructurePlural, [IsProjectiveSpace],
M: ElementsOfIncidenceStructure, [IsProjectiveSpace, IsPosInt],
M: ElementsOfIncidenceStructure, [IsProjectiveSpace],
M: \=, [ IsAllSubspacesOfProjectiveSpace, IsAllSubspacesOfProjectiveSpace ],
M: Size, [IsSubspacesOfProjectiveSpace and IsSubspacesOfProjectiveSpaceRep],
M: \in, [IsElementOfIncidenceStructure, IsElementsOfIncidenceStructure], 1*SUM_FLAGS+3,
M: \in, [IsElementOfIncidenceStructure, IsAllElementsOfIncidenceStructure], 1*SUM_FLAGS+3,
M: VectorSpaceToElement, [IsProjectiveSpace, IsCMatRep],
M: VectorSpaceToElement, [IsProjectiveSpace, IsPlistRep],
M: VectorSpaceToElement, [IsProjectiveSpace, IsGF2MatrixRep],
M: VectorSpaceToElement, [IsProjectiveSpace, Is8BitMatrixRep],
M: VectorSpaceToElement, [IsProjectiveSpace, IsCVecRep],
M: VectorSpaceToElement, [IsProjectiveSpace, IsRowVector],
M: VectorSpaceToElement, [IsProjectiveSpace, Is8BitVectorRep],
M: UnderlyingVectorSpace, [IsSubspaceOfProjectiveSpace],
M: ProjectiveDimension, [ IsSubspaceOfProjectiveSpace ],
M: Dimension, [ IsSubspaceOfProjectiveSpace ],
M: StandardFrame, [IsSubspaceOfProjectiveSpace],
M: Coordinates, [IsSubspaceOfProjectiveSpace],
M: DualCoordinatesOfHyperplane, [IsSubspaceOfProjectiveSpace],
M: HyperplaneByDualCoordinates, [IsProjectiveSpace,IsList],
M: EquationOfHyperplane, [IsSubspaceOfProjectiveSpace],
M: Span, [ IsEmptySubspace, IsProjectiveSpace ],
M: Span, [ IsProjectiveSpace, IsEmptySubspace ],
M: Meet, [ IsEmptySubspace, IsProjectiveSpace ],
M: Meet, [ IsProjectiveSpace, IsEmptySubspace ],
M: ShadowOfElement, [IsProjectiveSpace, IsSubspaceOfProjectiveSpace, IsPosInt],
M: Size, [IsShadowSubspacesOfProjectiveSpace and IsShadowSubspacesOfProjectiveSpaceRep ],
M: CollineationGroup, [ IsProjectiveSpace and IsProjectiveSpaceRep ],
M: ProjectivityGroup, [ IsProjectiveSpace ],
M: SpecialProjectivityGroup, [ IsProjectiveSpace ],
M: \^, [IsElementOfIncidenceStructure, IsProjGrpElWithFrob],
M: \^, [IsElementOfIncidenceStructure, IsProjGrpElWithFrobWithPSIsom],
M: AsList, [IsSubspacesOfProjectiveSpace],
M: Iterator, [IsSubspacesOfProjectiveSpace],
M: FlagOfIncidenceStructure, [ IsProjectiveSpace, IsSubspaceOfProjectiveSpaceCollection ],
M: FlagOfIncidenceStructure, [ IsProjectiveSpace, IsList and IsEmpty ],
M: UnderlyingVectorSpace, [ IsFlagOfProjectiveSpace and IsFlagOfIncidenceStructureRep ],
M: ViewObj, [ IsFlagOfProjectiveSpace and IsFlagOfIncidenceStructureRep ],
M: PrintObj, [ IsFlagOfProjectiveSpace and IsFlagOfIncidenceStructureRep ],
```

```
M: Display, [ IsFlagOfProjectiveSpace and IsFlagOfIncidenceStructureRep ],
M: ShadowOfFlag, [IsProjectiveSpace, IsFlagOfProjectiveSpace, IsPosInt],
M: Iterator, [IsShadowSubspacesOfProjectiveSpace and IsShadowSubspacesOfProjectiveSpaceRep ],
M: \in, [ IsProjectiveSpace, IsSubspaceOfProjectiveSpace ],
M: \in, [ IsProjectiveSpace, IsEmptySubspace ],
M: \in, [IsSubspaceOfProjectiveSpace, IsProjectiveSpace],
M: \in, [IsProjectiveSpace, IsSubspaceOfProjectiveSpace],
M: IsIncident, [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace],
M: Span, [IsProjectiveSpace, IsSubspaceOfProjectiveSpace],
M: Span, [IsSubspaceOfProjectiveSpace, IsProjectiveSpace],
M: Span, [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace],
M: Span, [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace, IsBool],
M: Span, [ IsHomogeneousList and IsSubspaceOfProjectiveSpaceCollection ],
M: Span, [ IsList ],
M: Span, [IsList, IsBool],
M: Meet, [IsProjectiveSpace, IsSubspaceOfProjectiveSpace],
M: Meet, [IsSubspaceOfProjectiveSpace, IsProjectiveSpace],
M: Meet, [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace],
M: Meet, [ IsHomogeneousList and IsSubspaceOfProjectiveSpaceCollection],
M: Meet, [ IsList ],
M: RandomSubspace, [IsProjectiveSpace,IsInt],
M: RandomSubspace, [IsSubspaceOfProjectiveSpace,IsInt],
M: RandomSubspace, [IsProjectiveSpace],
M: Random, [ IsSubspacesOfProjectiveSpace ],
M: Random, [ IsAllSubspacesOfProjectiveSpace ],
M: Random, [ IsShadowSubspacesOfProjectiveSpace ],
M: BaerSublineOnThreePoints, [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace, IsSubspa
M: BaerSubplaneOnQuadrangle, InstallMethod(BaerSubplaneOnQuadrangle,[IsSubspaceOfProjectiveSpace
M: ComplementSpace, [IsVectorSpace, IsFFECollColl],
M: ElationOfProjectiveSpace, [ IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace, IsSubsp
M: ProjectiveElationGroup, [ IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace ],
M: ProjectiveElationGroup, [ IsSubspaceOfProjectiveSpace ],
M: HomologyOfProjectiveSpace, [ IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace, IsSubs
M: ProjectiveHomologyGroup, [ IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace ],
M: IncidenceGraph, [ IsProjectiveSpace ],


correlations.gi: methods


M: IdentityMappingOfElementsOfProjectiveSpace, [IsProjectiveSpace],
M: StandardDualityOfProjectiveSpace, [IsProjectiveSpace],
M: IsCollineation, InstallMethod(IsCollineation,[IsProjGrpElWithFrobWithPSIsom],
M: IsCorrelation, InstallMethod(IsCorrelation,[IsProjGrpElWithFrobWithPSIsom],
M: IsCorrelation, InstallMethod(IsCorrelation,[IsProjGrpElWithFrob],
M: IsCorrelation, InstallMethod(IsCorrelation,[IsProjGrpEl],
M: IsProjectivity, [ IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: IsStrictlySemilinear, [ IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: IsProjectivityGroup, InstallMethod(IsProjectivityGroup,[IsProjGroupWithFrobWithPSIsom],
M: IsCollineationGroup, InstallMethod(IsCollineationGroup,[IsProjGroupWithFrobWithPSIsom],
M: ViewObj, [IsStandardDualityOfProjectiveSpace and IsSPMappingByFunctionWithInverseRep],
M: Display, [IsStandardDualityOfProjectiveSpace and IsSPMappingByFunctionWithInverseRep],
M: PrintObj, [IsStandardDualityOfProjectiveSpace and IsSPMappingByFunctionWithInverseRep],
M: \*, [IsStandardDualityOfProjectiveSpace, IsStandardDualityOfProjectiveSpace],
M: \*, [IsIdentityMappingOfElementsOfProjectiveSpace, IsStandardDualityOfProjectiveSpace],
```

```
M: \*, [IsStandardDualityOfProjectiveSpace, IsIdentityMappingOfElementsOfProjectiveSpace],
M: \*, [IsIdentityMappingOfElementsOfProjectiveSpace, IsIdentityMappingOfElementsOfProjectiveSpa
M: \^, [ IsProjectiveSpaceIsomorphism, IsZeroCyc ],
M: \=, [IsStandardDualityOfProjectiveSpace, IsStandardDualityOfProjectiveSpace],
M: \=, [IsStandardDualityOfProjectiveSpace, IsIdentityMappingOfElementsOfProjectiveSpace],
M: \=, [IsIdentityMappingOfElementsOfProjectiveSpace, IsStandardDualityOfProjectiveSpace],
M: \=, [IsIdentityMappingOfElementsOfProjectiveSpace, IsIdentityMappingOfElementsOfProjectiveSpa
M: ProjElWithFrobWithPSIsom, [IsCMatRep and IsFFECollColl, IsRingHomomorphism and IsMultiplicati
M: ProjElWithFrobWithPSIsom, [IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativ
M: ProjElWithFrobWithPSIsom, [IsCMatRep and IsFFECollColl, IsRingHomomorphism and IsMultiplicati
M: ProjElWithFrobWithPSIsom, [IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativ
M: ProjElWithFrobWithPSIsom, [IsCMatRep and IsFFECollColl, IsRingHomomorphism and IsMultiplicati
M: ProjElWithFrobWithPSIsom, [IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativ
M: ViewObj, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: Display, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: PrintObj, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: Representative, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: BaseField, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: BaseField, [IsProjGroupWithFrobWithPSIsom],
M: \=, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep, IsProjGrpElWithFrobW
M: IsOne, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: OneImmutable, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: OneImmutable, [IsGroup and IsProjGrpElWithFrobWithPSIsom],
M: OneSameMutability, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: \^, [ IsCVecRep and IsFFECollection, IsIdentityMappingOfElementsOfProjectiveSpace ],
M: \^, [ IsVector and IsFFECollection, IsIdentityMappingOfElementsOfProjectiveSpace ],
M: \^, [ IsVector and IsFFECollection and IsGF2VectorRep, IsIdentityMappingOfElementsOfProjectiv
M: \^, [ IsVector and IsFFECollection and Is8BitVectorRep, IsIdentityMappingOfElementsOfProjecti
M: \^, [ IsCMatRep and IsFFECollColl, IsStandardDualityOfProjectiveSpace ],
M: \^, [ IsMatrix and IsFFECollColl, IsStandardDualityOfProjectiveSpace ],
M: \^, [ IsCMatRep and IsFFECollColl, IsIdentityMappingOfElementsOfProjectiveSpace ],
M: \^, [ IsMatrix and IsFFECollColl, IsIdentityMappingOfElementsOfProjectiveSpace ],
M: \^, [ IsSubspaceOfProjectiveSpace, IsIdentityMappingOfElementsOfProjectiveSpace ],
M: \^, [ IsSubspaceOfProjectiveSpace, IsStandardDualityOfProjectiveSpace ],
M: \*, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep, IsProjGrpElWithFrobW
M: \<, [IsProjGrpElWithFrobWithPSIsom, IsProjGrpElWithFrobWithPSIsom],
M: InverseSameMutability, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: InverseMutable, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep],
M: \*, [IsProjGrpElWithFrob and IsProjGrpElWithFrobRep, IsProjGrpElWithFrobWithPSIsom and IsProj
M: \*, [IsProjGrpElWithFrobWithPSIsom and IsProjGrpElWithFrobWithPSIsomRep, IsProjGrpElWithFrob
M: ProjElsWithFrobWithPSIsom, [IsList, IsField],
M: CorrelationCollineationGroup, [ IsProjectiveSpace and IsProjectiveSpaceRep ],
M: CorrelationOfProjectiveSpace, [ IsMatrix and IsFFECollColl, IsField],
M: CorrelationOfProjectiveSpace, [ IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultipli
M: CorrelationOfProjectiveSpace, [ IsMatrix and IsFFECollColl, IsField, IsStandardDualityOfProje
M: CorrelationOfProjectiveSpace, [ IsMatrix and IsFFECollColl, IsField, IsIdentityMappingOfEleme
M: CorrelationOfProjectiveSpace, [ IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultipli
M: CorrelationOfProjectiveSpace, [ IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultipli
M: CorrelationOfProjectiveSpace, [ IsProjectiveSpace, IsMatrix and IsFFECollColl, IsRingHomomorp
M: CorrelationOfProjectiveSpace, [ IsProjectiveSpace, IsMatrix and IsFFECollColl, IsRingHomomorp
M: Correlation, [ IsProjectiveSpace, IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultip
M: Correlation, [ IsProjectiveSpace, IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultip
M: MatrixOfCorrelation, InstallMethod(MatrixOfCorrelation,[IsProjGrpElWithFrobWithPSIsomand IsPr
```

```
M: FieldAutomorphism, InstallMethod(FieldAutomorphism,[IsProjGrpElWithFrobWithPSIsomand IsProjGr
M: ProjectiveSpaceIsomorphism, InstallMethod(ProjectiveSpaceIsomorphism,[IsProjGrpElWithFrobWith
M: Embedding, [IsProjectiveGroupWithFrob, IsProjGroupWithFrobWithPSIsom],
M: Dimension, [IsProjGroupWithFrobWithPSIsom],
M: ActionOnAllPointsHyperplanes, [ IsProjGroupWithFrobWithPSIsom ],
M: CanComputeActionOnPoints, [IsProjGroupWithFrobWithPSIsom],
M: SetAsNiceMono, [IsProjGroupWithFrobWithPSIsom, IsGroupHomomorphism and IsInjective],
M: NiceMonomorphism, [IsProjGroupWithFrobWithPSIsom and CanComputeActionOnPoints and IsHandledBy
M: NiceMonomorphism, [IsProjGroupWithFrobWithPSIsom and IsHandledByNiceMonomorphism], 50,
M: ViewObj, [IsProjGroupWithFrobWithPSIsom],
M: ViewObj, [IsProjGroupWithFrobWithPSIsom and IsTrivial],
M: ViewObj, [IsProjGroupWithFrobWithPSIsom and HasGeneratorsOfGroup],
M: ViewObj, [IsProjGroupWithFrobWithPSIsom and HasSize],
M: ViewObj, [IsProjGroupWithFrobWithPSIsom and HasGeneratorsOfGroup and HasSize],
M: PolarityOfProjectiveSpaceOp, [IsSesquilinearForm and IsFormRep],
M: ViewObj, [IsPolarityOfProjectiveSpace and IsPolarityOfProjectiveSpaceRep],
M: PrintObj, [IsPolarityOfProjectiveSpace and IsPolarityOfProjectiveSpaceRep],
M: Display, [IsPolarityOfProjectiveSpace and IsPolarityOfProjectiveSpaceRep],
M: PolarityOfProjectiveSpace, [IsSesquilinearForm and IsFormRep],
M: PolarityOfProjectiveSpace, [IsMatrix,IsField and IsFinite],
M: PolarityOfProjectiveSpace, [IsMatrix, IsFrobeniusAutomorphism, IsField and IsFinite],
M: HermitianPolarityOfProjectiveSpace, [IsMatrix,IsField and IsFinite],
M: GramMatrix, [IsPolarityOfProjectiveSpace and IsPolarityOfProjectiveSpaceRep],
M: BaseField, [IsPolarityOfProjectiveSpace and IsPolarityOfProjectiveSpaceRep],
M: CompanionAutomorphism, [IsPolarityOfProjectiveSpace and IsPolarityOfProjectiveSpaceRep],
M: SesquilinearForm, [IsPolarityOfProjectiveSpace and IsPolarityOfProjectiveSpaceRep],
M: IsHermitianPolarityOfProjectiveSpace, [IsPolarityOfProjectiveSpace and IsPolarityOfProjective
M: IsOrthogonalPolarityOfProjectiveSpace, [IsPolarityOfProjectiveSpace and IsPolarityOfProjectiv
M: IsSymplecticPolarityOfProjectiveSpace, [IsPolarityOfProjectiveSpace and IsPolarityOfProjectiv
M: IsPseudoPolarityOfProjectiveSpace, [IsPolarityOfProjectiveSpace and IsPolarityOfProjectiveSpa
M: \^, [ IsSubspaceOfProjectiveSpace, IsPolarityOfProjectiveSpace],


polarspace.gi: methods

M: Wrap, [IsClassicalPolarSpace, IsPosInt, IsObject],
M: PolarSpace, [ IsSesquilinearForm, IsField, IsGroup, IsFunction ],
M: PolarSpaceStandard, [ IsSesquilinearForm, IsBool ],
M: PolarSpaceStandard, [ IsQuadraticForm, IsBool ],
M: PolarSpace, [ IsSesquilinearForm ],
M: PolarSpace, [ IsQuadraticForm ],
M: PolarSpace, [ IsHermitianForm ],
M: CanonicalOrbitRepresentativeForSubspaces, [IsString, IsPosInt, IsField],
M: EllipticQuadric, [ IsPosInt, IsField ],
M: EllipticQuadric, [ IsPosInt, IsPosInt ],
M: SymplecticSpace, [ IsPosInt, IsField ],
M: SymplecticSpace, [ IsPosInt, IsPosInt ],
M: ParabolicQuadric, [ IsPosInt, IsField ],
M: ParabolicQuadric, [ IsPosInt, IsPosInt ],
M: HyperbolicQuadric, [ IsPosInt, IsField ],
M: HyperbolicQuadric, [ IsPosInt, IsPosInt ],
M: HermitianPolarSpace, [ IsPosInt, IsField ],
M: HermitianPolarSpace, [ IsPosInt, IsPosInt ],
M: StandardPolarSpace, [ IsClassicalPolarSpace ],
```

```
M: IsCanonicalPolarSpace, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: CanonicalPolarSpace, [ IsClassicalPolarSpace ],
M: QuadraticForm, [ IsClassicalPolarSpace ],
M: PolarSpaceType, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: CompanionAutomorphism, [ IsClassicalPolarSpace ],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: ViewString, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsEllipticQuadric],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsEllipticQuadric and IsSta
M: ViewString, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsEllipticQuadric and Is
M: ViewString, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsEllipticQuadric],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsSymplecticSpace],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsSymplecticSpace and IsSta
M: ViewString, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsSymplecticSpace and Is
M: ViewString, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsSymplecticSpace],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsParabolicQuadric ],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsParabolicQuadric and IsSt
M: ViewString, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsParabolicQuadric and I
M: ViewString, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsParabolicQuadric ],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsParabolicQuadric ],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHyperbolicQuadric ],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHyperbolicQuadric and IsS
M: ViewString, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHyperbolicQuadric and
M: ViewString, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHyperbolicQuadric],
M: ViewObj, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHermitianPolarSpace ],
M: ViewObj, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHermitianPolarSpace and Is
M: ViewString, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHermitianPolarSpace and
M: ViewString, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHermitianPolarSpace],
M: PrintObj, InstallMethod(PrintObj,[IsClassicalPolarSpaceandIsClassicalPolarSpaceRep],
M: Display, InstallMethod(Display,[IsClassicalPolarSpaceandIsClassicalPolarSpaceRep],
M: PrintObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsEllipticQuadric ],
M: Display, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsEllipticQuadric ],
M: PrintObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsSymplecticSpace ],
M: Display, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsSymplecticSpace ],
M: PrintObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsParabolicQuadric ],
M: Display, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsParabolicQuadric ],
M: PrintObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHyperbolicQuadric ],
M: Display, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHyperbolicQuadric ],
M: PrintObj, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHermitianPolarSpace ],
M: Display, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHermitianPolarSpace ],
M: IsomorphismCanonicalPolarSpace, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: IsomorphismCanonicalPolarSpaceWithIntertwiner, [ IsClassicalPolarSpace and IsClassicalPolarSp
M: RankAttr, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: TypesOfElementsOfIncidenceStructure, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: TypesOfElementsOfIncidenceStructurePlural, [IsClassicalPolarSpace],
M: Order, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: RepresentativesOfElements, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: \QUO, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep, IsSubspaceOfClassicalPolarSpace],
M: Size, [IsSubspacesOfClassicalPolarSpace],
M: VectorSpaceToElement, [IsClassicalPolarSpace, IsCMatRep],
M: VectorSpaceToElement, [IsClassicalPolarSpace, IsCVecRep],
M: VectorSpaceToElement, [IsClassicalPolarSpace, IsPlistRep],
M: VectorSpaceToElement, [IsClassicalPolarSpace, IsGF2MatrixRep],
```

```
M: VectorSpaceToElement, [IsClassicalPolarSpace, Is8BitMatrixRep],
M: VectorSpaceToElement, [IsClassicalPolarSpace, IsRowVector],
M: VectorSpaceToElement, [IsClassicalPolarSpace, Is8BitVectorRep],
M: \in, [IsElementOfIncidenceStructure, IsClassicalPolarSpace],
M: Span, [IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace, IsBool],
M: Meet, [IsSubspaceOfClassicalPolarSpace, IsSubspaceOfClassicalPolarSpace],
M: ElementsOfIncidenceStructure, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep, IsPosInt],
M: ElementsOfIncidenceStructure, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep],
M: NumberOfTotallySingularSubspaces, [IsClassicalPolarSpace, IsPosInt],
M: TypeOfSubspace, [ IsClassicalPolarSpace, IsSubspaceOfProjectiveSpace ],
M: FlagOfIncidenceStructure, [ IsClassicalPolarSpace, IsSubspaceOfProjectiveSpaceCollection ],
M: FlagOfIncidenceStructure, [ IsClassicalPolarSpace, IsList and IsEmpty ],
M: ViewObj, [ IsFlagOfClassicalPolarSpace and IsFlagOfIncidenceStructureRep ],
M: PrintObj, [ IsFlagOfClassicalPolarSpace and IsFlagOfIncidenceStructureRep ],
M: Display, [ IsFlagOfClassicalPolarSpace and IsFlagOfIncidenceStructureRep ],
M: RandomSubspace, [ IsClassicalPolarSpace, IsPosInt ],
M: Random, [ IsSubspacesOfClassicalPolarSpace ],
M: Iterator, [IsSubspacesOfClassicalPolarSpace],
M: ShadowOfElement, [IsClassicalPolarSpace, IsSubspaceOfProjectiveSpace, IsPosInt],
M: Iterator, [ IsShadowSubspacesOfClassicalPolarSpace ],
M: Size, [IsShadowSubspacesOfClassicalPolarSpace andIsShadowSubspacesOfClassicalPolarSpaceRep ],
M: IsCollinear, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep, IsSubspaceOfProjectiveSpace
M: PolarityOfProjectiveSpace, [IsClassicalPolarSpace],
M: PolarSpace, [ IsPolarityOfProjectiveSpace ],
M: GeometryOfAbsolutePoints, [ IsPolarityOfProjectiveSpace ],
M: AbsolutePoints, [ IsPolarityOfProjectiveSpace ],
M: AbsolutePoints, [ IsPolarityOfProjectiveSpace ],
M: PolarMap, [IsClassicalPolarSpace],
M: CollineationGroup, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: SpecialIsometryGroup, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: IsometryGroup, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: SimilarityGroup, InstallMethod(SimilarityGroup,[IsClassicalPolarSpaceandIsClassicalPolarSpace
M: IsParabolicQuadric, [IsClassicalPolarSpace],
M: IsParabolicQuadric, [IsClassicalPolarSpace],
M: IsHyperbolicQuadric, [IsClassicalPolarSpace],
M: IsHyperbolicQuadric, [IsClassicalPolarSpace],
M: IsEllipticQuadric, [IsClassicalPolarSpace],
M: IsEllipticQuadric, [IsClassicalPolarSpace],
M: IsSymplecticSpace, [IsClassicalPolarSpace],
M: IsHermitianPolarSpace, [IsClassicalPolarSpace],
M: DefiningListOfPolynomials, [IsProjectiveVariety and IsClassicalPolarSpace and IsClassicalPola
M: NucleusOfParabolicQuadric, [ IsClassicalPolarSpace ],
M: TangentSpace, [ IsSubspaceOfClassicalPolarSpace ],
M: TangentSpace, [ IsClassicalPolarSpace, IsSubspaceOfProjectiveSpace ],
M: Pole, [ IsClassicalPolarSpace, IsSubspaceOfProjectiveSpace ],
M: IncidenceGraph, [ IsClassicalPolarSpace ],


morphisms.gi: methods


M: GeometryMorphismByFunction, [ IsAnyElementsOfIncidenceStructure, IsAnyElementsOfIncidenceStru
M: GeometryMorphismByFunction, [ IsAnyElementsOfIncidenceStructure, IsAnyElementsOfIncidenceStru
M: GeometryMorphismByFunction, [ IsAnyElementsOfIncidenceStructure, IsAnyElementsOfIncidenceStru
M: ViewObj, [ IsGeometryMorphism ],
```

```
M: PrintObj, [ IsGeometryMorphism ],
M: Display, [ IsGeometryMorphism ],
M: ViewObj, [ IsGeometryMorphism and IsMappingByFunctionWithInverseRep ],
M: ViewObj, [ IsGeometryMorphism and IsMappingByFunctionRep ],
M: PrintObj, [ IsGeometryMorphism and IsMappingByFunctionRep ],
M: Display, [ IsGeometryMorphism and IsMappingByFunctionRep ],
M: ImageElm, [IsGeometryMorphism, IsElementOfIncidenceStructure],
M: \^, [IsElementOfIncidenceStructure, IsGeometryMorphism],
M: ImagesSet, [IsGeometryMorphism, IsElementOfIncidenceStructureCollection],
M: PreImageElm, [IsGeometryMorphism, IsElementOfIncidenceStructure],
M: PreImagesSet, [IsGeometryMorphism, IsElementOfIncidenceStructureCollection],
M: NaturalEmbeddingBySubspace, [ IsProjectiveSpace, IsProjectiveSpace, IsSubspaceOfProjectiveSpa
M: NaturalEmbeddingBySubspaceNC, [ IsProjectiveSpace, IsProjectiveSpace, IsSubspaceOfProjectiveS
M: NaturalEmbeddingBySubspace, [ IsClassicalPolarSpace, IsClassicalPolarSpace, IsSubspaceOfProje
M: NaturalEmbeddingBySubspaceNC, [ IsClassicalPolarSpace, IsClassicalPolarSpace, IsSubspaceOfPro
M: IsomorphismPolarSpaces, [ IsClassicalPolarSpace, IsClassicalPolarSpace, IsBool ],
M: IsomorphismPolarSpaces, [ IsClassicalPolarSpace, IsClassicalPolarSpace ],
M: IsomorphismPolarSpacesNC, [ IsClassicalPolarSpace, IsClassicalPolarSpace, IsBool ],
M: IsomorphismPolarSpacesNC, [ IsClassicalPolarSpace, IsClassicalPolarSpace ],
M: ShrinkMat, [ IsBasis, IsMatrix ],
M: ShrinkMat, [ IsField,IsField, IsMatrix ],
M: ShrinkVec, [ IsField, IsField, IsVector ],
M: ShrinkVec, [ IsField, IsField, IsVector, IsBasis ],
M: BlownUpProjectiveSpace, [ IsBasis, IsProjectiveSpace ],
M: BlownUpProjectiveSpaceBySubfield, [ IsField, IsProjectiveSpace ],
M: BlownUpSubspaceOfProjectiveSpace, [ IsBasis, IsSubspaceOfProjectiveSpace ],
M: BlownUpSubspaceOfProjectiveSpaceBySubfield, [ IsField, IsSubspaceOfProjectiveSpace],
M: IsDesarguesianSpreadElement, [ IsBasis, IsSubspaceOfProjectiveSpace ],
M: IsBlownUpSubspaceOfProjectiveSpace, [ IsBasis, IsSubspaceOfProjectiveSpace ],
M: NaturalEmbeddingByFieldReduction, [ IsProjectiveSpace, IsField, IsBasis ],
M: NaturalEmbeddingByFieldReduction, [ IsProjectiveSpace, IsField ],
M: NaturalEmbeddingByFieldReduction, [ IsProjectiveSpace, IsProjectiveSpace, IsBasis ],
M: NaturalEmbeddingByFieldReduction, [ IsProjectiveSpace, IsProjectiveSpace ],
M: BilinearFormFieldReduction, [ IsBilinearForm, IsField, IsFFE, IsBasis ],
M: BilinearFormFieldReduction, [ IsBilinearForm, IsField, IsFFE ],
M: QuadraticFormFieldReduction, [ IsQuadraticForm, IsField, IsFFE, IsBasis ],
M: QuadraticFormFieldReduction, [ IsQuadraticForm, IsField, IsFFE ],
M: HermitianFormFieldReduction, [ IsHermitianForm, IsField, IsFFE, IsBasis ],
M: HermitianFormFieldReduction, [ IsHermitianForm, IsField, IsFFE ],
M: NaturalEmbeddingByFieldReduction, [IsClassicalPolarSpace, IsField, IsFFE, IsBasis, IsBool],
M: NaturalEmbeddingByFieldReduction, [IsClassicalPolarSpace, IsField, IsFFE, IsBasis],
M: NaturalEmbeddingByFieldReduction, [IsClassicalPolarSpace, IsField, IsFFE, IsBool],
M: NaturalEmbeddingByFieldReduction, [IsClassicalPolarSpace, IsField, IsFFE],
M: NaturalEmbeddingByFieldReduction, [IsClassicalPolarSpace, IsField, IsBool],
M: NaturalEmbeddingByFieldReduction, [IsClassicalPolarSpace, IsField],
M: NaturalEmbeddingByFieldReduction, [ IsClassicalPolarSpace, IsClassicalPolarSpace ],
M: NaturalEmbeddingByFieldReduction, [IsClassicalPolarSpace, IsClassicalPolarSpace, IsBool],
M: NaturalEmbeddingByFieldReduction, [IsClassicalPolarSpace, IsClassicalPolarSpace],
M: NaturalEmbeddingBySubfield, [ IsProjectiveSpace, IsProjectiveSpace ],
M: NaturalEmbeddingBySubfield, [ IsClassicalPolarSpace, IsClassicalPolarSpace, IsBool ],
M: NaturalEmbeddingBySubfield, [ IsClassicalPolarSpace, IsClassicalPolarSpace ],
M: NaturalProjectionBySubspace, [ IsProjectiveSpace, IsSubspaceOfProjectiveSpace ],
M: NaturalProjectionBySubspaceNC, [ IsProjectiveSpace, IsSubspaceOfProjectiveSpace ],
```

```
M: \QUO, [ IsProjectiveSpace and IsProjectiveSpaceRep, IsSubspaceOfProjectiveSpace],
M: NaturalProjectionBySubspace, [ IsClassicalPolarSpace, IsSubspaceOfClassicalPolarSpace ],
M: NaturalProjectionBySubspaceNC, [ IsClassicalPolarSpace, IsSubspaceOfClassicalPolarSpace ],
M: PluckerCoordinates, [ IsMatrix ],
M: InversePluckerCoordinates, [ IsVector ],
M: PluckerCoordinates, [ IsSubspaceOfProjectiveSpace ],
M: KleinCorrespondence, [ IsField, IsBool ],
M: KleinCorrespondence, [ IsPosInt ],
M: KleinCorrespondence, [ IsPosInt, IsBool ],
M: KleinCorrespondence, [ IsField ],
M: KleinCorrespondence, [ IsClassicalPolarSpace, IsBool ],
M: KleinCorrespondence, [ IsClassicalPolarSpace ],
M: KleinCorrespondenceExtended, [ IsField, IsBool ],
M: KleinCorrespondenceExtended, [ IsPosInt ],
M: KleinCorrespondenceExtended, [ IsClassicalPolarSpace, IsBool ],
M: KleinCorrespondenceExtended, [ IsClassicalPolarSpace ],
M: NaturalDualitySymplectic, [ IsClassicalGQ, IsClassicalGQ, IsBool, IsBool ],
M: NaturalDualityHermitian, [ IsClassicalGQ, IsClassicalGQ, IsBool, IsBool ],
M: NaturalDuality, [ IsClassicalGQ, IsClassicalGQ, IsBool ],
M: NaturalDuality, [ IsClassicalGQ, IsClassicalGQ ],
M: NaturalDuality, [ IsClassicalGQ, IsBool ],
M: NaturalDuality, [ IsClassicalGQ ],
M: IsomorphismPolarSpacesProjectionFromNucleus, [ IsClassicalPolarSpace, IsClassicalPolarSpace,
M: SelfDualitySymplectic, [ IsClassicalGQ, IsBool ],
M: SelfDualityParabolic, [ IsClassicalGQ, IsBool ],
M: SelfDuality, [ IsClassicalGQ, IsBool ],
M: SelfDuality, [ IsClassicalGQ ],


enumerators.gi: methods


M: AntonEnumerator, [IsSubspacesOfClassicalPolarSpace],
M: EnumeratorByOrbit, [ IsSubspacesOfClassicalPolarSpace ],
M: AsList, [IsSubspacesOfClassicalPolarSpace],
M: AsSortedList, [IsSubspacesOfClassicalPolarSpace],
M: AsSSortedList, [IsSubspacesOfClassicalPolarSpace],
M: Enumerator, [ IsSubspacesOfClassicalPolarSpace ],
M: Enumerator, [IsShadowSubspacesOfClassicalPolarSpace and IsShadowSubspacesOfClassicalPolarSpac


diagram.gi: methods


M: CosetGeometry, InstallMethod(CosetGeometry,"forgroupsandlistofsubgroups",[IsGroup,IsHomogeneo
M: Rank2Residues, InstallMethod(Rank2Residues,[IsIncidenceGeometry],
M: MakeRank2Residue, InstallMethod(MakeRank2Residue,[IsRank2Residue],
M: \^, [IsElementOfCosetGeometry, IsMultiplicativeElementWithInverse],
M: \^, [IsFlagOfCosetGeometry, IsMultiplicativeElementWithInverse],
M: FlagOfIncidenceStructure, [ IsCosetGeometry, IsElementOfIncidenceStructureCollection ],
M: FlagOfIncidenceStructure, [ IsCosetGeometry, IsList and IsEmpty ],
M: \=, InstallOtherMethod(\=,[IsCosetGeometry,IsCosetGeometry ],
M: ElementsOfIncidenceStructure, InstallMethod(ElementsOfIncidenceStructure,[IsCosetGeometry,IsP
M: ElementsOfIncidenceStructure, [IsCosetGeometry],
M: RandomElement, [IsCosetGeometry],
M: RandomChamber, [IsCosetGeometry],
M: RandomFlag, [IsCosetGeometry],
```

```
M: Random, [IsAllElementsOfCosetGeometry],
M: Size, InstallMethod(Size,[IsElementsOfCosetGeometry],
M: Wrap, [IsCosetGeometry, IsPosInt, IsObject],
M: Iterator, [IsElementsOfCosetGeometry],
M: IsIncident, [IsElementOfCosetGeometry, IsElementOfCosetGeometry],
M: ParabolicSubgroups, [ IsCosetGeometry ], cg -> cg!.parabolics );
M: AmbientGroup, [ IsCosetGeometry ], cg -> cg!.group );
M: BorelSubgroup, [ IsCosetGeometry ], cg -> Intersection(cg!.parabolics) );
M: IsFlagTransitiveGeometry, [ IsCosetGeometry ],
M: IsFirmGeometry, [ IsCosetGeometry ],
M: IsThinGeometry, [ IsCosetGeometry ],
M: IsThickGeometry, [ IsCosetGeometry ],
M: IsConnected, [ IsCosetGeometry ],
M: IsResiduallyConnected, [ IsCosetGeometry ],
M: StandardFlagOfCosetGeometry, [ IsCosetGeometry ],
M: FlagToStandardFlag, [ IsCosetGeometry, IsFlagOfCosetGeometry ],
M: CanonicalResidueOfFlag, [ IsCosetGeometry, IsFlagOfCosetGeometry ],
M: ResidueOfFlag, [ IsFlagOfCosetGeometry ],
M: IncidenceGraph, InstallMethod(IncidenceGraph,[IsCosetGeometryandIsHandledByNiceMonomorphism],
M: IncidenceGraph, InstallMethod(IncidenceGraph,[IsCosetGeometry],
M: AutGroupIncidenceStructureWithNauty, [ IsCosetGeometry ],
M: CorGroupIncidenceStructureWithNauty, [ IsCosetGeometry ],
M: IsIsomorphicIncidenceStructureWithNauty, [ IsCosetGeometry, IsCosetGeometry ],
M: ViewObj, [ IsDiagram and IsDiagramRep ],
M: ViewObj, [ IsDiagram and IsDiagramRep and HasGeometryOfDiagram],
M: ViewObj, [ IsCosetGeometry and IsCosetGeometryRep ],
M: ViewObj, [ IsFlagOfCosetGeometry ],
M: PrintObj, [ IsFlagOfCosetGeometry ],
M: PrintObj, [ IsCosetGeometry and IsCosetGeometryRep ],
M: ViewObj, [ IsElementsOfCosetGeometry and IsElementsOfCosetGeometryRep ],
M: PrintObj, InstallMethod(PrintObj,"forcosetgeometry",[IsElementsOfCosetGeometryand IsElementsO
M: ViewObj, InstallMethod(ViewObj,"forcosetgeometry",[IsElementOfCosetGeometry],
M: PrintObj, InstallMethod(PrintObj,"forelementofcosetgeometry",[IsElementOfCosetGeometry],
M: ViewObj, InstallMethod(ViewObj,"forvertexofdiagram",[IsVertexOfDiagramandIsVertexOfDiagramRep
M: PrintObj, InstallMethod(PrintObj,"forvertexofdiagram",[IsVertexOfDiagramandIsVertexOfDiagramR
M: ViewObj, InstallMethod(ViewObj,"foredgeofdiagram",[IsEdgeOfDiagramandIsEdgeOfDiagramRep],
M: PrintObj, InstallMethod(PrintObj,"foredgeofdiagram",[IsEdgeOfDiagramandIsEdgeOfDiagramRep],
M: ViewObj, InstallMethod(ViewObj,"forrank2residue",[IsRank2ResidueandIsRank2ResidueRep],
M: PrintObj, InstallMethod(PrintObj,"forrank2residue",[IsRank2ResidueandIsRank2ResidueRep],
M: \=, InstallMethod(\=,[IsVertexOfDiagramandIsVertexOfDiagramRep, IsVertexOfDiagram and IsVerte
M: \=, InstallMethod(\=,[IsEdgeOfDiagramandIsEdgeOfDiagramRep, IsEdgeOfDiagram and IsEdgeOfDiagr
M: DiagramOfGeometry, InstallMethod(DiagramOfGeometry,"forflag-transitivecosetgeometry",[IsCoset
M: Display, InstallMethod(Display,[IsDiagramandIsDiagramRep],
M: DiagramOfGeometry, InstallMethod(DiagramOfGeometry,"foraprojectivespace",[IsProjectiveSpace],
M: Rk2GeoDiameter, InstallMethod(Rk2GeoDiameter,"foracosetgeometry",[IsCosetGeometry, IsPosInt],
M: Rk2GeoGonality, InstallMethod(Rk2GeoGonality,"foracosetgeometry",[IsCosetGeometry],
M: GeometryOfRank2Residue, InstallMethod(GeometryOfRank2Residue,"forarank2residue",[IsRank2Resid
M: Rank2Parameters, InstallMethod(Rank2Parameters,"foracosetgeometryofrank2",[IsCosetGeometry],
M: \<, [ IsElementOfCosetGeometry and IsElementOfCosetGeometryRep, IsElementOfCosetGeometry and
M: DiagramOfGeometry, InstallMethod(DiagramOfGeometry,[IsClassicalPolarSpace],

varieties.gi: methods
```

```
M: AlgebraicVariety, [ IsProjectiveSpace, IsPolynomialRing, IsList ],
M: AlgebraicVariety, [ IsProjectiveSpace, IsList ],
M: AlgebraicVariety, [ IsAffineSpace, IsPolynomialRing, IsList ],
M: AlgebraicVariety, [ IsAffineSpace, IsList ],
M: ProjectiveVariety, [ IsProjectiveSpace, IsPolynomialRing, IsList ],
M: ProjectiveVariety, [ IsProjectiveSpace, IsList ],
M: ViewObj, [ IsProjectiveVariety and IsProjectiveVarietyRep ],
M: PrintObj, [ IsProjectiveVariety and IsProjectiveVarietyRep ],
M: Display, [ IsProjectiveVariety and IsProjectiveVarietyRep ],
M: HermitianVariety, [IsPosInt, IsField],
M: HermitianVariety, [IsPosInt, IsPosInt],
M: HermitianVariety, [IsProjectiveSpace,IsPolynomialRing, IsPolynomial],
M: HermitianVariety, [IsProjectiveSpace, IsPolynomial],
M: ViewObj, [ IsHermitianVariety and IsHermitianVarietyRep ],
M: PrintObj, [ IsHermitianVariety and IsHermitianVarietyRep ],
M: Display, [ IsHermitianVariety and IsHermitianVarietyRep ],
M: QuadraticVariety, [IsProjectiveSpace,IsPolynomialRing, IsPolynomial],
M: QuadraticVariety, [IsProjectiveSpace, IsPolynomial],
M: QuadraticVariety, [IsPosInt, IsField, IsString],
M: QuadraticVariety, [IsPosInt, IsField],
M: QuadraticVariety, [IsPosInt, IsPosInt],
M: QuadraticVariety, [IsPosInt, IsPosInt, IsString],
M: ViewObj, [ IsQuadraticVariety and IsQuadraticVarietyRep ],
M: PrintObj, [ IsQuadraticVariety and IsQuadraticVarietyRep ],
M: Display, [ IsQuadraticVariety and IsQuadraticVarietyRep ],
M: PolarSpace, [IsProjectiveVariety and IsProjectiveVarietyRep],
M: AffineVariety, [ IsAffineSpace, IsPolynomialRing, IsList ],
M: AffineVariety, [ IsAffineSpace, IsList ],
M: AlgebraicVariety, [ IsAffineSpace, IsList ],
M: ViewObj, [ IsAffineVariety and IsAffineVarietyRep ],
M: PrintObj, [ IsAffineVariety and IsAffineVarietyRep ],
M: Display, [ IsAffineVariety and IsAffineVarietyRep ],
M: \in, [IsElementOfIncidenceStructure, IsAlgebraicVariety],
M: PointsOfAlgebraicVariety, [IsAlgebraicVariety and IsAlgebraicVarietyRep],
M: ViewObj, [ IsPointsOfAlgebraicVariety and IsPointsOfAlgebraicVarietyRep ],
M: PrintObj, [ IsPointsOfAlgebraicVariety and IsPointsOfAlgebraicVarietyRep ],
M: Points, [IsAlgebraicVariety and IsAlgebraicVarietyRep],
M: \in, [IsElementOfIncidenceStructure, IsPointsOfAlgebraicVariety], 1*SUM_FLAGS+3,
M: Iterator, [IsPointsOfAlgebraicVariety],
M: Enumerator, [IsPointsOfAlgebraicVariety],
M: AmbientSpace, [IsAlgebraicVariety and IsAlgebraicVarietyRep],
M: SegreMap, [ IsHomogeneousList ],
M: SegreMap, [IsHomogeneousList, IsField ],
M: SegreMap, [IsProjectiveSpace, IsProjectiveSpace ],
M: SegreMap, [ IsPosInt, IsPosInt, IsField ],
M: SegreMap, [ IsPosInt, IsPosInt, IsPosInt ],
M: Source, [ IsSegreMap ],
M: ViewObj, [ IsSegreMap and IsSegreMapRep ],
M: PrintObj, [ IsSegreMap and IsSegreMapRep ],
M: SegreVariety, [IsHomogeneousList],
M: SegreVariety, [IsHomogeneousList, IsField ],
M: SegreVariety, [IsProjectiveSpace, IsProjectiveSpace ],
M: SegreVariety, [ IsPosInt, IsPosInt, IsField ],
```

```
M: SegreVariety, [ IsPosInt, IsPosInt, IsPosInt ],
M: ViewObj, [ IsSegreVariety and IsSegreVarietyRep ],
M: PrintObj, [ IsSegreVariety and IsSegreVarietyRep ],
M: SegreMap, [IsSegreVariety],
M: PointsOfSegreVariety, [IsSegreVariety and IsSegreVarietyRep],
M: ViewObj, [ IsPointsOfSegreVariety and IsPointsOfSegreVarietyRep ],
M: Points, [IsSegreVariety and IsSegreVarietyRep],
M: Iterator, [IsPointsOfSegreVariety],
M: Enumerator, [IsPointsOfSegreVariety],
M: Size, [IsPointsOfSegreVariety],
M: ImageElm, [IsSegreMap, IsList],
M: \^, [IsList, IsSegreMap],
M: ImagesSet, [IsSegreMap, IsList],
M: VeroneseMap, [IsProjectiveSpace],
M: ViewObj, [ IsVeroneseMap and IsVeroneseMapRep ],
M: PrintObj, [ IsVeroneseMap and IsVeroneseMapRep ],
M: VeroneseVariety, [IsProjectiveSpace],
M: VeroneseVariety, [ IsPosInt, IsField ],
M: VeroneseVariety, [ IsPosInt, IsPosInt ],
M: ViewObj, [ IsVeroneseVariety and IsVeroneseVarietyRep ],
M: PrintObj, [ IsVeroneseVariety and IsVeroneseVarietyRep ],
M: VeroneseMap, [IsVeroneseVariety],
M: PointsOfVeroneseVariety, [IsVeroneseVariety and IsVeroneseVarietyRep],
M: ViewObj, [ IsPointsOfVeroneseVariety and IsPointsOfVeroneseVarietyRep ],
M: Points, [IsVeroneseVariety and IsVeroneseVarietyRep],
M: Iterator, [IsPointsOfVeroneseVariety],
M: Enumerator, [IsPointsOfVeroneseVariety],
M: Size, [IsPointsOfVeroneseVariety],
M: ImageElm, [IsGeometryMap, IsElementOfIncidenceStructure],
M: \^, [IsElementOfIncidenceStructure, IsGeometryMap],
M: ImagesSet, [IsGeometryMap, IsElementOfIncidenceStructureCollection],
M: Source, [ IsGeometryMap ],
M: Range, [ IsGeometryMap ],
M: GrassmannCoordinates, [ IsSubspaceOfProjectiveSpace ],
M: GrassmannMap, [ IsPosInt, IsProjectiveSpace ],
M: GrassmannMap, [ IsPosInt, IsPosInt, IsPosInt ],
M: GrassmannMap, [ IsSubspacesOfProjectiveSpace ],
M: ViewObj, [ IsGrassmannMap and IsGrassmannMapRep ],
M: PrintObj, [ IsGrassmannMap and IsGrassmannMapRep ],
M: GrassmannVariety, [ IsPosInt, IsProjectiveSpace ],
M: GrassmannVariety, [ IsSubspacesOfProjectiveSpace ],
M: ViewObj, [ IsGrassmannVariety and IsGrassmannVarietyRep ],
M: PrintObj, [ IsGrassmannVariety and IsGrassmannVarietyRep ],
M: GrassmannMap, [IsGrassmannVariety],
M: PointsOfGrassmannVariety, [IsGrassmannVariety and IsGrassmannVarietyRep],
M: ViewObj, [ IsPointsOfGrassmannVariety and IsPointsOfGrassmannVarietyRep ],
M: Points, [IsGrassmannVariety and IsGrassmannVarietyRep],
M: Iterator, [IsPointsOfGrassmannVariety],
M: Enumerator, [IsPointsOfGrassmannVariety],
M: Size, [IsPointsOfGrassmannVariety],

affinespace.gi: methods
```

```
M: AffineSpace, [ IsPosInt, IsField ],
M: AffineSpace, [ IsPosInt, IsPosInt ],
M: ViewObj, InstallMethod(ViewObj,[IsAffineSpaceandIsAffineSpaceRep],
M: PrintObj, InstallMethod(PrintObj,[IsAffineSpaceandIsAffineSpaceRep],
M: \=, [IsAffineSpace, IsAffineSpace],
M: Dimension, [ IsAffineSpace and IsAffineSpaceRep ],
M: UnderlyingVectorSpace, [ IsAffineSpace and IsAffineSpaceRep ],
M: AmbientSpace, [IsSubspaceOfAffineSpace],
M: BaseField, [IsAffineSpace and IsAffineSpaceRep],
M: BaseField, [IsSubspaceOfAffineSpace],
M: TypesOfElementsOfIncidenceStructure, [IsAffineSpace],
M: TypesOfElementsOfIncidenceStructurePlural, [IsAffineSpace],
M: VectorSpaceTransversalElement, [IsVectorSpace, IsFFECollColl, IsVector],
M: VectorSpaceTransversal, [IsVectorSpace, IsFFECollColl],
M: ViewObj, [ IsVectorSpaceTransversal and IsVectorSpaceTransversalRep ],
M: PrintObj, [ IsVectorSpaceTransversal and IsVectorSpaceTransversalRep ],
M: Wrap, [IsAffineSpace, IsPosInt, IsObject],
M: ViewObj, [ IsSubspacesOfAffineSpace and IsSubspacesOfAffineSpaceRep ],
M: PrintObj, [ IsSubspacesOfAffineSpace and IsAllSubspacesOfProjectiveSpaceRep ],
M: Display, [ IsSubspaceOfAffineSpace ],
M: AffineSubspace, [IsAffineSpace, IsRowVector, IsPlistRep],
M: AffineSubspace, [IsAffineSpace, IsRowVector],
M: AffineSubspace, [IsAffineSpace, IsCVecRep],
M: AffineSubspace, [IsAffineSpace, IsRowVector, Is8BitMatrixRep],
M: AffineSubspace, [IsAffineSpace, IsRowVector, IsGF2MatrixRep],
M: AffineSubspace, [IsAffineSpace, IsCVecRep, IsCMatRep],
M: ObjectToElement, [ IsAffineSpace, IsList],
M: ObjectToElement, [ IsAffineSpace, IsPosInt, IsList],
M: RandomSubspace, [ IsAffineSpace, IsInt ],
M: Random, [ IsSubspacesOfAffineSpace ],
M: ElementsOfIncidenceStructure, [IsAffineSpace],
M: ElementsOfIncidenceStructure, [ IsAffineSpace, IsPosInt],
M: Points, [IsAffineSpace],
M: Lines, [IsAffineSpace],
M: Planes, [IsAffineSpace],
M: Solids, [IsAffineSpace],
M: Hyperplanes, [IsAffineSpace],
M: Size, [IsSubspacesOfAffineSpace],
M: FlagOfIncidenceStructure, [ IsAffineSpace, IsSubspaceOfAffineSpaceCollection ],
M: FlagOfIncidenceStructure, [ IsAffineSpace, IsList and IsEmpty ],
M: ViewObj, [ IsFlagOfAffineSpace and IsFlagOfIncidenceStructureRep ],
M: PrintObj, [ IsFlagOfAffineSpace and IsFlagOfIncidenceStructureRep ],
M: Display, [ IsFlagOfAffineSpace and IsFlagOfIncidenceStructureRep ],
M: Enumerator, [ IsVectorSpaceTransversal ],
M: Enumerator, [ IsSubspacesOfAffineSpace ],
M: Iterator, [IsSubspacesOfAffineSpace],
M: \in, [IsSubspaceOfAffineSpace, IsAffineSpace],
M: \in, [ IsAffineSpace, IsSubspaceOfAffineSpace ],
M: \in, [IsSubspaceOfAffineSpace, IsSubspaceOfAffineSpace],
M: IsIncident, [IsSubspaceOfAffineSpace, IsSubspaceOfAffineSpace],
M: Span, [IsSubspaceOfAffineSpace, IsSubspaceOfAffineSpace],
M: Meet, [IsSubspaceOfAffineSpace, IsSubspaceOfAffineSpace],
M: IsParallel, [ IsSubspaceOfAffineSpace, IsSubspaceOfAffineSpace ],
```

```
M: ProjectiveCompletion, [ IsAffineSpace ],
M: ShadowOfElement, [IsAffineSpace, IsSubspaceOfAffineSpace, IsPosInt],
M: ShadowOfFlag, [IsAffineSpace, IsFlagOfIncidenceStructure, IsPosInt],
M: ParallelClass, [IsAffineSpace, IsSubspaceOfAffineSpace],
M: ParallelClass, [ IsSubspaceOfAffineSpace ],
M: Iterator, [IsParallelClassOfAffineSpace and IsParallelClassOfAffineSpaceRep ],
M: Size, [IsShadowSubspacesOfAffineSpace and IsShadowSubspacesOfAffineSpaceRep ],
M: Iterator, [IsShadowSubspacesOfAffineSpace and IsShadowSubspacesOfAffineSpaceRep ],
M: ViewObj, [ IsShadowSubspacesOfAffineSpace and IsShadowSubspacesOfAffineSpaceRep ],
M: ViewObj, [ IsParallelClassOfAffineSpace and IsParallelClassOfAffineSpaceRep ],
M: Points, InstallMethod(Points,[IsSubspaceOfAffineSpace],
M: Points, InstallMethod(Points,[IsAffineSpace,IsSubspaceOfAffineSpace],
M: Lines, InstallMethod(Lines,[IsSubspaceOfAffineSpace],
M: Lines, InstallMethod(Lines,[IsAffineSpace,IsSubspaceOfAffineSpace],
M: Planes, InstallMethod(Planes,[IsSubspaceOfAffineSpace],
M: Planes, InstallMethod(Planes,[IsAffineSpace,IsSubspaceOfAffineSpace],
M: Solids, InstallMethod(Solids,[IsSubspaceOfAffineSpace],
M: Solids, InstallMethod(Solids,[IsAffineSpace,IsSubspaceOfAffineSpace],
M: IncidenceGraph, [ IsAffineSpace ],


affinegroup.gi: methods

M: AffineGroup, [ IsAffineSpace ],
M: CollineationGroup, [ IsAffineSpace ],
M: \^, [IsSubspaceOfAffineSpace, IsProjGrpElWithFrob],


gpolygons.gi: methods

M: GeneralisedPolygonByBlocks, [ IsHomogeneousList ],
M: GeneralisedPolygonByIncidenceMatrix, [ IsMatrix ],
M: GeneralisedPolygonByElements, [ IsSet, IsSet, IsFunction ],
M: GeneralisedPolygonByElements, [ IsSet, IsSet, IsFunction, IsGroup, IsFunction ],
M: ViewObj, [ IsProjectivePlaneCategory and IsGeneralisedPolygonRep],
M: ViewObj, [ IsGeneralisedQuadrangle and IsGeneralisedPolygonRep],
M: ViewObj, [ IsGeneralisedHexagon and IsGeneralisedPolygonRep],
M: ViewObj, [ IsGeneralisedOctagon and IsGeneralisedPolygonRep],
M: ViewObj, [ IsGeneralisedPolygon and IsGeneralisedPolygonRep],
M: ViewObj, [ IsWeakGeneralisedPolygon and IsGeneralisedPolygonRep],
M: Order, [ IsWeakGeneralisedPolygon ],
M: UnderlyingObject, [ IsElementOfGeneralisedPolygon ],
M: ObjectToElement, [ IsGeneralisedPolygon and IsGeneralisedPolygonRep, IsPosInt, IsObject],
M: ObjectToElement, [ IsGeneralisedPolygon and IsGeneralisedPolygonRep, IsObject],
M: ElementsOfIncidenceStructure, [IsGeneralisedPolygon and IsGeneralisedPolygonRep, IsPosInt],
M: ElementsOfIncidenceStructure, [IsWeakGeneralisedPolygon and IsGeneralisedPolygonRep, IsPosInt
M: Points, [IsGeneralisedPolygon and IsGeneralisedPolygonRep],
M: Lines, [IsGeneralisedPolygon and IsGeneralisedPolygonRep],
M: ViewObj, [ IsElementsOfGeneralisedPolygon and IsElementsOfGeneralisedPolygonRep ],
M: PrintObj, [ IsElementsOfGeneralisedPolygon and IsElementsOfGeneralisedPolygonRep ],
M: Size, [IsElementsOfGeneralisedPolygon],
M: Iterator, [ IsElementsOfGeneralisedPolygon and IsElementsOfGeneralisedPolygonRep],
M: Iterator, [IsShadowElementsOfGeneralisedPolygon and IsShadowElementsOfGeneralisedPolygonRep ]
M: Random, [ IsElementsOfGeneralisedPolygon and IsElementsOfGeneralisedPolygonRep ],
M: IsIncident, [IsElementOfGeneralisedPolygon, IsElementOfGeneralisedPolygon],
```

```
M: Span, [ IsElementOfGeneralisedPolygon, IsElementOfGeneralisedPolygon ],
M: Meet, [ IsElementOfGeneralisedPolygon, IsElementOfGeneralisedPolygon ],
M: Wrap, [IsGeneralisedPolygon, IsPosInt, IsObject],
M: TypesOfElementsOfIncidenceStructurePlural, [ IsGeneralisedPolygon and IsGeneralisedPolygonRep
M: ShadowOfElement, [IsGeneralisedPolygon and IsGeneralisedPolygonRep, IsElementOfGeneralisedPol
M: ViewObj, [ IsShadowElementsOfGeneralisedPolygon and IsShadowElementsOfGeneralisedPolygonRep ]
M: Points, [ IsElementOfGeneralisedPolygon ],
M: Lines, [ IsElementOfGeneralisedPolygon ],
M: DistanceBetweenElements, [ IsElementOfGeneralisedPolygon, IsElementOfGeneralisedPolygon],
M: IncidenceGraph, [ IsGeneralisedPolygon ],
M: IncidenceMatrixOfGeneralisedPolygon, [ IsGeneralisedPolygon ],
M: CollineationGroup, [ IsGeneralisedPolygon and IsGeneralisedPolygonRep ],
M: BlockDesignOfGeneralisedPolygon, [ IsGeneralisedPolygon and IsGeneralisedPolygonRep ],
M: DistanceBetweenElements, [ IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace ],
M: IncidenceGraph, [ IsDesarguesianPlane ],
M: DistanceBetweenElements, [ IsSubspaceOfClassicalPolarSpace, IsSubspaceOfClassicalPolarSpace ]
M: IncidenceGraph, [ IsClassicalGQ ],
M: Wrap, [IsClassicalGeneralisedHexagon, IsPosInt, IsObject],
M: SplitCayleyHexagon, [ IsField and IsFinite ],
M: SplitCayleyHexagon, [ IsPosInt ],
M: SplitCayleyHexagon, [ IsClassicalPolarSpace ],
M: TwistedTrialityHexagon, [ IsField and IsFinite ],
M: TwistedTrialityHexagon, [ IsPosInt ],
M: TwistedTrialityHexagon, [ IsClassicalPolarSpace ],
M: Display, [ IsGeneralisedHexagon and IsLieGeometry ],
M: G2fining, [ IsPosInt, IsField and IsFinite ],
M: 3D4fining, [ IsField and IsFinite ],
M: CollineationGroup, [ IsClassicalGeneralisedHexagon],
M: IncidenceGraph, [ IsClassicalGeneralisedHexagon and IsGeneralisedPolygonRep ],
M: VectorSpaceToElement, [ IsClassicalGeneralisedHexagon, IsCVecRep],
M: VectorSpaceToElement, [ IsClassicalGeneralisedHexagon, IsRowVector ],
M: VectorSpaceToElement, [ IsClassicalGeneralisedHexagon, Is8BitVectorRep ],
M: VectorSpaceToElement, [ IsClassicalGeneralisedHexagon, IsPlistRep],
M: VectorSpaceToElement, [ IsClassicalGeneralisedHexagon, IsGF2MatrixRep],
M: VectorSpaceToElement, [ IsClassicalGeneralisedHexagon, Is8BitMatrixRep],
M: VectorSpaceToElement, [ IsClassicalGeneralisedHexagon, IsCMatRep],
M: \in, [ IsElementOfIncidenceStructure, IsClassicalGeneralisedHexagon ],
M: ObjectToElement, [ IsClassicalGeneralisedHexagon, IsObject],
M: ViewObj, [ IsElementOfKantorFamily ],
M: PrintObj, [ IsElementOfKantorFamily ],
M: Wrap, [IsElationGQByKantorFamily, IsPosInt, IsPosInt, IsObject],
M: \=, [IsElementOfKantorFamily, IsElementOfKantorFamily],
M: \<, [IsElementOfKantorFamily, IsElementOfKantorFamily],
M: IsKantorFamily, [IsGroup, IsList, IsList],
M: EGQByKantorFamily, [IsGroup, IsList, IsList],
M: Display, [ IsElationGQByKantorFamily ],
M: UnderlyingObject, [ IsElementOfKantorFamily ],
M: ObjectToElement, [ IsElationGQByKantorFamily, IsPosInt, IsRightCoset ],
M: ObjectToElement, [ IsElationGQByKantorFamily, IsRightCoset ],
M: ObjectToElement, [ IsElationGQByKantorFamily, IsPosInt, IsMultiplicativeElementWithInverse ],
M: ObjectToElement, [ IsElationGQByKantorFamily, IsMultiplicativeElementWithInverse ],
M: ObjectToElement, [ IsElationGQByKantorFamily, IsPosInt, IsMagmaWithInverses ],
M: ObjectToElement, [ IsElationGQByKantorFamily, IsMagmaWithInverses ],
```

```
M: IsAnisotropic, [IsFFECollColl, IsField and IsFinite],
M: IsqClan, [ IsFFECollCollColl, IsField and IsFinite],
M: qClan, [ IsFFECollCollColl, IsField ],
M: ViewObj, [ IsqClanObj and IsqClanRep ],
M: PrintObj, [ IsqClanObj and IsqClanRep ],
M: AsList, [IsqClanObj and IsqClanRep],
M: AsSet, [IsqClanObj and IsqClanRep],
M: BaseField, [IsqClanObj and IsqClanRep],
M: IsLinearqClan, [ IsqClanObj ],
M: LinearqClan, [ IsPosInt ],
M: FisherThasWalkerKantorBettenqClan, [ IsPosInt ],
M: KantorMonomialqClan, [ IsPosInt ],
M: KantorKnuthqClan, [ IsPosInt ],
M: FisherqClan, [ IsPosInt ],
M: KantorFamilyByqClan, [ IsqClanObj and IsqClanRep ],
M: EGQByqClan, [ IsqClanObj and IsqClanRep ],
M: IncidenceGraph, [ IsElationGQ and IsGeneralisedPolygonRep ],
M: BLTSetByqClan, [ IsqClanObj and IsqClanRep ],
M: EGQByBLTSet, [IsList, IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace],
M: EGQByBLTSet, [ IsList ],
M: Display, [ IsElationGQByBLTSet ],
M: DefiningPlanesOfEGQByBLTSet, [ IsElationGQByBLTSet ],
M: ObjectToElement, [ IsElationGQByBLTSet, IsPosInt, IsSubspaceOfClassicalPolarSpace],
M: ObjectToElement, [ IsElationGQByBLTSet, IsSubspaceOfClassicalPolarSpace],
M: CollineationSubgroup, [ IsElationGQByBLTSet ],
M: FlockGQByqClan, InstallMethod(FlockGQByqClan,[IsqClanObj],


orbits-stabilisers.gi: methods


M: FiningOrbit, [ IsProjectiveGroupWithFrob, IsElementOfIncidenceStructure, IsFunction],
M: FiningOrbit, [ IsProjectiveGroupWithFrob, IsSubspaceOfProjectiveSpace ],
M: FiningOrbit, [ IsProjectiveGroupWithFrob, IsSubspaceOfAffineSpace ],
M: FiningOrbit, [ IsProjectiveGroupWithFrob, CategoryCollections(IsElementOfIncidenceStructure),
M: FiningOrbit, [ IsProjectiveGroupWithFrob, CategoryCollections(IsElementOfIncidenceStructure)
M: FiningOrbits, [ IsGroup, IsHomogeneousList, IsFunction],
M: FiningOrbits, [ IsProjectiveGroupWithFrob, IsSubspaceOfProjectiveSpaceCollection and IsHomoge
M: FiningOrbits, [ IsProjectiveGroupWithFrob, IsSubspaceOfAffineSpaceCollection and IsHomogeneou
M: FiningOrbits, [ IsProjectiveGroupWithFrob, IsSubspacesOfProjectiveSpace ],
M: FiningOrbits, [ IsProjectiveGroupWithFrob, IsSubspacesOfAffineSpace ],
M: FiningOrbits, [ IsProjectiveGroupWithFrob, IsShadowSubspacesOfProjectiveSpace],
M: FiningOrbits, [ IsProjectiveGroupWithFrob, IsShadowSubspacesOfAffineSpace],
M: FiningOrbits, [ IsProjectiveGroupWithFrob, IsShadowSubspacesOfClassicalPolarSpace],
M: FiningOrbits, [ IsProjectiveGroupWithFrob, IsParallelClassOfAffineSpace],
M: FiningElementStabiliserOp, [ IsGroup, IsElementOfIncidenceStructure, IsFunction],
M: FiningStabiliser, [ IsProjectiveGroupWithFrob, IsSubspaceOfProjectiveSpace],
M: FiningStabiliser, [ IsProjectiveGroupWithFrob, IsSubspaceOfAffineSpace],
M: FiningStabiliserOrb, [IsProjectiveGroupWithFrob, IsSubspaceOfProjectiveSpace],
M: FiningStabiliserOrb, [IsProjectiveGroupWithFrob, IsSubspaceOfAffineSpace],
M: FiningSetwiseStabiliser, [IsProjectiveGroupWithFrob, IsSubspaceOfProjectiveSpaceCollection an
M: FiningSetwiseStabiliser, [IsProjectiveGroupWithFrob, IsSubspaceOfAffineSpaceCollection and Is
M: FiningStabiliserPerm, InstallMethod(FiningStabiliserPerm,[IsProjectiveGroupWithFrob,IsElement
M: FiningStabiliserPerm2, [IsProjectiveGroupWithFrob, IsElementOfIncidenceStructure],
M: FixedSubspaces, [IsProjectiveGroupWithFrob, IsProjectiveSpace],
```

A.3 The filter graph(s)
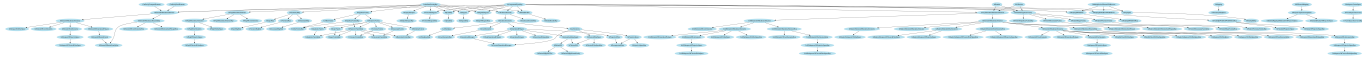
Appendix B

The finite classical groups in FinInG

B.1 Standard forms used to produce the finite classical groups.

An overview of operations is given that produce gram matrices to construct standard forms. The notion *standard form* is explained in Section 7.2, in the context of canonical and standard polar spaces.

B.1.1 CanonicalGramMatrix

▷ CanonicalGramMatrix(*type, d, f*) (operation)

Returns: a gram matrix usable as input to construct a sesquilinear form

The arguments `d` and `f` are the vector dimension and the fininte field respectively. The argument `type` is eiter "symplectic", "hermitian", "hyperbolic", "elliptic" or "parabolic".

If `type` equals "symplectic", the gram matrix is

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \ldots & 0 & 0 \\ -1 & 0 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 & 0 \\ 0 & 0 & -1 & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 0 & 1 \\ 0 & 0 & 0 & 0 & \ldots & -1 & 0 \end{pmatrix}.$$

If `type` equals "hermitian", the gram matrix is the identity matrix of dimension `d` over the field $f = GF(q)$

If `type` equals "hyperbolic", the gram matrix is

$$\begin{pmatrix} 0 & a & 0 & 0 & \ldots & 0 & 0 \\ a & 0 & 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 0 & a & \ldots & 0 & 0 \\ 0 & 0 & a & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 0 & a \\ 0 & 0 & 0 & 0 & \ldots & a & 0 \end{pmatrix}.$$

with $a = \frac{p+1}{2}$ if $p + 1 \equiv 0 \bmod 4, q = p^h$ and $a = 1$ otherwise.

If `type` equals "ellipic", the gram matrix is

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & t & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & a & \dots & 0 & 0 \\ 0 & 0 & a & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & a \\ 0 & 0 & 0 & 0 & \dots & a & 0 \end{pmatrix}.$$

with t the primitive root of $GF(q)$ if $q \equiv 1 \bmod 4$ or $q \equiv 2 \bmod 4$, and $t = 1$ otherwise; and $a = \frac{p+1}{2}$ if $p + 1 \equiv 0 \bmod 4, q = p^h$ and $a = 1$ otherwise.

If `type` equals "parabolic", the gram matrix is

$$\begin{pmatrix} t & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & a & \dots & 0 & 0 \\ 0 & a & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & a \\ 0 & 0 & 0 & \dots & a & 0 \end{pmatrix}.$$

with t the primitive root of $GF(p)$ and $a = t\frac{p+1}{2}$ if $q \equiv 5 \bmod 8$ or $q \equiv 7 \bmod 8$, and $t = a = 1$ otherwise.

There is no error message when asking for a hyperbolic, elliptic or parabolic type if the characteristic of the field f is even. In such a case, a matrix is returned, which is of course not suitable to create a bilinear form that corresponds with an orthogonal polar space. For this reason, `CanonicalGramMatrix` is not a operation designed for the user.

B.1.2 CanonicalQuadraticForm

▷ CanonicalQuadraticForm(*type, d, f*)   (operation)

Returns: a gram matrix usable as input to construct a quadratic form

The arguments `d` and `f` are the vector dimension and the fininte field respectively. The argument `type` is eiter "hyperbolic", "elliptic" or "parabolic". The matrix returned can be used to construct a quadratic form.

If `type` equals "hyperbolic", the gram matrix returned will result in the quadratic form $x_1 x_2 + x_3 x_4 + \dots + x_{d-1} x_d$

If `type` equals "elliptic", the gram matrix returned will result in the quadratic form $x_1^2 + x_1 x_2 + v x_2^2 + x_3 x_4 + \dots + x_{d-1} x_d$ with $v = \alpha^i$, with \alpha the primitive element of the multiplicative group of $GF(q)$, which is in GAP Z(q), and i the first number in $[0, 1, ..., q-2]$ for which $x^2 + x + v$ is irreducible over $GF(q)$.

If `type` equals "parabolic", the gram matrix returned will result in the quadratic form $x_1^2 + x_2 x_3 + \dots x_{d-1} x_d$

This function is intended to be used only when the characteristic of `f` is two, but there is no error message is this is not the case. For this reason, `CanonicalQuadraticForm` is not an operation designed for the user.

B.2 Direct commands to construct the projective classical groups in FinInG

As explained in Chapter 7, Section 7.7, we have assumed that the user asks for the projective classical groups in an indirect way, i.e. as a (subgroup) of the collineation group of a classical polar space. However, shortcuts to these groups exist. More information on the notations can be found in Section 7.7.

B.2.1 SOdesargues

▷ SOdesargues(*e, d, f*)   (operation)

Returns: the special isometry group of a canonical orthogonal polar space

The argument *e* determines the type of the othogonal polar space, i.e. -1,0,1 for an elliptic, hyperbolic, parabolic orthogonal space, respectively. The argument *d* is the dimension of the underlying vector space, *f* is the finite field. The method relies on SO, a GAP command returning the appropriate matrix group. Internally, the invariant form is asked, and the base chage to our canonical form is obtained using the package form

```
_____ Example _____
gap> SOdesargues(-1,6,GF(9));
PSO(-1,6,9)
gap> SOdesargues(0,7,GF(11));
PSO(0,7,11)
gap> SOdesargues(1,8,GF(16));
PSO(1,8,16)
```

B.2.2 GOdesargues

▷ GOdesargues(*e, d, f*)   (operation)

Returns: the isometry group of a canonical orthogonal polar space

The argument *e* determines the type of the othogonal polar space, i.e. -1,0,1 for an elliptic, hyperbolic, parabolic orthogonal space, respectively. The argument *d* is the dimension of the underlying vector space, *f* is the finite field. The method relies on GO, a GAP command returning the appropriate matrix group. Internally, the invariant form is asked, and the base chage to our canonical form is obtained using the package form

```
_____ Example _____
gap> GOdesargues(-1,6,GF(9));
PGO(-1,6,9)
gap> GOdesargues(0,7,GF(11));
PGO(0,7,11)
gap> GOdesargues(1,8,GF(16));
PGO(1,8,16)
```

B.2.3 SUdesargues

▷ SUdesargues(*d, f*)   (operation)

Returns: the special isometry group of a canonical hermitian polar space

The argument `d` is the dimension of the underlying vector space, `f` is the finite field. The method relies on `SU`, a GAP command returning the appropriate matrix group. Internally, the invariant form is asked, and the base chage to our canonical form is obtained using the package `form`

```
——————— Example ———————
gap> SUdesargues(4,GF(9));
PSU(4,3^2)
```

B.2.4 GUdesargues

▷ GUdesargues(`d, f`)   (operation)

Returns: the isometry/similarity group of a canonical hermitian polar space

The argument `d` is the dimension of the underlying vector space, `f` is the finite field. The method relies on `GU`, a GAP command returning the appropriate matrix group. Internally, the invariant form is asked, and the base chage to our canonical form is obtained using the package `form`

```
——————— Example ———————
gap> GUdesargues(4,GF(9));
PGU(4,3^2)
```

B.2.5 Spdesargues

▷ Spdesargues(`d, f`)   (operation)

Returns: the (special) isometry group of a canonical symplectic polar space

The argument `d` is the dimension of the underlying vector space, `f` is the finite field. The method relies on `Sp`, a GAP command returning the appropriate matrix group. Internally, the invariant form is asked, and the base chage to our canonical form is obtained using the package `form`

```
——————— Example ———————
gap> Spdesargues(6,GF(11));
PSp(6,11)
```

B.2.6 GeneralSymplecticGroup

▷ GeneralSymplecticGroup(`d, f`)   (operation)

Returns: the isometry group of a canonical symplectic form

The argument `d` is the dimension of the underlying vector space, `f` is the finite field. Internally, the invariant form is asked, and the base chage to our canonical form is obtained using the package `form`

```
——————— Example ———————
gap> GeneralSymplecticGroup(6,GF(7));
GSp(6,7)
```

B.2.7 GSpdesargues

▷ GSpdesargues(`d, f`)   (operation)

Returns: the similarity group of a canonical symplectic polar space

The argument `d` is the dimension of the underlying vector space, `f` is the finite field. The method relies on Sp, a GAP command returning the appropriate matrix group. Internally, the invariant form is asked, and the base chage to our canonical form is obtained using the package form

—————————————— Example ——————————————
```
gap> GSpdesargues(4,GF(9));
PGSp(4,9)
```

B.2.8 GammaSp

▷ GammaSp(`d`, `f`)   (operation)

Returns: the collineation group of a canonical symplectic polar space
The argument `d` is the dimension of the underlying vector space, `f` is the finite field. The method relies on GeneralSymplecticGroup, and adds the frobenius automorphism.

—————————————— Example ——————————————
```
gap> GammaSp(4,GF(9));
PGammaSp(4,9)
```

B.2.9 DeltaOminus

▷ DeltaOminus(`d`, `f`)   (operation)

Returns: the similarity group of a canonical elliptic orthogonal polar space
The argument `d` is the dimension of the underlying vector space, `f` is the finite field. The method relies on GOdesargues, and computes the generators to be added.

—————————————— Example ——————————————
```
gap> DeltaOminus(6,GF(7));
PDeltaO-(6,7)
```

B.2.10 DeltaOplus

▷ DeltaOplus(`d`, `f`)   (operation)

Returns: the similarity group of a canonical hyperbolic orthogonal polar space
The argument `d` is the dimension of the underlying vector space, `f` is the finite field. The method relies on GOdesargues, and computes the generators to be added.

—————————————— Example ——————————————
```
gap> DeltaOplus(8,GF(7));
PDeltaO+(8,7)
```

B.2.11 GammaOminus

▷ GammaOminus(`d`, `f`)   (operation)

Returns: the collineation group of a canonical elliptic orthogonal polar space
The argument `d` is the dimension of the underlying vector space, `f` is the finite field. The method relies on DeltaOminus, and computes the generators to be added.

——————————— Example ———————————
```
gap> GammaOminus(4,GF(25));
PGammaO-(4,25)
```

B.2.12   GammaO

▷ GammaO(*d, f*)   (operation)

    Returns: the collineation group of a canonical parabolic orthogonal polar space

    The argument *d* is the dimension of the underlying vector space, *f* is the finite field. The method relies on GO, a GAP command returning the appropriate matrix group. Internally, the invariant form is asked, and the base chage to our canonical form is obtained using the package form. Furthermore, the generators to be added are computed.

——————————— Example ———————————
```
gap> GammaO(5,GF(49));
PGammaO(5,49)
```

B.2.13   GammaOplus

▷ GammaOplus(*d, f*)   (operation)

    Returns: the collineation group of a canonical hyperbolic orthogonal polar space

    The argument *d* is the dimension of the underlying vector space, *f* is the finite field. The method relies on DeltaOplus, and computes the generators to be added.

——————————— Example ———————————
```
gap> GammaOplus(6,GF(64));
PGammaO+(6,64)
```

B.2.14   GammaU

▷ GammaU(*d, f*)   (operation)

    Returns: the collineation group of a canonical hermitian variety

    The argument *d* is the dimension of the underlying vector space, *f* is the finite field. The method relies on GU, a GAP command returning the appropriate matrix group. Internally, the invariant form is asked, and the base chage to our canonical form is obtained using the package form. Furthermore, the generators to be added are computed.

——————————— Example ———————————
```
gap> GammaU(4,GF(81));
PGammaU(4,9^2)
```

B.2.15   G2fining

▷ G2fining(*d, f*)   (operation)

    Returns: the chevalley group G_2(q)

    This group is the group of projectivities stabilising the split Cayley hexagon embedded in the paraboloc quadric $Q(6,q) : X_0X_4 + X_1X_5 + X_2X_6 = X_3^2$. *f* must be a finite field and *d* must be 5 or 6.

When *d* is 5, *F* must be a field of even order, and then the returned group consists of projectivities of $W(5,q)$. The generators of this group are described exaplicitely in [VM98], Appendix D. A correction can be found in [Off00]. However, also this source contains a mistake.

B.2.16   3D4fining

▷ 3D4fining(*d*, *f*)   (operation)
    Returns: the chevalley group 3D4(q)
    d must equal 7 and *f* must be a field of order q^3 This group is the group of collineations stabilising the twisted triality hexagon embedded in the hyperbolic quadric $Q + (7,q)$: $X_0X_4 + X_1X_5 + X_2X_6 + X_3X_7$ The generators of this group are described exaplicitely in [VM98], Appendix D.

B.3   Basis of the collineation groups

The GenSS uses a function FindBasePointCandidates taking a group as one of the arguments. From a geometrical point of view, it is straightforward to construct a basis for a collineation group for the action on projective points.

B.3.1   FindBasePointCandidates

▷ FindBasePointCandidates(*g*, *opt*, *i*, *parentS*)   (operation)
    Returns: a record
    The returned record contains the base points for the action, and some other fields. The information in the other fields is determined from the arguments *opt* and *i*. More information on these details can be found in the manual of GenSS.
    Variations on this version of BasePointCandidates are found in FinInG used in previous versions of GenSS. These variations are already or will become obsolete in the (near) future.

Appendix C

Low level functions for morphisms

C.1 Field reduction and vector spaces

C.1.1 ShrinkVec

▷ ShrinkVec(*f1, f2, v, basis*) (operation)
▷ ShrinkVec(*f1, f2, v*) (operation)
    Returns: a vector

    The argument *f2* is a subfield of *f1* and v is vector in a vectorspace V over *f2*. The second flavour Returns return the vector of length d/t, where $d = dim(V)$, and $t = [f1 : f2]$. The first flavour uses the natural basis `Basis(AsVectorSpace(f2,f1))`. It is not checked whether *f2* is a subfield of *f1*, but it is checked whether the lengt of *v* is a multiple of the degree of the field extension.

C.1.2 ShrinkMat

▷ ShrinkMat(*basis, matrix*) (operation)
▷ ShrinkMat(*f1, f2, matrix*) (operation)
    Returns: a matrix

    Let $K = GF(q)$ and let $L = GF(q^d)$. Asumme that B is a basis for L as K vector space. Let $A = (a_{ij})$ be a matrix over L. The result of `BlownUpMat(B,A)` is the matrix $M = (m_{ij})$, where each entry $a = a_{ij}$ is replaced by the $d \times d$ matrix M_a , representing the linear map $x \mapsto ax$ with respect to the basis B. This means that if $B = \{b_1, b_2, \ldots, b_d\}$, then the row j is row of the d coefficients of ab_j with resepect to the basis B. The operation `ShrinkMat` implements the converse of `BlownUpMat`. It is checked if the input is a blown up matrix as follows. Let A be a $dm \times dn$ matrix. For each $d \times d$ block, say M, we need to check that the set $\{b_i^{(}-1)\sum_{j=1}^{d} m_{ij}b_j : i \in \{1,..,d\}\}$. has size one, since the unique element in that case is the element $a \in L$ represented as a linear map by M with respect to the basis B.

    The first flavour of this operation requires a given basis as first argument. The second flavour requires a field *f1* and a subfield *f2* as first two arguments and calls the first flavour with `Basis(AsVectorSpace(f2,f1))` as basis. It is not checked whether *f2* is a subfield of *f1*.

C.1.3 BlownUpProjectiveSpace

▷ BlownUpProjectiveSpace(*basis, pg1*) (operation)
    Returns: a projective space

Let `basis` be a basis of the field $GF(q^t)$ that is an extension of the basefield of the $r - 1$ dimensional projective space `pg1`. This operation returns the $rt - 1$ dimensional projective space over $GF(q)$. The basis itself is only used to determine the field $GF(q^t)$.

C.1.4 BlownUpProjectiveSpaceBySubfield

▷ BlownUpProjectiveSpaceBySubfield(*subfield, pg*)   (operation)

Returns: a projective space

Blows up a projective space *pg* with respect to the standard basis of the basefield of *pg* over the *subfield*.

C.1.5 BlownUpSubspaceOfProjectiveSpace

▷ BlownUpSubspaceOfProjectiveSpace(*basis, subspace*)   (operation)
▷ BlownUpSubspaceOfProjectiveSpace(*basis, space*)   (operation)

Returns: a subspace of a projective space

The first flavour blows up a *subspace* of a projective space with respect to the *basis* using field reduction and returns a subspace of the projective space obtained from blowing up the ambient projective space of *subspace* with respect to *basis* using field reduction. This operation relies on `BlownUpMat`.

C.1.6 BlownUpSubspaceOfProjectiveSpaceBySubfield

▷ BlownUpSubspaceOfProjectiveSpaceBySubfield(*subfield, subspace*)   (operation)

Returns: a subspace of a projective space

Blows up a *subspace* of a projective space with respect to the standard basis of the basefield of *subspace* over the *subfield*, using field reduction and returns it a subspace of the projective space obtained from blowing up the ambient projective space of *subspace* over the subfield.

C.1.7 IsDesarguesianSpreadElement

▷ IsDesarguesianSpreadElement(*basis, subspace*)   (operation)

Returns: true or false

Checks wether the *subspace* is a subspace which is obtained from a blowing up a projective point using field reduction with respect to *basis*.

C.2 Field reduction and forms

The embedding of polar spaces by field reduction is explained in detail in Section 10.5.3, and relies on the following three operations.

C.2.1 QuadraticFormFieldReduction

▷ QuadraticFormFieldReduction(*qf1, f2, alpha, basis*)   (operation)
▷ QuadraticFormFieldReduction(*qf1, f2, alpha*)   (operation)

Returns: a quadratic form

Let f be quadratic form determining a polar space over the field L This operation returns the quadratic form T_{\alpha} \circ f \circ \Phi^{-1} over a subfield K of L, as explained in Section 10.5.3.

C.2.2 BilinearFormFieldReduction

▷ BilinearFormFieldReduction(*bil11, f2, alpha, basis*)　　　　　(operation)
▷ BilinearFormFieldReduction(*bil11, f2, alpha*)　　　　　(operation)

Returns: a bilinear form

Let f be bilinear form determining a polar space over the field L This operation returns the quadratic form T_{\alpha} \circ f \circ \Phi^{-1} over a subfield K of L, as explained in Section 10.5.3.

C.2.3 HermitianFormFieldReduction

▷ HermitianFormFieldReduction(*hf1, f2, alpha, basis*)　　　　　(operation)
▷ HermitianFormFieldReduction(*hf1, f2, alpha*)　　　　　(operation)

Returns: a hermitian form

Let f be bilinear form determining a polar space over the field L This operation returns the quadratic form T_{\alpha} \circ f \circ \Phi^{-1} over a subfield K of L, as explained in Section 10.5.3.

C.3 Low level functions

C.3.1 PluckerCoordinates

▷ PluckerCoordinates(*matrix*)　　　　　(operation)
▷ InversePluckerCoordinates(*vector*)　　　　　(operation)

The first operation can also take a matrix representing a line of $PG(3,q)$ as argument. No checks are performed in this case. It returns the plucker coordinates of the argument as list of finite field elements. The second operation is the inverse of the first. No check is performed whether the argument represents a point of the correct hyperbolic quadric. Both operations are to be used internally only.

C.3.2 IsomorphismPolarSpacesProjectionFromNucleus

▷ IsomorphismPolarSpacesProjectionFromNucleus(*quadric, w, boolean*)　　　　　(operation)

This operation returns the isomorphism between a parabolic quadric and a sympletic polar space. Although it is checked whether the base field and rank of both polar spaces are equal, this operation is menat for internal use only. This operation is called by the operation IsomorphismPolarSpaces.

C.3.3 IsomorphismPolarSpacesNC

▷ IsomorphismPolarSpacesNC(*ps1, ps2*)　　　　　(operation)
▷ IsomorphismPolarSpacesNC(*ps1, ps2, boolean*)　　　　　(operation)

IsomorphismPolarSpacesNC is the version of IsomorphismPolarSpaces where no checks are built in, and which is only applicable when the two polar spaces are equivalent. As no checks are built in, this operation is to be used internally only.

C.3.4 NaturalEmbeddingBySubspaceNC

▷ NaturalEmbeddingBySubspaceNC(*geom1, geom2, v*)   (operation)

The operation NaturalEmbeddingBySubspaceNC is the "no check" version of NaturalEmbeddingBySubspace.

C.3.5 NaturalProjectionBySubspaceNC

▷ NaturalProjectionBySubspaceNC(*ps, v*)   (operation)

The operation NaturalEmbeddingBySubspaceNC is the "no check" version of NaturalEmbeddingBySubspace.

References

[BC13]   Francis Buekenhout and Arjeh M. Cohen. *Diagram geometry*, volume 57 of *Ergebnisse der Mathematik und ihrer Grenzgebiete. 3. Folge. A Series of Modern Surveys in Mathematics [Results in Mathematics and Related Areas. 3rd Series. A Series of Modern Surveys in Mathematics]*. Springer, Heidelberg, 2013. Related to classical groups and buildings. 38

[BS74]   Francis Buekenhout and Ernest Shult. On the foundations of polar geometry. *Geometriae Dedicata*, 3:155–170, 1974. 114

[Cam00a] Peter J. Cameron. *Classical Groups*. Online notes, http://www.maths.qmul.ac.uk/~pjc/class_gps/, 2000. 100

[Cam00b] Peter J. Cameron. *Projective and Polar Spaces*. Online notes, http://www.maths.qmul.ac.uk/~pjc/pps/, 2000. 115

[Gil08]   Nick Gill. Polar spaces and embeddings of classical groups. *to appear in New Zealand J. Math.*, 2008. 179

[HT91]   J. W. P. Hirschfeld and J. A. Thas. *General Galois geometries*. Oxford Mathematical Monographs. The Clarendon Press Oxford University Press, New York, 1991. Oxford Science Publications. 68, 106, 115

[KL90]   Peter Kleidman and Martin Liebeck. *The subgroup structure of the finite classical groups*, volume 129 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, Cambridge, 1990. 18, 106, 116, 178

[LVdV]   Michel Lavrauw and Geertrui Van de Voorde. Field reduction and linear sets in finite geometry. *to appear in AMS Contemp. Math.* 174, 179

[Off00]   Alan Offer. *Spreads and Ovoids of the Split Cayley Hexagon*. PhD thesis, The university of Adelaide, 2000. 300

[PT84]   S. E. Payne and J. A. Thas. *Finite generalized quadrangles*, volume 110 of *Research Notes in Mathematics*. Pitman (Advanced Publishing Program), Boston, MA, 1984. 223

[Tit59]   Jacques Tits. Sur la trialité et certains groupes qui s'en déduisent. *Inst. Hautes Études Sci. Publ. Math.*, (2):13–60, 1959. 218

[Tit74]   Jacques Tits. *Buildings of spherical type and finite BN-pairs*. Springer-Verlag, Berlin, 1974. Lecture Notes in Mathematics, Vol. 386. 114

[Vel59]   F. D. Veldkamp. Polar geometry. I, II, III, IV, V. *Nederl. Akad. Wetensch. Proc. Ser. A 62;
          63 = Indag. Math. 21 (1959), 512-551*, 22:207–212, 1959. 114

[VM98]   Hendrik Van Maldeghem.   *Generalized polygons*.   Modern Birkhäuser Classics.
          Birkhäuser/Springer Basel AG, Basel, 1998.   [2011 reprint of the 1998 original]
          [MR1725957]. 217, 218, 300

[VY65a]   Oswald Veblen and John Wesley Young. *Projective geometry. Vol. 1*. Blaisdell Publishing
          Co. Ginn and Co. New York-Toronto-London, 1965. 68, 151

[VY65b]   Oswald Veblen and John Wesley Young. *Projective geometry. Vol. 2 (by Oswald Veblen)*.
          Blaisdell Publishing Co. Ginn and Co. New York-Toronto-London, 1965. 68, 151

Index