GAP 4 Package FinInG

Finite Incidence Geometry

1.01

2012

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

Copyright

Acknowledgements

The development phase of FinInG started roughly in 2005. The idea to write a package for projective geometry in GAP was known 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.

Contents

Chapter 1

Introduction

1.1 Philosophy

FinInG 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 Overview over this manual

Chapter 2 describes the installation of this package. Chapter 3 contains some extended examples to introduce the user to the basic functionality and philosophy to get started. Chapter 4 contains a rigourous discription of the basic structures. This chapter can be omitted at first reading, since the set of consequent chapters is also self contained. Chapters 5, 6 and 7 deal with projective spaces, projective semilinear groups, and polarities of projective spaces, respectively. In Chapter 8 the functionality for classical polar spaces is treated and in Chapter 10 affine spaces and their groups are considered. *Geometry morphisms* between various geometries that are available in the package, are introduced and discussed in Chapter 11. The final three chapters, 12, 13, and 14 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.

1.3 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,.

[scale=0.5]devteamstasep08.jpg

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.

[scale=0.5]devteamvicapr11.jpg

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

[scale=0.5]devteamchidec11.jpg

The same survivors, staring at the destiny.

[scale=0.5]devteamdestiny.jpg

Chapter 2

Installation of the FinInG-Package

This package requires the GAP packages GAPDoc, version 0.99 or higher, Forms, version 1.2 or higher, Orb, version 2.0 or higher, GenSS, version 0.9 or higher, and GRAPE, version 4.3 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. Currently, the packages GAPDoc, GRAPE, Forms and Design are accepted GAP packages, the packages orb, GenSS and IO are deposited. All can be found through the GAP website. The recent development stage of FinInG is based on GAP4r4p12. We have done testing using the beta release of GAP4r5, and no installation differences occured. In this section, we describe in detail the installation procedure for FinInG, assuming the use of GAP4r4. We have also (succesfully) tested this procedure with GAP4r5 beta.

2.1 Installing FinInG under UNIX like systems

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.

$$/usr/local/gap4r4/pkg/$$

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

$$gunzip fining-...-tar.gz$$

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

$$tar-xf finin-...-tar$$

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

$$/usr/local/gap4r4/pkg/fining/.$$

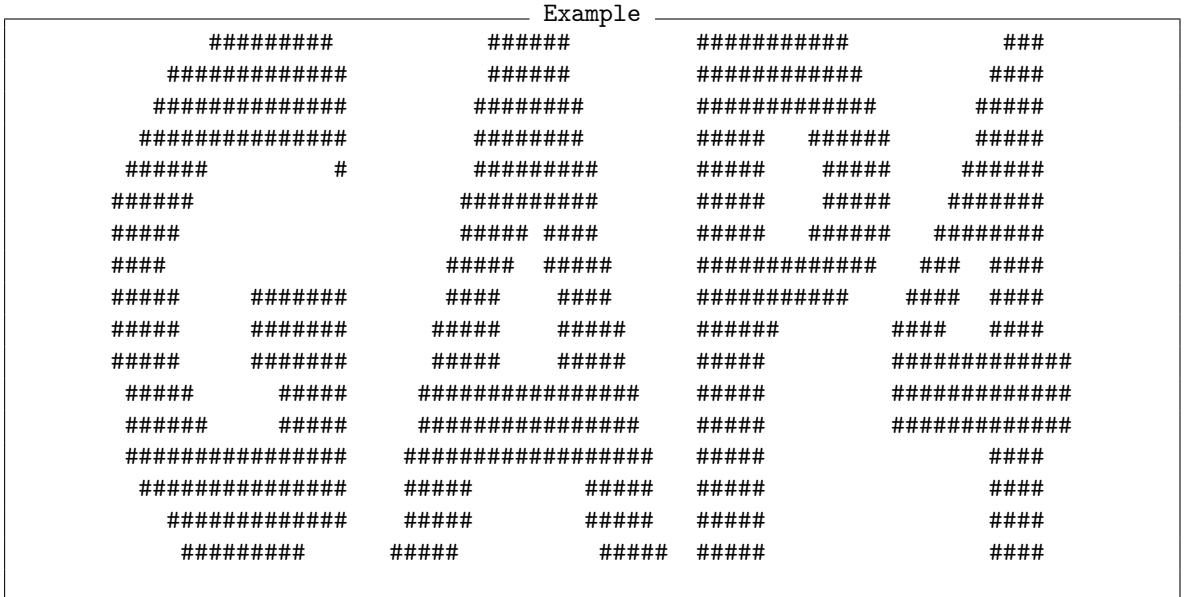
This directory contains a subdirectory "./doc", containing an html and pdf version of the manual. The html version is accessible by opening the file "chap0.html" in your favorite browser. The pdf version of the manual can be found in the file "manual.pdf". Please notice that you can unpack your archive

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 gap4r4. If you installed FinInG in your local pkg directory, e.g. "/home/yourself/pkg/", then move to "/home/yourself", and issue the command

$$gap4r4 - l"/usr/local/gap4r4; ./"$$

This will cause gap to startup and use as pkg directory both its own central pkg directory, i.e. "/usr/loca/gap4r4/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



```
      Information at:  http://www.gap-system.org
      Try '?help' for help. See also  '?copyright' and  '?authors'

   Loading the library. Please be patient, this may take a while.
GAP4, Version: 4.4.12 of 17-Dec-2008, i686-apple-darwin10.8.0-gcc
Components:  small 2.1, small2 2.0, small3 2.0, small4 1.0, small5 1.0,
             small6 1.0, small7 1.0, small8 1.0, small9 1.0, small10 0.2,
             id2 3.0, id3 2.1, id4 1.0, id5 1.0, id6 1.0, id9 1.0, id10 0.1,
             trans 1.0, prim 2.1  loaded.
Packages:    GAPDoc 1.3, IO 3.3, TomLib 1.1.4  loaded.
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. The easiest way to find out 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 4.3  (GRaph Algorithms using PErmutation groups),
by L.H.Soicher@qmul.ac.uk.

true
gap> LoadPackage("orb");
------------------------------------------------------------------------------
Loading  orb 3.8 (orb - Methods to enumerate orbits)
by Juergen Mueller (http://www.math.rwth-aachen.de/~Juergen.Mueller),
   Max Neunhoeffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef), and
   Felix Noeske (http://www.math.rwth-aachen.de/~Felix.Noeske).
------------------------------------------------------------------------------
true
gap> LoadPackage("genss");
------------------------------------------------------------------------------
Loading  genss 1.3 (genss - generic Schreier-Sims)
by Max Neunhoeffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef) and
   Felix Noeske (http://www.math.rwth-aachen.de/~Felix.Noeske).
------------------------------------------------------------------------------
true
```

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

Example

```
Packages:    GAPDoc 1.3, IO 3.3, TomLib 1.1.4  loaded.
gap> LoadPackage("fining");
------------------------------------------------------------------
Loading 'Forms' 1.2.3 (29/08/2011)
by John Bamberg (http://school.maths.uwa.edu.au/~bamberg/)
   Jan De Beule (http://cage.ugent.be/~jdebeule)
For help, type: ?Forms
------------------------------------------------------------------
------------------------------------------------------------------------------
Loading  orb 3.8 (orb - Methods to enumerate orbits)
by Juergen Mueller (http://www.math.rwth-aachen.de/~Juergen.Mueller),
   Max Neunhoeffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef), and
   Felix Noeske (http://www.math.rwth-aachen.de/~Felix.Noeske).
------------------------------------------------------------------------------
------------------------------------------------------------------------------
Loading  genss 1.3 (genss - generic Schreier-Sims)
by Max Neunhoeffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef) and
   Felix Noeske (http://www.math.rwth-aachen.de/~Felix.Noeske).
------------------------------------------------------------------------------

Loading  GRAPE 4.3  (GRaph Algorithms using PErmutation groups),
by L.H.Soicher@qmul.ac.uk.

------------------------------------------------------------------
```

```
      --------------      --------      ---------   -------------
      ___  ____/__(_)_________  _/________  ____/   __<  /__  __
      __  /_   __  /__  __ __  / __  __   / __      __  / _  / / /
      _  __/   _  / _  / / /_/ /  _  / / / /_/ /     _  /__/ /_/ /
      /_/      /_/  /_/ /_//___/  /_/ /_/____/      /_/_(_)____/
-------------------------------------------------------------------
Loading  FinInG 1.0 (Finite Incidence Geometry)
by John Bamberg (http://cage.ugent.be/~bamberg)
   Anton Betten (http://www.math.colostate.edu/~betten)
   Jan De Beule (http://cage.ugent.be/~jdebeule)
   Philippe Cara (http://homepages.vub.ac.be/~pcara)
   Michel Lavrauw (http://cage.ugent.be/~ml)
   Max Neunhoeffer (http://www.math.rwth-aachen.de/~Max.Neunhoeffer)
For help, type: ?FinInG
-------------------------------------------------------------------
loading: geometry, liegeometry, group, projectivespace, correlations, polarspace/morphis
ms, enumerators, diagram, varieties, affinespace/affinegroup, gpolygons
true
gap>
```

For the sake of completeness, we show the output under GAP4r5

Example

```
 *********   GAP, Version 4.5.2(beta) of 27-Nov-2011 (free software, GPL)
 *  GAP  *   http://www.gap-system.org
 *********   Architecture: x86_64-apple-darwin10.8.0-gcc-default64
 Libs used:  gmp
 Loading the library and packages ...
 Components: trans 1.0, prim 2.1, small* 1.0, id* 1.0
 Packages:   GAPDoc 1.4, IO 3.3, TomLib 1.2.2
 Try '?help' for help. See also  '?copyright' and  '?authors'
gap> LoadPackage("fining");
-------------------------------------------------------------------
Loading 'Forms' 1.2.3 (29/08/2011)
by John Bamberg (http://school.maths.uwa.edu.au/~bamberg/)
   Jan De Beule (http://cage.ugent.be/~jdebeule)
For help, type: ?Forms
-------------------------------------------------------------------
-------------------------------------------------------------------------------------------
Loading  orb 3.8 (orb - Methods to enumerate orbits)
by Juergen Mueller (http://www.math.rwth-aachen.de/~Juergen.Mueller),
   Max Neunhoeffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef), and
   Felix Noeske (http://www.math.rwth-aachen.de/~Felix.Noeske).
Homepage: http://www-groups.mcs.st-and.ac.uk/~neunhoef/Computer/Software/Gap/orb.html
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
Loading  genss 1.3 (genss - generic Schreier-Sims)
by Max Neunhoeffer (http://www-groups.mcs.st-and.ac.uk/~neunhoef) and
   Felix Noeske (http://www.math.rwth-aachen.de/~Felix.Noeske).
Homepage: http://www-groups.mcs.st-and.ac.uk/~neunhoef/Computer/Software/Gap/genss.html
-------------------------------------------------------------------------------------------

Loading  GRAPE 4.3  (GRaph Algorithms using PErmutation groups),
by L.H.Soicher@qmul.ac.uk.
```

```
loading: geometry, liegeometry, group, projectivespace, correlations, polarspace/morphis
ms, enumerators, diagram, varieties, affinespace/affinegroup, gpolygons
-------------------------------------------------------------------
      ______________       ________      _________   _____________
      ___  ____/__(_)_____________  _/________  ____/   __<  /__  __
      __  /_   __  /__  __ __  / __  __   / __      __  / _  / / /
      _  __/   _  / _  / / /_/ /  _  / / / /_/ /     _  /__/ /_/ /
      /_/      /_/  /_/ /_//___/  /_/ /_/____/      /_/_(_)____/
-------------------------------------------------------------------
Loading  FinInG 1.0 (Finite Incidence Geometry)
by John Bamberg (http://cage.ugent.be/~bamberg)
   Anton Betten (http://www.math.colostate.edu/~betten)
   Jan De Beule (http://cage.ugent.be/~jdebeule)
   Philippe Cara (http://homepages.vub.ac.be/~pcara)
   Michel Lavrauw (http://cage.ugent.be/~ml)
   Max Neunhoeffer (http://www.math.rwth-aachen.de/~Max.Neunhoeffer)
For help, type: ?FinInG
-------------------------------------------------------------------
true
gap>
```

2.2 Installing other required packages

The packages orb and IO require compilation and the linking of some binaries to the gap kernel. On some UNIX like system, dynamical linking seems to be impossible. In such a case, the binaries must be statically linked to the GAP kernel. We describe the procedure here, assuming that we have to link the binaries of both packages to the gap-kernel. We assume that gap is installed in the directory

$$/usr/local/gap4r4/$$

and that we use the central ./pkg/ directory for the packages IO and orb

1. Move to your gap directory, which is in this example /usr/loca/gap4r4/. Move further to the /bin directory, and loc
2. We are in the directory '/usr/loca/gap4r4/bin/i686-apple-darwin10.8.0-gcc/'. Locate the file 'gac'. This is a shell s
3. Recall that the pkg directoy is now '/usr/local/gap4r4/pkg/'. So the source files of orb and io are to be found in '/u
4. Issue the command './gac gap-static -p "-DIOSTATIC -I/usr/local/gap4r4/pkg/io/bin/i686-apple-darwin10.8.0-gcc
5. Adapt the shell script to start GAP. In this example, we assumed GAP is started up using 'gap'. Locate this shell s

Chapter 3

Examples

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

3.1 A simple example to get you started

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 exploring a bit the particular properties of the projective plane $PG(2,4)$. The projective plane is initalised, 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"
```

3.2 Polar Spaces

3.2.1 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.

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 ] ]
```

3.2.2 $W(3,3)$ inside $W(5,3)$

In this example, we embed $W(3,3)$ in $W(5,3)$.

Example

```
gap> w3 := SymplecticSpace(3, 3);
W(3, 3)
gap> w5 := SymplecticSpace(5, 3);
W(5, 3)
gap> pg := AmbientSpace( w5 );
ProjectiveSpace(5, 3)
gap> solids := ElementsOfIncidenceStructure(pg, 4);
<solids of ProjectiveSpace(5, 3)>
gap> iter := Iterator( solids );
<iterator>
gap> perp := PolarityOfProjectiveSpace( w5 );
<polarity of PG(5, GF(3)) >
gap> solid := NextIterator( iter );
<a solid in ProjectiveSpace(5, 3)>
gap> solid^perp;
<a line in ProjectiveSpace(5, 3)>
gap> em := NaturalEmbeddingBySubspace( w3, w5, solid );
<geometry morphism from <Elements of W(3, 3)> to <Elements of W(5, 3)>>
gap> points := Points( w3 );
<points of W(3, 3)>
gap> points2 := ImagesSet(em, AsSet(points));;
gap> ForAll(points2, x -> x in solid);
true
```

3.2.3 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
```

3.2.4 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...

Example

```
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
```

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 );
<projective collineation group of size 9170703360 with 2 generators>
gap> stabovoid := SetwiseStabilizer(g, OnProjSubspaces, ovoid)!.setstab;
#I  Computing adjusted stabilizer chain...
<projective collineation group with 12 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
```

3.3 Elation generalised quadrangles

3.3.1 The classical q-clan

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

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...
#I  Computing points from Kantor family...
#I  Computing lines from Kantor family...
<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 a Kantor family>
gap> x := Random(elations);
[ [ Z(3)^0, Z(3), 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, Z(3)^0 ], [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ]
gap> OnKantorFamily(p,x);
<a point of a Kantor family>
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
```

3.3.2 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 \phi 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), Z(3^2)^2, Z(3^2)^6 ]
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...
#I  Computing points from Kantor family...
#I  Computing lines from Kantor family...
<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  No intertwiner computed. One of the polar spaces must have a collineation group computed
#I  Now embedding dual BLT-set into W(5,q)...
#I  Computing points(1) of Knarr construction...
#I  Computing lines(1) of Knarr construction...
#I  Computing points(2) of Knarr construction...
#I  Computing lines(2) of Knarr construction...please wait
#I  Computing elation group...
<EGQ of order [ 81, 9 ] and basepoint [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3),
   0*Z(3), 0*Z(3) ]>
```

3.4 Diagram geometries

3.4.1 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. In a later version of our package, we might use `neato` to make a diagram with straight lines. Here is what the output looks like: PSL211.jpg

Chapter 4

Incidence Geometry

The term geometry, or incidence geometry, is interpreted in a broad sense in this package. The basis for the construction of the objects in this package is an abstract incidence geometry consisting of elements, types, and an incidence relation. To be more specific, an *incidence geometry* 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"). There are two axioms: (i) no two elements of the same type are incident; (ii) every maximal flag 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.

FinInG concerns itself primarily with the most commonly studied incidence geometries of rank more than 2: projective spaces, polar spaces, and affine spaces. However, some facility with generalised polygons has been included. 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 5.

In this chapter we describe functionality that is DECLARED for incidence structures, which does not imply that operations described here will work for arbitrarily user constructed incidence structures. Its aim is furthermore to allow the user to become a bit 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 of the commands. These can easily be found using the index.

4.1 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.

4.1.1 IsIncidenceStructure

▷ `IsIncidenceStructure` (Category)

Top level category for all objects representing an incidence structure.

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 Grassman-

nian of PG(4,2). This example is not meant to create this incidence structure in an efficient way, but just to demonstrate the general philiosophy.

4.1.2 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 if 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*.

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
```

4.1.3 IsIncidenceGeometry

▷ IsIncidenceGeometry (Category)

Category for all objects representing an incidence geometry.

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

4.1.4 Main categories in IsIncidenceGeometry

▷ IsLieGeometry (Category)

▷ IsAffineSpace (Category)

▷ IsGeneralisedPolygon (Category)

4.1.5 Main categories in IsLieGeometry

▷ IsProjectiveSpace (Category)
▷ IsClassicalPolarSpace (Category)

Lie geometries bundle projective spaces and classical polar spaces. 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.

The following categories for geometries are not considered as main categories.

4.1.6 Categories in IsGeneralisedPolygon

▷ IsGeneralisedQuadrangle (Category)
▷ IsGeneralisedHexagon (Category)
▷ IsGeneralisedOctogon (Category)

Within IsGeneralisedPolygon, categories are declared for generalised quadrangles, generalised hexagons, and generalised octogons.

4.1.7 IsElationGQ

▷ IsElationGQ (Category)

Within IsGeneralisedQuadrangle, this category is declared to construct elation generalised quadrangles.

4.1.8 IsClassicalGQ

▷ IsClassicalGQ (Category)

This category lies in IsElationGQ and IsClassicalPolarSpace.

4.1.9 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" ]
gap> CategoriesOfObject(ParabolicQuadric(4,16));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsLieGeometry",
  "IsClassicalPolarSpace", "IsGeneralisedPolygon", "IsGeneralisedQuadrangle",
  "IsClassicalGQ", "IsAlgebraicVariety", "IsProjectiveVariety" ]
```

4.1.10 TypesOfElementsOfIncidenceStructure

▷ TypesOfElementsOfIncidenceStructure(*inc*) (attribute)
▷ TypesOfElementsOfIncidenceStructurePlural(*inc*) (attribute)

Returns: a list of strings

Both attributes are declared for objects in the category IsIncidenceStructure, but only methods are installed for the geometries that are built in FinInG. Any incidence structure contains a set of types. This set is usually just the list [1 .. n]. If specific names are given to each type, this attribute returns the names for the particular incidence structure *inc*. The second variant returns the list of plurals of these names.

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" ]
```

4.1.11 Rank

▷ Rank(*inc*) (operation)
▷ RankAttr(*inc*) (attribute)

Returns: rank of *inc*, an object which must belong to the categorie 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
```

4.2 Elements of incidence structures

4.2.1 Main categories for individual elements of incidence structures

▷ IsElementOfIncidenceStructure (Category)
▷ IsElementOfIncidenceGeometry (Category)

▷ IsElementOfLieGeometry (Category)
▷ IsElementOfAffineSpace (Category)
▷ IsSubspaceOfProjectiveSpace (Category)
▷ IsSubspaceOfClassicalPolarSpace (Category)

A category IsElementOfIncidenceStructure for the individual elements of an incidence structure in the category IsIncStr, except for projective spaces and classical polar spaces. 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 of Split Cayley Hexagon of order 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" ]
```

4.2.2 Main categories for collections of elements of incidence structures

▷ IsElementsOfIncidenceStructure (Category)
▷ IsElementsOfIncidenceGeometry (Category)
▷ IsElementsOfLieGeometry (Category)
▷ IsElemenstOfAffineSpace (Category)
▷ IsSubspacesOfProjectiveSpace (Category)
▷ IsSubspacesOfClassicalPolarSpace (Category)

A category IsElementsOfIncidenceStructure for objects representing a set of elements of an incidence structure in the category IsIncStr, is declared, with an exception for projective spaces and polar spaces. The inclusion for different categories of geometries is followed for their collection of elements, except for IsSubspaceOfClassicalPolarSpace, which is a subcategory of IsSubspaceOfProjectiveSpace, 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, of course assuming that a method is installed for the particular incidence structure.

4.2.3 ElementsOfIncidenceStructure

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

Returns: a list 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 "points", "lines", "planes" or "solids" and the function returns the elements of type 1, 2, 3 or 4 respectively, of course if *inc* has elements of the deduced type. When no type is specified all elements of *inc* are returned.

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)>
```

4.2.4 Short names for ElementsOfIncidenceStructure

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

Returns: The elements of *inc* of type 1, 2, 3, and 4.

It is possible that *inc* is an incidence structure where the elements of type 1, 2, 3, and 4 are not called "points", "lines", "planes", and "solids", respectively. The methods don't check if names are given, and are just shortcuts to the operation `ElementsOfIncidenceStructure`.

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)>
```

4.2.5 IsIncident

▷ IsIncident(*u*, *v*) (operation)
▷ *(*u*, *v*) (operation)
▷ \in(*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 incidence 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, and is just calling IsIncident.

Example

```
gap> ps := ProjectiveSpace(4,7);
ProjectiveSpace(4, 7)
gap> p := VectorSpaceToElement(ps,[1,0,1,0,1]*Z(7)^0);
<a point in ProjectiveSpace(4, 7)>
gap> l := VectorSpaceToElement(ps,[[0,0,1,0,0],[1,0,0,0,1]]*Z(7)^0);
<a line in ProjectiveSpace(4, 7)>
gap> pl := VectorSpaceToElement(ps,[[1,1,1,0,0],[0,1,0,0,0],
>                                  [0,-1,0,0,1]]*Z(7)^0);
<a plane in ProjectiveSpace(4, 7)>
gap> p in l;
true
gap> l in pl;
false
gap> p in pl;
true
gap> IsIncident(p,l);
true
gap> IsIncident(l,p);
true
gap> IsIncident(pl,p);
true
gap> pl in p;
false
```

4.2.6 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. Our methods may refer to methods for random selection of e.g. subspaces of the underlying vector space, as in the example here, or refer to the use of a psuedo random element of an appropriate group, as is the case for subspaces of classical polar spaces.

Example

```
gap> Random(Hyperplanes(PG(5,7)));
<a proj. 4-space in ProjectiveSpace(5, 7)>
```

4.2.7 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 incidence structures was 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 of Split Cayley Hexagon of order 3>
gap> AmbientGeometry(l);
Split Cayley Hexagon of order 3
gap> p := Random(Points(EGQByBLTSet(BLTSetByqClan(LinearqClan(3)))));
#I  No intertwiner computed. One of the polar spaces must have a collineation group computed
#I  Now embedding dual BLT-set into W(5,q)...
#I  Computing points(1) of Knarr construction...
#I  Computing lines(1) of Knarr construction...
#I  Computing points(2) of Knarr construction...
#I  Computing lines(2) of Knarr construction...please wait
#I  Computing elation group...
<a point of <EGQ of order [ 9, 3 ] and basepoint
[ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ]>>
gap> AmbientGeometry(p);
<EGQ of order [ 9, 3 ] and basepoint [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3),
  0*Z(3) ]>
```

4.3 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 number of types. Recall that an incidence structure is an incidence geometry if every maximal flag is a chamber.

4.3.1 FlagOfIcidenceStructure

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

Returns: the flag 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)>
```

4.3.2 IsChamberOfIncidenceStructure

▷ IsChamberOfIncidenceStructure(*flag*) (operation)

Returns: true if and only if *flag* is a chamber

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
```

4.4 Shadow of elements

4.4.1 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 structure, 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
```

4.4.2 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 the elements of *flag*, or with the elements of *list*

Variant 2 and 4 convert *list* to a flag of *inc*, using FlagOfIcidenceStructure, which performs the necessary checks. Variant 1 and 2 rely on variant 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)>
```

4.4.3 ElementsIncidentWithElementOfIncidenceStructure

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

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

This operation is declared for objects v belonging to `IsElementOfIncidenceStructure`, but relies to particular methods installed for particular incidence structures, and refers always to `ShadowOfElement`, where the ambient geometry is derived from the element v, and using the integer j

4.4.4 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

It is possible that *inc* is an incidence structure where the elements of type 1, 2, 3, and 4 respectively are not called "points", "lines", "planes", and "solids" respectively. The methods don't check if names are given, and are just shortcuts to the operation `ShadowOfElement`. The second variant derives the incidence structure to be used as the ambient geometry of v. Please keep in mind that these methods are shortcuts to `ShadowOfElement`, which implies that asking e.g. `Lines(v)` with v a point, will indeed return the lines incident with a point.

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)>
```

4.5 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. We can either use `AsList` to get all of the elements of a projective/polar space efficiently, or we can ask for an iterator or enumerator of a

collection of elements. The word collection is important here. 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 )
```

takes almost no time at all. But if you want a random element from this set, you could be waiting a while. Instead, the user is better off using an iterator or an enumerator to access elements of this collection. We have such a facility for the elements of a projective or polar space. At the moment, we have made available iterators for projective spaces, and enumerators for polar spaces. We give basic examples of enumerators and iterators here. For AsList, we refer to the appropriate sections in the chapters on the particular geometries, since methods for AsList refer always to the group, and make use of the package orb. 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.

4.5.1 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.

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
```

For a collection of elements of a given type of certain incidence structures, FinInG also provides methods to compute an enumerator. In its simpliest 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.

4.5.2 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.

Example

```
gap> enum := Enumerator(Lines(ParabolicQuadric(6,2)));
EnumeratorOfSubspacesOfClassicalPolarSpace( <lines of Q(6, 2)> )
gap> s := Size(enum);
315
gap> n := Random([1..s]);
284
gap> l := enum!.ElementNumber(s,n);
<a line in Q(6, 2)>
gap> enum!.NumberElement(s,l);
278
```

4.6 Lie geometries

The category `IsLieGeometry` contains the categories `IsProjectiveSpace` and `IsClassicalPolarSpace`, and is bundling projective spaces and classical polar spaces. One common fact of these 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. Again, having a declaration does not imply that methods are installed for all particular Lie geometries.

4.6.1 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 )
```

4.6.2 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
```

4.6.3 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)
```

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.

4.6.4 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.

4.7 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.

4.7.1 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 represents possible 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 of the underlying vector space 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 >
```

4.7.2 ElementToVectorSpace

▷ `ElementToVectorSpace(v)` (operation)

Returns: the vector or list of vectors 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.

Example

```
gap> l := Random(Lines(PG(4,3)));
<a line in ProjectiveSpace(4, 3)>
gap> ElementToVectorSpace(l);
[ [ 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) ] ]
```

4.7.3 \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
```

4.7.4 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 that the type of v are asked.

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)>
```

4.8 Hard wired embeddings and converting elements

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 *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 on 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 arguents, 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.

4.8.1 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( ps, ElementToVectorSpace( 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 5

Projective Spaces

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

5.1 Projective Spaces and basic operations

A *projective space* is a point-line incidence geometry, satisfying few well known axioms. An axiomatic treatment can e.g. be found in [VY65a] and [VY65b]. Projective spaces are axiomatically, point-line geometries, but may contain higher dimensional projective subspaces too.

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 subspaces of the vector space. So the projective points correspond with the vectorlines, the projective lines correspond with the vectorplanes, etc. 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 are in FinInG considered as projective spaces. This abuses the terminology a bit, if one asks that projective spaces have rank at least 2.

5.1.1 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.

5.1.2 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)
```

5.1.3 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
```

5.1.4 BaseField

▷ `BaseField(`*ps*`)` (operation)

Returns: returns the base field for the projective space *ps*

Example

```
gap> BaseField(ProjectiveSpace(3,81));
GF(3^4)
```

5.1.5 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 )
```

5.1.6 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.

5.2 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 dimenion -1, corresponds with the zero dimensional vector space of the underlying vector space of $PG(n,q)$, and is hence represented by the zero vector of lenght $n+1$ over the underlying field $GF(q)$. The trivial subspace and the whole projective space are mathematically considerd as a subsace of the projective geometry, but not as elements of the incidence geometry, and hence do in FinInG not belong to the category `IsSubspaceOfProjectiveSpace`.

5.2.1 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);
 1 4 2 . 1 3 1
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;
```

5.2.2 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
```

5.2.3 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
```

5.2.4 ElmentsOfIncidenceStructure

▷ `ElmentsOfIncidenceStructure(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. An error message is produced when the projective space `as` 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;
```

5.2.5 Short names for ElementsOfIncidenceStructure

▷ `Points(as)` (operation)

▷ `Lines(as)` (operation)

▷ `Planes(as)` (operation)

▷ `Solids(as)` (operation)

▷ `Hyperplanes(as)` (operation)

Returns: The elements of `as` 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)>
```

5.2.6 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 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 := 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
```

5.2.7 StandardFrame

▷ StandardFrame(*ps*) (operation)

Returns: the standard frame of the projetive 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);
[ [ 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), 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) ],
  [ 0*Z(2), 0*Z(2), 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, Z(2)^0 ] ]
```

5.2.8 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 ]
```

5.2.9 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.

5.2.10 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.

5.2.11 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
```

5.2.12 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)
```

5.2.13 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)
```

5.2.14 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(Points(ps));
<a point in ProjectiveSpace(9, 49)>
gap> Random(Lines(ps));
<a line 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 = Z(49)
    1    .    .    .    .    . z^14 z^44 z^14    5
    .    1    .    .    .    . z^29 z^13 z^19 z^27
    .    .    1    .    .    . z^20 z^10 z^18 z^27
    .    .    .    1    .    .    3 z^30 z^18 z^14
    .    .    .    .    1    . z^10 z^28 z^47 z^29
    .    .    .    .    .    1  z^9 z^42 z^34 z^25
gap> RandomSubspace(ps,3);
<a solid in ProjectiveSpace(9, 49)>
gap> Display(last);
z = Z(49)
    1    .    .    . z^17 z^33  z^4    .  z^1 z^33
    .    1    .    . z^30    2    6    1 z^20 z^42
    .    .    1    . z^20 z^30 z^11 z^39    6    3
    .    .    .    1 z^21    1 z^11 z^45  z^1  z^9
gap> RandomSubspace(ps,7);
<a proj. 7-space in ProjectiveSpace(9, 49)>
gap> Display(last);
z = Z(49)
    1    .    .    .    .    .    .    . z^42 z^35
    .    1    .    .    .    .    .    . z^43  z^1
    .    .    1    .    .    .    .    . z^44    4
    .    .    .    1    .    .    .    . z^41 z^10
    .    .    .    .    1    .    .    . z^37 z^12
    .    .    .    .    .    1    .    . z^11 z^39
    .    .    .    .    .    .    1    . z^22 z^10
    .    .    .    .    .    .    .    1 z^43 z^22
gap> RandomSubspace(ps);
<a plane in ProjectiveSpace(9, 49)>
gap> RandomSubspace(ps);
```

```
<a proj. 6-space in ProjectiveSpace(9, 49)>
```

5.2.15 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)>
```

5.2.16 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. 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 argument for this operation, also as member of the argument of 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)
```

5.2.17 Meet

▷ `Meet(u, v)` (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 argument for this operation, also as member of the argument of the second variant of this operation.

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 >
```

5.2.18 FlagOfIncidenceStructure

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

Returns: the flag of the projetive 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)>
```

5.2.19 IsEmptyFlag

▷ `IsEmptyFlag(`*flag*`)` (operation)

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

5.2.20 IsChamberOfIncidenceStructure

▷ `IsChamberOfIncidenceStructure(`*flag*`)` (operation)

Returns: true if *flag* is a chamber flag

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
```

5.3 Shadows of Projective Subspaces

5.3.1 ShadowOfElement

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

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

Given the element *el* in the projective space *ps*, this operation returns the elements of *ps* of type *i* incident with *el*.

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)> ]
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)> ]
```

5.3.2 ShadowOfFlag

▷ ShadowOfFlag(*ps*, *flag*, *i*) (operation)
▷ ShadowOfFlag(*ps*, *flag*, *str*) (operation)

Returns: the shadow elements of type *i* in 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)>
```

5.3.3 ElementsIncidentWithElementOfIncidenceStructure

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

Returns: the elements of type *i* incident with *el*, in other words, the shadow of the elements of type i 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)> ]
```

5.3.4 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)>
```

5.4 Enumerating subspaces of a projective spaces

5.4.1 Iterator

▷ `Iterator(`*subspaces*`)` (operation)

Returns: an iterator for the collection *subspaces*

We refer to `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)>
```

5.4.2 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
```

5.4.3 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 explicitely, 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 6

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 5, 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 $x_0X_0+x_1X_1+\cdots+x_nX_n$. 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 6.8

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.

6.1 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.

6.1.1 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.

6.1.2 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 developpers to have the projectivity groups naturally embedded in the collineation groups. Correlations of projective spaces will be constructed in the category IsProjGrpElWithFrobWithPSIsom.

6.1.3 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: [ [ Z(2^2)^2, 0*Z(2), Z(2^2)^2, 0*Z(2) ],
  [ Z(2^2), Z(2^2), Z(2^2)^2, Z(2^2) ], [ Z(2^2)^2, 0*Z(2), Z(2^2), 0*Z(2) ],
  [ Z(2^2)^2, Z(2^2), Z(2^2)^2, Z(2)^0 ] ], F^0>
gap> IsProjectivity(g);
true
gap> g := Random(CollineationGroup(PG(3,4)));
< a collineation: [ [ Z(2)^0, 0*Z(2), Z(2^2), Z(2^2)^2 ],
  [ Z(2)^0, Z(2^2), Z(2^2), 0*Z(2) ], [ 0*Z(2), Z(2^2), 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, Z(2^2), 0*Z(2), Z(2)^0 ] ], F^0>
gap> IsProjectivity(g);
true
gap> g := Random(CorrelationCollineationGroup(PG(3,4)));
<projective element with Frobenius with projectivespace isomorphism
[ [ 0*Z(2), Z(2^2), Z(2^2)^2, Z(2^2) ], [ Z(2)^0, Z(2^2)^2, Z(2^2), 0*Z(2) ],
  [ Z(2^2)^2, Z(2^2), 0*Z(2), Z(2)^0 ],
```

```
  [ Z(2^2)^2, Z(2^2), Z(2^2)^2, Z(2)^0 ] ], F^
2, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
3,GF(2^2)) ) ) >
gap> IsProjectivity(g);
false
```

6.1.4 Collineations of projective spaces

▷ IsCollineation (Property)

IsCollineation is property. All element of $P\Gamma L(n+1,q)$ are collineations, and therefor 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: [ [ Z(3^3)^8, Z(3^3)^20, Z(3^3)^22 ],
  [ Z(3^3), Z(3^3)^7, Z(3^3)^11 ], [ Z(3^3)^19, Z(3^3), Z(3^3) ] ], F^0>
gap> IsCollineation(g);
true
gap> g := Random(CollineationGroup(PG(2,27)));
< a collineation: [ [ Z(3^3)^24, Z(3^3)^16, Z(3^3)^23 ],
  [ Z(3^3)^10, Z(3), Z(3) ], [ Z(3)^0, Z(3)^0, Z(3^3)^15 ] ], F^0>
gap> IsCollineation(g);
true
gap> g := Random(CorrelationCollineationGroup(PG(2,27)));
<projective element with Frobenius with projectivespace isomorphism
[ [ Z(3^3), Z(3^3)^19, Z(3^3)^14 ], [ Z(3^3)^14, Z(3^3)^20, Z(3^3)^2 ],
  [ Z(3^3)^17, 0*Z(3), Z(3)^0 ]
 ], F^0, IdentityMapping( <All elements of ProjectiveSpace(2, 27)> ) >
gap> IsCollineation(g);
true
```

6.1.5 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: [ [ Z(5^2)^9, Z(5)^0, Z(5^2)^3, Z(5^2)^3 ],
  [ Z(5^2)^9, Z(5^2)^19, Z(5)^0, Z(5^2)^13 ],
```

```
  [ Z(5^2)^14, Z(5)^3, Z(5^2)^9, Z(5^2)^2 ],
  [ Z(5^2)^9, Z(5)^0, Z(5^2)^20, Z(5)^0 ] ], F^0>
gap> IsStrictlySemilinear(g);
false
gap> g := Random(CollineationGroup(PG(3,25)));
< a collineation: [ [ 0*Z(5), Z(5^2)^2, 0*Z(5), Z(5^2)^21 ],
  [ Z(5^2)^4, Z(5)^0, Z(5^2)^9, Z(5)^3 ],
  [ Z(5^2)^15, Z(5^2)^2, Z(5)^3, Z(5^2)^10 ],
  [ Z(5), Z(5^2)^16, Z(5^2)^21, Z(5^2)^13 ] ], F^5>
gap> IsStrictlySemilinear(g);
true
gap> g := Random(CorrelationCollineationGroup(PG(3,25)));
<projective element with Frobenius with projectivespace isomorphism
[ [ Z(5^2)^14, Z(5^2)^2, Z(5^2)^20, Z(5^2)^4 ],
  [ Z(5)^0, Z(5^2)^2, Z(5^2)^14, Z(5^2)^23 ],
  [ Z(5^2)^22, Z(5^2)^20, 0*Z(5), Z(5^2)^7 ],
  [ Z(5^2)^2, Z(5^2)^11, Z(5^2)^23, Z(5) ] ], F^
5, IdentityMapping( <All elements of ProjectiveSpace(3, 25)> ) >
gap> IsStrictlySemilinear(g);
true
```

6.1.6 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: [ [ Z(7)^5, Z(7)^2, Z(7)^3, Z(7)^5, Z(7)^2 ],
  [ Z(7)^5, Z(7)^4, 0*Z(7), Z(7)^4, Z(7) ],
  [ Z(7), Z(7)^0, Z(7)^4, 0*Z(7), Z(7)^4 ],
  [ Z(7)^2, Z(7)^5, Z(7)^4, Z(7)^0, Z(7)^0 ],
  [ Z(7)^3, Z(7), Z(7)^5, Z(7)^3, Z(7)^0 ] ], F^0>
gap> IsCorrelationCollineation(g);
false
gap> IsCorrelation(g);
false
gap> g := Random(CorrelationCollineationGroup(PG(4,7)));
<projective element with Frobenius with projectivespace isomorphism
[ [ Z(7)^0, Z(7)^3, Z(7)^0, Z(7)^3, 0*Z(7) ],
  [ Z(7)^2, Z(7)^2, Z(7)^2, Z(7)^0, Z(7)^5 ],
  [ Z(7)^4, 0*Z(7), Z(7), Z(7)^5, Z(7)^3 ],
  [ 0*Z(7), Z(7)^0, Z(7)^3, Z(7), Z(7) ],
  [ Z(7)^0, Z(7)^3, Z(7)^0, Z(7)^3, Z(7)^0 ]
```

```
 ], F^0, IdentityMapping( <All elements of ProjectiveSpace(4, 7)> ) >
gap> IsCorrelationCollineation(g);
true
gap> IsCorrelation(g);
false
```

6.2 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`.

6.2.1 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: [ [ 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 ] ], F^0>
```

6.2.2 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: [ [ 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 ] ], 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(512));
< a collineation: [ [ 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 ] ], F^0>
```

6.2.3 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*.

6.2.4 IdentityMappingOfElementsOfProjectiveSpace

▷ `IdentityMappingOfElementsOfProjectiveSpace(`*ps*`)` (operation)

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

6.2.5 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> ElementToVectorSpace(h);
[ [ 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) ] ]
```

6.2.6 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 (6.2.5), or the identity mapping on the collection of subspaces of a projective space, created using IdentityMappingOfElementsOfProjectiveSpace (6.2.4). 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
[ [ 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 ]
 ], 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
[ [ 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 ] ], 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
[ [ 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 ]
 ], 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
[ [ 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 ] ], F^
7, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
2,GF(7^3)) ) ) >
```

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

6.3.1 Representative

▷ `Representative(`*g*`)` (operation)

g is a projectivity, collineation or correlation of a projective space. This function returns the reresentative 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);
[ [ [ Z(7)^4, Z(7^2)^17, Z(7^2)^9 ], [ Z(7^2)^12, Z(7^2)^47, Z(7^2)^31 ],
      [ Z(7^2)^11, Z(7^2)^29, Z(7^2)^31 ] ], FrobeniusAutomorphism( GF(7^2) )
 ]
```

6.3.2 UnderlyingMatrix

▷ `UnderlyingMatrix(`*g*`)` (operation)

g is a projectivity, collineation or correlation 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> UnderlyingMatrix(x);
[ [ Z(3)^0, Z(3)^0, 0*Z(3), Z(3)^0 ], [ Z(3)^0, Z(3), 0*Z(3), Z(3)^0 ],
  [ Z(3), 0*Z(3), Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3), Z(3)^0 ] ]
```

6.3.3 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: [ [ 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) ] ], F^0>
gap> BaseField(g);
GF(3^6)
```

6.3.4 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) )
```

6.3.5 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
```

```
[ [ 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 ] ], F^
7, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
2,GF(7^3)) ) ) >
gap> ProjectiveSpaceIsomorphism(phi);
StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
2,GF(7^3)) ) )
```

6.3.6 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: [ [ Z(3)^0, Z(3)^0, Z(3^2)^3, 0*Z(3), Z(3) ],
  [ Z(3), Z(3)^0, Z(3^2)^2, Z(3), Z(3^2)^6 ],
  [ Z(3^2)^2, Z(3)^0, Z(3)^0, Z(3^2)^7, 0*Z(3) ],
  [ Z(3^2), Z(3^2)^5, Z(3^2)^7, Z(3^2)^2, Z(3^2) ],
  [ Z(3), Z(3), 0*Z(3), Z(3), Z(3^2)^2 ] ], F^3>
gap> t := Order(x);
32
gap> IsOne(x^t);
true
```

6.4 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.

6.4.1 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)
```

6.4.2 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)
```

6.4.3 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)
```

6.4.4 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.

6.4.5 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.

6.5 Basic operations for projective groups

6.5.1 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.

6.5.2 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.

6.6 Natural embedding of a collineation group in a correlation 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: [ [ Z(2)^0, Z(2^2), Z(2)^0, Z(2^2)^2 ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2^2)^2 ], [ Z(2^2), Z(2^2)^2, 0*Z(2), Z(2^2)^2 ]
    , [ Z(2)^0, 0*Z(2), Z(2)^0, Z(2^2)^2 ] ], F^2>
gap> y := Random(CorrelationCollineationGroup(PG(3,4)));
<projective element with Frobenius with projectivespace isomorphism
[ [ 0*Z(2), Z(2^2), 0*Z(2), Z(2^2)^2 ], [ Z(2)^0, Z(2^2)^2, Z(2)^0, 0*Z(2) ],
  [ Z(2)^0, Z(2^2), Z(2^2)^2, 0*Z(2) ], [ Z(2^2), Z(2)^0, 0*Z(2), Z(2)^0 ]
 ], F^0, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
3,GF(2^2)) ) ) >
gap> x*y;
<projective element with Frobenius with projectivespace isomorphism
[ [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2^2)^2, 0*Z(2), Z(2^2), Z(2^2)^2 ],
  [ Z(2)^0, Z(2^2)^2, Z(2^2)^2, 0*Z(2) ],
  [ Z(2^2)^2, Z(2^2)^2, Z(2^2), Z(2)^0 ] ], F^
2, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
3,GF(2^2)) ) ) >
```

6.6.1 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 )
```

6.7 Basic action of projective group element

6.7.1 \^

▷ `\^(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.

6.8 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 5, 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, then p^ϕ is the point with underlying row vector $(y_0,y_1,\ldots,y_n)$, when ϕ is a correlation then is a hyperplane of $PG(n,q)$ with equation $y_0X_0+y_1X_1+\ldots+y_nX_n$. The action of collineations or correlations on points determines the action on subspaces of arbitrary dimension completely.

6.8.1 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` (6.8.3) 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> Display(p);
z = Z(27)
    1 z^14  z^5  z^7  z^7
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: [ [ 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 ] ], F^3>
gap> r := OnProjSubspaces(p,phi);
<a point in ProjectiveSpace(4, 27)>
gap> Display(r);
z = Z(27)
    1    .    . z^17
gap> vect := [[Z(3^3)^9,Z(3^3)^5,Z(3^3)^19,Z(3^3)^21,Z(3^3)^17],
>   [Z(3^3)^22,Z(3^3)^22,Z(3^3)^4,Z(3^3)^16,Z(3^3)^17],
>   [Z(3^3)^8,0*Z(3),Z(3^3)^24,Z(3),Z(3^3)^21]];
[ [ Z(3^3)^9, Z(3^3)^5, Z(3^3)^19, Z(3^3)^21, Z(3^3)^17 ],
  [ Z(3^3)^22, Z(3^3)^22, Z(3^3)^4, Z(3^3)^16, Z(3^3)^17 ],
  [ Z(3^3)^8, 0*Z(3), Z(3^3)^24, Z(3), Z(3^3)^21 ] ]
gap> s := VectorSpaceToElement(ps,vect);
<a plane in ProjectiveSpace(4, 27)>
gap> r := OnProjSubspaces(s,phi);
<a plane in ProjectiveSpace(4, 27)>
gap> Display(r);
z = Z(27)
    1    .    .  z^3
    .    1    . z^22
    .    .    1  z^3
```

6.8.2 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.

6.8.3 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
[ [ 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 ] ], 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
[ [ 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 ] ], 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)>
```

6.9 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 $V(n,F)$.

6.9.1 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: [ [ Z(3)^0, Z(3), 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) ],
  [ 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0 ] ], F^0>
```

6.9.2 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 $GF(q)$.

6.9.3 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: [ [ Z(3)^0, Z(3^4)^59, Z(3^4)^15, 0*Z(3) ],
  [ 0*Z(3), Z(3^4)^5, Z(3^4)^15, 0*Z(3) ],
  [ 0*Z(3), Z(3^4)^19, Z(3^4)^57, 0*Z(3) ],
  [ 0*Z(3), Z(3^4)^19, Z(3^4)^55, Z(3)^0 ] ], F^0>
```

6.9.4 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"
```

6.10 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 (automotic) 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). FinInG contains a switch to influence this behaviour.

6.10.1 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>
```

6.10.2 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>
```

6.10.3 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 `DESARGUES.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>
```

Chapter 7

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. Construction of correlations of a projective space is described in Chapter 6. In this chapter we describe methods and operations dealing with the construction and use of polarities of projective spaces in FinInG.

7.1 Creating polarities of projective spaces

Since polarities of a projective space necessarily have an involutory field automorphism as companion automorphism and the standardduality 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 polaritiy 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 by α is mapped to the element with underlying subspace $\alpha^\perp$, i.e. the subspace of $V(n+1,q)$ orhtogonal to α with relation 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 6.8) garantuees 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.

7.1.1 PolarityOfProjectiveSpace

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

Returns: a polarity of a projective space

the underlying correlation of the projective space is constructed using *mat*, *f*, the identity mapping as field automorphism and the standardduality of the projective space. It is checked whether the *mat* satisfies the necessary conditions to induce a polaritiy.

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)) >
```

7.1.2 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 *mat*, *frob*, *f* as matrix, field automorphism, field, and the standardduality of the projective space. It is checked whether the *mat* satisfies the necessary conditions to induce a polaritiy, 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 the method 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
```

7.1.3 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)) >
```

7.1.4 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;
```

7.2 Operations, attributes and properties for polarties of projective spaces

7.2.1 SesquilinearForm

▷ SesquilinearForm(*f*) (attribute)

Returns: a sesquilinear form

The sesquilinear form corresponding to the given polarity 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 >
```

7.2.2 BaseField

▷ BaseField(*f*) (attribute)

Returns: a field

the basefield over which the polarity 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)
```

7.2.3 GramMatrix

▷ GramMatrix(*f*) (attribute)

Returns: a matrix

the Gram matrix of the polarity.

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);
[ [ 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) ] ]
```

7.2.4 CompanionAutomorphism

▷ CompanionAutomorphism(*f*) (attribute)

Returns: a field automorphism

the involutory fieldautomorphism accompanying the polarity

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) )
```

7.2.5 IsHermitianPolarityOfProjectiveSpace

▷ IsHermitianPolarityOfProjectiveSpace(*f*) (property)

Returns: true or false

The polarity *f* 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
```

7.2.6 IsSymplecticPolarityOfProjectiveSpace

▷ IsSymplecticPolarityOfProjectiveSpace(*f*) (property)

Returns: true or false

The polarity *f* is a symplectic polarity of a projective space if and only if the underlying matrix is hermitian.

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
```

7.2.7 IsOrthogonalPolarityOfProjectiveSpace

▷ IsOrthogonalPolarityOfProjectiveSpace(*f*) (property)

Returns: true or false

The polarity *f* 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
```

7.2.8 IsPseudoPolarityOfProjectiveSpace

▷ `IsPseudoPolarityOfProjectiveSpace(f)` (property)

Returns: true or false

The polarity `f` 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(8));
<polarity of PG(3, GF(2^3)) >
gap> IsPseudoPolarityOfProjectiveSpace(phi);
true
```

7.3 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 vectorspace is totally isotropic with relation 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 speudo 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 8.

7.3.1 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 polartiy, 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)>
```

7.3.2 AbsolutePoints

▷ `AbsolutePoints(f)` (operation)

Returns: a set of points

This operation returns all points that are absolute with relation 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> ]
```

7.3.3 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 >
```

7.4 Commuting polarities

FinInG constructs polarites 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: [ [ 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) ] ], F^5>
```

Chapter 8

Finite Classical Polar Spaces

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

8.1 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. 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 fininte dimensional vector space over a finite field.

8.1.1 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 treament 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 either 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.

8.1.2 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 7 for more information on polarities of projective spaces, and more particularly to Section 7.3 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 documention 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
```

8.2 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 sequilinear form f on V. The couple (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 couple (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 (hence in odd characteristic) come into 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 representant of the canonical similarity class. Hence a *standard* classical polar space is always a *canonical* classical polar space, a cononical polar space is determined by a standard form upto a constant factor. In the following table, we summerize 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	$X_0Y_1-Y_0X_1+\ldots+X_{n-1}Y_n-Y_{n-1}X_n$	odd and even
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: 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 following five operations always return polar spaces induced by one of the above standard forms.

8.2.1 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
```

8.2.2 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
```

8.2.3 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 .
```

8.2.4 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 .
```

8.2.5 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 .
```

8.2.6 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*.

8.2.7 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*.

8.3 Basic operations for finite classical polar spaces

8.3.1 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 )
```

8.3.2 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)
```

8.3.3 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
```

8.3.4 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
```

8.3.5 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)
```

8.3.6 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
```

8.3.7 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
```

8.3.8 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
```

8.4 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 sequilinear 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 dimenion -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`.

8.4.1 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 mxn 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 >
```

8.4.2 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 >
gap> p := VectorSpaceToElement(ps,[1,1,1,0,1,1,1,0,1,0,0]*Z(7)^0);
<a point in H(10, 7^2)>
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 11 of *stdin*
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> e in p;
true
```

8.4.3 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
```

8.4.4 ElmentsOfIncidenceStructure

▷ `ElmentsOfIncidenceStructure(`*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(4,4);
H(4, 2^2)
gap> ElementsOfIncidenceStructure(ps,1);
<points of H(4, 2^2)>
gap> ElementsOfIncidenceStructure(ps,2);
<lines of H(4, 2^2)>
gap> ElementsOfIncidenceStructure(ps,3);
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;
```

8.4.5 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)
```

8.4.6 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) ]
```

8.4.7 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
```

8.4.8 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
```

8.4.9 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.

8.4.10 IsCollinear

▷ `IsCollinear(ps, u, v)` (operation)

Returns: Boolean

`u` and `v` are points of the polar space `ps`. This function returns True if `u` and `v` are incident with a common line and False otherwise.

8.4.11 Polarity

▷ `Polarity(ps)` (operation)

Returns: a function for the polarity

`ps` must be a polar space. This operation returns the polarity of the polar space `ps` in the form of a function.

8.4.12 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"
```

8.5 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

8.5.1 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)
```

8.5.2 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)
```

8.5.3 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)
```

8.5.4 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 field of definition $GF(q^2)$ has q prime, we return the similarity group.

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);
PGU(5,3^2)
```

8.6 Enumerating subspaces of polar spaces

8.6.1 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*.

8.6.2 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;
13605
gap> Length(planes);
891
```

8.6.3 Iterators for polar spaces

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

8.6.4 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)>
```

8.6.5 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 9

Actions, stabilisers and orbits

9.1 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>
gap> time;
13527
gap> line := Random(Lines(ps));
<a line in ProjectiveSpace(3, 8)>
gap> Stabilizer(g,line,OnProjSubspaces);
<projective collineation group of size 21849440256>
gap> time;
108345
```

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

9.1.1 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);
Split Cayley Hexagon of order 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 of Split Cayley Hexagon of order 3>
gap> FiningStabiliser(g,p);
<projective collineation group of size 11664 with 3 generators>
gap> line := Random(Lines(h));
<a line of Split Cayley Hexagon of order 3>
gap> FiningStabiliser(g,line);
<projective collineation group of size 11664 with 3 generators>
```

9.1.2 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);
Split Cayley Hexagon of order 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 of Split Cayley Hexagon of order 3>
gap> FiningStabiliserOrb(g,p);
<projective collineation group with 15 generators>
gap> line := Random(Lines(h));
<a line of Split Cayley Hexagon of order 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>
gap> time;
13759
gap> g2 := FiningStabiliser(g,p);
<projective collineation group of size 177223237632 with 2 generators>
gap> time;
312
gap> g3 := FiningStabiliserOrb(g,p);
<projective collineation group with 15 generators>
gap> time;
46
gap> g1=g2;
true
gap> g2=g3;
true
```

Computing the setwise stabiliser under a group is possible using `Stabilizer`. Not suprisingly, the computing time can also 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>
gap> time;
16079
```

The package GenSS provides an efficient operation to compute setwise stabilisers, and FinInG provides functionality to used these GenSS operation for the particular groups and (elements) of geometries.

9.1.3 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 5 generators>
```

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 720>
gap> time;
31095
gap> g2 := FiningSetwiseStabiliser(g,Set([l1,l2]));
#I  Computing adjusted stabilizer chain...
<projective collineation group with 4 generators>
gap> time;
56
gap> g1=g2;
true
```

Chapter 10

Affine Spaces

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

10.1 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 described as the "geometry you get" when you remove a hyperplane from a projective space. Conversely, adding to an affine spaces its hyperplane at infinity, yields a projective space. 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 fashion. 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

$$v+S.$$

Hence one can 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. Thus 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.

10.1.1 IsAffineSpace

▷ `IsAffineSpace` (Category)

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

10.1.2 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)
```

10.1.3 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
```

10.1.4 BaseField

▷ BaseField(*as*) (operation)

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

Example

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

10.1.5 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 )
```

10.1.6 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)
```

10.2 Subspaces of affine spaces

10.2.1 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 2 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)>
```

10.2.2 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, $1 \leq j \leq d$ this operation returns the collection of $j-1$ dimenaional 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;
```

10.2.3 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)>
```

10.2.4 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
```

10.2.5 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)
```

10.2.6 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)
```

10.2.7 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) ] ]
```

10.2.8 Meet

▷ `Meet(u, v)` (operation)

Returns: a 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 . .
```

10.2.9 IsParallel

▷ `IsParallel(u, v)` (operation)

Returns: true or false

The arguments u and v must be affine subspaces of a common affine space, of the same dimension. These two subspaces are parallel if and only if they are cosets of the same vector subspace, i.e. if the have the same subspace at infinity.

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
```

10.2.10 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 for it. 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 that which *v* contains as a component.

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)> ]
```

10.3 Shadows of Affine Subspaces

10.3.1 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 (such as 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)> ]
```

10.3.2 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 (such as 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)> ]
```

10.4 Iterators and enumerators

Recall from Section 5.4("Enumerating subspaces of a projective space", Chapter 5), 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.

10.4.1 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)>
```

10.4.2 Enumerator

▷ `Enumerator(subs)` (operation)

Returns: an enumerator for the 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) ] ]
```

One technical aspect of the design behind affine spaces in FinInG are having canonical transversals for subspaces of vector spaces. So we provide some documentation below for the interested user.

10.4.3 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 that the user apply methods normally used for this category to our objects (they won't work!). Our objects are only used in order to facilitate computing enumerators of subspace collections.

10.4.4 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.

10.4.5 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 $U+v$.

10.4.6 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.

10.5 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 collineations of an affine space $AG(d,F)$ form the group $A\Gamma L(d,F)$, which is generated by the translations T, matrices of $GL(d,F)$ and the automorphisms of the field F. Since the translations T form a normal subgroup of $A\Gamma L(d,F)$, we see that $A\Gamma L(d,F)$ 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: $\left(\begin{array}{cc|c} & & 0 \\ A & & \vdots \\ & & 0 \\ \hline x & & 1 \end{array}\right)$ We can also 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

$$(1,2,2)+\begin{pmatrix} \\ 1 \\ 2 \\ 1 \\ 2 \\ 2 \\ 1 \\ 1 \\ 2 \end{pmatrix}.$$

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
```

10.5.1 AffineGroup

▷ `AffineGroup(as)` (operation)

Returns: a group

This operation returnes the affine linear group $AGL(V)$ acting on the affine space with underlying vector space V. The elements of this group are collineations of the associated projective space. In order to get the full group of collineations of the affine space, one may need 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)
```

10.5.2 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)
```

10.5.3 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* is 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)>
```

Chapter 11

Geometry Morphisms

Here we describe what is meant by a *geometry morphism* in FinInG and the various operations and tools available to the user.

11.1 Geometry morphisms in FinInG

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 the Klein correspondence are examples of such geometry morphisms.

11.1.1 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 `ImagesElm`, `ImagesSet`, `PreImagesElm`, `PreImagesSet` work for geometry morphisms, as well as the overload operator `\^`. Since `Image` is a GAP function, we advise the user to not use this for geometry morphisms.

For some geometry morphisms, there is also an accompanying intertwiner for the automorphism groups of the source 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

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

11.1.2 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 be able to obtain the intertwiner by calling this operation (see the individual geometry morphism constructions). 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), such that the attribute `Intertwiner` becomes available. Here is a simple example of the intertwiner for the isomorphism of two polar spaces (see `IsomorphismPolarSpaces` (11.2.1)). 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 )
```

11.2 Isomorphisms between polar spaces

An important class of geometry morphisms in FinInG are the isomorphisms between polar spaces of the same kind that are induced by coordinate transformations.

11.2.1 IsomorphismPolarSpaces

▷ `IsomorphismPolarSpaces(ps1, ps2)` (operation)

▷ `IsomorphismPolarSpaces(ps1, ps2, boolean)` (operation)

Returns: a geometry morphism

The arguments *ps1* and *ps2* are equivalent polar spaces, and this function returns a geometry isomorphism between them. 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 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. That is, we also obtain a natural group isomorphism from the collineation group of *ps1* onto the collineation group of *ps2* (see also `Intertwiner` (11.4.3)).

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 <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 >> to <Elements of <polar space
 in ProjectiveSpace(5,GF(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 )
```

Both functions also have a "no check" version, which does not check whether *ps1* and *ps2* are polar spaces of the same type. ▷ `IsomorphismPolarSpacesNC(`*ps1, ps2*`)` (operation)

▷ IsomorphismPolarSpacesNC(*ps1, ps2, boolean*) (operation)

11.3 When will you use geometry morphisms?

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 things in GAP, the right thing to do is to embed one projective space into another, rather than having one projective space automatically 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 the subject of the following section.

11.4 Natural geometry morphisms

The most natural of geometry morphisms include, for example, the embedding of a projective space into another via a subspace, or the projection of a polar space to a smaller polar space of the same type via a totally isotropic subspace.

11.4.1 NaturalEmbeddingBySubspace

▷ NaturalEmbeddingBySubspace(*geom1, geom2, v*) (operation)
▷ NaturalEmbeddingBySubspaceNC(*geom1, geom2, v*) (operation)

Returns: a geometry morphism

The arguments *geom1* and *geom2* are both projective spaces, or both polar spaces, and *v* is an element of a projective or polar space. 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. The operation NaturalEmbeddingBySubspaceNC is the "no check" version of NaturalEmbeddingBySubspace.

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)>
```

Another example, this time with polar spaces:

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)>>
```

11.4.2 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.

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)>>
```

11.4.3 Intertwiner

▷ `Intertwiner(em)` (operation)

Returns: an intertwiner for a geometry morphism

The argument *em* is a geometry morphism constructed from $PG(r-1,q^t)$ into $PG(rt-1,q)$. The intertwiner of *em* will return a homomorphisms from the *homography* group of $PG(r-1,q^t)$ into the collineation group of $PG(rt-1,q)$. 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 := 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>
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
```

11.4.4 NaturalEmbeddingByFieldReduction

▷ `NaturalEmbeddingByFieldReduction(geom1, f2)` (operation)

▷ `NaturalEmbeddingByFieldReduction(geom1, f2, B)` (operation)

▷ `NaturalEmbeddingByFieldReduction(geom1, f2, boolean)` (operation)

Returns: a geometry morphism

The argument *geom1* is a classical polar space over a field L and *f2* is a subfield K of L, $L = GF(q^t)$ and $K = GF(q)$. This function returns a geometry morphism representing the natural embedding of *geom1* into a classical polar space S via field reduction, based on the following principle. Consider the trace map $T : L = GF(q^t) \to GF(q) : x \mapsto x^{q^t} + x^{q^{t-1}} + \ldots x$. The polar space *geom1* is the geometry associated to a quadratic or sequilinear form f, acting on an r-dimensional

vector space $V1$ over the finite field $GF(q^t)$. We first consider the rt-dimensional vector space $V2$ over the finite field $GF(q)$. There is a bijective map Φ from $V1$ to $V2$. Now it is easy to see that T \circ f \circ \Phi^{-1} will be a quadratic or sesquilinear form (depending on f being quadratic or sesquilinear) acting on $V2$, and hence, if not singular or degenerate, induce a polar space over the finite field $GF(q)$. An element of *geom1* is mapped to an element of the induced polar space over $GF(q)$ using the same principle as for the natural embedding by field reduction for projective spaces, of course now restricted to the elements of *geom1*. The only such possible embeddings are listed in the table below (see [Gil08]):

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

The geometry morphism also comes equipped with an intertwiner (see `Intertwiner` (11.4.3)). This intertwiner has as its domain the isometry group of *geom1*. The optional third argument *boolean* can take either `true` or `false` as input, and then this operation will or will not compute the intertwiner accordingly. The user may wish that the intertwiner is not computed when embedding into 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( PGSp(2,27), <projective collineation group of size
19656 with 3 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, 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 ] ]
```

11.4.5 BlownUpProjectiveSpace

▷ `BlownUpProjectiveSpace(`*basis, pg1*`)` (operation)

Returns: a projective space

Blows up the projective space *pg1* with respect to the *basis* using field reduction. If the argument *pg1* is has projective dimension $r-1$ over the finite field $GF(q^t)$, and *basis* is a basis of $GF(q^t)$ over $GF(q)$, then this functions returns a projective space of dimension $rt-1$ over GF(q).

11.4.6 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*.

11.4.7 BlownUpSubspaceOfProjectiveSpace

▷ `BlownUpSubspaceOfProjectiveSpace(basis, subspace)` (operation)

Returns: a subspace of a projective space

Blows up a *subspace* of a projective space with respect to the *basis* using field reduction and returns it a subspace of the projective space obtained from blowing up the ambient projective space of *subspace* with respect to *basis* using field reduction.

11.4.8 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.

11.4.9 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*.

11.4.10 NaturalEmbeddingBySubField

▷ `NaturalEmbeddingBySubField(geom1, geom2)` (operation)

▷ `NaturalEmbeddingBySubField(geom1, geom2, boolean)` (operation)

Returns: a geometry morphism

The arguments *geom1* and *geom2* are projective or polar spaces of the same dimension. This function returns a geometry morphism representing the natural embedding of *geom1* into *geom2* as a subfield geometry. If *geom1* and *geom2* are polar spaces, then the only such possible embeddings 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` (11.4.3)). 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 user may wish that the intertwiner is not computed when embedding into large polar spaces. The default (when calling the operation with two arguments) is set to `true`. Here is a simple example where the geometry morphism takes the points of $PG(2,3)$ and embeds them into $PG(2,9)$.

Example

```
gap> pg1 := ProjectiveSpace(2, 3);
ProjectiveSpace(2, 3)
gap> pg2 := ProjectiveSpace(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)> ]
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)> ]
```

In this example, we embed $W(5,3)$ in $H(5,3^2)$.

Example

```
gap> w := SymplecticSpace(5, 3);
W(5, 3)
gap> h := HermitianPolarSpace(5, 3^2);
H(5, 3^2)
gap> em := NaturalEmbeddingBySubfield(w, h);
<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
```

11.4.11 NaturalProjectionBySubspace

▷ `NaturalProjectionBySubspace(`*ps*`, `*v*`)` (operation)

▷ `NaturalProjectionBySubspaceNC(`*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, this operation returns a geometry morphism from the subspaces containing *v* to the subspaces of a smaller projective space over the same field. Similarly, if *ps* is a polar space, this operation returns a geometry morphism from the totally singular subspaces containing *v* to a polar space of smaller dimension, but of the same polar space type. The operation `NaturalProjectionBySubspaceNC` performs in exactly the same way as

NaturalProjectionBySubspace except that there are fewer checks such as 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. We should also mention here a shorthand for this operation which is basically and overload of the quotient operation. So, for example, SymplecticSpace(3, 3) / v achieves the same thing as NaturalProjectionBySubspace(SymplecticSpace(3,3), v).

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> ]
```

11.5 Some special kinds of geometry morphisms

In this section we provide some more specialised geometry morphisms, that are commonly used in finite geometry.

11.5.1 KleinCorrespondence

▷ KleinCorrespondence(*quadric*) (operation)

Returns: a geometry morphism

The argument *quadric* is a 5-dimensional hyperbolic quadric $Q^+(5,q)$, and this function returns the Klein correspondence from the lines of $PG(3,q)$ to the points of *quadric*.

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)>
```

11.5.2 NaturalDuality

▷ `NaturalDuality(gq)` (operation)

Returns: a geometry morphism

The argument *gq* is either the symplectic generalised quadrangle `W(3,q)` or the hermitian generalised quadrangle `H(3,q^2)`. By the Klein correspondence, the lines of $W(3,q)$ are mapped to the points of $Q(4,q)$, which results in a point-line duality from $W(3,q)$ onto $Q(4,q)$. Likewise, the Klein correspondence induces a duality between $H(3,q^2)$ and $Q^-(5,q)$. At the moment, the geometry morphism returned is a map from lines to points. This operation does not require that the input is the canonical version of the generalised quadrangle; it suffices that the input has the correct polarity type.

Example

```
gap> w := SymplecticSpace(3,5);
W(3, 5)
gap> lines:=AsList(Lines(w));;
gap> duality := NaturalDuality(w);
#I  No intertwiner computed. One of the polar spaces must have a collineation group computed
<geometry morphism from <lines of W(3, 5)> to <points 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)>
```

11.5.3 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) goes to 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 12

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` (12.7.1), the Segre varieties `SegreVariety` (12.6.1) and the Grassmann varieties `GrassmannVariety` (12.8.1); three classical projective varieties. These varieties have an associated *geometry map* (the `VeroneseMap` (12.7.3), `SegreMap` (12.6.3) and `GrassmannMap` (12.8.3)) and FinInG also provides some general functionality for these.

12.1 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.

12.1.1 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.

12.1.2 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*.

12.1.3 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.

12.1.4 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` (13.4.2) of the algebraic variety *var* whose coordinates satify the `DefiningListOfPolynomials` (12.1.2) of the algebraic variety *var*.

12.1.5 Iterator

▷ `Iterator(pts)` (operation)

Returns: an iterator

The argument *pts* is the set of `PointsOfAlgebraicVariety` (12.1.4) of an algebraic variety *var*. This operation returns an iterator for the points of an algebraic variety.

12.1.6 \in

▷ `\in(x, var)` (operation)

▷ `\in(x, pts)` (operation)

Returns: true or false

The argument *x* is a point of the `AmbientSpace` (13.4.2) of an algebraic variety `AlgebraicVariety` (12.4.1). This operation also works for a point *x* and the collection *pts* returned by `PointsOfAlgebraicVariety` (12.1.4).

12.2 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.

12.2.1 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)
```

12.3 Quadrics and Hermitian varieties

Quadrics (`QuadraticVariety` (12.3.3)) and Hermitian varieties (`HermitianVariety` (12.3.2)) 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.

12.3.1 HermitianVariety

▷ `HermitianVariety(`*pg*`,` *pring*`,` *pol*`)` (operation)
▷ `HermitianVariety(`*pg*`,` *pol*`)` (operation)

Returns: a hermitian variety in a projective space

The argument *pg* is a projective space, *pring* is a polynomial ring, and *pol* is polynomial.

12.3.2 HermitianVariety

▷ `HermitianVariety(`*n*`,` *F*`)` (operation)
▷ `HermitianVariety(`*n*`,` *q*`)` (operation)

Returns: a hermitian variety in a projective space

The argument *n* is an integer, the argument *F* is a finite field, and the argument *q* is a prime power. This function returns the hermitian variety associated to the standard hermitian form in the projective space of dimension n over the field F of order q.

12.3.3 QuadraticVariety

▷ `QuadraticVariety(`*pg*`,` *pring*`,` *pol*`)` (operation)
▷ `QuadraticVariety(`*pg*`,` *pol*`)` (operation)

Returns: a quadratic variety in a projective space

The argument *pg* is a projective space, *pring* is a polynomial ring, and *pol* is a polynomial.

12.3.4 QuadraticForm

▷ QuadraticForm(*var*) (attribute)

Returns: a quadratic form

When the argument *var* is a QuadraticVariety (12.3.3), this returns the associated quadratic form.

12.3.5 SesquilinearForm

▷ SesquilinearForm(*var*) (attribute)

Returns: a hermitian form

If the argument *var* is a HermitianVariety (12.3.2), this returns the associated hermitian form.

12.3.6 PolarSpace

▷ PolarSpace(*var*) (operation)

the argument *var* is a projective algebraic variety. When its list of defininin 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 >
```

12.4 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.

12.4.1 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*.

12.5 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` (12.6.3), the `VeroneseMap` (12.7.3), and the `GrassmannMap` (12.8.3).

12.5.1 Source

▷ `Source(`*gm*`)` (operation)

Returns: the source of a geometry map

The argument *gm* is a geometry map.

12.5.2 Range

▷ `Range(`*gm*`)` (operation)

Returns: the range of a geometry map

The argument *gm* is a geometry map.

12.5.3 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` (12.8.4) of the geometry map *gm*.

12.5.4 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` (12.8.4) of the geometry map *gm*.

12.5.5 \^

▷ `\^(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` (12.8.4) of the geometry map *gm*.

12.6 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*.

12.6.1 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 $n_1n_2\ldots n_k-1$.

12.6.2 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` (13.4.2) of the Segre variety. This set of points corresponds to the image of the `SegreMap` (12.6.3).

12.6.3 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 (12.6.2)) in the projective space of dimension $n_1n_2\ldots n_k-1$. 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 ]
```

12.6.4 Source

▷ Source(*sm*) (operation)

Returns: the source of a Segre map

The argument *sm* is a SegreMap (12.6.3). This operation returns the cartesian product of the list consisting of the pointsets of the projective spaces that were used to construct the SegreMap (12.6.3).

12.7 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*.

12.7.1 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 Veronese variety.

12.7.2 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` (13.4.2) of the Veronese variety. This set of points corresponds to the image of the `VeroneseMap` (12.7.3).

12.7.3 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` (12.7.2)) 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
```

12.7.4 Source

▷ `Source(vm)` (operation)

Returns: the source of a Veronese map

The argument *vm* is a `VeroneseMap` (12.7.3). This operation returns the pointset of the projective space that was used to construct the `VeroneseMap` (12.7.3).

12.8 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*.

12.8.1 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.

12.8.2 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` (13.4.2) of the Grassmann variety. This set of points corresponds to the image of the `GrassmannMap` (12.8.3).

12.8.3 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` (12.8.2)). When a Grassmann variety `gv` is given as input, the operation returns the associated Grassmann map.

12.8.4 Source

▷ `Source(gm)` (operation)

Returns: the source of a Grassmann map

The argument `gm` is a `GrassmannMap` (12.8.3). This operation returns the set of subspaces of the projective space that was used to construct the `GrassmannMap` (12.8.3).

Chapter 13

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 ($n = 8$).

13.1 Projective planes

13.1.1 ProjectivePlaneByBlocks

▷ `ProjectivePlaneByBlocks(l)` (operation)

Returns: a projective plane

The argument `l` is a finite homogeneous list consisting of ordered sets of a common size $n+1$ from the number 1 up to n^2+n+1. This operation returns the projective plane of order n.

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> pp := ProjectivePlaneByBlocks( blocks );
<projective plane of order 4>
```

13.1.2 ProjectivePlaneByIncidenceMatrix

▷ `ProjectivePlaneByIncidenceMatrix(mat)` (operation)

Returns: a projective plane

The argument `mat` is a square matrix with entries from $\{0,1\}$; the incidence matrix of a projective plane. The rows represent the lines of the projective plane and the columns represent the points. That

is, the (i, j)-entry of `mat` is equal to 0 or 1 according to whether the i-th line is incident or not incident with the j-th points.

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, 0, 1, 1, 0, 0, 0, 0, 1, 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, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0 ],
>   [ 0, 0, 0, 1, 0, 0, 1, 0, 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 := ProjectivePlaneByIncidenceMatrix( incmat );
<projective plane of order 4>
```

13.2 Generalised quadrangles

The classical generalised quadrangles were treated in the chapter on polar spaces (Chapter 8), and here we provide operations which create elation generalised quadrangles arising from Kantor families. Suppose we have 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 we have an *elation generalised quadrangle* with base point P and elation group G. Such an elation generalised quadrangle is equivalent to a *Kantor family* of subgroups of G: a set of $t+1$ subgroups F of order s and a set of $t+1$ subgroups F^* of order st such that (i) each element of F is a subgroup of one element of F^* and intersects the other elements of F^* trivially, and (ii) any three elements A,B,C of F satisfy $AB \cap C = 1$. For more information, we refer to the standard work in this field of Payne and Thas [PT84].

13.2.1 IsKantorFamily

▷ `IsKantorFamily(grp, f1, f2)` (operation)

Returns: true or false

This operation tests to see if (`f1`, `f2`) forms a Kantor family of subgroups for the group `grp`. The elements of `f1` are smaller than the elements of `f2`.

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) ];
[ <pc group with 1 generators>, <pc group with 1 generators>,
  <pc group with 1 generators>, <pc group with 1 generators> ]
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 ]) ];
[ <pc group with 2 generators>, <pc group with 2 generators>,
  <pc group with 2 generators>, <pc group with 2 generators> ]
gap> IsKantorFamily( g, flist1, flist2 );
#I  Checking tangency condition...
#I  Checking triple condition...
true
```

explain how to construct an EGQ from a Kantor family?

13.2.2 EGQByKantorFamily

▷ EGQByKantorFamily(*grp, f1, f2*) (operation)

Returns: an elation generalised quadrangle

The argument *grp* is a finite group and *f1* and *f2* are each lists of subgroups of *grp* which form a Kantor family. The i-th member of *f1* must be a subgroup of the i-th member of *f2*. We should mention that this operation does not check that the input is a valid Kantor family, as this would slow this operation down. Thus if the user is unsure of their input, they would best use the operation IsKantorFamily (13.2.1) beforehand. In the following example we construct the unique generalised quadrangle of order 3.

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);
#I  Computing points from Kantor family...
#I  Computing lines from Kantor family...
<EGQ of order [ 3, 3 ] and basepoint 0>
```

Let C be a set of 2×2 upper triangular matrices over $GF(q)$, which are indexed by $GF(q)$. If the pairwise difference of any two elements of C is anisotropic, that is, represents a nondegenerate binary quadratic form, then we say that C is a *q-clan*. This concept was introduced by Stanley Payne [Pay85] to construct Kantor families for flock generalised quadrangles.

13.2.3 IsqClan

▷ IsqClan(*list*, *f*) (operation)

Returns: true or false

This operation tests to see if *list* defines a q-Clan over the field *f*.

Example

```
gap> f := GF(3);
GF(3)
gap> id := IdentityMat(2, f);;
gap> clan := List( f, t -> t * id );;
gap> IsqClan( clan, f );
true
```

13.2.4 qClan

▷ qClan(*list*, *f*) (operation)

Returns: the q clan of matrices in *list*

This operation tests to see if *list* defines a q-Clan over the field *f*, and returs the q-Clan.

Example

```
gap> f := GF(7);
GF(7)
gap> id := IdentityMat(2, f);;
gap> clan := List( f, t -> t * id );;
gap> clan := qClan( clan, f );
<q-clan over GF(7)>
```

13.2.5 EGQByqClan

▷ EGQByqClan(*qclan*) (operation)

Returns: an elation generalised quadrangle

The argument *qclan* is a q-Clan. In the following example, we construct an elation generalised quadrangle from a q-Clan that is actually isomorphic with the classical generalised quadrangle of order (9, 3) (i.e., H(3,9)). We do not explicitely compute the isomorphism, but compute, using a detour via the incidence graph, a group isomorphic with the complete collineation group of the elation GQ, which turns out the have the same size as $P\Gamma U(4,9)$

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...
#I  Computing points from Kantor family...
#I  Computing lines from Kantor family...
<EGQ of order [ 9, 3 ] and basepoint 0>
gap> incgraph := IncidenceGraphOfGeneralisedPolygon(egq);;
```

```
#I  Computing incidence graph of generalised polygon...
gap> group := AutomorphismGroup(incgraph);
<permutation group with 6 generators>
gap> Order(group);
26127360
gap> Order(CollineationGroup(HermitianPolarSpace(3,9)));
#I  Computing nice monomorphism...
26127360
```

13.2.6 KantorFamilyByqClan

▷ KantorFamilyByqClan(*qclan, f*) (operation)

Returns: a kantor family

The argument *qclan* is a q-Clan, and the corresponding Kantor family is returned. Internally, the operation EGQByqClan EGQByqClan (13.2.5) will use this method to construct the elation generalised quadrangle with the operation EGQByKantorFamily.

Example

```
gap> f := GF(7);
GF(7)
gap> id := IdentityMat(2, f);;
gap> list := List( f, t -> t * id );;
gap> clan := qClan(list,f);
<q-clan over GF(7)>
gap> fam := KantorFamilyByqClan(clan);
[ <matrix group with 8 generators>,
  [ <matrix group with 2 generators>, <matrix group with 2 generators>,
      <matrix group with 2 generators>, <matrix group with 2 generators>,
      <matrix group with 2 generators>, <matrix group with 2 generators>,
      <matrix group with 2 generators>, <matrix group with 4 generators> ],
  [ <matrix group with 4 generators>, <matrix group with 4 generators>,
      <matrix group with 4 generators>, <matrix group with 4 generators>,
      <matrix group with 4 generators>, <matrix group with 4 generators>,
      <matrix group with 4 generators>, <matrix group with 6 generators> ] ]
gap> egq := EGQByKantorFamily(fam[1],fam[2],fam[3]);
#I  Computing points from Kantor family...
#I  Computing lines from Kantor family...
<EGQ of order [ 49, 7 ] and basepoint 0>
```

13.2.7 Particular q-Clans

▷ LinearqClan(*q*) (operation)

▷ FisherThasWalkerKantorBettenqClan(*q*) (operation)

▷ KantorMonomialqClan(*q*) (operation)

▷ KantorKnuthqClan(*q*) (operation)

▷ FisherqClan(*q*) (operation)

Returns: a q-Clan

The argument *q* is a prime power. These operations return a particular q-Clan. Should we add references here since describing all these q-Clans is definitely beyond the scope here?

Example

```
gap> LinearqClan(4);
Error, Couldn't find nonsquare called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> LinearqClan(5);
<q-clan over GF(5)>
gap> FisherThasWalkerKantorBettenqClan(9);
Error, q must be congruent to 2 mod (3) called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> FisherThasWalkerKantorBettenqClan(11);
<q-clan over GF(11)>
gap> KantorMonomialqClan(17);
<q-clan over GF(17)>
gap> KantorKnuthqClan(25);
<q-clan over GF(5^2)>
gap> FisherqClan(23);
<q-clan over GF(23)>
```

A *BLT-set* is a set S of points of the parabolic quadric $Q(4,q)$, which is a classial generalised quadrangle, such that for any three points of S, there is no point of $Q(4,q)$ collinear with all three of the points. BLT-sets, which were introduced by Bader, Lunardon and Thas [BLT90], give rise to q-clans, and hence to flock generalised quadrangles.

13.2.8 BLTSetByqClan

▷ `BLTSetByqClan(qclan, f)` (operation)

Returns: a list of points of Q(4,q)

The argument `qclan` is a list of matrices (i.e., IsFFECollCollColl) which form a q-Clan, and `f` is the defining field. This field must have odd order. This operation returns a BLT-set for the parabolic quadric defined by the bilinear form with Gram matrix $\begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & w^{(q+1)/2} & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$ where w is a primitive root of $GF(q)$.

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> Span(blt);
<a solid in ProjectiveSpace(4, 9)>
gap> clan := LinearqClan(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> Span(blt);
<a plane in ProjectiveSpace(4, 9)>
```

13.2.9 EGQByBLTSet

▷ `EGQByBLTSet(list, point, solid)` (operation)

▷ `EGQByBLTSet(list)` (operation)

Returns: an elation generalised quadrangle

The argument *list* is a list of points of a BLT-set of $Q(4,q)$, where q is odd. The user may enter the point and solid as extra arguments which are used in the Knarr construction of the elation generalised quadrangle from the BLT-set. Otherwise, we take the $W(5,q)$ in the Knarr construction to be defined by the canonical form used in FinInG, and we take *point* and *solid* to be the elements $[1,0,0,0,0,0]$ and $[[1,0,0,0,0,1],[0,0,1,0,0,0],[0,0,0,1,0,0],[0,0,0,0,1,0]]$ respectively. We show how we can construct the classical generalised quadrangle of order (9, 3) (i.e., H(3,9)) from a conic of $Q(4,3)$.

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> geo := AmbientGeometry( bltset[1] );
Q(4, 3): -x_1*x_5-x_2*x_4+x_3^2=0
gap> Display( geo );
Q(4, 3)
Non-degenerate parabolic bilinear form
```

```
Gram Matrix:
 . . . . 1
 . . . 1 .
 . . 1 . .
 . 1 . . .
 1 . . . .
Polynomial: -x_1*x_5-x_2*x_4+x_3^2
Witt Index: 2
gap> egq := EGQByBLTSet( bltset );
#I  No intertwiner computed. One of the polar spaces must have a collineation
group computed
#I  Computing nice monomorphism...
#I  Now embedding dual BLT-set into W(5,q)...
#I  Computing points(1) of Knarr construction...
#I  Computing lines(1) of Knarr construction...
#I  Computing points(2) of Knarr construction...
#I  Computing lines(2) of Knarr construction...please wait
#I  Computing elation group...
<EGQ of order [ 9, 3 ] and basepoint [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3),
  0*Z(3) ]>
```

13.2.10 ElationGroup

▷ `ElationGroup(`*egq*`)` (attribute)

Returns: a group

This method returns the elation group of order s^2t of the elation generalised quadrangle *egq*, which has order (s,t). This is the stored as an attribute of *egq*. Note that the type of the group is of course dependent on the model from which *egq* was constructed.

Example

```
gap> clan := FisherThasWalkerKantorBettenqClan(11);
<q-clan over GF(11)>
gap> egq := EGQByqClan(clan);
#I  Computed Kantor family. Now computing EGQ...
#I  Computing points from Kantor family...
#I  Computing lines from Kantor family...
<EGQ of order [ 121, 11 ] and basepoint 0>
gap> group := ElationGroup(egq);
<matrix group of size 161051 with 8 generators>
```

13.2.11 BasePointOfEGQ

▷ `BasePointOfEGQ(`*egq*`)` (attribute)

Returns: a point of *egq*

This method returns the base point for the elation generalised quadrangle *egq*, that is, a point for which the elation group of *egq* fixes every line through it. This is the stored as an attribute of *egq*.

Example

```
gap> clan := LinearqClan(3);
<q-clan over GF(3)>
```

```
gap> egq := EGQByqClan(clan);
#I  Computed Kantor family. Now computing EGQ...
#I  Computing points from Kantor family...
#I  Computing lines from Kantor family...
<EGQ of order [ 9, 3 ] and basepoint 0>
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> egq2 := EGQByBLTSet(blt);
#I  No intertwiner computed. One of the polar spaces must have a collineation
group computed
#I  Computing nice monomorphism...
#I  Now embedding dual BLT-set into W(5,q)...
#I  Computing points(1) of Knarr construction...
#I  Computing lines(1) of Knarr construction...
#I  Computing points(2) of Knarr construction...
#I  Computing lines(2) of Knarr construction...please wait
#I  Computing elation group...
<EGQ of order [ 9, 3 ] and basepoint [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3),
  0*Z(3) ]>
gap> BasePointOfEGQ(egq);
<a point of a Kantor family>
gap> Display(last);
0
gap> BasePointOfEGQ(egq2);
<a point of <EGQ of order [ 9, 3 ] and basepoint
[ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ]>>
gap> Display(last);
[ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ]
```

13.3 Generalised hexagons

Due to the sheer sizes of generalised octagons, they have not yet been included into FinInG. The only other known families of generalised hexagons (up to duality) are the Split Cayley hexagons and the Twisted Triality hexagons.

13.3.1 SplitCayleyHexagon

▷ `SplitCayleyHexagon(f)` (operation)
▷ `SplitCayleyHexagon(q)` (operation)

Returns: a generalised hexagon of order (q,q)

The Split Cayley hexagons were first constructed by Jacques Tits via the absolute points and lines of a triality of the 7-dimensional hyperbolic quadric. The input is either a finite field `f` or a prime power `q`, and a generalised hexagon is returned consisting of points and lines of `Q(6, q)` if `q` is odd, or of `W(5,q)` if `q` is even. The forms for these polar spaces are, respectively, `Q(x):= x_1x_5+x_2x_6+x_3x_7-x_4^2` and `B(x,y):=x_1y_4+x_2y_5+x_3y_6+x_4y_1+x_5y_2+x_6y_3`.

Example

```
gap> hexagon := SplitCayleyHexagon( 3 );
Split Cayley Hexagon of order 3
gap> points := Points( hexagon );
<points of Split Cayley Hexagon of order 3>
gap> lines := AsList( Lines(hexagon) );;
gap> lines[1];
<a line of Split Cayley Hexagon of order 3>
gap> AmbientSpace( hexagon );
ProjectiveSpace(6, 3)
gap> coll := CollineationGroup( hexagon );
G_2(3)
gap> DisplayCompositionSeries( coll );
G (size 4245696)
 | G(2,3)
1 (size 1)
```

13.3.2 TwistedTrialityHexagon

▷ `TwistedTrialityHexagon(f)` (operation)

▷ `TwistedTrialityHexagon(q)` (operation)

Returns: a generalised hexagon of order $(q, \sqrt[3]{q})$

Just like the Split Cayley hexagons (see `SplitCayleyHexagon` (13.3.1)), the Twisted Triality hexagons arise as absolute points and lines of a triality. The input is either a finite field *f* or a prime power *q*, where the order of the field is a cube, and a generalised hexagon is returned consisting of points and lines of `Q^+(7, q)`, defined by the form *Q(x):= x_1x_5+x_2x_6+x_3x_7+x_4x_8*. The smallest Twisted Triality hexagon has 2457 points and 819 lines.

13.3.3 AmbientSpace and AmbientPolarSpace

▷ `AmbientSpace(geo)` (attribute)

▷ `AmbientPolarSpace(geo)` (attribute)

Returns: a projective space, a classical polar space respectively

All generalised hexagons in FinInG are Lie geometries, constructed inside a classical polar space. As for all Lie geometries, the first attribute will return the ambient projective space. The second attribute returns the polar space in which *geo* is constructed.

Example

```
gap> hexagon := SplitCayleyHexagon( 3 );
Split Cayley Hexagon of order 3
gap> AmbientSpace(hexagon);
ProjectiveSpace(6, 3)
gap> AmbientPolarSpace(hexagon);
standard Q(6, 3)
gap> hexagon := SplitCayleyHexagon( 4 );
Split Cayley Hexagon of order 4
gap> AmbientSpace(hexagon);
ProjectiveSpace(5, 4)
gap> AmbientPolarSpace(hexagon);
standard W(5, 4)
```

```
gap> hexagon := TwistedTrialityHexagon( 5^3 );
Twisted Triality Hexagon of order [ 125, 5 ]
gap> AmbientSpace(hexagon);
ProjectiveSpace(7, 125)
gap> AmbientPolarSpace(hexagon);
standard Q+(7, 125)
```

13.4 General attributes and operations for generalised polygons

13.4.1 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*.

13.4.2 AmbientSpace

▷ `AmbientSpace(`*gp*`)` (attribute)

Returns: an incidence geometry

Some of our generalised polygons have a natural ambient space, for example, the Split Cayley hexagons in odd characteristic are naturally embedded in the 6-dimensional parabolic quadrics. Therefore, for some generalised polygons the user can use this method to return the natural ambient geometry for the generalised polygon, provided such a geometry exists.

13.4.3 CollineationGroup

▷ `CollineationGroup(`*gp*`)` (attribute)

Returns: a group

Some of our generalised polygons come equipped automatically with a collineation group. For example, the generalised hexagons have their collineation groups already installed, and so do the classical generalised quadrangles. However, the collineation group of a projective plane is calculated via using the package Grape. We refer to `CollineationAction` (13.4.4) for an example.

13.4.4 CollineationAction

▷ `CollineationAction(`*gp*`)` (attribute)

Returns: a function

Unlike some of the other geometries in FinInG, the collineations of generalised polygons to not have a uniform representation. Thus depending on the generalised polygon we are working with, a group element and its action could be very different. For example, we use ordinary permutations when acting on the elements of a projective plane (modulo some wrapping), whereas elation generalised quadrangles arising from Kantor families must employ a completely different group action. So our collineation groups come equipped with the attribute `CollineationAction`, which is a function with input a pair *(x,g)* where *x* is an element of *gp*, and *g* is a collineation.

Example

```
gap> LoadPackage("grape");
true
gap> Print("Collineations of projective planes...\n");
Collineations of projective planes...
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> pp := ProjectivePlaneByBlocks( blocks );
<projective plane of order 4>
gap> coll := CollineationGroup( pp );
#I  Computing incidence graph of projective plane...
<permutation group with 8 generators>
gap> DisplayCompositionSeries( coll );
G (8 gens, size 120960)
 | Z(2)
S (4 gens, size 60480)
 | Z(3)
S (3 gens, size 20160)
 | A(2,4) = L(3,4)
1 (0 gens, size 1)
gap> Display( CollineationAction(coll) );
function ( x, g )
    if x!.type = 1  then
        return Wrap( plane, 1, OnPoints( x!.obj, g ) );
    elif x!.type = 2  then
        return Wrap( plane, 2, OnSets( x!.obj, g ) );
    fi;
    return;
end
gap>
gap> Print("Collineations of generalised hexagons...\n");
Collineations of generalised hexagons...
gap> hex := SplitCayleyHexagon( 5 );
Split Cayley Hexagon of order 5
gap> coll := CollineationGroup( hex );
G_2(5)
gap> CollineationAction(coll) = OnProjSubspaces;
true
gap> Print("Collineations of elation generalised quadrangles...\n");
Collineations of elation generalised quadrangles...
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);
#I  Computing points from Kantor family...
```

```
#I  Computing lines from Kantor family...
<EGQ of order [ 3, 3 ] and basepoint 0>
gap> elations := ElationGroup( egq );
<pc group of size 27 with 3 generators>
gap> CollineationAction(elations) = OnKantorFamily;
true
gap> HasCollineationGroup( egq );
false
```

13.4.5 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, then 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");
------------------------------------------------------------------------------
Loading  DESIGN 1.4 (The Design Package for GAP)
by Leonard H. Soicher (http://www.maths.qmul.ac.uk/~leonard/).
------------------------------------------------------------------------------
#W BIND_GLOBAL: variable 'BlockDesign' already has a value
true
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...
#I  Computing points from Kantor family...
#I  Computing lines from Kantor family...
<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 5 generators>
gap> NrBlockDesignPoints( design );
280
gap> NrBlockDesignBlocks( design );
```

```
112
gap> DisplayCompositionSeries(aut);
G (5 gens, size 26127360)
 | Z(2)
S (4 gens, size 13063680)
 | Z(2)
S (4 gens, size 6531840)
 | Z(2)
S (3 gens, size 3265920)
 | 2A(3,3) = U(4,3) ~ 2D(3,3) = O-(6,3)
1 (0 gens, size 1)
```

13.4.6 IncidenceGraphOfGeneralisedPolygon

▷ `IncidenceGraphOfGeneralisedPolygon(`*gp*`)` (attribute)

Returns: a graph

This method allows one to use the GAP package GRAPE 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, then the incidence graph is computed with this extra information and thus the resulting graph 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 incidence graph. We should also point out that this method returns a *mutable* attribute of *gp*, so that accquired 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.

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> pp := ProjectivePlaneByBlocks( blocks );
<projective plane of order 4>
gap> incgraph := IncidenceGraphOfGeneralisedPolygon( pp );;
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 (3 gens, size 120960)
 | Z(2)
```

```
S (3 gens, size 60480)
 | Z(3)
S (2 gens, size 20160)
 | A(2,4) = L(3,4)
1 (0 gens, size 1)
```

13.4.7 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 (so you need to load this package first). The rows of the matrix correspond to the points of *gp*, and the columns correspond to the lines.

Example

```
gap> gp := SymplecticSpace(3,2);
W(3, 2)
gap> mat := IncidenceMatrixOfGeneralisedPolygon(gp);
#I  Computing nice monomorphism...
#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, 0, 1, 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, 0, 1, 0, 0, 0, 1, 1 ] ]
```

Chapter 14

Coset Geometries and Diagrams

This part of FinInG depends on GRAPE.

14.1 Coset Geometries

Suppose we have an *incidence geometry* Γ (as defined in chapter 4), together with a group G of automorphisms of Γ is also given such that G is transitive on the set of *chambers* of Γ (also defined in chapter 4). 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 4 for definitions).

14.1.1 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) ])
```

14.1.2 IsIncident

▷ `IsIncident(ele1, ele2)` (operation)

Returns: true if ans only if *ele1* and *ele2* are incident

ele1 and *ele2* must be two elements in the same coset geometry.

14.1.3 ParabolicSubgroups

▷ `ParabolicSubgroups(cg)` (operation)

Returns: the list of parabolic subgroups defining the coset geometry *cg*

14.1.4 AmbientGroup

▷ `AmbientGroup(cg)` (operation)

Returns: the group used to define the coset geometry *cg*

cg must be a coset geometry.

14.1.5 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 subgrops.

14.1.6 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 is quite time consuming. You can bind the attribute `IsFlagTransitiveGeometry` if you are sure the coset geometry is indeed flag-transitive.

14.1.7 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.

14.1.8 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.

14.1.9 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.

14.1.10 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). `cg` must be a coset geometry.

14.1.11 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`.

14.1.12 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`.

14.1.13 ResidueOfFlag

▷ `ResidueOfFlag(cg, fl)` (operation)

Returns: the residue of `fl` in `cg`.

`cg` must be a coset geometry. CHECK the back-mapping. Still not quite right. I'll have another look.

14.1.14 IncidenceGraph

▷ `IncidenceGraph(cg)` (operation)

Returns: incidence graph of `cg`.

`cg` must be a coset geometry. The graph returned is a GRAPE object. Be sure the GRAPE is loaded! All GRAPE functionality can now be used to analyse `cg` via its incidence graph.

14.1.15 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.

14.1.16 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).

14.1.17 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`.

14.2 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. 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 $I \setminus \{i, j\}$ 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 the 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 graph with vertex set I and edges whenever the point diamater, the line diameter and the gonality are all greater than 2.

14.2.1 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 test is time consuming. The command IsFlagTransitiveGeometry can be used to check flag-transitivity if needed.

14.2.2 DrawDiagram

▷ DrawDiagram(*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 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.

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 0 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 2 are the 4-subsets of the vertex set of K_8 and the elements of type 3 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), (1,5)(2,4)(3,7)(6,8),
>       (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(cg,[point]);
CosetGeometry( Group( [ (1,3,5), (1,7,2)(3,8)(5,6) ] ) )
gap> diagc3:=DiagramOfGeometry(residue);
< Diagram of CosetGeometry( Group( [ (1,3,5), (1,7,2)(3,8)(5,6) ] ) ,
[ Group( [ (2,3,8), (2,3,6), (5,8,6), (2,7,5,6,8) ] ),
  Group( [ (2,5)(6,8), (1,7,3), (1,7)(5,6), (5,8,6), (2,6)(5,8) ] ),
  Group( [ (1,6,3,2)(5,8), (2,7)(3,8) ] ) ] ) >
gap> DrawDiagram(diagc3, "a7geo");
gap> #Exec("gv a7geo.ps");
```

The produced diagrams are included here: Neumaier's A_8

neuma8.pdf

The A_7 geometry:

[scale=0.5]a7geo.pdf

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 liested an 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

0: IncidenceStructure: [IsList, IsFunction, IsFunction, IsList]
0: ElementsOfIncidenceStructure: [IsIncidenceStructure]
0: ElementsOfIncidenceStructure: [IsIncidenceStructure, IsPosInt]
0: ElementsOfIncidenceStructure: [IsIncidenceStructure, IsString]
0: Points: [IsIncidenceStructure]
0: Lines: [IsIncidenceStructure]
0: Planes: [IsIncidenceStructure]
0: Solids: [IsIncidenceStructure]
0: FlagOfIncidenceStructure: [IsIncidenceStructure, IsElementOfIncidenceStructureCollection]
0: FlagOfIncidenceStructure: [IsIncidenceStructure, IsListandIsEmpty]
0: ChamberOfIncidenceStructure: [IsElementOfIncidenceStructureCollection]
0: IsIncident: [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure]
0: IsCollinear: [IsIncidenceStructure, IsElementOfIncidenceStructure, IsElementOfIncidenceStruct
0: Span: [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure]
```

```
O: Meet: [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure]
O: RandomFlag: [IsIncidenceStructure]
O: RandomChamber: [IsIncidenceStructure]
O: Type: [IsElementOfIncidenceStructureandIsElementOfIncidenceStructureRep]
O: Type: [IsElementsOfIncidenceStructureandIsElementsOfIncidenceStructureRep]
O: Wrap: [IsIncidenceGeometry, IsPosInt, IsObject]
O: Unwrap: [IsElementOfIncidenceStructure]
O: \^: [IsElementOfIncidenceStructure, IsUnwrapper]
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: VectorSpaceToElement: [IsLieGeometry, IsRowVector]
O: VectorSpaceToElement: [IsLieGeometry, Is8BitVectorRep]
O: VectorSpaceToElement: [IsLieGeometry, IsPlistRep]
O: VectorSpaceToElement: [IsLieGeometry, Is8BitMatrixRep]
O: VectorSpaceToElement: [IsLieGeometry, IsGF2MatrixRep]
O: ElementToVectorSpace: [IsElementOfLieGeometry]
O: EmptySubspace: [IsLieGeometry]
O: \^: [IsEmptySubspace, IsUnwrapper]
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: ProjElWithFrob: [IsMatrixandIsFFECollColl, IsMapping]
O: ProjElWithFrob: [IsMatrixandIsFFECollColl, IsMapping, IsField]
O: ProjElsWithFrob: [IsList]
O: ProjElsWithFrob: [IsList, IsField]
O: ProjectiveSemilinearMap: [IsList, IsField]
O: ProjectiveSemilinearMap: [IsList, IsMapping, IsField]
O: ProjectivityByImageOfStandardFrameNC: [IsProjectiveSpace, IsList]
O: UnderlyingMatrix: [IsProjGrpElWithFrobandIsProjGrpElWithFrobRep]
O: UnderlyingMatrix: [IsProjGrpElandIsProjGrpElRep]
O: FieldAutomorphism: [IsProjGrpElWithFrobandIsProjGrpElWithFrobRep]
O: ActionOnAllProjPoints: [IsProjectiveGroup]
O: SetAsNiceMono: [IsProjectiveGroup, IsGroupHomomorphism]
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]

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, IsMapping, IsField, IsStandardDualityOfProjectiveSpace
O: UnderlyingMatrix: [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]
O: PolarSpace: [IsForm, IsField, IsGroup, IsFunction]
O: PolarSpace: [IsForm]
O: Polarity: [IsClassicalPolarSpace]
O: IsTotallySingular: [IsClassicalPolarSpaceandIsClassicalPolarSpaceRep,
O: IsTotallyIsotropic: [IsClassicalPolarSpaceandIsClassicalPolarSpaceRep,
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: HermitianVariety: [IsPosInt, IsField]
O: HermitianVariety: [IsPosInt, IsPosInt]
O: Span: [IsSubspaceOfClassicalPolarSpace, IsSubspaceOfClassicalPolarSpace, IsBool]

morphisms.gd: operations

O: GeometryMorphismByFunction:   [ IsAnyElementsOfIncidenceStructure,  IsAnyElementsOfIncidenceS
O: GeometryMorphismByFunction:   [ IsAnyElementsOfIncidenceStructure,  IsAnyElementsOfIncidenceS
O: GeometryMorphismByFunction:   [ IsAnyElementsOfIncidenceStructure,  IsAnyElementsOfIncidenceS
O: IsomorphismPolarSpaces:                    [ IsClassicalPolarSpace,  IsClassicalPolarSpace,
O: IsomorphismPolarSpaces:                    [ IsClassicalPolarSpace,  IsClassicalPolarSpace
O: IsomorphismPolarSpacesNC:  [ IsClassicalPolarSpace,  IsClassicalPolarSpace,  IsBool ]
O: IsomorphismPolarSpacesNC:                   [ IsClassicalPolarSpace,  IsClassicalPolarSpac
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: 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,  IsProjectiveSpac
O: NaturalEmbeddingByFieldReduction:                    [ IsProjectiveSpace,  IsProjectiveSpac
O: NaturalEmbeddingByFieldReduction:                    [ IsClassicalPolarSpace,  IsClassicalP
O: NaturalEmbeddingByFieldReduction:  [ IsClassicalPolarSpace,  IsClassicalPolarSpace]
O: NaturalEmbeddingBySubfield:                   [ IsProjectiveSpace,  IsProjectiveSpace ]
O: NaturalEmbeddingBySubfield:  [ IsClassicalPolarSpace,  IsClassicalPolarSpace,  IsBool ]
O: NaturalEmbeddingBySubfield:                   [ IsClassicalPolarSpace,  IsClassicalPolarSp
O: PluckerCoordinates: [IsSubspaceOfProjectiveSpace]
O: InversePluckerCoordinates: [IsSubspaceOfProjectiveSpace]
O: KleinCorrespondence: [IsClassicalPolarSpace]
O: NaturalDuality: [IsSymplecticSpaceandIsGeneralisedPolygon]
O: NaturalDuality: [IsHermitianVarietyandIsGeneralisedPolygon]
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, IsHomogeneousList]
O: ResidueOfFlag: [IsCosetGeometry, IsHomogeneousList]
O: CanonicalResidueOfFlag: [IsCosetGeometry, IsHomogeneousList]
O: Rk2GeoDiameter: [IsCosetGeometry, IsPosInt]
O: GeometryOfRank2Residue: [IsRank2Residue]
O: GeometryFromLabelledGraph: [IsObjectandIS_REC]
O: IncidenceGraph: [IsCosetGeometry]
O: Rank2Residues: [IsIncidenceGeometry]
O: MakeRank2Residue: [IsRank2Residue]

varieties.gd: operations

O: AlgebraicVariety: [IsProjectiveSpace, IsList]
O: AlgebraicVariety: [IsAffineSpace, IsList]
O: PointsOfAlgebraicVariety: [IsAlgebraicVariety]
O: Points: [IsAlgebraicVariety]
O: ProjectiveVariety: [IsProjectiveSpace, IsPolynomialRing, IsList]
O: ProjectiveVariety: [IsProjectiveSpace, IsList]
O: DualCoordinatesOfHyperplane: [IsSubspaceOfProjectiveSpace]
O: HyperplaneByDualCoordinates: [IsProjectiveSpace, IsList]
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: GrassmannVariety: [IsPosInt, IsPosInt, IsField]
O: GrassmannVariety: [IsPosInt, IsPosInt, IsPosInt]
O: ConicOnFivePoints: [IsHomogeneousListand                    IsSubspaceOfProjectiveS
```

```
O: PolarSpace: [IsProjectiveVariety]

affinespace.gd: operations

O: VectorSpaceTransversal: [IsVectorSpace, IsFFECollColl]
O: VectorSpaceTransversalElement: [IsVectorSpace, IsFFECollColl, IsVector]
O: ComplementSpace: [IsVectorSpace, IsFFECollColl]
O: AffineSpace: [IsPosInt, IsField]
O: AffineSpace: [IsPosInt, IsPosInt]
O: AffineSubspace: [IsAffineSpace, IsRowVector]
O: AffineSubspace: [IsAffineSpace, IsRowVector, IsPlistRep]
O: AffineSubspace: [IsAffineSpace, IsRowVector, Is8BitMatrixRep]
O: AffineSubspace: [IsAffineSpace, IsRowVector, IsGF2MatrixRep]
O: RandomSubspace: [IsAffineSpace, IsInt]
O: IsParallel: [IsSubspaceOfAffineSpace, IsSubspaceOfAffineSpace]
O: ParallelClass: [IsAffineSpace, IsSubspaceOfAffineSpace]
O: ParallelClass: [IsSubspaceOfAffineSpace]

affinegroup.gd: operations


gpolygons.gd: operations

O: SplitCayleyHexagon: [IsFieldandIsFinite]
O: SplitCayleyHexagon: [IsPosInt]
O: TwistedTrialityHexagon: [IsFieldandIsFinite]
O: TwistedTrialityHexagon: [IsPosInt]
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: EGQByKantorFamily: [IsGroup, IsList, IsList]
O: Wrap: [IsElationGQByKantorFamily, IsPosInt, IsPosInt, IsObject]
O: IsKantorFamily: [IsGroup, IsList, IsList]
O: EGQByBLTSet: [IsList, IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace]
O: EGQByBLTSet: [IsList]
O: FlockGQByqClan: [IsqClanObj]
O: BLTSetByqClan: [IsqClanObjandIsqClanRep]
O: KantorFamilyByqClan: [IsqClanObjandIsqClanRep]
O: EGQByqClan: [IsqClanObjandIsqClanRep]
O: ProjectivePlaneByBlocks: [IsHomogeneousList]
O: ProjectivePlaneByIncidenceMatrix: [IsMatrix]
O: BlockDesignOfGeneralisedPolygon: [IsGeneralisedPolygon]
O: IncidenceGraphOfGeneralisedPolygon: [IsGeneralisedPolygon]
```

Example

```
Attributes
```

```
geometry.gd: attributes

A: IsChamberOfIncidenceStructure: IsFlagOfIncidenceStructure
A: IsEmptyFlag: IsFlagOfIncidenceStructure
A: RankAttr: IsIncidenceStructure
A: TypesOfElementsOfIncidenceStructure: IsIncidenceStructure
A: TypesOfElementsOfIncidenceStructurePlural: IsIncidenceStructure
A: CollineationGroup: IsIncidenceStructure
A: CorrelationGroup: IsIncidenceStructure
A: CollineationAction: IsIncidenceStructure
A: CorrelationAction: IsIncidenceStructure
A: RepresentativesOfElements: IsIncidenceStructure
A: AmbientGeometry: IsIncidenceStructure
A: AmbientGeometry: IsElementOfIncidenceStructureandIsElementOfIncidenceStructureRep
A: AmbientGeometry: IsElementsOfIncidenceStructureandIsElementsOfIncidenceStructureRep
A: AmbientGeometry: IsAllElementsOfIncidenceStructureandIsAllElementsOfIncidenceStructureRep

liegeometry.gd: attributes

A: AmbientSpace: IsLieGeometry
A: AmbientSpace: IsElementOfLieGeometry
A: ProjectiveDimension: IsLieGeometry
A: ProjectiveDimension: IsElementOfLieGeometry
A: ProjectiveDimension: IsEmptySubspace

group.gd: attributes

A: Dimension: IsProjectiveGroup
A: Dimension: IsProjectiveGroupWithFrob

projectivespace.gd: attributes

A: HomographyGroup: IsProjectiveSpace
A: SpecialHomographyGroup: IsProjectiveSpace
A: Dimension: 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

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

affinespace.gd: attributes

A: AmbientSpace: IsAffineSpace

affinegroup.gd: attributes
```

```
A: AffineGroup: IsAffineSpace

gpolygons.gd: attributes

A: Order: IsGeneralisedPolygon
A: AmbientSpace: IsGeneralisedPolygon
A: CollineationAction: IsGroup
A: ElationGroup: IsElationGQ
A: BasePointOfEGQ: IsElationGQ
A: IncidenceMatrixOfGeneralisedPolygon: IsGeneralisedPolygon
A: IsLinearqClan: IsqClanObj
```

Example

```
Properties

geometry.gd: properties


liegeometry.gd: properties


group.gd: properties

P: CanComputeActionOnPoints: IsProjectiveGroup
P: CanComputeActionOnPoints: IsProjectiveGroupWithFrob

projectivespace.gd: properties


correlations.gd: properties

P: CanComputeActionOnPoints: 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: IsHermitianVariety: IsClassicalPolarSpace
P: IsStandardPolarSpace: IsClassicalPolarSpace

morphisms.gd: properties


enumerators.gd: properties
```

```
diagram.gd: properties


varieties.gd: properties


affinespace.gd: properties


affinegroup.gd: properties


gpolygons.gd: properties
```

A.2.2 Functions/Methods

Example

```
Functions

geometry.gi: global functions

F: HashFuncForElements

liegeometry.gi: global functions


group.gi: global functions

F: MakeAllProjectivePoints
F: IsScalarMatrix
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: OnProjSubspacesReversing

polarspace.gi: global functions
```

```
morphisms.gi: global functions

F: LeukBasis

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: 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: OnKantorFamily
```

Example

```
Methods

geometry.gi: methods

M: Wrap, [IsIncidenceGeometry, IsPosInt, IsObject],
M: Unwrap, [IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep],
M: \^, [IsElementOfIncidenceStructure, IsUnwrapper ],
M: AmbientGeometry, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: AmbientGeometry, [ IsElementsOfIncidenceStructure and IsElementsOfIncidenceStructureRep ],
M: AmbientGeometry, [ IsAllElementsOfIncidenceStructure and IsAllElementsOfIncidenceStructureRep
M: Type, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: Type, [IsElementsOfIncidenceStructure and IsElementsOfIncidenceStructureRep],
M: Rank, [IsIncidenceStructure],
M: \=, [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure],
M: \<, [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure],
M: \*, [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure],
M: ChooseHashFunction, [ IsElementOfIncidenceStructure, IsPosInt ],
M: ViewObj, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: PrintObj, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: Display, [ IsElementOfIncidenceStructure and IsElementOfIncidenceStructureRep ],
M: ViewObj, [ IsAllElementsOfIncidenceStructure ],
M: PrintObj, [ IsAllElementsOfIncidenceStructure ],
```

```
M: ElementsOfIncidenceStructure, [IsIncidenceStructure, IsString],
M: IsChamberOfIncidenceStructure, [ IsFlagOfIncidenceStructure and IsFlagOfIncidenceStructureRep
M: ShadowOfElement, [IsIncidenceStructure, IsElementOfIncidenceStructure, IsString],
M: ShadowOfFlag, [IsIncidenceStructure, IsFlagOfIncidenceStructure, IsString],
M: ShadowOfFlag, [IsIncidenceStructure, IsList, IsPosInt],
M: ShadowOfFlag, [IsIncidenceStructure, IsList, IsString],
M: Enumerator, [IsElementsOfIncidenceStructure],
M: Intersection2, [IsElementOfIncidenceStructure, IsElementOfIncidenceStructure],
M: IncidenceStructure, [ IsList, IsFunction, IsFunction, IsList ],
M: ViewObj, [ IsIncidenceStructure ],
M: PrintObj, [ IsIncidenceStructure ],
M: Display, [ IsIncidenceStructure ],

liegeometry.gi: methods

M: Wrap, [IsLieGeometry, IsPosInt, IsObject],
M: ElementToVectorSpace, [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: ViewObj, InstallMethod(ViewObj,[IsEmptySubspace],
M: PrintObj, InstallMethod(PrintObj,[IsEmptySubspace],
M: Display, InstallMethod(Display,[IsEmptySubspace],
M: \=, [IsEmptySubspace, IsEmptySubspace],
M: \^, [ IsEmptySubspace, IsUnwrapper ],
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: ElementsIncidentWithElementOfIncidenceStructure, [ IsElementOfLieGeometry, IsPosInt],
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: ViewObj, [ IsShadowElementsOfLieGeometry and IsShadowElementsOfLieGeometryRep ],
M: \in, [IsElementOfLieGeometry, IsElementOfLieGeometry],
```

```
M: Random, [ IsSubspacesVectorSpace ],
M: RandomSubspace, [IsVectorSpace,IsInt],
M: ElementToElement, [IsLieGeometry, IsElementOfLieGeometry],
M: ConvertElement, [IsProjectiveSpace, IsElementOfLieGeometry],
M: ConvertElementNC, [IsLieGeometry, IsElementOfLieGeometry],

group.gi: methods

M: ProjEl, [IsMatrix and IsFFECollColl],
M: ProjEls, [IsList],
M: Projectivity, InstallMethod(Projectivity,[IsMatrixandIsFFECollColl,IsField],
M: ProjElWithFrob, [IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativeElementWi
M: ProjElWithFrob, [IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativeElementWi
M: ProjElWithFrob, [IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativeElementWi
M: ProjElsWithFrob, [IsList, IsField],
M: ProjElsWithFrob, [IsList],
M: ProjectiveSemilinearMap, [ IsMatrix and IsFFECollColl, IsField],
M: ProjectiveSemilinearMap, [ IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativ
M: ProjectivityByImageOfStandardFrameNC, InstallMethod(ProjectivityByImageOfStandardFrameNC,[IsP
M: UnderlyingMatrix, InstallMethod(UnderlyingMatrix,[IsProjGrpElandIsProjGrpElRep],
M: UnderlyingMatrix, InstallMethod(UnderlyingMatrix,[IsProjGrpElWithFrobandIsProjGrpElWithFrobRe
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, IsFrobeniusAutomorphism ],
M: \^, [ IsVector and IsFFECollection, IsMapping and IsOne ],
M: \^, [ IsVector and IsFFECollection and IsGF2VectorRep, IsFrobeniusAutomorphism ],
M: \^, [ IsVector and IsFFECollection and IsGF2VectorRep, IsMapping and IsOne ],
```

```
M: \^, [ IsVector and IsFFECollection and Is8BitVectorRep, IsFrobeniusAutomorphism ],
M: \^, [ IsVector and IsFFECollection and Is8BitVectorRep, IsMapping and IsOne ],
M: \^, [ IsMatrix and IsFFECollColl, IsFrobeniusAutomorphism ],
M: \^, [ IsMatrix and IsFFECollColl, IsMapping and IsOne ],
M: \^, [ IsMatrix and IsFFECollColl and IsGF2MatrixRep, IsFrobeniusAutomorphism ],
M: \^, [ IsMatrix and IsFFECollColl and IsGF2MatrixRep, IsMapping and IsOne ],
M: \^, [ IsMatrix and IsFFECollColl and Is8BitMatrixRep, IsFrobeniusAutomorphism ],
M: \^, [ IsMatrix and IsFFECollColl and Is8BitMatrixRep, 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, [IsProjectiveGroup],
M: ViewObj, [IsProjectiveGroup and IsTrivial],
M: ViewObj, [IsProjectiveGroup and HasGeneratorsOfGroup],
M: ViewObj, [IsProjectiveGroup and HasSize],
M: ViewObj, [IsProjectiveGroup and HasGeneratorsOfGroup and HasSize],
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, [IsProjectiveGroup],
M: BaseField, [IsProjectiveGroupWithFrob],
M: Dimension, [IsProjectiveGroup],
M: Dimension, [IsProjectiveGroupWithFrob],
M: OneImmutable, # was: [IsGroup and IsProjectiveGroup], I think might be
M: OneImmutable, # was [IsGroup and IsProjectiveGroupWithFrob], I think might be
M: CanComputeActionOnPoints, [IsProjectiveGroup],
M: CanComputeActionOnPoints, [IsProjectiveGroupWithFrob],
M: ActionOnAllProjPoints, [ IsProjectiveGroup ],
M: ActionOnAllProjPoints, [ IsProjectiveGroupWithFrob ],
M: SetAsNiceMono, [IsProjectiveGroup, IsGroupHomomorphism and IsInjective],
M: SetAsNiceMono, [IsProjectiveGroupWithFrob, IsGroupHomomorphism and IsInjective],
M: NiceMonomorphism, [IsProjectiveGroup and CanComputeActionOnPoints and IsHandledByNiceMonomorp
M: NiceMonomorphism, [IsProjectiveGroupWithFrob and IsHandledByNiceMonomorphism],
M: NiceMonomorphism, [IsProjectiveGroupWithFrob and CanComputeActionOnPoints and IsHandledByNice
M: NiceMonomorphism, [IsProjectiveGroupWithFrob and IsHandledByNiceMonomorphism], 50,
M: FindBasePointCandidates, [IsProjectiveGroup,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: \^, [ IsSubspaceOfProjectiveSpace, IsUnwrapper ],
M: ProjectiveSpace, [ IsInt, IsField ],
M: ProjectiveSpace, [ IsInt, IsPosInt ],
M: ViewObj, InstallMethod(ViewObj,[IsProjectiveSpaceandIsProjectiveSpaceRep],
M: PrintObj, InstallMethod(PrintObj,[IsProjectiveSpaceandIsProjectiveSpaceRep],
M: Display, InstallMethod(Display,[IsProjectiveSpaceandIsProjectiveSpaceRep],
M: UnderlyingVectorSpace, [IsProjectiveSpace and IsProjectiveSpaceRep],
M: \=, [IsProjectiveSpace, IsProjectiveSpace],
M: ProjectiveDimension, [ IsProjectiveSpace and IsProjectiveSpaceRep ],
M: Dimension, [ IsProjectiveSpace and IsProjectiveSpaceRep ],
M: Rank, [ IsProjectiveSpace and IsProjectiveSpaceRep ],
M: BaseField, InstallMethod(BaseField,"foraprojectivespace",[IsProjectiveSpace],
M: BaseField, InstallMethod(BaseField,"foranelementofaprojectivespace",[IsSubspaceOfProjectiveSp
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, IsPlistRep],
M: VectorSpaceToElement, [IsProjectiveSpace, IsGF2MatrixRep],
M: VectorSpaceToElement, [IsProjectiveSpace, Is8BitMatrixRep],
M: VectorSpaceToElement, [IsProjectiveSpace, IsRowVector],
M: VectorSpaceToElement, [IsProjectiveSpace, Is8BitVectorRep],
M: UnderlyingVectorSpace, [IsSubspaceOfProjectiveSpace],
M: ProjectiveDimension, [ IsSubspaceOfProjectiveSpace ],
M: Dimension, [ IsSubspaceOfProjectiveSpace ],
M: StandardFrame, [IsSubspaceOfProjectiveSpace],
M: Coordinates, [IsSubspaceOfProjectiveSpace],
M: CoordinatesOfHyperplane, [IsSubspaceOfProjectiveSpace],
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: Hyperplanes, [ IsSubspaceOfProjectiveSpace ],
M: Hyperplanes, [ IsProjectiveSpace, IsSubspaceOfProjectiveSpace ],
```

```
M: CollineationGroup, [ IsProjectiveSpace and IsProjectiveSpaceRep ],
M: HomographyGroup, [ IsProjectiveSpace ],
M: SpecialHomographyGroup, [ IsProjectiveSpace ],
M: \^, [IsElementOfIncidenceStructure, IsProjGrpElWithFrob],
M: AsList, [IsSubspacesOfProjectiveSpace],
M: Iterator, [IsSubspacesOfProjectiveSpace],
M: FlagOfIncidenceStructure, [ IsProjectiveSpace, IsSubspaceOfProjectiveSpaceCollection ],
M: FlagOfIncidenceStructure, [ IsProjectiveSpace, IsList and IsEmpty ],
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

correlations.gi: methods

M: IdentityMappingOfElementsOfProjectiveSpace, [IsProjectiveSpace],
M: StandardDualityOfProjectiveSpace, [IsProjectiveSpace],
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, [IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativ
M: ProjElWithFrobWithPSIsom, [IsMatrix and IsFFECollColl, IsRingHomomorphism and IsMultiplicativ
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: \^, [ IsVector and IsFFECollection, IsIdentityMappingOfElementsOfProjectiveSpace ],
M: \^, [ IsVector and IsFFECollection and IsGF2VectorRep, IsIdentityMappingOfElementsOfProjectiv
M: \^, [ IsVector and IsFFECollection and Is8BitVectorRep, IsIdentityMappingOfElementsOfProjecti
M: \^, [ IsMatrix and IsFFECollColl, IsStandardDualityOfProjectiveSpace ],
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: CorrelationGroup, [ 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, IsRingHomomorphism and IsMultipli
M: UnderlyingMatrix, InstallMethod(UnderlyingMatrix,[IsProjGrpElWithFrobWithPSIsomand IsProjGrpE
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 ],
M: PolarSpaceStandard, [ IsQuadraticForm ],
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: HermitianVariety, [ IsPosInt, IsField ],
M: HermitianVariety, [ IsPosInt, IsPosInt ],
M: BaseField, InstallMethod(BaseField,"forapolarspace",[IsClassicalPolarSpace],
M: UnderlyingVectorSpace, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep],
M: ProjectiveDimension, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: Dimension, [ IsClassicalPolarSpace ],
M: QuadraticForm, [ IsClassicalPolarSpace ],
M: PolarSpaceType, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: CompanionAutomorphism, [ IsClassicalPolarSpace ],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep ],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsEllipticQuadric],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsEllipticQuadric and IsSta
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsSymplecticSpace],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsSymplecticSpace and IsSta
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsParabolicQuadric ],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsParabolicQuadric and IsSt
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHyperbolicQuadric ],
M: ViewObj, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHyperbolicQuadric and IsS
M: ViewObj, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHermitianVariety ],
M: ViewObj, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHermitianVariety and IsSta
```

```
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 IsHermitianVariety ],
M: Display, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep and IsHermitianVariety ],
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, 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: RandomSubspace, [ IsClassicalPolarSpace, IsPosInt ],
M: Random, [ IsSubspacesOfClassicalPolarSpace ],
M: Iterator, [IsSubspacesOfClassicalPolarSpace],
M: ShadowOfElement, [IsClassicalPolarSpace, IsElementOfIncidenceStructure, IsPosInt],
M: Size, [IsShadowSubspacesOfClassicalPolarSpace andIsShadowSubspacesOfClassicalPolarSpaceRep ],
M: IsCollinear, [IsClassicalPolarSpace and IsClassicalPolarSpaceRep, IsElementOfIncidenceStructu
M: PolarityOfProjectiveSpace, [IsClassicalPolarSpace],
M: PolarSpace, [ IsPolarityOfProjectiveSpace ],
M: GeometryOfAbsolutePoints, [ IsPolarityOfProjectiveSpace ],
M: AbsolutePoints, [ IsPolarityOfProjectiveSpace ],
M: IsTotallyIsotropic, [ IsClassicalPolarSpace and IsClassicalPolarSpaceRep, IsSubspaceOfProject
M: Polarity, [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: DefiningListOfPolynomials, [IsProjectiveVariety and IsClassicalPolarSpace and IsClassicalPola

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: 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, IsProjectiveSpace, IsBasis ],
M: NaturalEmbeddingByFieldReduction, [ IsProjectiveSpace, IsProjectiveSpace ],
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, [ IsSubspaceOfProjectiveSpace ],
M: InversePluckerCoordinates, [ IsSubspaceOfProjectiveSpace ],
M: KleinCorrespondence, [ IsClassicalPolarSpace ],
M: NaturalDuality, [IsSymplecticSpace and IsGeneralisedPolygon ],
M: NaturalDuality, [ IsHermitianVariety and IsGeneralisedPolygon ],
```

```
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: ElementsOfIncidenceStructure, InstallMethod(ElementsOfIncidenceStructure,[IsCosetGeometry,IsP
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: IsConnected, [ IsCosetGeometry ],
M: IsResiduallyConnected, [ IsCosetGeometry ],
M: StandardFlagOfCosetGeometry, [ IsCosetGeometry ],
M: FlagToStandardFlag, [ IsCosetGeometry, IsHomogeneousList ],
M: CanonicalResidueOfFlag, [ IsCosetGeometry, IsHomogeneousList ],
M: ResidueOfFlag, [ IsCosetGeometry, IsHomogeneousList ],
M: IncidenceGraph, InstallMethod(IncidenceGraph,[IsCosetGeometryandIsHandledByNiceMonomorphism],
M: IncidenceGraph, InstallMethod(IncidenceGraph,[IsCosetGeometry],
M: ViewObj, [ IsDiagram and IsDiagramRep ],
M: ViewObj, [ IsDiagram and IsDiagramRep and HasGeometryOfDiagram],
M: ViewObj, [ IsCosetGeometry and IsCosetGeometryRep ],
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,"forcosetgeometry",[IsCosetGeometry],
M: Display, InstallMethod(Display,[IsDiagramandIsDiagramRep],
M: DiagramOfGeometry, InstallMethod(DiagramOfGeometry,"foraprojectivespace",[IsProjectiveSpace],
```

```
M: Rk2GeoDiameter, InstallMethod(Rk2GeoDiameter,"foracosetgeometry",[IsCosetGeometry, IsPosInt],
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: ProjectiveVariety, [ IsProjectiveSpace, IsPolynomialRing, IsList ],
M: ProjectiveVariety, [ IsProjectiveSpace, IsList ],
M: AlgebraicVariety, [ IsProjectiveSpace, IsList ],
M: ViewObj, [ IsProjectiveVariety and IsProjectiveVarietyRep ],
M: PrintObj, [ IsProjectiveVariety and IsProjectiveVarietyRep ],
M: DualCoordinatesOfHyperplane, [IsSubspaceOfProjectiveSpace],
M: HyperplaneByDualCoordinates, [IsProjectiveSpace,IsList],
M: AffineVariety, [ IsAffineSpace, IsPolynomialRing, IsList ],
M: AffineVariety, [ IsAffineSpace, IsList ],
M: AlgebraicVariety, [ IsAffineSpace, IsList ],
M: ViewObj, [ IsAffineVariety and IsAffineVarietyRep ],
M: PrintObj, [ IsAffineVariety and IsAffineVarietyRep ],
M: \in, [IsElementOfIncidenceStructure, IsAlgebraicVariety],
M: PointsOfAlgebraicVariety, [IsAlgebraicVariety and IsAlgebraicVarietyRep],
M: ViewObj, [ IsPointsOfAlgebraicVariety and IsPointsOfAlgebraicVarietyRep ],
M: Points, [IsAlgebraicVariety and IsAlgebraicVarietyRep],
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: SegreVariety, [IsHomogeneousList],
M: SegreVariety, [IsHomogeneousList, IsField ],
M: ViewObj, [ IsSegreVariety and IsSegreVarietyRep ],
M: PrintObj, [ IsSegreVariety and IsSegreVarietyRep ],
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: VeroneseMap, [IsProjectiveSpace],
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: ConicOnFivePoints, [ IsHomogeneousList and IsSubspaceOfProjectiveSpaceCollection ],
M: GrassmannCoordinates, [ IsSubspaceOfProjectiveSpace ],
M: GrassmannMap, [ IsPosInt, IsProjectiveSpace ],
M: GrassmannMap, [ IsPosInt, IsPosInt, IsPosInt ],
M: PolarSpace, [IsProjectiveVariety and IsProjectiveVarietyRep],

affinespace.gi: methods

M: AffineSpace, InstallMethod(AffineSpace,[IsPosInt,IsField],
M: AffineSpace, [ IsPosInt, IsPosInt ],
M: RankAttr, [ IsAffineSpace and IsAffineSpaceRep ],
M: TypesOfElementsOfIncidenceStructure, InstallMethod(TypesOfElementsOfIncidenceStructure,"foran
M: TypesOfElementsOfIncidenceStructurePlural, [IsAffineSpace],
M: Wrap, [IsAffineSpace, IsPosInt, IsObject],
M: AffineSubspace, [IsAffineSpace, IsRowVector, IsPlistRep],
M: AffineSubspace, [IsAffineSpace, IsRowVector],
M: AffineSubspace, [IsAffineSpace, IsRowVector, Is8BitMatrixRep],
M: AffineSubspace, [IsAffineSpace, IsRowVector, IsGF2MatrixRep],
M: RandomSubspace, [ IsAffineSpace, IsInt ],
M: Random, [ IsAllSubspacesOfAffineSpace ],
M: \in, [IsSubspaceOfAffineSpace, IsAffineSpace],
M: ElementsOfIncidenceStructure, InstallMethod(ElementsOfIncidenceStructure,[IsAffineSpace],
M: ElementsOfIncidenceStructure, InstallMethod(ElementsOfIncidenceStructure,[IsAffineSpace,IsPos
M: Points, InstallMethod(Points,[IsAffineSpace],
M: Lines, InstallMethod(Lines,[IsAffineSpace],
M: Planes, InstallMethod(Planes,[IsAffineSpace],
M: Solids, InstallMethod(Solids,[IsAffineSpace],
M: Size, [IsAllSubspacesOfAffineSpace],
M: ComplementSpace, InstallMethod(ComplementSpace,[IsVectorSpace,IsFFECollColl],
M: VectorSpaceTransversalElement, InstallMethod(VectorSpaceTransversalElement,[IsVectorSpace,IsF
M: VectorSpaceTransversal, InstallMethod(VectorSpaceTransversal,[IsVectorSpace,IsFFECollColl],
M: Enumerator, InstallMethod(Enumerator,[IsVectorSpaceTransversal],
M: Iterator, [IsAllSubspacesOfAffineSpace],
M: Enumerator, [ IsAllSubspacesOfAffineSpace ],
M: ViewObj, InstallMethod(ViewObj,[IsAffineSpaceandIsAffineSpaceRep],
M: PrintObj, InstallMethod(PrintObj,[IsAffineSpaceandIsAffineSpaceRep],
M: ViewObj, InstallMethod(ViewObj,[IsAllSubspacesOfAffineSpaceand IsAllSubspacesOfAffineSpaceRep
M: PrintObj, InstallMethod(PrintObj,[IsAllSubspacesOfAffineSpaceand IsAllSubspacesOfProjectiveSp
M: Display, InstallMethod(Display,[IsSubspaceOfAffineSpace],
M: ViewObj, InstallMethod(ViewObj,[IsVectorSpaceTransversalandIsVectorSpaceTransversalRep],
M: IsIncident, InstallMethod(IsIncident,[IsSubspaceOfAffineSpace,IsSubspaceOfAffineSpace],
M: Span, InstallMethod(Span,[IsSubspaceOfAffineSpace,IsSubspaceOfAffineSpace],
M: Meet, InstallMethod(Meet,[IsSubspaceOfAffineSpace,IsSubspaceOfAffineSpace],
M: IsParallel, [ IsSubspaceOfAffineSpace, IsSubspaceOfAffineSpace ],
M: ProjectiveCompletion, InstallMethod(ProjectiveCompletion,[IsAffineSpace],
M: ShadowOfElement, InstallMethod(ShadowOfElement,[IsAffineSpace,IsSubspaceOfAffineSpace,IsPosIn
M: ShadowOfFlag, InstallMethod(ShadowOfFlag,[IsAffineSpace,IsFlagOfIncidenceStructure,IsPosInt],
```

```
M: ParallelClass, [IsAffineSpace, IsSubspaceOfAffineSpace],
M: ParallelClass, InstallMethod(ParallelClass,"foranaffinesubspace",[IsSubspaceOfAffineSpace],
M: Iterator, [IsParallelClassOfAffineSpace and IsParallelClassOfAffineSpaceRep ],
M: Size, InstallMethod(Size,[IsShadowSubspacesOfAffineSpaceand IsShadowSubspacesOfAffineSpaceRep
M: Iterator, [IsShadowSubspacesOfAffineSpace and IsShadowSubspacesOfAffineSpaceRep ],
M: ViewObj, InstallMethod(ViewObj,[IsShadowSubspacesOfAffineSpaceand IsShadowSubspacesOfAffineSp
M: ViewObj, InstallMethod(ViewObj,[IsParallelClassOfAffineSpaceand IsParallelClassOfAffineSpaceR
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],

affinegroup.gi: methods

M: AffineGroup, InstallMethod(AffineGroup,[IsAffineSpace],
M: CollineationGroup, InstallMethod(CollineationGroup,[IsAffineSpace],
M: \^, InstallOtherMethod(\^,[IsSubspaceOfAffineSpace,IsProjGrpElWithFrob],

gpolygons.gi: methods

M: ElementsOfIncidenceStructure, [IsGeneralisedPolygon and IsGeneralisedPolygonRep, IsPosInt],
M: ElementsOfIncidenceStructure, [IsElationGQByKantorFamily, IsPosInt],
M: ElementsOfIncidenceStructure, [IsGeneralisedHexagon and IsGeneralisedPolygonRep, IsPosInt],
M: Points, [IsGeneralisedPolygon and IsGeneralisedPolygonRep],
M: Lines, [IsGeneralisedPolygon and IsGeneralisedPolygonRep],
M: Size, [IsAllElementsOfGeneralisedPolygon],
M: Iterator, [IsAllElementsOfGeneralisedPolygon],
M: IsIncident, [IsElementOfGeneralisedPolygon, IsElementOfGeneralisedPolygon],
M: Wrap, [IsGeneralisedPolygon, IsPosInt, IsObject],
M: SplitCayleyHexagon, [ IsField and IsFinite ],
M: SplitCayleyHexagon, [ IsPosInt ],
M: TwistedTrialityHexagon, [ IsField and IsFinite ],
M: TwistedTrialityHexagon, [ IsPosInt ],
M: Wrap, [IsGeneralisedHexagon and IsLieGeometry, IsPosInt, IsObject],
M: Iterator, [IsAllElementsOfGeneralisedHexagon],
M: \=, [IsElementOfKantorFamily, IsElementOfKantorFamily],
M: \<, [IsElementOfKantorFamily, IsElementOfKantorFamily],
M: IsKantorFamily, [IsGroup, IsList, IsList],
M: EGQByKantorFamily, [IsGroup, IsList, IsList],
M: Iterator, [IsAllElementsOfKantorFamily],
M: IsIncident, [IsElementOfKantorFamily, IsElementOfKantorFamily],
M: Wrap, [IsElationGQByKantorFamily, IsPosInt, IsPosInt, IsObject],
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: BLTSetByqClan, [ IsqClanObj and IsqClanRep ],
M: EGQByBLTSet, [ IsList ],
M: FlockGQByqClan, InstallMethod(FlockGQByqClan,[IsqClanObj],
M: EGQByBLTSet, [IsList, IsSubspaceOfProjectiveSpace, IsSubspaceOfProjectiveSpace],
M: DiagramOfGeometry, InstallMethod(DiagramOfGeometry,[IsGeneralisedQuadrangle],
M: ProjectivePlaneByBlocks, InstallMethod(ProjectivePlaneByBlocks,[IsHomogeneousList],
M: ProjectivePlaneByIncidenceMatrix, InstallMethod(ProjectivePlaneByIncidenceMatrix,[IsMatrix],
M: CollineationGroup, [ IsProjectivePlane and IsGeneralisedPolygonRep ],
M: Span, [IsElementOfGeneralisedPolygon, IsElementOfGeneralisedPolygon], ## do we want special f
M: Meet, [IsElementOfGeneralisedPolygon, IsElementOfGeneralisedPolygon], ## do we want special f
M: \^, InstallOtherMethod(\^,[IsElementOfGeneralisedPolygon,IsPerm],
M: BlockDesignOfGeneralisedPolygon, [ IsProjectivePlane and IsGeneralisedPolygonRep ],
M: BlockDesignOfGeneralisedPolygon, [ IsGeneralisedPolygon and IsGeneralisedPolygonRep ],
M: IncidenceGraphOfGeneralisedPolygon, [ IsGeneralisedPolygon ],
M: IncidenceGraphOfGeneralisedPolygon, [ IsProjectivePlane and IsGeneralisedPolygonRep ],
M: IncidenceMatrixOfGeneralisedPolygon, [ IsGeneralisedPolygon ],
M: ViewObj, [ IsGeneralisedPolygon and IsGeneralisedPolygonRep and HasOrder],
M: PrintObj, InstallMethod(PrintObj,[IsGeneralisedPolygonandIsGeneralisedPolygonRepandHasOrder],
M: ViewObj, InstallMethod(ViewObj,[IsElationGQandHasOrder],
M: ViewObj, InstallMethod(ViewObj,[IsElationGQandHasOrderandHasBasePointOfEGQ],
M: PrintObj, InstallMethod(PrintObj,[IsElationGQandHasOrder],
M: ViewObj, [ IsProjectivePlane and HasOrder],
M: PrintObj, InstallMethod(PrintObj,[IsProjectivePlaneandHasOrder],
M: ViewObj, InstallMethod(ViewObj,[IsGeneralisedQuadrangleandHasOrder],
M: PrintObj, InstallMethod(PrintObj,[IsGeneralisedQuadrangleandHasOrder],
M: ViewObj, [ IsClassicalGQ and HasOrder and IsEllipticQuadric],
M: PrintObj, [ IsClassicalGQ and HasOrder and IsEllipticQuadric ],
M: Display, [ IsClassicalGQ and HasOrder and IsEllipticQuadric ],
M: ViewObj, [ IsClassicalGQ and HasOrder and IsSymplecticSpace],
M: PrintObj, [ IsClassicalGQ and HasOrder and IsSymplecticSpace ],
M: ViewObj, [ IsClassicalGQ and HasOrder and IsParabolicQuadric ],
M: PrintObj, [ IsClassicalGQ and HasOrder and IsParabolicQuadric ],
M: ViewObj, [ IsClassicalGQ and HasOrder and IsHyperbolicQuadric ],
M: PrintObj, [ IsClassicalGQ and HasOrder and IsHyperbolicQuadric ],
M: ViewObj, [ IsClassicalGQ and HasOrder and IsHermitianVariety ],
M: PrintObj, [ IsClassicalGQ and HasOrder and IsHermitianVariety ],
M: ViewObj, InstallMethod(ViewObj,[IsClassicalGQandHasOrder],
M: PrintObj, InstallMethod(PrintObj,[IsClassicalGQandHasOrder],
M: ViewObj, [ IsGeneralisedHexagon and HasOrder ],
M: PrintObj, [ IsGeneralisedHexagon and HasOrder ],
M: ViewObj, [ IsGeneralisedOctogon and HasOrder ],
M: PrintObj, [ IsGeneralisedOctogon and HasOrder ],
M: ViewObj, [ IsAllElementsOfGeneralisedPolygon and IsAllElementsOfGeneralisedPolygonRep ],
M: PrintObj, [ IsAllElementsOfGeneralisedPolygon and IsAllElementsOfGeneralisedPolygonRep ],
```

```
M: ViewObj, [ IsElementOfKantorFamily ],
M: PrintObj, [ IsElementOfKantorFamily ],
M: ViewObj, [ IsElementOfGeneralisedPolygon ],
M: PrintObj, [ IsElementOfGeneralisedPolygon ],
```

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 8.2, 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

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ -1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 0 & -1 & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & 0 & 0 & \dots & -1 & 0 \end{pmatrix}.$$

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

$$\begin{pmatrix} 0 & a & 0 & 0 & \dots & 0 & 0 \\ a & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & a & \dots & 0 & 0 \\ 0 & 0 & a & 0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & a \\ 0 & 0 & 0 & 0 & \dots & a & 0 \end{pmatrix}.$$

with $a = \frac{p+1}{2}$ 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 $x_1x_2 + x_3x_4 + \ldots + x_{d-1}x_d$

If `type` equals "elliptic", the gram matrix returned will result in the quadratic form $x_1^2 + x_1x_2 + \nu x_2^2 + x_3x_4 + \ldots + x_{d-1}x_d$with $\nu = \alpha^i$, 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 $x^2 + x + \nu$ is irreducible over $GF(q)$.

If `type` equals "parabolic", the gram matrix returned will result in the quadratic form $x_1^2 + x_2x_3 + \ldots x_{d-1}x_d$

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 8, Section 8.5, 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 8.5.

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)
```

References

[BLT90] Laura Bader, Guglielmo Lunardon, and Joseph A. Thas. Derivation of flocks of quadratic cones. *Forum Math.*, 2(2):163–174, 1990. 149

[BS74] Francis Buekenhout and Ernest Shult. On the foundations of polar geometry. *Geometriae Dedicata*, 3:155–170, 1974. 83

[Cam00a] Peter J. Cameron. *Classical Groups*. Online notes, http://www.maths.qmul.ac.uk/~pjc/class_gps/, 2000. 70

[Cam00b] Peter J. Cameron. *Projective and Polar Spaces*. Online notes, http://www.maths.qmul.ac.uk/~pjc/pps/, 2000. 84

[Gil08] Nick Gill. Polar spaces and embeddings of classical groups. *to appear in New Zealand J. Math.*, 2008. 128

[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. 37, 75, 84

[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. 75, 85, 130

[Pay85] Stanley E. Payne. A new infinite family of generalized quadrangles. In *Proceedings of the sixteenth Southeastern international conference on combinatorics, graph theory and computing (Boca Raton, Fla., 1985)*, volume 49, pages 115–128, 1985. 146

[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. 145

[Tit74] Jacques Tits. *Buildings of spherical type and finite BN-pairs*. Springer-Verlag, Berlin, 1974. Lecture Notes in Mathematics, Vol. 386. 83

[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. 83

[VY65a] Oswald Veblen and John Wesley Young. *Projective geometry. Vol. 1*. Blaisdell Publishing Co. Ginn and Co. New York-Toronto-London, 1965. 37, 109

[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. 37, 109

Index