GAP 4 Package FinInG

Finite Incidence Geometry

1.01

March 2009

John Bamberg, Anton Betten, Philippe Cara, Jan De Beule, Michel Lavrauw, Maska Law, Max Neunhoeffer, Michael Pauley, Sven Reichard

Copyright

Contents

Chapter 1

Introduction

1.1 Philosophy

FinInG is a package for computation in finite 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. *More text on other chapters to be written.*

Finally, Chapter 3 shows instructive examples for the usage of this package.

1.3 The Development Team

This is the development team, from left to right: Philippe Cara, Michel Lavrauw, Max Neunhöffer, Jan De Beule and John Bamberg, meeting in St. Andrews in september 2008.

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

Chapter 2

Installation of the FinInG-Package

This package is dependent on three other packages: Forms, GenSS and Orb. If you do not already have these packages installed on your system, you will need to do so in order to use FinInG. To install this package just extract the package's archive file to the GAP `pkg` directory.

By default the FinInG package is not automatically loaded by GAP when it is installed. You must load the package with `LoadPackage("fining");` before its functions become available. Please, send us an e-mail if you have any questions, remarks, suggestions, etc. concerning this package. Also, we would like to hear about applications of this package.

The Authors

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.

Example
```
gap> pg := ProjectiveSpace(2, 4);
PG(2, 4)
gap> points := Points( pg );
<points of PG(2, 4)>
gap> pointslist := AsList( points );
[ <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)>,
  <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)>,
  <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)>,
  <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)>,
  <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)>,
  <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)>,
  <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)> ]
gap> Display( pointslist[1] );
[ 0*Z(2), 0*Z(2), Z(2)^0 ]
```

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

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

```
gap> Display(pointslist[1]);
[ 0*Z(2), 0*Z(2), Z(2)^0 ]
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 PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)>,
  <a point in PG(2, 4)> ]
gap> pairs := Combinations(frame,2);
[ [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ],
  [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ],
  [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ],
  [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ],
  [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ],
  [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ] ]
gap> secants := List(pairs,p -> Span(p[1],p[2]));
[ <a line in PG(2, 4)>, <a line in PG(2, 4)>, <a line in PG(2, 4)>,
  <a line in PG(2, 4)>, <a line in PG(2, 4)>, <a line in PG(2, 4)> ]
gap> leftover := Filtered(pointslist,t->not ForAny(secants,s->t in s));
[ <a point in PG(2, 4)>, <a point in PG(2, 4)> ]
gap> hyperoval := Union(frame,leftover);
[ <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)>,
  <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)> ]
gap> g := CollineationGroup(pg);
PGammaL(3,4)
gap> stab := Stabilizer(g,Set(hyperoval),OnSets);
<projective semilinear group of size 720>
gap> StructureDescription(stab);
"S6"
```

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 PG(2, 4)>, <a point in PG(2, 4)> ],
  [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ],
  [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ],
  [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ],
  [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ],
  [ <a point in PG(2, 4)>, <a point in PG(2, 4)> ] ]
gap> secants := List( pairs, p -> Join(p[1], p[2]) );
[ <a line in PG(2, 4)>, <a line in PG(2, 4)>, <a line in PG(2, 4)>,
  <a line in PG(2, 4)>, <a line in PG(2, 4)>, <a line in PG(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 PG(2, 4)>, <a point in PG(2, 4)> ]
gap> hyperoval := Union( frame, leftover );
[ <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)>,
  <a point in PG(2, 4)>, <a point in PG(2, 4)>, <a point in PG(2, 4)> ]
```

This hyperoval has S_6 as its stabiliser, which can easily be calculated:

Example
```
gap> g := CollineationGroup(pg);
PGammaL(3,4)
gap> stab := Stabilizer(g, Set(hyperoval), OnSetsProjSubspaces);
<projective group with Frobenius 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:=HermitianVariety(2, 7^2);
H(2, 7^2)
gap> pg := AmbientSpace( h );
PG(2, 49)
gap> lines := Lines( pg );
<lines of PG(2, 49)>
gap> curve := AsList( Points( h ) );;
#I  Computing nice monomorphism...
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 );
PG(5, 3)
gap> solids := ElementsOfIncidenceStructure(pg, 4);
<solids of PG(5, 3)>
gap> iter := Iterator( solids );
<iterator>
gap> perp := PolarityOfProjectiveSpace( w5 );
<polarity of PG(5, GF(3)), < immutable compressed matrix 6x6 over GF(
3) >, F^0 >
gap> solid := NextIterator( iter );
<a solid in PG(5, 3)>
gap> solid^perp;
<a line in PG(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));;
#I  Computing nice monomorphism...
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);
#I  Computing nice monomorphism...
PSp(6,3)
gap> syl := SylowSubgroup(g, 13);
<projective semilinear 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> 13s := Filtered(partialspreads, i -> Size(i) = 13);;
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));;
```

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).

Example
```
gap> id := IdentityMat(7, GF(3));;
gap> form := QuadraticFormByMatrix(id, GF(3));
< quadratic form >
gap> ps := PolarSpace( form );
<polar space of dimension 6 over GF(3)>
```

The construction of the ovoid (a la Shult):

```
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...

```
gap> planes := AsList( Planes( ps ) );;
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.

```
gap> g := IsometryGroup( ps );
<projective semilinear group of size 9170703360 with 2 generators>
gap> stabovoid := SetwiseStabilizer(g, OnProjSubspaces, ovoid)!.setstab;
<projective semilinear group with 14 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.

```
gap> f := GF(3);
GF(3)
gap> id := IdentityMat(2, f);;
gap> clan := List( f, t -> t*id );;
gap> IsqClan( clan, f );
true
gap> egq := EGQByqClan( clan, f );
#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), 0*Z(3), Z(3) ], [ 0*Z(3), Z(3)^0, 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 ] ]
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, f );
[ <a point in Q(4, 3)>, <a point in Q(4, 3)>, <a point in Q(4, 3)>,
  <a point in Q(4, 3)> ]
gap> q4q := AmbientGeometry( blt[1] );
Q(4, 3)
gap> span := Join( blt );
<a plane in PG(4, 3)>
gap> Print("Now we see if this BLT-set is a conic\n");
Now we see if this BLT-set is a conic
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> SetInfoLevel( InfoDesargues, 0 );
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> egq1 := EGQByqClan( qclan, f );
<EGQ of order [ 81, 9 ] and basepoint 0>
gap> blt := BLTSetByqClan( qclan, f );
[ <a point in Q(4, 9)>, <a point in Q(4, 9)>, <a point in Q(4, 9)>,
  <a point in Q(4, 9)>, <a point in Q(4, 9)>, <a point in Q(4, 9)>,
  <a point in Q(4, 9)>, <a point in Q(4, 9)>, <a point in Q(4, 9)>,
  <a point in Q(4, 9)> ]
gap> egq2 := EGQByBLTSet( blt );
<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

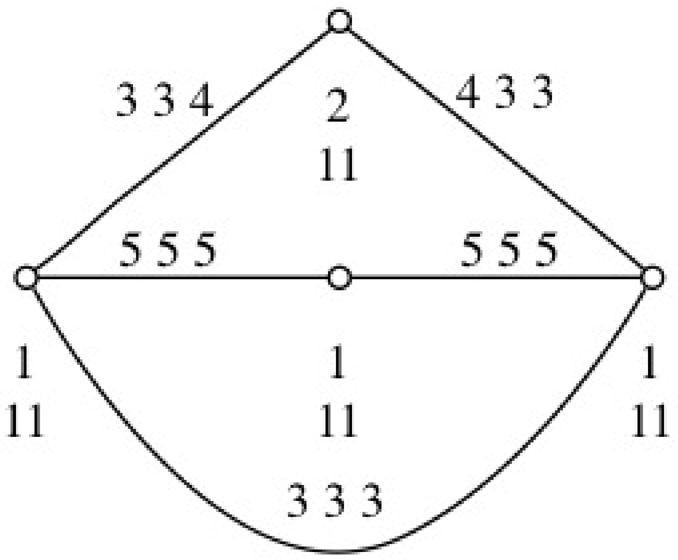
3.4.1 A rank 4 geometry for $PSL(2,11)$

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)(11,1\
2) ]);
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 );
<varieties of type 2 of Gamma>
gap> IsFlagTransitiveGeometry( cg );
true
gap> DrawDiagram( DiagramOfGeometry(cg), "PSL211");
```

The output of this example uses `dotty` which is a sophisticated graph drawing program. We could have also used `neato` to make a diagram with straight lines, which we may offer in a later version of our package. Here is what the output looks like:



Chapter 4

Incidence Geometry

The term geometry, or incidence geometry, is interpreted broadly 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 introduced the reader to yet in this manual. For further information on projective spaces, see Chapter 8.

probably more detailed information is needed here. E.g. we start to explain Incidence Geometry, but suddenly, we talk about Incidence Structures in the next sections. Not clear at all if we mean the same thing or not, and, we have Incidence Geometries *and* Incidence Structures in this package. Essentially, 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 geometries.

4.1 Incidence structures

particular and general examples of incidence structures. Show categorie graph?

4.1.1 IsIncidenceStructure

◊ `IsIncidenceStructure` (Category)

General Top level category for all objects representing an incidence structure. `IsIncidenceStructure`.

4.1.2 IsIncidenceGeometry

◊ `IsIncidenceGeometry` (Category)

General category for all objects representing an incidence geometry.

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

4.1.3 Main categories in `IsIncidenceGeometry`

◊ `IsLieGeometry` (Category)
◊ `IsAffineSpace` (Category)
◊ `IsGeneralisedPolygon` (Category)

4.1.4 Main categories in `IsLieGeometry`

◊ `IsProjectiveSpace` (Category)
◊ `IsClassicalPolarSpace` (Category)

Lie geometries bundle projective spaces and classical polar spaces. Both classes of geometries have their category, as a subcategory of `IsLieGeometry`.

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

4.1.5 Categories in `IsGeneralisedPolygon`

◊ `IsGeneralisedQuadrangle` (Category)
◊ `IsGeneralisedHexagon` (Category)
◊ `IsGeneralisedOctogon` (Category)

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

4.1.6 IsElationGQ

◊ `IsElationGQ` (Category)

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

4.1.7 IsClassicalGQ

◊ `IsClassicalGQ` (Category)

This category lies in `IsElationGQ` and `IsClassicalPolarSpace`.

4.1.8 Examples of categories of incidence geometries

```
gap> CategoriesOfObject(ProjectiveSpace(5,7));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsLieGeometry",
  "IsProjectiveSpace" ]
gap> CategoriesOfObject(HermitianVariety(5,9));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsLieGeometry",
  "IsClassicalPolarSpace" ]
gap> CategoriesOfObject(AffineSpace(3,3));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsAffineSpace" ]
gap> CategoriesOfObject(SymplecticSpace(3,11));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsLieGeometry",
  "IsClassicalPolarSpace", "IsGeneralisedPolygon", "IsGeneralisedQuadrangle",
  "IsElationGQ", "IsClassicalGQ" ]
gap> CategoriesOfObject(SplitCayleyHexagon(9));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsGeneralisedPolygon",
  "IsGeneralisedHexagon" ]
gap> CategoriesOfObject(ParabolicQuadric(4,16));
[ "IsIncidenceStructure", "IsIncidenceGeometry", "IsLieGeometry",
  "IsClassicalPolarSpace", "IsGeneralisedPolygon", "IsGeneralisedQuadrangle",
  "IsElationGQ", "IsClassicalGQ" ]
```

4.1.9 TypesOfElementsOfIncidenceStructure

◊ TypesOfElementsOfIncidenceStructure(*ig*) (operation)
◊ TypesOfElementsOfIncidenceStructurePlural(*ig*) (operation)

Returns: a list of strings

Any incidence structure contains elements of a set of types. Names can be given to an element of each type, and this operation returns the names for the particular *ig*. The second variant returs the list of plurals of these names.

```
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. 5",
  "affine. subspaces of dim. 6", "affine. subspaces of dim. 7" ]
```

4.1.10 Rank

◊ Rank(*ig*) (operation)

Returns: rank of *ig*

```
```

4.1.11 AmbientGeometry

◊ AmbientGeometry(*ig*) (operation)

Returns:

Example

```
```

4.1.12 AmbientSpace

◊ AmbientSpace(*ig*) (operation)

Returns:

Example

```
```

4.1.13 IsIncident

◊ IsIncident(*ig*) (operation)

Returns:

Example

```
```

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)

In FinInG a category for the indidual elements of an incidence structure in a category `IsIncStr` is declared and named `IsElementOfIncStr`. The inclusion for different categories of geometries is followed for their elements, except for `IsSubspaceOfClassicalPolarSpace`, which is a subcategory of `IsSubspaceOfProjectiveSpace`, while `IsClassicalPolarSpace` is not a subcategory of `IsProjectiveSpace`. The reasons for this construction are obvious.

Example

```
gap> Random(Lines(SplitCayleyHexagon(3)));
<a line in Q(6, 3)>
gap> CategoriesOfObject(last);
[ "IsElementOfIncidenceStructure", "IsElementOfIncidenceGeometry",
  "IsElementOfLieGeometry", "IsSubspaceOfProjectiveSpace",
  "IsSubspaceOfClassicalPolarSpace" ]
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)

In FinInG for an incidence structure *ig* in the category `IsIncStr`, a category `IsElementsOfIncStr` is declared for objects representing a set of elements of *ig*, all of the same type. E.g. the set of all elements of a given type of *ig* will be constructed in `IsElementsOfIncStr`. The chain of inclusions for `IsElementsOfIncStr` follows the chain of inclusions of `IsElementOfIncStr`.

Example
```
```

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(ig)` (operation)
◊ `ElementsOfIncidenceStructure(ig, j)` (operation)
◊ `ElementsOfIncidenceStructure(ig, str)` (operation)

Returns: a list of elements

ig must be an incidence geometry. *j* must be a type of element of *ig*. This function returns a list of all elements of *ig* of type *j*, and an error is displayed if *ig* has no elements of type *j*. 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 *ig* has elements of the deduced type.

Example
```
gap> ps := ProjectiveSpace(3,3);
ProjectiveSpace(3, 3)
gap> l := ElementsOfIncidenceStructure(ps,2);
<lines of ProjectiveSpace(3, 3)>
gap> ps := EllipticQuadric(5,9);
Q-(5, 9)
gap> lines := ElementsOfIncidenceStructure(ps,2);
<lines of Q-(5, 9)>
gap> planes := ElementsOfIncidenceStructure(ps,3);
Error, <geo> has no elements of type <j> called from
<function>( <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> 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(*ps*) (operation)
◊ Lines(*ps*) (operation)
◊ Planes(*ps*) (operation)

Returns: The elements of *ps* of respective type 1,2, and 3.

Example
```
gap> Points(HermitianVariety(2,64));
<points of H(2, 8^2)>
gap> Lines(EllipticQuadric(5,2));
<lines of Q-(5, 2)>
gap> Planes(SymplecticSpace(7,3));
<planes of W(7, 3)>
```

4.2.5 Hyperplanes

◊ Hyperplanes(*ps*) (operation)

Returns: a list of hyerplanes

This function returns a list of all elements of *ps* of the maximum type. *ps* must be an incidence geometry, but only methods for projective and affine spaces are installed, since a “hyperplane” has either no, either a completely different meaning in other geometires than projective and affine spaces. This operation is perhaps more synonomous and better suited when using projective and affine spaces, however one can also use it for an arbitrary incidence geometry.

Example
```
gap ps := ProjectiveSpace(5,5);
PG(5, 5)
gap> Hyperplanes(ps);
<proj. 4-subspaces of PG(5, 5)>
```

4.2.6 IsIncident

◊ IsIncident(*u*, *v*) (operation)

Returns: Boolean

u and *v* must be elements of an incidence structure. This function returns

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

4.2.7 Random

◊ `Random(u, v)` (operation)

Returns: Boolean

u and *v* must be elements of an incidence structure. This function returns

Example

```
```

4.2.8 Size

◊ `Size(u, v)` (operation)

Returns: Boolean

u and *v* must be elements of an incidence structure. This function returns

Example

```
```

4.3 Shadows of elements

4.3.1 ShadowOfElement

◊ `ShadowOfElement(geom, v, j)` (operation)

Returns: a collection of elements

geom is an incidence geometry. *v* must be an element of *geom*. This function returns a list of all elements of *geom* of type *j* which are incident with *v*. This operation may not be installed for all incidence geometries. In some case, such as when we are working with objects of projective or polar spaces, we can just type `Points(v)` for the set of points of the ambient geometry incident with *v*. We can perform a similar operation with `Lines`, `Planes` and `Solids`.

Example

```
gap> ps := ProjectiveSpace(3,3);
PG(3, 3)
gap> pi := Random(Planes(ps));
<a plane in PG(3, 3)>
```

```
gap> lines := ShadowOfElement(ps,pi,2);
<shadow lines in PG(3, 3)>
gap> Size(lines);
13
```

4.3.2 ShadowOfFlag

◊ `ShadowOfFlag(`*`geom, v, j`*`)` (operation)

Returns: a collection of elements

geom must be aan incidence geometry. *v* must be a list of elements of *geom*. This function returns a list of all elements of *geom* of type *j* which are incident with every element of the list. The function assumes that the list *v* is a flag; that is, every element of *v* is incident with every other element of *v*. If *v* is not a flag then the return value is unspecified. This operation may not be installed for all incidence geometries.

Example

```
gap> ps := ProjectiveSpace(3,3);
PG(3, 3)
gap> pi := Random(Planes(ps));
<a plane in PG(3, 3)>
gap> x := Random( ShadowOfElement(ps, pi, 1) );
<a point in PG(3, 3)>
gap> IsIncident(x,pi);
true
gap> lines := ShadowOfElement(ps,pi,2);
<shadow lines in PG(3, 3)>
gap> Size(lines);
13
```

Chapter 5

Projective Spaces

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

5.1 Creating Projective Spaces and basic operations

A *projective space* is a point-line incidence geometry, satisfying few well known axioms. In FinInG, we deal with finite Desarguesion projective spaces. It is well known that these geometries can be described completely using vector spaces over finite fields. Hence, the underlying vector space and matrix group are to our advantage. We refer the reader to [HT91] for the necessary background theory (if it is not otherwise provided).

5.1.1 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)
```

5.1.2 ProjectiveDimension

◊ `ProjectiveDimension(ps)` (operation)
◊ `Dimension(ps)` (operation)
◊ `Rank(ps)` (operation)

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
```

5.1.3 BaseField

◊ `BaseField(ps)` (operation)

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

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

5.1.4 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 )
```

5.2 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 basis vectors for the subspace. An exceptional case is when *v* is a zero-vector, whereby 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);
[ Z(7), Z(7)^5, Z(7)^3, 0*Z(7), Z(7), Z(7)^2, Z(7) ]
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 vectorspace 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;
```

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*

Example
```
gap> EmptySubspace;
< trivial subspace >
gap> line := Random(Lines(PG(5,9)));
<a line in ProjectiveSpace(5, 9)>
gap> EmptySubspace * line;
true
gap> EmptySubspace * PG(3,11);
true
```

5.2.3 ProjectiveDimension

◊ `ProjectiveDimension(`*`sub`*`)` (operation)

◊ `Dimension(`*`sub`*`)` (operation)

Returns: the projective dimension of a subspace of a projective space. This operation 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*, $1 lt j lt d$ this operation returns the collection of $j-1$ dimenaional subspaces.

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);
[ Z(7), Z(7)^5, Z(7)^3, 0*Z(7), Z(7), Z(7)^2, Z(7) ]
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 vectorspace 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;
```

5.2.5 StandardFrame

◊ StandardFrame(*ps*) (operation)

Returns: returns the standard frame in the projective space *ps*

Example
```
gap> sf := StandardFrame(PG(5,16));
[ <a point in ProjectiveSpace(5, 16)>, <a point in ProjectiveSpace(5, 16)>,
  <a point in ProjectiveSpace(5, 16)>, <a point in ProjectiveSpace(5, 16)>,
  <a point in ProjectiveSpace(5, 16)>, <a point in ProjectiveSpace(5, 16)>,
  <a point in ProjectiveSpace(5, 16)> ]
gap> Display(sf);
[ [ 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 ] ]
```

5.2.6 Coordinates

◊ Coordinates(*p*) (operation)

Returns: the coordinates of the projective point *p*

Example
```
gap> sf := StandardFrame(PG(5,16));
[ <a point in ProjectiveSpace(5, 16)>, <a point in ProjectiveSpace(5, 16)>,
  <a point in ProjectiveSpace(5, 16)>, <a point in ProjectiveSpace(5, 16)>,
  <a point in ProjectiveSpace(5, 16)>, <a point in ProjectiveSpace(5, 16)>,
  <a point in ProjectiveSpace(5, 16)> ]
gap> Display(sf);
[ [ 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 ] ]
```

5.2.7 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
```

5.2.8 AmbientSpace

◇ AmbientSpace(*el*) (operation)

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

This operation is also applicable on the trivial subspace.

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)
```

5.2.9 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.

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)
```

5.2.10 AsList

◇ AsList(*subspaces*) (operation)

Returns: an Orb object or list

Example
```
John's example works, but it is not clear whether it fits in this part of the manual. I'll ask John
the difference between using orb and not using orb to list e.g. all lines of PG(3,4), is in the exa
```

5.2.11 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(8,16);
ProjectiveSpace(8, 16)
gap> RandomSubspace(ps);
<a line in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a plane in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a proj. 7-space in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a line in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a line in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a solid in ProjectiveSpace(8, 16)>
gap> RandomSubspace(ps);
<a point in ProjectiveSpace(8, 16)>
```

5.2.12 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(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^6  z^5 z^13    .
    .    1    .    .    .    . z^43 z^33 z^29    3
    .    .    1    .    .    . z^14    6 z^12  z^9
    .    .    .    1    .    . z^47 z^27 z^12 z^22
    .    .    .    .    1    .  z^3    1 z^31 z^44
    .    .    .    .    .    1 z^27 z^42 z^34 z^34
gap> RandomSubspace(ps,3);
<a solid in ProjectiveSpace(9, 49)>
gap> Display(last);
z = Z(49)
```

```
    1    .    .    . z^25  z^3 z^31 z^44 z^38  z^5
    .    1    .    . z^41  z^9    1 z^31 z^23 z^45
    .    .    1    .  z^4 z^21 z^20 z^37    4 z^25
    .    .    .    1 z^28 z^25    2    4 z^23 z^29
gap> RandomSubspace(ps,7);
<a proj. 7-space in ProjectiveSpace(9, 49)>
gap> Display(last);
z = Z(49)
    1    .    .    .    .    .    .    .  z^6  z^6
    .    1    .    .    .    .    .    .    6 z^43
    .    .    1    .    .    .    .    .    2 z^29
    .    .    .    1    .    .    .    .  z^4    2
    .    .    .    .    1    .    .    .  z^1  z^2
    .    .    .    .    .    1    .    .    4 z^30
    .    .    .    .    .    .    1    .  z^6 z^37
    .    .    .    .    .    .    .    1 z^27 z^31
gap> RandomSubspace(ps);
<a proj. 6-space in ProjectiveSpace(9, 49)>
gap> RandomSubspace(ps);
<a solid in ProjectiveSpace(9, 49)>
```

5.2.13 Span

◇ Span(*u*, *v*) (operation)

◇ Span(*list*) (operation)

Returns: an element

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

Example

```
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)
```

5.2.14 Meet

◊ Meet(*u*, *v*) (operation)

Returns: an element

When *u* and *v* are elements of a projective or polar 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);
< trivial subspace >
gap> Display(s);
< trivial 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]);
< trivial subspace >
```

5.2.15 IsIncident

◊ IsIncident(*v*, *geo*) (operation)
◊ *(*v*, *geo*) (operation)
◊ \in(*v*, *geo*) (operation)

Returns: true or false

Implentation to be reconsidered. Incidence is only applicable on elements of an Incidence structure. in must be applicable on subspaces, hence also trivial one.

Example

```
gap> ps := ProjectiveSpace(5,9);
ProjectiveSpace(5, 9)
gap> p := Random(Points(ps));
<a point in ProjectiveSpace(5, 9)>
gap> r := Random(Solids(ps));
<a solid in ProjectiveSpace(5, 9)>
gap> IsIncident(p,r);
false
gap> IsIncident(r,p);
false
gap> p*r;
false
gap> r*p;
```

```
false
gap> p in r;
false
gap> r in p;
false
gap> EmptySubspace(ps) in r;
true
gap> r in ps;
true
```

5.2.16 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>( <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> flag := FlagOfIncidenceStructure(ps,[]);
<a flag of ProjectiveSpace(4, 5)>
```

5.2.17 IsEmptyFlag

◊ IsEmptyFlag(*flag*) (operation)

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

5.2.18 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)>, <a point in ProjectiveSpace(4, 3)>,
  <a point in ProjectiveSpace(4, 3)> ]
gap> shadowlines := ShadowOfElement(ps,plane,2);
<shadow lines in ProjectiveSpace(4, 3)>
```

```
gap> List(shadowlines);
[ <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)>, <a line in ProjectiveSpace(4, 3)>,
  <a line in ProjectiveSpace(4, 3)> ]
```

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 ShadowOfFlag

◇ ShadowOfFlag(*ps, list, i*) (operation)
◇ ShadowOfFlag(*ps, list, str*) (operation)

Returns: the shadow elements of type *i* in the flag determined by *list*, i.e. the elements of type *i* incident with all elements of the flag. The second variant determines the type *i* from the position of *str* in the list returned by `TypesOfElementsOfIncidenceStructurePlural`

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

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)>
```

5.3.4 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)> ]
```

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 is induced by a semi-linear 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 $n+1$-dimensional square 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 $\Gamma L(n+1,q)$. It is clear that 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* of $PG(n,q)$, and the group $PGL(n+1,q)$ is called the *projectivity 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 $PG(n,q)$.

Consider the projective space $PG(n,q)$. As described in Chapter 8, 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 $A_1 \cdot A_2^{\theta_2^{-1}}$ 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 correlation 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 *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 $x_0X_0+x_1X_1+\ldots+x_nX_n$. 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 $(x_0,x_1,\dots,x_n)$ to the projective hyperplane represented by row vector $(y_0,y_1,\dots,y_n) = ((x_0,x_1,\dots,x_n)A)^{\theta}$.

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 $PG(n,q)$ form a group, called the *correlation group* of $PG(n,q)$. The convention determines completely the action of correlations on all elements of a projective space, and it follows that the product of two elements of the correlation 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 group with underlying matrix $A_1 \cdot (A_2^{\theta_2^{-1}})_2^{\delta}$, underlying field automorphism $\theta_1\theta_2$, and underlying projective space automorphism $\delta_1\delta_2$.

Action functions for collineations and correlations on the subspaces of a projective space are described in detail in Section 6.6

6.1 Projectivities, collineations and correlations of projective spaces

In FinInG, different categories are created for projectivities, collineations and correlations of a projective space.

6.1.1 Categories for group elements

◊ `IsProjGrpEl` (Category)
◊ `IsProjGrpElWithFrob` (Category)
◊ `IsProjGrpElWithFrobWith` (Category)

`IsProjGrpEl` is a category in which elements of $PGL(n+1,q)$ can be constructed as objects in this category. So projectivities in the mathematical sense can be constructed as objects in this category. `IsProjGrpElWithFrob` is a category in which elements of $P\Gamma L(n+1,q)$ can be constructed. So collineations in the mathematical sense (and thus also projectivities) can be constructed as objects in this category.

6.1.2 Projectivity

◊ `Projectivity(mat, f)` (operation)

Returns: a projectivity of a projective space

mat must be a non-singular matrix over the finite field *f*. Creates an element of a projectivity group. The returned object belongs to `IsProjGrpEl`.

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));
<projective element [ [ 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 ] ]>
```

6.1.3 ProjectiveSemilinearMap

◊ ProjectiveSemilinearMap(*mat*, *frob*, *f*) (operation)
◊ ProjectiveSemilinearMap(*mat*, *f*) (operation)
◊ CollineationOfProjectiveSpace(*mat*, *frob*, *f*) (operation)
◊ CollineationOfProjectiveSpace(*mat*, *f*) (operation)

mat is a nonsingular matrix, *frob* is a field automorphism, and *f* is a field. This function (and its synonym) returns the collineation with matrix *mat* and automorphism *frob*. 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`

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));
<projective semilinear element: [ [ 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));
<projective semilinear element: [ [ 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>
```

6.1.4 StandardDualityOfProjectiveSpace

◊ StandardDualityOfProjectiveSpace(*ps*) (operation)

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

Example
```
gap> ps := ProjectiveSpace(4,5);
PG(4, 5)
gap> delta := StandardDualityOfProjectiveSpace(ps);
StandardDuality( Elements( ProjectiveSpace(4,GF(5)) ) )
gap> delta^2;
IdentityMapping( <Elements of PG(4, 5)> )
gap> p := VectorSpaceToElement(ps,[1,2,3,0,1]*Z(5)^0);
<a point in PG(4, 5)>
gap> h := p^delta;
```

```
<a solid in PG(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) ] ]
```

6.1.5 CorrelationOfProjectiveSpace

◊ `CorrelationOfProjectiveSpace(mat, f)` (operation)
◊ `CorrelationOfProjectiveSpace(mat, frob, f)` (operation)
◊ `CorrelationOfProjectiveSpace(mat, f, delta)` (operation)
◊ `CorrelationOfProjectiveSpace(mat, frob, f, 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` (6.1.4). 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( <Elements of PG(
3, 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( <Elements of PG(
3, 343)> ) >
gap> delta := StandardDualityOfProjectiveSpace(ProjectiveSpace(2,GF(7^3)));
StandardDuality( ElementsOfIncidenceStructure( 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( ElementsOfIncidenceStructure( 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( ElementsOfIncidenceStructure( ProjectiveSpace(2,GF(7^
3)) ) ) >
```

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

6.2.1 Representations for group elements

◊ `IsProjGrpElRep` (Representation)
◊ `IsProjGrpElWithFrobRep` (Representation)
◊ `IsProjGrpElWithFrobWithPSIsom` (Representation)

As we have seen, in FinInG, a projectivity is described by a matrix, a collineation of a projective space by a matrix and a field automorphism, and a correlation of a projective space by a matrix, a field automorphism and a isomorphism of the projective space that is either the standard duality or the identity mapping. Also the basefield is stored as a component in the representation.

6.2.2 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));
PGL(4,3)
gap> x:=Random(g);;
gap> UnderlyingMatrix(x);
[ [ 0*Z(3), Z(3), Z(3), Z(3)^0 ], [ Z(3)^0, 0*Z(3), Z(3)^0, Z(3)^0 ],
  [ 0*Z(3), Z(3)^0, Z(3), Z(3) ], [ Z(3)^0, Z(3)^0, 0*Z(3), Z(3)^0 ] ]
```

6.2.3 BaseField

◊ `BaseField(`*g*`)` (operation)

Returns: a field

g is a projectivity, collineation or correlation of a projective space. This function returns the matrix 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));
<projective element [ [ 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> BaseField(g);
GF(3^6)
```

6.2.4 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.

```
gap> g:=CollineationGroup( ProjectiveSpace(3,9));
PGammaL(4,9)
gap> x:=Random(g);;
gap> FieldAutomorphism(x);
IdentityMapping( GF(3^2) )
```

6.2.5 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*.

```
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 Collineation groups of projective or polar spaces

6.3.1 CollineationGroup

◊ CollineationGroup(*geom*) (operation)

Returns: a group of collineations of geom

geom must be an incidence geometry. If *geom* is a projective space, $PG(n,q)$, then this operation returns the full group of collineations $\Gamma L(n+1,q)$ of the projective space. If *geom* is a polar space, then this operation returns the full group of semi-similarities of *geom*.

Example
```
gap> p1 := ProjectiveSpace(3,3);
PG(3, 3)
gap> CollineationGroup(p1);
PGL(4,3)
gap> p2 := ProjectiveSpace(4,81);
PG(4, 81)
gap> CollineationGroup(p2);
PGammaL(5,81)
gap> p3 := EllipticQuadric(3,16);
Q-(3, 16)
gap> CollineationGroup(p3);
#I  Computing nice monomorphism...
PGammaO-(4,16)
gap> p4 := SymplecticSpace(3,9);
W(3, 9)
gap> CollineationGroup(p4);
#I  Computing nice monomorphism...
PGammaSp(4,9)
```

6.3.2 SimilarityGroup

◊ `SimilarityGroup(`*geom*`)` (operation)

Returns: a group of collineations of geom

geom must be a polar space. This operation returns the full group of similarities of *geom* (those collineations which preserve the form up to a scalar).

Example
```
gap> w := SymplecticSpace(5,3);
W(5, 3)
gap> SimilarityGroup(w);
#I  Computing nice monomorphism...
PGSp(6,3)
```

6.3.3 IsometryGroup

◊ `IsometryGroup(`*geom*`)` (operation)

Returns: a group of collineations of geom

geom must be a polar space. This operation returns the full group of isometries of *geom* (those collineations which preserve the form).

Example
```
gap> w := SymplecticSpace(3,8);
W(3, 8)
gap> IsometryGroup(w);
#I  Computing nice monomorphism...
PSp(4,8)
```

6.3.4 SpecialIsometryGroup

◇ `SpecialIsometryGroup(`*`geom`*`)` (operation)

Returns: a group of collineations of geom

geom must be a polar space. This operation returns those isometries of *geom* which have unit determinant.

Example
```
gap> hq := HyperbolicQuadric(5,3);
Q+(5, 3)
gap> SpecialIsometryGroup(hq);
#I  Computing nice monomorphism...
PSO(1,6,3)
```

6.4 Basic operations for projective groups

6.4.1 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.

6.4.2 Dimension

◇ `Dimension(`*`g`*`)` (operation)

Returns: a number

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

6.5 Collineation group as a subgroup of 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)));
<projective semilinear element: [ [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2^2) ],
  [ 0*Z(2), Z(2^2)^2, Z(2)^0, Z(2)^0 ],
  [ Z(2^2)^2, 0*Z(2), Z(2^2)^2, Z(2^2)^2 ],
  [ 0*Z(2), 0*Z(2), Z(2^2)^2, Z(2^2) ] ], F^2>
gap> y := Random(CorrelationGroup(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
```

```
[ [ 0*Z(2), Z(2^2)^2, Z(2)^0, 0*Z(2) ], [ Z(2)^0, Z(2^2)^2, Z(2)^0, Z(2)^0 ],
  [ Z(2)^0, Z(2^2)^2, Z(2)^0, Z(2^2) ], [ Z(2^2), 0*Z(2), Z(2)^0, Z(2^2) ]
 ], F^2, StandardDuality( AllElementsOfIncidenceStructure( ProjectiveSpace(
3,GF(2^2)) ) ) >
```

6.6 Projective group actions

Consider the projective space $PG(n,q)$. As described in Chapter 8, 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 $(x_0,x_1,\dots,x_n)$, and a collineation or correlation ϕ with underlying matrix A and field automorphism θ. Define the row vector $(y_0,y_1,\dots,y_n) = ((x_0,x_1,\dots,x_n)A)^\theta$. When ϕ is a collineation, then p^ϕ is the point with underlying row vector $(y_0,y_1,\dots,y_n)$, when ϕ is a correlation then is a hyperplane of $PG(n,q)$ with equation $y_0X_0+y_1X_1+\dots+y_nX_n$. The action of collineations or correlations on points determines the action on subspaces of arbitrary dimension completely.

6.6.1 OnProjSubspaces

◊ `OnProjSubspaces(subspace, el)` (function)

subspace is a subspace of a projective or polar space. *el* must be a projective semilinear map. This function return the image of *subspace* under *el*, which is a subspace of the same dimension.

Example
```
gap> ps := ProjectiveSpace(4,27);
PG(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 PG(4, 27)>
gap> Display(p);
[ Z(3)^0, Z(3^3)^14, Z(3^3)^5, Z(3^3)^7, Z(3^3)^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));
<projective element with Frobenius:
[ [ 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 PG(4, 27)>
gap> Display(r);
[ Z(3)^0, 0*Z(3), 0*Z(3), Z(3^3)^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 PG(4, 27)>
gap> r := OnProjSubspaces(s,phi);
<a plane in PG(4, 27)>
gap> Display(r);
z = Z(27)
    1    .    .  z^3
    .    1    . z^22
    .    .    1  z^3
```

6.6.2 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.

6.6.3 OnProjSubspacesReverseing

◊ `OnProjSubspacesReverseing(subspace, el)` (function)

subspace is a subspace of a projective or polar space. *el* must be an element of the correlation group of the ambient geometry of *subspace*. This function return the image of *subspace* under *el*, which is a subspace of the same dimension if *el* is a collineation.

Example

```
gap> ps := ProjectiveSpace(3,27);
PG(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( Elements( 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( Elements( ProjectiveSpace(3,GF(3^3)) ) ) >
gap> p := Random(Points(ps));
<a point in PG(3, 27)>
gap> OnProjSubspacesReversing(p,phi);
<a plane in PG(3, 27)>
```

```
gap> l := Random(Lines(ps));
<a line in PG(3, 27)>
gap> OnProjSubspacesReversing(p,phi);
<a plane in PG(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( <Elements of PG(4, 27)> ) >
gap> OnProjSubspacesReversing(p,psi);
<a point in PG(3, 27)>
gap> OnProjSubspacesReversing(l,psi);
<a line in PG(3, 27)>
```

6.7 Nice Monomorphisms

6.7.1 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`.

Example
```
gap> NiceMonomorphism(CollineationGroup(ProjectiveSpace(6,7)));
Error, action on projective points not feasible to calculate
 ...
gap> DESARGUES.LimitForCanComputeActionOnPoints := 500000;
500000
gap> NiceMonomorphism(CollineationGroup(ProjectiveSpace(6,7)));
<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		$A^T = A$, 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 6.6) 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 := HermitianVariety(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> c
alled 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;
```

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 zeor 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 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 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 $PG(n,q)$.

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 over GF(2^4)>
```

```
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);
<All elements of Q+(3, 3)>
gap> List(points);
[ <a point in Q+(3, 3)>, <a point in Q+(3, 3)>, <a point in Q+(3, 3)>,
  <a point in Q+(3, 3)>, <a point in Q+(3, 3)>, <a point in Q+(3, 3)>,
  <a point in Q+(3, 3)>, <a point in Q+(3, 3)>, <a point in Q+(3, 3)>,
  <a point in Q+(3, 3)>, <a point in Q+(3, 3)>, <a point in Q+(3, 3)>,
  <a point in Q+(3, 3)>, <a point in Q+(3, 3)>, <a point in Q+(3, 3)>,
  <a point in Q+(3, 3)> ]
```

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>( <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> 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 over GF(7)>
```

7.4 Commuting polarities

FinInG constructs packages as correlations. This allows polarities to be multiplies easily, resulting in is 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.

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 := CorrelationGroup(PG(3,25));
<projective group with Frobenius with proj. space isomorphism of size
37190822760000000000 with 4 generators>
gap> h := CollineationGroup(PG(3,25));
PGammaL(4,25)
gap> hom := Embedding(h,g);
MappingByFunction( PGammaL(4,25), <projective group with Frobenius with proj.
space isomorphism of size 37190822760000000000 with
4 generators>, function( y ) ... end )
gap> coll := PreImagesRepresentative(hom,phi*psi);
<projective semilinear element: [ [ 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>
```

$A^T = -A$. $A^\theta = A^T$.

Chapter 8

Classical Polar Spaces

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

8.1 Creating Polar Spaces

A *polar space* is a point-line incidence geometry, also satisfying few well known axioms. 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*. As in the previous case, the underlying vector space and matrix group are to our advantage. We refer the reader to [HT91] and [Cam00] for the necessary background theory (if it is not otherwise provided), and we follow the approach of [Cam00] 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. It is well known that the geometry consisting of the subspaces of $PG(n,q)$ whose underlying vector subspace is totally isotropic with relation to f, is a polar space P. 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 $V(n+1,q)$. It is well known that the geometry consisting of the subspaces of $PG(n,q)$ whose underlying vector subspace is totally singular with relation to q, is a polar space P. 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.

8.1.1 PolarSpace

◊ `PolarSpace(form)` (operation)

Returns: a classical polar space

form must be a bilinear, quadratic, or hermitian form created by use of the GAP package 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 of rank 3 over GF(5^2)>
gap> r := PolynomialRing(GF(32),4);
PolynomialRing(..., [ 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 of rank 3 over GF(2^5)>
```

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 of dimension 6 over GF(3)>
gap> ## The construction of the ovoid: ##
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 ) );;
gap> ForAll(planes, p -> Number(ovoid, x -> x in p) = 1);
true
```

8.2 Canonical 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-degenerate 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 is dependent of 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 is dependent of 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.

In the following table, we summerize the above information on polar spaces, together with the canonical forms that are chosen in FinInG.

polar space	canonical form	characteristic	projective dimensior
hermitian variety	$X_0^{q+1}+X_1^{q+1}+\ldots+X_n^{q+1}=0$	odd and even	odd and even
symplectic space	$X_0Y_1-Y_0X_1+\ldots+X_{n-1}Y_n-Y_{n-1}X_n=0$	odd and even	odd
hyperbolic quadric	$X_0X_1+\ldots+X_{n-1}X_n=0$	odd and even	odd
parabolic quadric	$X_0^2+X_1X_2+\ldots+X_{n-1}X_n=0$	odd and even	even
elliptic quadric	$\nu X_0^2+X_1^2+X_2X_3+\ldots+X_{n-1}X_n=0$, ν a non-square	odd	odd
elliptic quadric	$dX_0^2+X_0X_1+X_1^2+X_2X_3+\ldots+X_{n-1}X_n=0$, $Tr(d)=1$	even	odd

Table: finite classical polar spaces

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)
Non-degenerate symplectic form
Gram Matrix:
 . 1 . .
 1 . . .
 . . . 1
 . . 1 .
Witt Index: 2
```

8.2.2 HermitianVariety

◊ `HermitianVariety(d, F)` (operation)
◊ `HermitianVariety(d, q)` (operation)

Returns: a Hermitian variety

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

Example

```
gap> ps := HermitianVariety(2,25);
H(2, 5^2)
gap> Display(ps);
H(2, 25)
Hermitian form
Gram Matrix:
 1 . .
 . 1 .
 . . 1
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)
Non-degenerate parabolic bilinear form
Gram Matrix:
 1 . .
 . . 2
 . 2 .
Witt Index: 1
gap> ps := ParabolicQuadric(4,16);
Q(4, 16)
gap> Display(ps);
Q(4, 16)
Non-singular parabolic quadratic form
Gram Matrix:
 1 . . . .
 . . 1 . .
 . . . . .
 . . . . 1
 . . . . .
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)
Non-degenerate hyperbolic bilinear form
Gram Matrix:
 . 2 . . . .
 2 . . . . .
 . . . 2 . .
 . . 2 . . .
 . . . . . 2
 . . . . 2 .
Witt Index: 3
gap> ps := HyperbolicQuadric(3,4);
Q+(3, 4)
gap> Display(ps);
Q+(3, 4)
Non-singular hyperbolic quadratic form
Gram Matrix:
 . 1 . .
 . . . .
 . . . 1
 . . . .
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)
Non-degenerate elliptic bilinear form
Gram Matrix:
 1 . . .
 . 1 . .
 . . . 2
 . . 2 .
Witt Index: 1
gap> ps := EllipticQuadric(5,8);
Q-(5, 8)
gap> Display(ps);
Q-(5, 8)
Non-singular elliptic quadratic form
Gram Matrix:
 1 1 . . . .
 . 1 . . . .
 . . . 1 . .
 . . . . . .
 . . . . . 1
 . . . . . .
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 similar to the given polar space *P* of the classical polar space with underlying form *form*

8.3 Basic operations finite classical polar spaces

8.3.1 UnderlyingVectorSpace

◊ `UnderlyingVectorSpace(`*ps*`)` (operation)

Returns: a vector space

```
Example
```

Equality of projective spaces? Projective dimension of a projective space and of an element

8.3.2 ProjectiveDimension

◊ `ProjectiveDimension(`*`ps`*`)` (operation)
◊ `Dimension(`*`ps`*`)` (operation)
◊ `Rank(`*`ps`*`)` (operation)

Returns:

```
Example
```

8.3.3 StandardFrame

◊ `StandardFrame(`*`ps`*`)` (operation)

Returns:

```
Example
```

8.3.4 Coordinates

◊ `Coordinates(`*`v`*`)` (operation)

Returns:

```
Example
```

8.3.5 EquationOfHyperplane

◊ `EquationOfHyperplane(`*`ps`*`)` (operation)

Returns:

```
Example
```

8.3.6 BaseField

◊ `BaseField(`*`ps`*`)` (operation)

Returns:

```
Example
```

8.3.7 AsList

◊ `AsList(`*`subspaces`*`)` (operation)

Returns: an Orb object or list

```
Example
```

8.3.8 Random

◊ Random(*subspaces*) (operation)

Returns:

Example

```
```

8.3.9 RandomSubspace

◊ RandomSubspace(*pg, i*) (operation)

Returns:

Example

```
```

8.3.10 VectorSpaceToElement

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

Returns: an element

geo is a projective space or a polar space, such as `ProjectiveSpace(3,3)` or `EllipticQuadric(7,5)`, and *v* is either a row vector (for points) or an mxn matrix (for an $(m-1)$-subspace of a geometry of projective dimension $n-1$). 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 (note: by convention, the empty set is the unique subspace of projective dimension -1). When *geo* is a polar space, it is checked whether *v* determines a point or an element *geo*.

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);
[ Z(7), Z(7)^5, Z(7)^3, 0*Z(7), Z(7), Z(7)^2, Z(7) ]
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 vectorspace 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;
```

8.3.11 EmptySubspace

◊ EmptySubspace (global variable)

Returns: a GAP object

EmptySubspace is a GAP object which represents the ubiquitous trivial subspace of a projective or polar space, and is contained in every projective space.

Example
```
gap> EmptySubspace;
< trivial subspace >
gap> line := Random(Lines(PG(5,9)));
<a line in ProjectiveSpace(5, 9)>
gap> EmptySubspace * line;
true
gap> EmptySubspace * PG(3,11);
true
```

8.3.12 \in

◊ \in(*v*, *geo*) (operation)

Returns: true or false

v is an element of an incidence structure. It is checked whether *v* is a subspace of the polar space *geo*.

Example
```
gap> ps := ProjectiveSpace(5,9);
ProjectiveSpace(5, 9)
gap> p := Random(Points(ps));
<a point in ProjectiveSpace(5, 9)>
gap> r := Random(Solids(ps));
<a solid in ProjectiveSpace(5, 9)>
gap> IsIncident(p,r);
false
gap> IsIncident(r,p);
false
gap> p*r;
false
gap> r*p;
false
gap> p in r;
false
gap> r in p;
false
gap> EmptySubspace(ps) in r;
true
gap> r in ps;
true
```

8.3.13 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
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)
```

8.3.14 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.

8.3.15 IsCollinear

◊ `IsCollinear(ps, u, v)` (operation)

Returns: Boolean

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

8.3.16 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. (Put more here when Jan has implemented polarities).

Example
```
gap> pq := ParabolicQuadric(4,3);
Q(4, 3)
gap> perp := Polarity(pq);
function( v ) ... end
gap> lines := Lines(ps);
<lines of Q(4, 3)>
gap> l:=Random(lines);
<a line in Q(4, 3)>
gap> perp(l);
<a plane in PG(4, 3)>
```

8.3.17 AmbientSpace

◊ `AmbientSpace(ps)` (operation)

Returns: the ambient projective space

ps is a polar space. This function returns the ambient projective spce of *ps*.

8.3.18 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 := HermitianVariety(2, 3^2);
H(2, 3^2)
gap> h2 := HermitianVariety(3, 3^2);
H(3, 3^2)
gap> pg := AmbientSpace( h2 );
PG(3, 9)
gap> pi := VectorSpaceToElement( pg, [[1,0,0,0],[0,1,0,0],[0,0,1,0]] * Z(9)^0 );
<a plane in PG(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 PG(3, 9)>
gap> TypeOfSubspace(h2, pi);
"degenerate"
```

8.4 All the elements or just one at a time

In FinInG, 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. So too do 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.

8.4.1 Enumerators for polar spaces

If you are not familiar with “enumerators” in GAP, it is worthy to explain a little bit about them in this section. An `enumerator` is a particular object in GAP which allows the user to compute the i-th entry in a collection. Mathematically speaking, it is a bijection from the positive integers to the given collection. So for example, the rationals are totally ordered and there is an enumerator in GAP so that the user can access rational numbers one at a time:

Example
```
gap> enum:=Enumerator(Rationals);
<enumerator of Rationals>
gap> enum[10];
3/2
```

For more on enumerators, see the relevant section in the GAP manual.

8.4.2 Enumerator

◇ `Enumerator(`*`elements`*`)` (operation)

Returns: an enumerator

`elements` is a collection of elements, such as `Points( ParabolicQuadric( 4, 3 ))`. This function returns an enumerator for *`elements`*.

Example
```
gap> lines := Lines( ParabolicQuadric(6, 3) );
<lines of Q(6, 3)>
gap> enum := Enumerator( lines );
EnumeratorOfSubspacesOfClassicalPolarSpace( <lines of Q(6, 3)> )
gap> enum[10];
<a line in Q(6, 3)>
```

8.4.3 Iterators for projective spaces

An `iterator` is a particular object in GAP which allows the user to compute the NEXT entry in a collection. So iterators are a way to loop over the elements of a (countable) collection or a list, without repetition. The most important operations for the user are `NextIterator` and `IsDoneIterator`. We show an example of how this works with the rationals.

Example
```
gap> iter := Iterator( Rationals );
<iterator>
gap> x := NextIterator( iter );
0
gap> x := NextIterator( iter );
1
gap> x := NextIterator( iter );
-1
gap> x := NextIterator( iter );
1/2
gap> x := NextIterator( iter );
2
gap> IsDoneIterator( iter );
false
```

For more on iterators, see the relevant section in the GAP manual.

8.4.4 Iterator

◊ `Iterator(`*`elements`*`)` (operation)

Returns: an iterator

`elements` is a collection of elements, such as `Points( ProjectiveSpace( 4, 3 ))`. This function returns an iterator for *`elements`*.

Example
```
gap> lines := Lines( ProjectiveSpace(6, 3) );
<lines of PG(6, 3)>
gap> iter := Iterator(lines);
<iterator>
gap> x := NextIterator( iter );
<a line in PG(6, 3)>
gap> x := NextIterator( iter );
<a line in PG(6, 3)>
gap> Display( x );
 1 . . . . . .
 . 1 . . . . 1
gap> IsDoneIterator( iter );
false
```

Chapter 9

Affine Geometry

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

9.1 Overview

An affine space can be loosely described as the “geometry you get” when you remove a hyperplane from a projective space. Lines which were concurrent in a point of the subtracted hyperplane, are now *parallel*. Conversely, one can “complete” an affine space naturally to produce a projective space, by adding a *hyperplane at infinity*. In order to implement (Desarguesian) affine spaces in FinInG, we have to represent the subspaces in a standard fashion. The common representation is that of a coset of a vector subspace

$$v+S.$$

Hence one can think of an affine variety as consisting of: (i) a projective variety, and (ii) a “direction”. Thus in FinInG, we represent a variety of rank at least 2 by a pair

$$[v, mat]$$

where `v` is a row vector and `mat` is a matrix (representing a basis of the associated projective variety). For affine points, we simply use vectors. Here is a basic example of how we can work with affine spaces in FinInG.

Example
```
gap> ag := AffineSpace(3,3);
AG(3, 3)
gap> points := Points(ag);;
gap> x := Random(points);
<a point in AG(3, 3)>
gap> Display(x);
Affine point: [ Z(3), Z(3), 0*Z(3) ]
gap> planes := AsList( Planes(ag) );;
gap> p := Random(planes);
<a plane in AG(3, 3)>
gap> Display(p);
Affine plane:
Coset representative: [ 0*Z(3), 0*Z(3), 0*Z(3) ]
Coset (direction): [ [ Z(3)^0, 0*Z(3), Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3) ] ]
gap> g := CollineationGroup( ag );
AGL(3,3)
```

9.2 Construction of affine spaces and subspaces

9.2.1 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 (Desarguesian) affine space of dimension *d* over *F*. In the second form, *q* is a prime power specifying the size of the field. We have also installed the synonym `AG`, as this is the customary shorthand notation for an affine space.

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

9.2.2 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)>
```

9.3 Basic operations

9.3.1 Span

◊ `Span(u, v)` (operation)

Returns: a subspace

u and *v* are subspaces of an affine space. This function returns the join of the two subspaces, that is, 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 := Join(p, r);
<a line in AG(4, 5)>
gap> l^_;
[ [ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5) ], [ [ Z(5)^0, Z(5)^2, 0*Z(5), 0*Z(5) ] ] ]
gap> Display(l);
Affine line:
Coset representative: [ 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5) ]
Coset (direction): [ [ Z(5)^0, Z(5)^2, 0*Z(5), 0*Z(5) ] ]
```

9.3.2 Meet

◊ `Meet(u, v)` (operation)

Returns: a subspace

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 [ ].

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: [ Z(5)^0, Z(5)^0, 0*Z(5), 0*Z(5) ]
```

9.3.3 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)> ]
```

9.3.4 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( as ) );
<a point in AG(3, 3)>
gap> flag := [x, l];
[ <a point in AG(3, 3)>, <a line in AG(3, 3)> ]
gap> shadow := ShadowOfFlag( as, flag, 3 );
<shadow planes in AG(3, 3)>
gap> AsList(shadow);
[ <a plane in AG(3, 3)>, <a plane in AG(3, 3)>, <a plane in AG(3, 3)>,
  <a plane in AG(3, 3)> ]
```

9.3.5 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.

9.3.6 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)> ]
```

9.4 Iterators and enumerators

Recall from “All the subspaces or just one at a time” from Chapter 8, that an iterator allows us to obtain elements from a collection one at a time in sequence, whereas an enumerator for a collection give us a way of picking out the i-th element. In FinInG we have enumerators and iterators for subspace collections of affine spaces.

9.4.1 Iterator

◊ `Iterator(`*subs*`)` (operation)

Returns: an iterator for the given subspaces collection

subs is a collection of subspaces of an affine space, such as `Points( AffineSpace(3, 3) )`.

Example
```
gap> ag := AffineSpace(3, 3);
AG(3, 3)
gap> lines := Lines( ag );
<lines of AG(3, 3)>
gap> iter := Iterator( lines );
<iterator>
gap> l := NextIterator( iter );
<a line in AG(3, 3)>
```

9.4.2 Enumerator

◊ `Enumerator(`*subs*`)` (operation)

Returns: an enumerator for the 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.

9.4.3 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.

9.4.4 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.

9.4.5 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 $U+v$.

9.4.6 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.

9.5 Affine groups

A *collineation* of an affine space is a permutation of the points which preserves the relation of collinearity within the affine space. The fundamental theorem of affine geometry states that the 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 $\Gamma L(d,F)$. In FinInG, we represent the affine groups as projective semilinear transformations so that we can use all the functionality that exists for collineations of projective spaces. Suppose we have an affine transformation of the form $x+A$ where x is a vector representing a translation, and A is a matrix in $GL(d,q)$. Then by using the natural embedding of $AGL(d,q)$ in $PGL(d+1,q)$, we can write this collineation as a matrix: $\left(\begin{array}{c|c} \begin{matrix} A & 0 \end{matrix} & \begin{matrix} 0 \\ \\ 0 \end{matrix} \\ \hline x & 1 \end{array}\right)$ 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);
<projective element with Frobenius, underlying matrix:
 2 1 2 .
 1 2 2 .
 1 1 2 .
 1 2 2 1
, F^0>
```

Here we see that this affine transformation is

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

9.5.1 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)
```

9.5.2 CollineationGroup

◊ CollineationGroup(*as*) (operation)

Returns: a group

This operation returnes the affine semilinear group $A\Gamma L(V)$ acting on the affine space with underlying vector space V. The elements of this group are collineations of the associated projective space. Note that if the defining field has prime order, then $A\Gamma L(V) = AGL(V)$.

```
Example
gap> as := AffineSpace(4, 8);
AG(4, 8)
gap> g := CollineationGroup( as );
AGammaL(4,8)
```

9.5.3 OnAffineSpaces

◊ OnAffineSpaces(*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.

Chapter 10

Geometry Morphisms

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

10.1 Geometry morphisms in FinInG

A *geometry morphism* from `P` to `P'` is defined to be a map from the elements of `P` to the elements of `P'` which preserves incidence and induces a function from the type set of `P` to the type set of `P'`. 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.

10.1.1 IsGeometryMorphism

◊ `IsGeometryMorphism` (family)

The category `IsGeometryMorphism` represents a special object in FinInG which carries attributes and the given element map. The element map is given as a `IsGeneralMapping`, and so has a source and range.

Example
```
gap> ShowImpliedFilters(IsGeometryMorphism);
Implies:
   IsGeneralMapping
   IsTotal
   Tester(IsTotal)
   IsSingleValued
   Tester(IsSingleValued)
```

The usual operations of `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 `P` to `P'`, an intertwiner ϕ is a map from the automorphism group of `P` to the automorphism group of `P'`, such that for every element p of `P` and every automorphism g of `P`, we have

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

10.1.2 Intertwiner

◊ `Intertwiner(f)` (attribute)

Returns: a group homomorphism

The arguments `f` is a geometry morphism. If `f` comes equipped with a natural intertwiner from the the 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). Here is a simple example of the intertwiner for the isomorphism of two polar spaces (see `IsomorphismPolarSpaces` (10.1.3)).

Example
```
gap> form := BilinearFormByMatrix( IdentityMat(3,GF(3)), GF(3) );
< bilinear form >
gap> ps := PolarSpace(form);
<polar space of rank 2 over GF(3)>
gap> pq := ParabolicQuadric(2,3);
Q(2, 3)
gap> iso := IsomorphismPolarSpaces(ps, pq);
<geometry morphism from <Elements of <polar space of rank 2 over GF(
3)>> to <Elements of Q(2, 3)>>
gap> KnownAttributesOfObject(iso);
[ "Range", "Source", "Intertwiner" ]
gap> hom := Intertwiner(iso);;
```

10.1.3 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` (10.1.2)).

Example
```
gap> id := IdentityMat(6, GF(5));;
gap> form := BilinearFormByMatrix( id, GF(5) );
< bilinear form >
gap> ps := PolarSpace( form );
<polar space of rank 5 over GF(5)>
gap> PolarSpaceType( ps );
"hyperbolic"
```

```
gap> quadric := HyperbolicQuadric( 5, 5 );
Q+(5, 5)
gap> iso := IsomorphismPolarSpaces( ps, quadric );
<geometry morphism from <Elements of <polar space of rank 5 over GF(
5)>> to <Elements of Q+(5, 5)>>
gap> HasCollineationGroup( ps );
true
gap> hom := Intertwiner( iso );;
gap> ImagesSet(hom, SpecialIsometryGroup( ps ));
<projective group with Frobenius of size 14508000000 with 3 generators>
```

Both functions also have a ”no check” version. ◊ `IsomorphismPolarSpacesNC(`*`ps1, ps2`*`)` (operation)

◊ `IsomorphismPolarSpacesNC(`*`ps1, ps2, boolean`*`)` (operation)

Returns: a geometry morphism

10.2 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.

10.3 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.

10.3.1 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);
PG(2, 3)
```

```
gap> geom2 := ProjectiveSpace(3, 3);
PG(3, 3)
gap> planes := Planes(geom2);
<planes of PG(3, 3)>
gap> hyp := Random(planes);
<a plane in PG(3, 3)>
gap> em := NaturalEmbeddingBySubspace(geom1, geom2, hyp);
<geometry morphism from <Elements of PG(2, 3)> to <Elements of PG(3, 3)>>
gap> points := Points(geom1);
<points of PG(2, 3)>
gap> x := Random(points);
<a point in PG(2, 3)>
gap> x^em;
<a point in PG(3, 3)>
```

Another example, this time with polar spaces:

Example
```
gap> h1 := HermitianVariety(2, 3^2);
H(2, 3^2)
gap> h2 := HermitianVariety(3, 3^2);
H(3, 3^2)
gap> pg := AmbientSpace( h2 );
PG(3, 9)
gap> pi := VectorSpaceToElement( pg, [[1,0,0,0],[0,1,0,0],[0,0,1,0]] * Z(9)^0 );
<a plane in PG(3, 9)>
gap> em := NaturalEmbeddingBySubspace( h1, h2, pi );
<geometry morphism from <Elements of H(2, 3^2)> to <Elements of H(3, 3^2)>>
```

10.3.2 NaturalEmbeddingByFieldReduction

◇ `NaturalEmbeddingByFieldReduction(`*geom1*, *geom2*`)` (operation)
◇ `NaturalEmbeddingByFieldReduction(`*geom1*, *geom2*, *B*`)` (operation)
◇ `NaturalEmbeddingByFieldReduction(`*geom1*, *geom2*, *boolean*`)` (operation)

Returns: a geometry morphism

The arguments `geom1` and `geom2` are projective or polar spaces. This function returns a geometry morphism representing the natural embedding of `geom1` into `geom2` via field reduction. By *natural* for projective spaces, we mean that the embedding is induced by considering the field F_1 of `geom1` as a vector space over the field F_2 of `geom2`, perhaps with a choice of basis `B` in the case we have projective spaces. If `geom1` and `geom2` are polar spaces, then the only such possible embeddings are listed in the table below (see [Gil08]):

Polar Space 1	Polar Space 2	Conditions
$W(2n-1,q^a)$	$W(2na-1,q)$	–
$Q^+(2n-1,q^a)$	$Q^+(2na-1,q)$	–
$Q^-(2n-1,q^a)$	$Q^-(2na-1,q)$	–
$Q(2n,q^{2a})$	$Q^+(2(2n+1)a-1,q)$	q=1 mod 4
$Q(2n,q^{2a})$	$Q^-(2(2n+1)a-1,q)$	q=3 mod 4
$Q(2n,q^{2a+1})$	$Q((2a+1)(2n+1)-1,q)$	q odd
$H(n,q^{2a+1})$	$H((2a+1)(n+1)-1,q)$	q square
$H(2n,q^{2a})$	$Q^+(2a(2n+1)-1,q)$	q odd square
$H(2n-1,q^{2a})$	$Q^-(4an-1,q)$	q odd square
$H(n,q^{2a})$	$W(2n-1,q)$	–

Table: Field reduction of polar spaces

The geometry morphism also comes equipped with an intertwiner (see `Intertwiner` (10.1.2)). In the case polar spaces, 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 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 $PG(2,9)$ and maps them to the lines of $PG(5,3)$.

Example
```
gap> pg1 := ProjectiveSpace(2,9);
PG(2, 9)
gap> pg2 := ProjectiveSpace(5,3);
PG(5, 3)
gap> em := NaturalEmbeddingByFieldReduction(pg1, pg2);
<geometry morphism from <Elements of PG(2, 9)> to <Elements of PG(5, 3)>>
gap> line := Random( Lines(pg1) );
<a line in PG(2, 9)>
gap> solid := line ^ em;
<a solid in PG(5, 3)>
gap> l := em!.prefun(solid);
<a line in PG(2, 9)>
```

Suppose we have field reduction from a polar space P_1 to a polar space P_2, and suppose that they are both defined by sesquilinear forms. Let M be the Gram matrix for the sesquilinear form defining P_1 and let $\{b_1,..,b_m\}$ be a basis for the larger defining field of F P_1 over the smaller defining field K of P_2. Now the `BlownUpMat` command takes as input a matrix *mat* over F and returns the matrix of the linear transformation on the row space K^{mn} with respect to the K-basis whose vectors are $\{b_1v_1,...b_mv_1,...,b_mv_n\}$, where $\{v_1,...,v_n\}$ is a basis for F^n. Hence if we have a singular vector $x=(x_1,...,x_n)$ of P_1, then the blow-up of x will be singular if and only if for all $i,j \in \{1,...,m\}$, we have

$$\sum_{k=1}^{n} Coeff(x_k b_i) \cdot Coeff(M_k \cdot x b_j) = 0$$

where $Coeff$ is the map which takes the coefficients of an element of F with respect to $\{b_1,..,b_m\}$, and M_k is the k-th row of M.

In this example, we consider the image of the Hermitian variety $H(2,25)$ in $Q^-(5,5)$.

Example

```
gap> h := HermitianVariety(2, 5^2);
H(2, 5^2)
gap> quadric := EllipticQuadric(5, 5);
Q-(5, 5)
gap> em := NaturalEmbeddingByFieldReduction(h, quadric);
#I  Testing degeneracy of the *associated bilinear form*
<geometry morphism from <Elements of H(2, 5^2)> to <Elements of Q-(5, 5)>>
gap> points := AsList(Points(h));;
gap> image := ImagesSet(em, points);;
gap> image[1];
<a line in Q-(5, 5)>
gap> hom := Intertwiner( em );;
gap> g := Range( hom );
<projective group with Frobenius of size 378000 with 2 generators>
gap> OrbitLengths(g, image);
[ 126 ]
```

10.3.3 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).

10.3.4 BlownUpProjectiveSpaceBySubfield

◊ `BlownUpProjectiveSpaceBySubfield(`*`subfield, pg`*`)` (operation)

Returns: a projective space

Blows up a projective space *pg* with respect to the standard basis of the basefield of *pg* over the *subfield*.

10.3.5 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.

10.3.6 BlownUpSubspaceOfProjectiveSpaceBySubfield

◊ `BlownUpSubspaceOfProjectiveSpaceBySubfield(`*`subfield, subspace`*`)` (operation)

Returns: a subspace of a projective space

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

10.3.7 IsDesarguesianSpreadElement

◊ `IsDesarguesianSpreadElement(basis, subspace)` (operation)

Returns: true or false

Checks wether the *subspace* is a subspace which is obtained from a blowing up a projective point using field reduction with respect to *basis*.

10.3.8 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` (10.1.2)). The optional third argument *boolean* can take either `true` or `false` as input, and then our operation will or will not compute the intertwiner accordingly. The 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);
PG(2, 3)
gap> pg2 := ProjectiveSpace(2, 9);
PG(2, 9)
gap> em := NaturalEmbeddingBySubfield(pg1,pg2);
<geometry morphism from <Elements of PG(2, 3)> to <Elements of PG(2, 9)>>
gap> points := AsList(Points( pg1 ));
[ <a point in PG(2, 3)>, <a point in PG(2, 3)>, <a point in PG(2, 3)>,
  <a point in PG(2, 3)>, <a point in PG(2, 3)>, <a point in PG(2, 3)>,
  <a point in PG(2, 3)>, <a point in PG(2, 3)>, <a point in PG(2, 3)>,
  <a point in PG(2, 3)>, <a point in PG(2, 3)>, <a point in PG(2, 3)>,
  <a point in PG(2, 3)> ]
gap> image := ImagesSet(em, points);
[ <a point in PG(2, 9)>, <a point in PG(2, 9)>, <a point in PG(2, 9)>,
  <a point in PG(2, 9)>, <a point in PG(2, 9)>, <a point in PG(2, 9)>,
  <a point in PG(2, 9)>, <a point in PG(2, 9)>, <a point in PG(2, 9)>,
  <a point in PG(2, 9)>, <a point in PG(2, 9)>, <a point in PG(2, 9)>,
  <a point in PG(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 := HermitianVariety(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
```

10.3.9 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 of rank 3 over GF(3)>>>
gap> image := ImagesSet(proj, planes_on_x);
[ <a line in Q+(3, 3)>, <a line in Q+(3, 3)>, <a line in Q+(3, 3)>,
  <a line in Q+(3, 3)>, <a line in Q+(3, 3)>, <a line in Q+(3, 3)>,
  <a line in Q+(3, 3)>, <a line in Q+(3, 3)> ]
```

10.4 Some special kinds of geometry morphisms

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

10.4.1 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 );
#I  Finding base change...
#I  Computing nice monomorphism...
<geometry morphism from <lines of PG(3, 3)> to <points of Q+(5, 3)>>
gap> pg := ProjectiveSpace(3, 3);
PG(3, 3)
gap> l := Random( Lines(pg) );
<a line in PG(3, 3)>
gap> l^k;
<a point in Q+(5, 3)>
```

10.4.2 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));;
#I  Computing nice monomorphism...
gap> duality := NaturalDuality(w);
#I  Finding base change...
#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 PG(3, 5)>
```

10.4.3 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 <Elements of PG(3, 5)>>
gap> p := Random( Points(as) );
<a point in AG(3, 5)>
gap> p^map;
<a point in PG(3, 5)>
```

Chapter 11

Algebraic Varieties

In FinInG we provide some basic functionality for algebraic varieties defined over finite fields. The algebraic varieties in FinInG are defined by a list of multivariate polynomials over a finite field, and an ambient geometry. This ambient geometry is either a projective space, and then the algebraic variety is called a *projective variety*, or an affine geometry, and then the algebraic variety is called an *affine variety*. In this chapter we give a brief overview of the features of FinInG concerning these two types of algebraic varieties.

11.1 Projective Varieties

A *projective variety* in FinInG is an algebraic variety in a projective space defined by a list of homogeneous polynomials over a finite field.

11.1.1 ProjectiveVariety

◇ `ProjectiveVariety(`*`pg, pring, pollist`*`)` (operation)
◇ `ProjectiveVariety(`*`pg, pollist`*`)` (operation)
◇ `AlgebraicVariety(`*`pg, pring, pollist`*`)` (operation)
◇ `AlgebraicVariety(`*`pg, pollist`*`)` (operation)

Returns: a projective algebraic variety

The argument *`pg`* is a 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 homogeneous polynomials in *`pring`*.

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);
PG(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]);
```

```
Algebraic Variety V( [ x_1*x_3-x_2^2, x_1^2*x_4-x_4^3 ] ) in PG(3, 9)
```

11.2 Affine Varieties

An *affine variety* in FinInG is an algebraic variety in an affine space defined by a list of polynomials over a finite field.

11.2.1 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`*.

Example
```

```

Chapter 12

Generalised Polygons

A *generalised n-gon* is a point/line geometry whose incidence graph is bipartite of diameter n and girth $2n$. Although these rank 2 structures are very much a subdomain of Grape and Design, their significance in finite geometry warrants their inclusion in FinInG. By the famous theorem of Feit and Higman, a generalised n-gon which has at least three points on every line, must have n in $\{2,3,4,6,8\}$. The case $n = 2$ concerns the complete multipartite graphs, which we disregard. The more interesting cases are accordingly projective planes ($n = 3$), generalised quadrangles ($n = 4$), generalised hexagons ($n = 6$) and generalised octagons ($n = 8$).

12.1 Projective planes

12.1.1 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>
```

12.1.2 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, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0 ],
>    [ 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0 ],
>    [ 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1 ],
>    [ 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0 ],
>    [ 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0 ],
>    [ 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1 ],
>    [ 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0 ],
>    [ 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0 ] ];;
gap> pp := ProjectivePlaneByIncidenceMatrix( incmat );
<projective plane of order 4>
```

12.2 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 $AB \cap C = 1$. The standard text for generalised quadrangles is Payne and Thas [PT84].

12.2.1 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*. See the example for "EGQByKantorFamily".

12.2.2 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` (12.2.1) 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 );
true
gap> egq := EGQByKantorFamily(g, flist1, flist2);
<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*. These sets of matrices were introduced by Stanley Payne [Pay85] to construct Kantor families for flock generalised quadrangles.

12.2.3 IsqClan

◊ `IsqClan(`*`qclan, f`*`)` (operation)

Returns: true or false

This operation tests to see if *qclan* defines a q-Clan over the field *f*. See the example for `EGQByqClan` (12.2.4).

12.2.4 EGQByqClan

◊ `EGQByqClan(`*`qclan, f`*`)` (operation)

Returns: an elation generalised quadrangle

The argument *qclan* is a list of matrices (i.e., IsFFECollCollColl) which form a q-Clan, and *f* is the defining field. In the following example, we construct the classical generalised quadrangle of order (9, 3) (i.e., H(3,9)).

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> egq := EGQByqClan( clan, f );
<EGQ of order [ 9, 3 ] and basepoint 0>
```

12.2.5 KantorFamilyByqClan

◊ `KantorFamilyByqClan(qclan, f)` (operation)

Returns: a kantor family

The argument `qclan` is a list of matrices (i.e., IsFFECollCollColl) which form a q-Clan, and `f` is the defining field. The operation returns a triple `[g, list1, list2]` where `g` is a group of order s^2t, `list1` is a list of subgroups os order s and `list2` is a list of subgroups of order st.

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

12.2.6 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 $GF(q)$. See `EGQByBLTSet` (12.2.7) for an example of how to use this operation.

12.2.7 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 `ParabolicQuadric(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> f := GF(3);
GF(3)
gap> id := IdentityMat(2, f);;
gap> clan := List( f, t -> t * id );;
gap> bltset := BLTSetByqClan( clan, f );
[ <a point in Q(4, 3)>, <a point in Q(4, 3)>, <a point in Q(4, 3)>, <a point in Q(4, 3)> ]
gap> geo := AmbientGeometry( bltset[1] );
Q(4, 3)
gap> Display( geo );
Q(4, 3)
Non-degenerate parabolic bilinear form
```

```
Gram Matrix:
 . . . . 1
 . . . 1 .
 . . 1 . .
 . 1 . . .
 1 . . . .
Witt Index: 2
gap> egq := EGQByBLTSet( bltset );
<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) ]>
```

12.2.8 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*.

12.2.9 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*. (Note, some elation generalised quadrangles are known to have more than choice of base point; so we are assuming that *egq* has a particular choice for its base point.)

12.3 Generalised hexagons and octagons

Due to the sheer sizes of generalised octagons, they have not yet been included into FinInG, and there is (at the moment) limited functionality with the twisted triality hexagons. The only other known family of generalised hexagons (up to duality) are the Split Cayley hexagons.

12.3.1 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.

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

12.3.2 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` (12.3.1)), 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)`. The smallest Twisted Triality hexagon has 2457 points and 819 lines.

12.4 General attributes and operations of generalised polygons

12.4.1 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*.

12.4.2 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.

12.4.3 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` (12.4.4) for an example.

12.4.4 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 (3 gens, size 60480)
 | Z(3)
S (2 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> Print("Collineations of generalised hexagons...\n");
Collineations of generalised hexagons...
gap> hex := SplitCayleyHexagon( 5 );
<generalised hexagon of order [ 5, 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);
<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
```

12.4.5 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> f := GF(3);
GF(3)
gap> id := IdentityMat(2, f);;
gap> clan := List( f, t -> t*id );;
gap> egq := EGQByqClan( clan, f );
<EGQ of order [ 9, 3 ] and basepoint 0>
gap> HasElationGroup( egq );
true
gap> design := BlockDesignOfGeneralisedPolygon( egq );;
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)
```

12.4.6 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)
```

12.4.7 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.

Chapter 13

Coset Geometries and Diagrams

13.1 Preliminary version of the manual

Suppose we have a flag-transitive incidence geometry Γ. This means that a group G of automorphisms of Γ is also given such that G is transitive on the set of chambers of Γ. 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 oservation 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.

13.1.1 CosetGeometry

◊ `CosetGeometry(G, l)` (operation)

Returns: Returns the coset geometry 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 geometry* 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 ] ) )
```

13.1.2 IsIncident

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

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

`ele1` and `ele2` must be two elements in thes same coset geometry.

13.1.3 ParabolicSubgroups

◇ `ParabolicSubgroups(`*cg*`)` (operation)

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

13.1.4 AmbientGroup

◇ `AmbientGroup(`*cg*`)` (operation)

Returns: the group of the coset geometry *cg*

cg must be a coset geometry.

13.1.5 DiagramOfGeometry

◇ `DiagramOfGeometry(`*Gamma*`)` (operation)

Returns: The diagram of the geometry *Gamma*

Gamma must be a coset geometry.

13.1.6 DrawDiagram

◇ `DrawDiagram(`*Diag, filename*`)` (operation)

Returns: Does not return anything but wirtes 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.

13.1.7 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 untersection of all parabolic subgrops.

13.1.8 IsFlagTransitiveGeometry

◇ `IsFlagTransitiveGeometry(`*cg*`)` (operation)

Returns: true if and only if the group *G* defining *cg* acts flag-transitively.

cg must be a coset geometry.

13.1.9 IsFirmGeometry

◇ `IsFirmGeometry(`*cg*`)` (operation)

Returns: true if and only if *cg* is firm.

An incidence geometry is said to be firm if every nonmaximal flag is contained in at least two chambers. *cg* must be a coset geometry.

13.1.10 IsConnected

◊ `IsConnected(cg)` (operation)

Returns: true if and only if *cg* is connected.

A geometry is connected if and only if its incidence graph is connected. *cg* must be a coset geometry.

13.1.11 IsResiduallyConnected

◊ `IsResiduallyConnected(cg)` (operation)

Returns: true if and only if *cg* is residually connected.

A geometry is residually connected the incidence graphs of all its residues of rank at least 2 are connected. *cg* must be a coset geometry.

13.1.12 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. Maybe this function should be called StandardChamberOfCosetGeometry.

13.1.13 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*. The order of the elements of the chamber should be the same as the order of the parabolics defining *cg*.

13.1.14 CanonicalResidueOfFlag

◊ `CanonicalResidueOfFlag(cg, fl)` (operation)

Returns: coset geometry isomorphic to residue of *fl* in *cg*

cg must be a coset geometry and *fl* must be a flag in that geometry. The returned coset geometry for a flag {giGi} is ...

13.1.15 ResidueOfFlag

◊ `ResidueOfFlag(cg, fl)` (operation)

Returns: The residue of *fl* in *cg*.

CHECK the back-mapping.

13.1.16 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!

13.1.17 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.

References

[BLT90] Laura Bader, Guglielmo Lunardon, and Joseph A. Thas. Derivation of flocks of quadratic cones. *Forum Math.*, 2(2):163–174, 1990. 97

[Cam00] Peter J. Cameron. *Projective and Polar Spaces*. Online notes, `http://www.maths.qmul.ac.uk/~pjc/pps/`, 2000. 61

[Gil08] Nick Gill. Polar spaces and embeddings of classical groups. *to appear in New Zealand J. Math.*, 2008. 85

[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. 29, 53, 61

[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. 53, 62, 88

[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. 96

[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. 95

Index