PG

A share package for GAP

by

Jan De Beule

Patrick Govaerts

under supervision of

Leo Storme

Department of Pure Mathematics and Computer Algebra
University of Ghent

Contents

1 PG: Projective Geometries

1.1 Introduction

PG , Projective Geometries is a share package for GAP. It is completely written in GAP-language, so it is available for every GAP user. Its aim is to allow to work with finite projective geometries. At present, basic concepts have been implemented such as points, hyperplanes, subspaces and collineations. Furthermore we have implemented quadrics and Hermitian varieties together with a wide variety of functions dealing with these objects.

The implementation of the projective points is similar to the implementation of the hyperplanes. They both have a "vector" representation, namely a list with coordinates. For the projective points this is the usual vector representation. A hyperplane defined by the equation $b_0X_0 + b_1X_1 + \ldots + b_nX_n = 0$ is represented by the vector $[b_0, b_1, \ldots, b_n]$. Although the user can enter a projective point or a hyperplane by a vector that is not normalised, GAP will always return a normalised vector in the sense that the first non-zero element is one.

In order to make certain operations with projective geometry more efficient, such as describing an incidence structure or creating graphs from projective geometries, projective points and hyperplanes are also numbered, bu using a cyclic Singer group on PG(n,q). A number between 1 and $\frac{q^{n+1}-1}{q-1}$ is assigned to each projective point and each hyperplane from PG(n,q). The use of a cyclic Singer group allows to uses the corresponding difference sets of the projective spaces [Hir98].

Subspaces of a PG(n, q) can be constructed in two different ways. The user can construct a subspace as the intersection of some hyperplanes stored in a given list, or a subspace spanned by some projective points, also stored in a list. This mechanism results in subspaces that can belong to different representations, although the user should not worry about this. All defined operations for subspaces are allowed for each representation. The main result is an easy construction of subspaces, while the system handles the conversion between the representations automatically. Of course some basic operations are defined for subspaces, such as computing all the projective points.

A collineation of PΓL($n+1$, q) is uniquely defined by a non-singular $(n+1) \times (n+1)$ matrix over GF(q) and an automorphism of the field GF(q). The user can easily construct a collineation. Also implemented are images of most of the above described objects under collineations. Furthermore there are some functions about the whole group PΓL($n+1$, q).

A quadric in PG(n, q) is a set of projective points whose coordinates satisfy a homogeneous equation of degree two over GF(q) in at most $n+1$ different variables. Quadrics are implemented in that way, although there are two basic possibilities for the user to construct quadrics. One can use matrices or polynomials to construct quadrics. For all quadrics we implemented a wide set of functions. We did the same for Hermitian varieties. For our definition of Hermitian varieties and for more information, we refer to the corresponding chapter.

1.2 Installation and Usage

Installation of PG is like installation of a standard share package for GAP. Create a subdirectory `pkg` in your homedirectory. Place the file `pg.zoo` in the newly created subdirectory and unpack with the command

```
you@unix:~ > unzoo -x pg.zoo [enter]
```

Start GAP4 and issue the following command

```
gap> GAP_ROOT_PATHS
```

GAP will give you its homedirectory, e.g. `["/usr/local/src/gap4r1"]`.

When you want to make use of PG start GAP from your homedirectory with the option

`-l "./;/usr/local/src/gap4r1"`.

Now issue the following command

```
gap> RequirePackage("pg");
```

All functionality from PG should now be available.

1.3 Notations

We assume that the user is a bit familiar in using GAP. When we are talking about operations, attributes, objects, properties, ..., we assume the user to know the meaning of this. In order to make that possible our manual follows the same conventions and notations of the GAP manual. We repeat one important convention here.

1 ▶ `Oper(` *arg1*`,` *arg2*`[,` *opt*`] )` F

stands for the command `Oper` that takes two arguments *arg1* and *arg2* and an optional third argument *opt*. `F` stands for function. Other possible characters are `A`, `P`, `O`, `C` en `R` standing for respectively "Attribute", "Property", "Operation", "Category" and "Representation". For the meaning of these concepts we refer to the GAP4 Reference Manual.

2 Basic Functions and objects

In this chapter we will deal with a wide variety of functions about projective geometries, projective points and hyperplanes. We will discuss various aspects of all the new objects that can be created.

The implementation of the projective points is similar to the implementation of the hyperplanes. They both have a "vector"-representation, namely a list with coordinates. For the projective points this is the usual vector-representation. A hyperplane defined by the equation $b_0X_0 + b_1X_1 + \ldots + b_nX_n = 0$, is represented by the vector $[b_0, b_1, \ldots, b_n]$.

2.1 Projective Geometries

1 ▶ `ProjectiveGeometry( n, q )` F

Given a positive integer n, $n \geq 1$, and a prime power q, the function `ProjectiveGeometry` returns the projective geometry PG(n,q). Note that within a GAP-session, a projective geometry of given dimension n and order q is only constructed once. Constructed projective geometries are stored in a global variable. Asking a second time for a particular projective geometry retruns the already constructed projective geometry. The list of already constructed geometries is stored in the variable `ListOfPGs`.

```
gap> P := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> Q := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> ListOfPGs;
[ , [ ,,,, PG( 2, 5 ) ] ]
gap> IsIdenticalObj(P,Q);
true
gap> R := ProjectiveGeometry(5,4);
PG( 5, 4 )
gap> ListOfPGs;
[ , [ ,,,, PG( 2, 5 ) ],,, [ ,,, PG( 5, 4 ) ] ]
```

2 ▶ `FieldOfPG( P )` A

When P is the projective geometry PG(n,q), `FieldOfPG( P )` returns the finite field GF(q).

3 ▶ `Size( P )` A

`Size( P )`, with P the projective geometry PG(n,q) returns the size of P, i.e. the number of points in PG(n,q).

```
gap> FieldOfPG(ProjectiveGeometry(3,25));
GF(5^2)
gap> P := ProjectiveGeometry(6,3);
PG( 6, 3 )
gap> FieldOfPG(P);
GF(3)
gap> Size(P);
1093
gap> Size(ProjectiveGeometry(4,9));
7381
```

4 ▶ `Polynomial( P )` A

Given a projective geometry P, `Polynomial( P )` computes a primitive polynomial *pol* of degree $n+1$ over GF(q), when P is the projective geometry PG(n,q). It is returned as a list. The i-th position of this list contains the coefficient of X^{i-1} in *pol*, $i = 1, 2, \ldots, n+2$. This function is used in the computation of all projective points (see below).

```
gap> P := ProjectiveGeometry(6,27);
PG( 6, 27 )
gap> Polynomial(P);
[ Z(3^3)^22, Z(3^3)^17, Z(3^3)^20, Z(3)^0, Z(3^3), Z(3^3)^20, Z(3^3)^14,
   Z(3)^0 ]
```

2.2 Constructing projective points

1 ▶ `ProjectivePoints( P )` A

This attribute returns a list containing all the projective points af the projective geometry P. Example: compute all the points of the Fano plane (i.e. PG$(2,2)$). Control the returned number of points with `Size`

```
gap> P := ProjectiveGeometry(2,2);
PG( 2, 2 )
gap> ProjectivePoints(P);
[ [ 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, 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, Z(2)^0 ], [ Z(2)^0, Z(2)^0, Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), Z(2)^0 ] ]
gap> Print(last,"\n");
[ ProjectivePoint( PG( 2, 2 ), [ Z(2)^0, 0*Z(2), 0*Z(2) ] ),
  ProjectivePoint( PG( 2, 2 ), [ 0*Z(2), Z(2)^0, 0*Z(2) ] ),
  ProjectivePoint( PG( 2, 2 ), [ 0*Z(2), 0*Z(2), Z(2)^0 ] ),
  ProjectivePoint( PG( 2, 2 ), [ Z(2)^0, Z(2)^0, 0*Z(2) ] ),
  ProjectivePoint( PG( 2, 2 ), [ 0*Z(2), Z(2)^0, Z(2)^0 ] ),
  ProjectivePoint( PG( 2, 2 ), [ Z(2)^0, Z(2)^0, Z(2)^0 ] ),
  ProjectivePoint( PG( 2, 2 ), [ Z(2)^0, 0*Z(2), Z(2)^0 ] ) ]
gap> Size(P);
7
```

It is important to know that this function is rather time consuming, so never use this function for "big" geometries, unless it is really necessary. We give a comparative example:

```
gap> P := ProjectiveGeometry(2,2);
PG( 2, 2 )
gap> Size(P);
7
gap> ProjectivePoints(P);;
gap> time;
210
gap> Q := ProjectiveGeometry(5,8);
PG( 5, 8 )
gap> Size(Q);
37449
gap> ProjectivePoints(Q);;
gap> time;
11060
gap> ProjectivePoints(Q);;
gap> time;
0
```

In this example the projective geometry Q is indeed big. We will discuss a faster way of computing all projective points using subspaces. As one can see, invoking the attribute `ProjectivePoints` a second time takes no CPU-time, because we installed `ProjectivePoints` as an attribute and a projective geometry as an `IsAttributeStoringRep`-object. We should also mention that CPU-times will vary depending on the system on which GAP is running. Nevertheless this way of constructing all the projective points of a projective geometry has the advantage that, due to the algorithm we used, we already have an oredering on all the projective points. This ordering is useful when assigning an integer to each point. Once we have computed all projective points, we assign an integer to each point, namely the position of the point in the returned list. (for functions that deal with this order, see below).

2 ▶ `ProjectivePoint(` *P*, *vect* `)` O

This operation constructs the projective point in the projective geometry P with coordinate vector *vect*. Controls about the argument *vect* are built in, so if $P = \mathrm{PG}(n,q)$ then *vect* should be a non-zero vector of the $(n+1)$-dimensional vector space over $\mathrm{GF}(q)$. The coordinate vector of the returned point will be normalised, i.e. the first non-zero component will be equal to `One(` *field* `)` were *field* is the finite field $\mathrm{GF}(q)$.

```
gap> P := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> a := [1,0,0]*Z(5);
[ Z(5), 0*Z(5), 0*Z(5) ]
gap> p := ProjectivePoint(P,a);
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> Print(p,"\n");
ProjectivePoint( PG( 2, 5 ), [ Z(5)^0, 0*Z(5), 0*Z(5) ] )
gap> q := ProjectivePoint(ProjectiveGeometry(5,4),[1,0,1,0,1,0]*Z(4));
[ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ]
gap> Print(q,"\n");
ProjectivePoint( PG( 5, 4 ), [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0,
0*Z(2) ] )
```

2.3 Constructing hyperplanes

1 ▶ `Hyperplanes( P )` A

This attribute returns a list containing all hyperplanes of the projective geometry P.

```
gap> P := ProjectiveGeometry(2,2);
PG( 2, 2 )
gap> Hyperplanes(P);
[ [ 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ Z(2)^0, Z(2)^0, Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, Z(2)^0 ], [ Z(2)^0, 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, 0*Z(2) ] ]
gap> Print(last,"\n");
[ Hyperplane( PG( 2, 2 ), [ 0*Z(2), 0*Z(2), Z(2)^0 ] ),
  Hyperplane( PG( 2, 2 ), [ Z(2)^0, 0*Z(2), 0*Z(2) ] ),
  Hyperplane( PG( 2, 2 ), [ Z(2)^0, Z(2)^0, 0*Z(2) ] ),
  Hyperplane( PG( 2, 2 ), [ Z(2)^0, Z(2)^0, Z(2)^0 ] ),
  Hyperplane( PG( 2, 2 ), [ 0*Z(2), Z(2)^0, Z(2)^0 ] ),
  Hyperplane( PG( 2, 2 ), [ Z(2)^0, 0*Z(2), Z(2)^0 ] ),
  Hyperplane( PG( 2, 2 ), [ 0*Z(2), Z(2)^0, 0*Z(2) ] ) ]
```

Because this attribute uses the same algorithm as the attribute `ProjectivePoints`, it is also time consuming for "big" geometries, but it has the same advantages as described for the projective points.

2 ▶ `Hyperplane( P, vect )` O

This operation constructs the hyperplane in the projective geometry P with coordinate vector *vect*. Controls about the argument *vect* are built in, so if $P = \mathrm{PG}(n,q)$ then *vect* should be a non-zero vector of the $(n+1)$-dimensional vector space over $\mathrm{GF}(q)$. The coordinate vector of the returned hyperplane will be normalised, i.e. the first non-zero component will be equal to `One( field )` were *field* is the finite field $\mathrm{GF}(q)$.

```
gap> P := ProjectiveGeometry(4,9);
PG( 4, 9 )
gap> a := [1,0,1,1,1]*Z(9);
[ Z(3^2), 0*Z(3), Z(3^2), Z(3^2), Z(3^2) ]
gap> h := Hyperplane(P,a);
[ Z(3)^0, 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0 ]
gap> Print(h,"\n");
Hyperplane( PG( 4, 9 ), [ Z(3)^0, 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0 ] )
gap> k := Hyperplane(ProjectiveGeometry(5,7),[1,1,1,1,1,1]*Z(7));
[ Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0 ]
gap> Print(k,"\n");
Hyperplane( PG( 5, 7 ), [ Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0, Z(7)^0 ] )
```

2.4 Basic functions and operations for projective points

1 ▶ p `=` q
▶ `\=(` p, q `)` O

Equality of two projective points can be tested with the operation `\=`.

```
gap> P := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> p := ProjectivePoint(P,[1,0,0]*Z(5));
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> q := ProjectivePoint(P,[1,0,0]*Z(5));
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> p=q;
true
gap> \=(p,q);
true
```

2 ► `FieldOfPG(` *p* `)` A

returns the field GF(q) when p is a projective point from PG(n, q).

3 ► `PGOfSubspace(` *p* `)` A

if p is a projective point from the projective geometry PG(n,q), then `PGOfSubspace` returns PG(n,q).

```
gap> P := ProjectiveGeometry(6,3);
PG( 6, 3 )
gap> p := ProjectivePoint(P,[1,1,1,1,1,1,1]*Z(3));
[ Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ]
gap> FieldOfPG(p);
GF(3)
gap> PGOfSubspace(p);
PG( 6, 3 )
```

4 ► *p*`^`*i*
► `\^(`*p*`,`*i*`)` O

The infix operator `^` is defined as follows for a projective point p and an integer i. If $p = [x_0, x_1, \ldots, x_n]$ then $p^i = [x_0^i, x_1^i, \ldots, x_n^i]$. This can be useful when you want to compute quickly the image of a projective point by the collineation defined by a Frobenius automorphism of GF(q).

```
gap> r := ProjectivePoint(ProjectiveGeometry(2,4),[Z(4),Z(4)^3,Z(4)^0]);
[ Z(2)^0, Z(2^2)^2, Z(2^2)^2 ]
gap> r^2;
[ Z(2)^0, Z(2^2), Z(2^2) ]
```

5 ► *p* `+` *q*
► `\+(` *p*`,` *q* `)` O

The sum of two distinct binary projective points can be computed with the operation `\+`

```
gap> P := ProjectiveGeometry(2,2);
PG( 2, 2 )
gap> p := ProjectivePoint(P,[1,0,0]*Z(2));
[ Z(2)^0, 0*Z(2), 0*Z(2) ]
gap> q := ProjectivePoint(P,[0,1,1]*Z(2));
[ 0*Z(2), Z(2)^0, Z(2)^0 ]
gap> p+q;
[ Z(2)^0, Z(2)^0, Z(2)^0 ]
gap> \+(p,q);
[ Z(2)^0, Z(2)^0, Z(2)^0 ]
```

6 ► `PointToNumber(` *arg* `)` A

When *arg* is a projective point belonging to the projective geometry P, this function returns the number of p, i.e., as mentioned above, the position of p in the list `ProjectivePoints(`P`)`. When *arg* is a list of

projective points, this function returns a list with all the numbers of the corresponding projective points in *arg*.

```
gap> P := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> p := ProjectivePoint(P,Z(5)*[1,0,0]);
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> q := ProjectivePoint(P,Z(5)*[0,1,1]);
[ 0*Z(5), Z(5)^0, Z(5)^0 ]
gap> PointToNumber(p);
1
gap> PointToNumber([p,q]);
[ 1, 5 ]
```

The use of this function causes the computation of all projective points, which can be time consuming.

7 ► `NumberToPoint( P, arg )` O

The argument P of this function should be a projective geometry PG(n,q), while *arg* should be an integer i or a list of integers $[i_1, \ldots, i_k]$, where $i, i_1, \ldots, i_k \in \{1, \ldots, (q^{n+1}-1)/(q-1)\}$. If *arg* is an integer i then NumberToPoint returns the projective point with number i from P. If it is a list $[i_1, \ldots, i_k]$, then `NumberToPoint` returns a list of length k, whose j^{th} position contains the i_j^{th} point of P.

```
gap> P := ProjectiveGeometry(3,3);
PG( 3, 3 )
gap> NumberToPoint(P,2);
[ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ]
gap> NumberToPoint(P,[2,7,9]);
[ [ 0*Z(3), Z(3)^0, 0*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*Z(3) ] ]
gap> Print(last,"\n");
[ ProjectivePoint( PG( 3, 3 ), [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ] ),
  ProjectivePoint( PG( 3, 3 ), [ Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ] ),
  ProjectivePoint( PG( 3, 3 ), [ Z(3)^0, Z(3), Z(3), 0*Z(3) ] ) ]
```

The use of this function causes the computation of all projective points, so you should never use this function unless strictly needed.

8 ► `PointToVector( arg )` A

If *arg* is a projective point, then this function returns the coordinate vector of *arg*. If *arg* is a list of projective points from the same projective geometry, then `PointToVector` returns a list that contains the coordinate vectors of the corresponding projective points in *arg*.

```
gap> P := ProjectiveGeometry(2,25);
PG( 2, 25 )
gap> p := ProjectivePoint(P,[1,0,0]*Z(25));
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> q := ProjectivePoint(P,[1,1,1]*Z(25));
[ Z(5)^0, Z(5)^0, Z(5)^0 ]
gap> PointToVector(p);
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> PointToVector([p,q]);
[ [ Z(5)^0, 0*Z(5), 0*Z(5) ], [ Z(5)^0, Z(5)^0, Z(5)^0 ] ]
```

9 ► `Line( p, q )` O

Given two projective points p and q from the same projective geometry, this function returns a list containing all the points on the line through the points p and q

```
gap> P := ProjectiveGeometry(3,16);
PG( 3, 16 )
gap> p := ProjectivePoint(P,[1,0,0,0]*Z(16));
[ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ]
gap> q := ProjectivePoint(P,[0,1,1,0]*Z(16));
[ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> Line(p,q);
[ [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ], [ Z(2)^0, Z(2^4)^3, Z(2^4)^3, 0*Z(2) ],
  [ Z(2)^0, Z(2^4)^2, Z(2^4)^2, 0*Z(2) ],
  [ Z(2)^0, Z(2^4)^6, Z(2^4)^6, 0*Z(2) ], [ Z(2)^0, Z(2^4), Z(2^4), 0*Z(2) ],
  [ Z(2)^0, Z(2^4)^9, Z(2^4)^9, 0*Z(2) ], [ Z(2)^0, Z(2^2), Z(2^2), 0*Z(2) ],
  [ Z(2)^0, Z(2^4)^11, Z(2^4)^11, 0*Z(2) ], [ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
    , [ Z(2)^0, Z(2^4)^14, Z(2^4)^14, 0*Z(2) ],
  [ Z(2)^0, Z(2^4)^8, Z(2^4)^8, 0*Z(2) ],
  [ Z(2)^0, Z(2^4)^13, Z(2^4)^13, 0*Z(2) ],
  [ Z(2)^0, Z(2^4)^4, Z(2^4)^4, 0*Z(2) ],
  [ Z(2)^0, Z(2^4)^7, Z(2^4)^7, 0*Z(2) ],
  [ Z(2)^0, Z(2^2)^2, Z(2^2)^2, 0*Z(2) ],
  [ Z(2)^0, Z(2^4)^12, Z(2^4)^12, 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ] ]
gap> Length(last);
17
```

2.5 Basic functions and operations for hyperplanes

1 ► h `=` k
► `\=(` h, k `)` O

Equality of two hyperplanes can be tested with the operation `\=`.

```
gap> P := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> p := Hyperplane(P,[1,0,0]*Z(5));
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> q := Hyperplane(P,[1,0,0]*Z(5));
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> p=q;
true
gap> \=(p,q);
true
```

2 ► `FieldOfPG(` h `)` A

returns the field GF(q) when h is a hyperplane from PG(n, q).

3 ► `PGOfSubspace(` h `)` A

if h is a hyperplane from the projective geometry PG(n,q), then `PGOfSubspace` returns PG(n,q).

```
gap> P := ProjectiveGeometry(6,3);
PG( 6, 3 )
gap> h := Hyperplane(P,[1,1,1,1,1,1,1]*Z(3));
[ Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ]
gap> FieldOfPG(h);
GF(3)
gap> PGOfSubspace(h);
PG( 6, 3 )
```

4► `Size( h )` A

`Size` returns the number of points in the hyperplane h.

```
gap> P := ProjectiveGeometry(6,3);
PG( 6, 3 )
gap> h := Hyperplane(P,[1,1,1,1,1,1,1]*Z(3));
[ Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ]
gap> Size(h);
364
```

5► h`^`i
► `\^(`h`,`i`)` O

The infix operator `^` is defined as follows for a hyperplane h and an integer i. If $h = [x_0, x_1, \ldots, x_n]$ then $h^i = [x_0^i, x_1^i, \ldots, x_n^i]$. This can be useful when you want to compute quickly the image of a hyperplane by the collineation defined by a Frobenius automorphism of GF(q).

```
gap> r := Hyperplane(ProjectiveGeometry(2,4),[Z(4),Z(4)^3,Z(4)^0]);
[ Z(2)^0, Z(2^2)^2, Z(2^2)^2 ]
gap> r^2;
[ Z(2)^0, Z(2^2), Z(2^2) ]
```

6► `HyperplaneToNumber( arg )` A

When *arg* is a hyperplane belonging to the projective geometry P, this function returns the number of *arg*, i.e., as mentioned above, the position of *arg* in the list `Hyperplanes(`P`)`. When *arg* is a list of hyperplanes, this function returns a list with all the numbers of the corresponding hyperplanes in *arg*.

```
gap> P := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> h := Hyperplane(P,Z(5)*[1,0,0]);
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> k := Hyperplane(P,Z(5)*[0,1,1]);
[ 0*Z(5), Z(5)^0, Z(5)^0 ]
gap> HyperplaneToNumber(h);
2
gap> HyperplaneToNumber([h,k]);
[ 2, 29 ]
```

The use of this function causes the computation of all hyperplanes, which can be time consuming.

7► `NumberToHyperplane( P, arg )` O

The argument P of this function should be a projective geometry PG(n,q), while *arg* should be an integer i or a list of integers $[i_1, \ldots, i_k]$, where $i, i_1, \ldots, i_k \in \{1, \ldots, (q^{n+1}-1)/(q-1)\}$. If *arg* is an integer i, then `NumberToHyperplane` returns the hyperplane with number i from P. If it is a list $[i_1, \ldots, i_k]$, then `NumberToPoint` returns a list of length k, whose j^{th} position contains the i_j^{th} point of P.

```
gap> P := ProjectiveGeometry(3,3);
PG( 3, 3 )
gap> NumberToHyperplane(P,2);
[ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ]
gap> NumberToHyperplane(P,[2,7,9]);
[ [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, Z(3), 0*Z(3) ],
  [ Z(3)^0, Z(3)^0, 0*Z(3), Z(3)^0 ] ]
gap> Print(last,"\n");
[ Hyperplane( PG( 3, 3 ), [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ] ),
  Hyperplane( PG( 3, 3 ), [ 0*Z(3), Z(3)^0, Z(3), 0*Z(3) ] ),
  Hyperplane( PG( 3, 3 ), [ Z(3)^0, Z(3)^0, 0*Z(3), Z(3)^0 ] ) ]
```

The use of this function causes the computation of all hyperplanes, which can be time consuming.

8 ▶ `HyperplaneToVect(` *arg* `)` O

If *arg* is a hyperplane, then this function returns the coordinate vector of *arg*. If *arg* is a list of hyperplanes from the same projective geometry, then `HyperplaneToVector` returns a list that contains the coordinate vectors of the hyperplanes in *arg*.

```
gap> P := ProjectiveGeometry(2,25);
PG( 2, 25 )
gap> h := Hyperplane(P,[1,0,0]*Z(25));
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> k := Hyperplane(P,[1,1,1]*Z(25));
[ Z(5)^0, Z(5)^0, Z(5)^0 ]
gap> HyperplaneToVector(h);
[ Z(5)^0, 0*Z(5), 0*Z(5) ]
gap> HyperplaneToVector([h,k]);
[ [ Z(5)^0, 0*Z(5), 0*Z(5) ], [ Z(5)^0, Z(5)^0, Z(5)^0 ] ]
```

2.6 Functions for relations between projective points and hyperplanes

1 ▶ `DiffSetPoints(` P `)` A

`DiffSetPoints` returns a list with the numbers of points that lie in the hyperplane $X_n = 0$ of P=PG(n,q).

```
gap> P := ProjectiveGeometry(3,2);
PG( 3, 2 )
gap> DiffSetPoints(P);
[ 1, 2, 3, 5, 6, 9, 11 ]
```

We note that the use of this function causes the computation of all projective points, but not computation of the hyperplanes.

2 ▶ `DiffSetHyperplanes(` P `)` A

`DiffSetHyperplanes` returns the list containing the numbers of the hyperplanes that go through the point $[1,0,\ldots,0]$ of P = PG(n,q).

```
gap> P := ProjectiveGeometry(3,2);
PG( 3, 2 )
gap> DiffSetHyperplanes(P);
[ 1, 6, 8, 11, 12, 14, 15 ]
```

We note that the use of this function causes the computation of all hyperplanes, but not the computation of the projective points.

3 ► `HyperplanesAsDiffSets( P )` A

Given a projective geometry $P = \mathrm{PG}(n,q)$, this function returns a list of length $(q^{n+1}-1)/(q-1)$. The i^{th} position of this list contains a list with the numbers of all the projective points in the i-th hyperplane of P.

```
gap> P := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> HyperplanesAsDiffSets(P);
[ [ 1, 2, 4, 11, 15, 27 ], [ 2, 3, 5, 12, 16, 28 ], [ 3, 4, 6, 13, 17, 29 ],
  [ 4, 5, 7, 14, 18, 30 ], [ 5, 6, 8, 15, 19, 31 ], [ 6, 7, 9, 16, 20, 1 ],
  [ 7, 8, 10, 17, 21, 2 ], [ 8, 9, 11, 18, 22, 3 ], [ 9, 10, 12, 19, 23, 4 ],
  [ 10, 11, 13, 20, 24, 5 ], [ 11, 12, 14, 21, 25, 6 ],
  [ 12, 13, 15, 22, 26, 7 ], [ 13, 14, 16, 23, 27, 8 ],
  [ 14, 15, 17, 24, 28, 9 ], [ 15, 16, 18, 25, 29, 10 ],
  [ 16, 17, 19, 26, 30, 11 ], [ 17, 18, 20, 27, 31, 12 ],
  [ 18, 19, 21, 28, 1, 13 ], [ 19, 20, 22, 29, 2, 14 ],
  [ 20, 21, 23, 30, 3, 15 ], [ 21, 22, 24, 31, 4, 16 ],
  [ 22, 23, 25, 1, 5, 17 ], [ 23, 24, 26, 2, 6, 18 ],
  [ 24, 25, 27, 3, 7, 19 ], [ 25, 26, 28, 4, 8, 20 ],
  [ 26, 27, 29, 5, 9, 21 ], [ 27, 28, 30, 6, 10, 22 ],
  [ 28, 29, 31, 7, 11, 23 ], [ 29, 30, 1, 8, 12, 24 ],
  [ 30, 31, 2, 9, 13, 25 ], [ 31, 1, 3, 10, 14, 26 ] ]
```

We note that the use of this function causes the computation of all projective points, but not the computation of the hyperplanes.

4 ► `ProjectivePointsAsDiffSets( P )` A

Given a projective geometry $P = \mathrm{PG}(n,q)$, this function returns a list of length $(q^{n+1}-1)/(q-1)$. The i^{th} position of this list contains a list with the numbers of all the hyperplanes through the i^{th} projective point of P.

```
gap> P := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> ProjectivePointsAsDiffSets(P);
[ [ 1, 6, 18, 22, 29, 31 ], [ 2, 7, 19, 23, 30, 1 ], [ 3, 8, 20, 24, 31, 2 ],
  [ 4, 9, 21, 25, 1, 3 ], [ 5, 10, 22, 26, 2, 4 ], [ 6, 11, 23, 27, 3, 5 ],
  [ 7, 12, 24, 28, 4, 6 ], [ 8, 13, 25, 29, 5, 7 ], [ 9, 14, 26, 30, 6, 8 ],
  [ 10, 15, 27, 31, 7, 9 ], [ 11, 16, 28, 1, 8, 10 ],
  [ 12, 17, 29, 2, 9, 11 ], [ 13, 18, 30, 3, 10, 12 ],
  [ 14, 19, 31, 4, 11, 13 ], [ 15, 20, 1, 5, 12, 14 ],
  [ 16, 21, 2, 6, 13, 15 ], [ 17, 22, 3, 7, 14, 16 ],
  [ 18, 23, 4, 8, 15, 17 ], [ 19, 24, 5, 9, 16, 18 ],
  [ 20, 25, 6, 10, 17, 19 ], [ 21, 26, 7, 11, 18, 20 ],
  [ 22, 27, 8, 12, 19, 21 ], [ 23, 28, 9, 13, 20, 22 ],
  [ 24, 29, 10, 14, 21, 23 ], [ 25, 30, 11, 15, 22, 24 ],
  [ 26, 31, 12, 16, 23, 25 ], [ 27, 1, 13, 17, 24, 26 ],
  [ 28, 2, 14, 18, 25, 27 ], [ 29, 3, 15, 19, 26, 28 ],
  [ 30, 4, 16, 20, 27, 29 ], [ 31, 5, 17, 21, 28, 30 ] ]
```

We note that the use of this function causes the computation of all projective points **and** all hyperplanes of P.

5 ► `PointsInHyperplane( h )` O

This operation returns a list of all the projective points that lie in the hyperplane h.

```
gap> P := ProjectiveGeometry(3,3);
PG( 3, 3 )
gap> h := Hyperplane(P,[1,0,2,1]*Z(3));
[ Z(3)^0, 0*Z(3), Z(3), Z(3)^0 ]
gap> PointsInHyperplane(h);
[ [ Z(3)^0, Z(3)^0, Z(3), Z(3)^0 ], [ Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3) ],
  [ 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0 ], [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), Z(3)^0, Z(3), Z(3) ], [ Z(3)^0, Z(3), 0*Z(3), Z(3) ],
  [ Z(3)^0, Z(3), Z(3)^0, 0*Z(3) ], [ Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3) ],
  [ Z(3)^0, Z(3)^0, 0*Z(3), Z(3) ], [ Z(3)^0, Z(3), Z(3), Z(3)^0 ],
  [ Z(3)^0, 0*Z(3), Z(3), Z(3)^0 ], [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3)^0 ],
  [ Z(3)^0, 0*Z(3), 0*Z(3), Z(3) ] ]
```

We note that the use of this function causes the computation of all projective points **and** all hyperplanes of P.

6 ▶ `ProjectivePoints( h )` A

This attribute returns a list off all the projective points that lie in the hyperplane h. Computing these projective points this way does not require the calculation of all projective points of the projective geometry $\mathrm{PG}(n, q)$ to which h belongs. This function is much faster than `PointsInHyperplane`.

7 ▶ `HyperplanesThroughPoint( p )` O

This operation returns a list of all the hyperplanes that go through the point p.

```
gap> P := ProjectiveGeometry(3,3);
PG( 3, 3 )
gap> p := ProjectivePoint(P,[1,0,2,1]*Z(3));
[ Z(3)^0, 0*Z(3), Z(3), Z(3)^0 ]
gap> HyperplanesThroughPoint(p);
[ [ Z(3)^0, Z(3), Z(3)^0, 0*Z(3) ], [ Z(3)^0, Z(3)^0, Z(3), Z(3)^0 ],
  [ Z(3)^0, Z(3), Z(3), Z(3)^0 ], [ Z(3)^0, 0*Z(3), Z(3), Z(3)^0 ],
  [ 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0 ], [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3)^0 ],
  [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ], [ Z(3)^0, 0*Z(3), 0*Z(3), Z(3) ],
  [ Z(3)^0, Z(3)^0, Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0, Z(3), Z(3) ],
  [ Z(3)^0, Z(3)^0, 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3) ],
  [ Z(3)^0, Z(3), 0*Z(3), Z(3) ] ]
```

We note that the use of this function causes the computation of all projective points **and** all hyperplanes of P.

8 ▶ p `in` h
▶ `\in(` p, h `)` O

p `in` h simply tests whether the projective point p lies in the hyperplane h.

```
gap> P := ProjectiveGeometry(4,3);
PG( 4, 3 )
gap> p := ProjectivePoint(P,[1,0,1,0,1]*Z(3));
[ Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ]
gap> h := Hyperplane(P,[1,0,2,0,0]*Z(3));
[ Z(3)^0, 0*Z(3), Z(3), 0*Z(3), 0*Z(3) ]
gap> p in h;
true
gap> p := ProjectivePoint(P,[1,0,0,0,0]*Z(3));
```

```
[ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ]
gap> p in h;
false
```

9 ▶ `IntersectionNumbers(` *list* `)`
▶ `IntersectionNumbers(` *P*, *list* `)`

Suppose *list* is a dense list of projective points. This function prints the number n_k of hyperplanes that have m_k points in common with *list*. It also returns a list (not necessarily all positions are bound). On position i we find the number of hyperplanes that have $i-1$ points in common with *list*. When only a set of numbers of points in a projective geometry P is available, the second variant has to be used. In the next example, the point set we use is the point set of a non-singular quadric in $\mathrm{PG}(2,17)$, given by the equation $X_1^2 - X_0X_2 = 0$.

```
gap> P := ProjectiveGeometry(2,17);
PG( 2, 17 )
gap> points := [];
[  ]
gap> for a in GF(17) do
> Add(points,ProjectivePoint(P,[a,Z(17)^0,a^2]));
> od;
gap> Add(points,ProjectivePoint(P,[0*Z(17),0*Z(17),Z(17)^0]));
gap> IntersectionNumbers(points);
136 hyperplanes are intersecting in 0 point(s)
18 hyperplanes are intersecting in 1 point(s)
153 hyperplanes are intersecting in 2 point(s)

[ 136, 18, 153 ]
gap> list := PointToNumber(points);
[ 2, 97, 90, 105, 42, 208, 254, 17, 86, 62, 225, 248, 218, 138, 168, 289,
  275, 3 ]
gap> IntersectionNumbers(P,list);
136 hyperplanes are intersecting in 0 point(s)
18 hyperplanes are intersecting in 1 point(s)
153 hyperplanes are intersecting in 2 point(s)

[ 136, 18, 153 ]
```

There are 18 tangent lines, 153 secant lines and 136 lines that have no point in common with the quadric. One can easily check these results.

3 Subspaces of a projective geometry

In this chapter we will describe the construction of subspaces in a given projective geometry. With "spanned by points" we mean taking all possible linear combinations of the coordinate vectors of the projective points.

3.1 Constructing subspaces

1 ▶ `SubspaceOfPG(` *list* `)` O

The argument *list* of this operation is a non-empty list containing either projective points or hyperplanes from the same projective geometry. When *list* contains projective points, the function checks whether they belong to the same projective geometry P and returns the subspace of P spanned by all the projective points in *list*. When *list* contains hyperplanes, the function checks whether the hyperplanes belong to the same projective geometry P and returns the subspace of P that is the intersection of all the hyperplanes in *list*. This operation only **constructs** the appropriate subspace, no calculations about e.g. the dimension or the projective points in the subspace are done. If a subspace is constructed with points, it belongs to the representation `IsSubspaceSpanRep`, otherwise it belongs to the representation `IsSubspaceIntersRep`. Note that there also exists an attribute `PGOfSubspace` for points, hyerplanes and subspaces; this return the projective geometry in which the subspace lies.

```
gap> P := ProjectiveGeometry(5,5);
PG( 5, 5 )
gap> h := Hyperplane(P,[1,0,0,0,0,0]*Z(5));
[ Z(5)^0, 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5) ]
gap> k := Hyperplane(P,[1,0,1,0,0,0]*Z(5));
[ Z(5)^0, 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5), 0*Z(5) ]
gap> l := Hyperplane(P,[0,0,0,0,1,1]*Z(5));
[ 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0, Z(5)^0 ]
gap> S := SubspaceOfPG([h,k,l]);
Subspace of PG( 5, 5 )
gap> Print(S,"\n");
Subspace of PG( 5, 5 ) by intersection of:
[ Hyperplane( PG( 5, 5 ), [ Z(5)^0, 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5) ] )
    , Hyperplane( PG( 5, 5 ), [ Z(5)^0, 0*Z(5), Z(5)^0, 0*Z(5), 0*Z(5),
      0*Z(5) ] ),
  Hyperplane( PG( 5, 5 ), [ 0*Z(5), 0*Z(5), 0*Z(5), 0*Z(5), Z(5)^0, Z(5)^0
    ] ) ]
gap> p := ProjectivePoint(P,[0,1,1,0,0,0]*Z(5));
[ 0*Z(5), Z(5)^0, Z(5)^0, 0*Z(5), 0*Z(5), 0*Z(5) ]
gap> q := ProjectivePoint(P,[1,0,0,1,0,1]*Z(5));
[ Z(5)^0, 0*Z(5), 0*Z(5), Z(5)^0, 0*Z(5), Z(5)^0 ]
gap> R := SubspaceOfPG([p,q]);
Subspace of PG( 5, 5 )
gap> Print(R,"\n");
Subspace of PG( 5, 5 ) spanned by:
```

```
[ ProjectivePoint( PG( 5, 5 ), [ 0*Z(5), Z(5)^0, Z(5)^0, 0*Z(5), 0*Z(5),
      0*Z(5) ] ),
  ProjectivePoint( PG( 5, 5 ), [ Z(5)^0, 0*Z(5), 0*Z(5), Z(5)^0, 0*Z(5),
      Z(5)^0 ] ) ]
```

2 ▶ `SubspaceOfPG( p )` O
▶ `SubspaceOfPG( H )` O

To construct a subspace that is only a point p or a hyperplane H, one needs not to use a list as argument. The first variant returns the subspace that is actually a single projective point p, while the second variant returns a subspace that is equal to the hyperplane H.

```
gap> P := ProjectiveGeometry(7,8);
PG( 7, 8 )
gap> p := ProjectivePoint(P,[1,0,1,0,1,0,1,0]*Z(8)^0);
[ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ]
gap> S := SubspaceOfPG(a);
Subspace of PG( 7, 8 )
gap> Print(S,"\n");
Subspace of PG( 7, 8 ) spanned by:
[ ProjectivePoint( PG( 7, 8 ), [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0,
      0*Z(2), Z(2)^0, 0*Z(2) ] ) ]
gap> H := Hyperplane(P,[0,0,1,1,0,0,1,1]*Z(8)^0);
[ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ]
gap> R := SubspaceOfPG(H);
Subspace of PG( 7, 8 )
gap> Print(R,"\n");
Subspace of PG( 7, 8 ) by intersection of:
[ Hyperplane( PG( 7, 8 ), [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2), 0*Z(2),
      Z(2)^0, Z(2)^0 ] ) ]
```

3.2 Basic Functions

1 ▶ `PGOfSubspace( S )` A

returns $\mathrm{PG}(n, q)$ if S is a subspace of $\mathrm{PG}(n, q)$.

```
gap> P := ProjectiveGeometry(5,13);
PG( 5, 13 )
gap> a := Hyperplane(P,[1,0,1,0,0,0]*Z(13));
[ Z(13)^0, 0*Z(13), Z(13)^0, 0*Z(13), 0*Z(13), 0*Z(13) ]
gap> b := Hyperplane(P,[0,0,0,1,1,1]*Z(13));
[ 0*Z(13), 0*Z(13), 0*Z(13), Z(13)^0, Z(13)^0, Z(13)^0 ]
gap> S := SubspaceOfPG([a,b]);
Subspace of PG( 5, 13 )
gap> PGOfSubspace(S);
PG( 5, 13 )
```

2 ▶ `Dimension( S )` A

returns the geometrical dimension of the subspace S.

3 ▶ `IsEmpty( S )` P

returns `true` if and only if the subspace S of $\mathrm{PG}(n, q)$ is empty, i.e. if the geometrical dimension of S equals -1.

4 ▶ `IsWholePG( S )` P

returns `true` if and only if the subspace S of $\mathrm{PG}(n, q)$ is equal to $\mathrm{PG}(n, q)$ itself (as set of projective points), i.e. if the geometrical dimension of S equals n.

```
gap> P := ProjectiveGeometry(2,2);
PG( 2, 2 )
gap> a := ProjectivePoint(P,[1,0,0]*Z(2));
[ Z(2)^0, 0*Z(2), 0*Z(2) ]
gap> b := ProjectivePoint(P,[0,1,0]*Z(2));
[ 0*Z(2), Z(2)^0, 0*Z(2) ]
gap> c := ProjectivePoint(P,[0,0,1]*Z(2));
[ 0*Z(2), 0*Z(2), Z(2)^0 ]
gap> S := SubspaceOfPG([a,b]);
Subspace of PG( 2, 2 )
gap> Dimension(S);
1
gap> R := SubspaceOfPG([a,c,b]);
Subspace of PG( 2, 2 )
gap> Dimension(R);
2
gap> IsWholePG(R);
true
gap> A := Hyperplane(P,[1,0,0]*Z(2));
[ Z(2)^0, 0*Z(2), 0*Z(2) ]
gap> B := Hyperplane(P,[1,1,0]*Z(2));
[ Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> C := Hyperplane(P,[1,1,1]*Z(2));
[ Z(2)^0, Z(2)^0, Z(2)^0 ]
gap> T := SubspaceOfPG([B,C]);
Subspace of PG( 2, 2 )
gap> Dimension(T);
0
gap> U := SubspaceOfPG([A,B,C]);
Subspace of PG( 2, 2 )
gap> IsEmpty(U);
true
```

5 ▶ `Reduce( S )` O

When constructing a subspace with a list of hyperplanes or projective points, the system does not check whether the hyperplanes or points are linearly independent. The operation `Reduce` reduces the defining set of the subspace S (i.e. the list of hyperplanes or projective points that was given for the construction of S) to a list of linearly independent hyperplanes or projective points. The operation does not return any objects.

```
gap> P := ProjectiveGeometry(4,3);
PG( 4, 3 )
gap> a := ProjectivePoint(P,[1,0,0,0,0]*Z(3)^0);
[ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ]
gap> b := ProjectivePoint(P,[0,1,1,0,1]*Z(3)^0);
[ 0*Z(3), Z(3)^0, Z(3)^0, 0*Z(3), Z(3)^0 ]
gap> c := ProjectivePoint(P,[0,0,0,1,0]*Z(3)^0);
[ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ]
gap> S := SubspaceOfPG([a,a,b,c,a]);
```

```
Subspace of PG( 4, 3 )
gap> Print(S,"\n");
Subspace of PG( 4, 3 ) spanned by:
[ ProjectivePoint( PG( 4, 3 ), [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ] ),
  ProjectivePoint( PG( 4, 3 ), [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ] ),
  ProjectivePoint( PG( 4, 3 ), [ 0*Z(3), Z(3)^0, Z(3)^0, 0*Z(3), Z(3)^0 ] ),
  ProjectivePoint( PG( 4, 3 ), [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ] ),
  ProjectivePoint( PG( 4, 3 ), [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ] ) ]
gap> Reduce(S);
Subspace of PG( 4, 3 )
gap> Print(S,"\n");
Subspace of PG( 4, 3 ) spanned by:
[ ProjectivePoint( PG( 4, 3 ), [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ] ),
  ProjectivePoint( PG( 4, 3 ), [ 0*Z(3), Z(3)^0, Z(3)^0, 0*Z(3), Z(3)^0 ] ),
  ProjectivePoint( PG( 4, 3 ), [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ] ) ]
```

6 ► `AsSpanningPoints( S )` O

If S is a subspace of $\mathrm{PG}(n, q)$ constructed as the intersection of hyperplanes, then `AsSpanningPoints` returns the same subspace of $\mathrm{PG}(n, q)$ but now constructed as the linear span of projective points.

7 ► `AsIntersectingHyperplanes( S )` O

If S is a subspace of $\mathrm{PG}(n, q)$ constructed as the linear span of projective points, then `AsIntersectingHyperplanes` returns the same subspace of $\mathrm{PG}(n, q)$ but now constructed as the intersection of hyperplanes.

```
gap> P := ProjectiveGeometry(4,3);
PG( 4, 3 )
gap> a := [1,0,0,0,0]*Z(3)^0;
[ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ]
gap> b := [0,1,0,0,0]*Z(3)^0;
[ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ]
gap> lijst := [ProjectivePoint(P,a),ProjectivePoint(P,b)];
[ [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ] ]
gap> S := SubspaceOfPG(lijst);
Subspace of PG( 4, 3 )
gap> Print(S);
Subspace of PG( 4, 3 ) spanned by:
[ ProjectivePoint( PG( 4, 3 ), [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ] )
    ,
  ProjectivePoint( PG( 4, 3 ), [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3)
     ] ) ]
gap> R := AsIntersectingHyperplanes(S);
Subspace of PG( 4, 3 )
gap> Print(R);
Subspace of PG( 4, 3 ) by intersection of:
[ Hyperplane( PG( 4, 3 ), [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ] ),
  Hyperplane( PG( 4, 3 ), [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ] ),
  Hyperplane( PG( 4, 3 ), [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] )
 ]
gap> lijst := [Hyperplane(P,a),Hyperplane(P,b)];
[ [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ] ]
```

```
gap> S := SubspaceOfPG(lijst);
Subspace of PG( 4, 3 )
gap> Print(S,"\n");
Subspace of PG( 4, 3 ) by intersection of:
[ Hyperplane( PG( 4, 3 ), [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ] ),
  Hyperplane( PG( 4, 3 ), [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ] )
 ]
gap> R := AsSpanningPoints(S);
Subspace of PG( 4, 3 )
gap> Print(R,"\n");
Subspace of PG( 4, 3 ) spanned by:
[ ProjectivePoint( PG( 4, 3 ), [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ] )
    ,
  ProjectivePoint( PG( 4, 3 ), [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ] )
    ,
  ProjectivePoint( PG( 4, 3 ), [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0
     ] ) ]
```

8 ▶ `ProjectivePoints( S )` A

returns a list with all projective points of a subspace S of a projective geometry.

```
gap> P := ProjectiveGeometry(3,4);
PG( 3, 4 )
gap> p := ProjectivePoint(P,[1,0,1,0]*Z(4)^0);
[ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ]
gap> q := ProjectivePoint(P,[0,1,1,1]*Z(4)^0);
[ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ]
gap> S := SubspaceOfPG([p,q]);
Subspace of PG( 3, 4 )
gap> ProjectivePoints(S);
[ [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ], [ Z(2)^0, Z(2)^0, 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, Z(2^2), Z(2^2)^2, Z(2^2) ], [ Z(2)^0, Z(2^2)^2, Z(2^2), Z(2^2)^2 ]
    , [ Z(2)^0, 0*Z(2), Z(2)^0, 0*Z(2) ] ]
gap> H := Hyperplane(P,[1,1,1,1]*Z(4)^0);
[ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ]
gap> I := Hyperplane(P,[0,1,1,0]*Z(4)^0);
[ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> R := SubspaceOfPG([H,I]);
Subspace of PG( 3, 4 )
gap> ProjectivePoints(R);
[ [ Z(2)^0, 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(2^2)^2, Z(2^2)^2, Z(2)^0 ], [ Z(2)^0, Z(2^2), Z(2^2), Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, Z(2)^0, 0*Z(2) ] ]
```

Here we also obtain an alternative way to compute all projective points of P= PG(n, q). We simply take a subspace S that is equal to P. All the points of P are all the points of S. As already mentioned, after using this way to compute all projective points of P, we are not able to assign a number to the projective points.

```
gap> P := ProjectiveGeometry(5,4);
PG( 5, 4 )
gap> list := [];
[ ]
gap> matrix := IdentityMat(6,GF(4));
[ [ 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 ] ]
gap> for i in [1..6] do
> list[i] := ProjectivePoint(P,matrix[i]);
> od;
gap> S := SubspaceOfPG(list);
Subspace of PG( 5, 4 )
gap> IsWholePG(S);
true
gap> points1 := ProjectivePoints(S);;
gap> time;
280
gap> Length(points1);
1365
gap> points2 := ProjectivePoints(P);;
gap> time;
600
gap> Length(points2);
1365
```

3.3 Miscellaneous functions for subspaces

1 ▶ *p* `in` *S*
▶ `\in(` *p*, *S* `)` O

returns `true` if and only if the projective point p lies in the subspace S

2 ▶ `Intersection(` *S1*, *S2*, ... `)` F
▶ `Intersection(` *list* `)` F

Suppose *S1*, *S2*, ..., are subspaces of the same PG(n, q), and *list* is a dense list of subspaces of the same PG(n, q). The first variant returns the intersection of the subspaces *S1*, *S2*, ..., which is again a subspace of PG(n, q) (possibly empty). The second variant returns the intersection of all the subspaces in *list*.

3 ▶ `ProjectivePoint(` *S* `)` O

When S is a 0-dimensional subspace, then S is only one point. In that case, `ProjectivePoint(`S`)` returns that projective point. If S is not a 0-dimensional subspace, then this function returns an error.

4 ▶ `Hyperplane(` *S* `)` O

When S is an $(n-1)$-dimensional subspace of PG(n, q), then S is a hyperplane. In that case, `Hyperplane(`S`)` returns that hyperplane. If S is not an $(n-1)$-dimensional subspace, then an error is returned.

4 Collineations and the collineation group

A collineation of $\mathrm{P\Gamma L}(n+1,q)$ is uniquely defined by a non-singular $(n+1)\times(n+1)$-matrix and an automorphism of the finite field $\mathrm{GF}(q)$.

If $q=p^h$, p prime, then the automorphisms of $\mathrm{GF}(q)$ are $f_i:\mathrm{GF}(q)\to\mathrm{GF}(q)$: $x\mapsto x^{p^i}$, $i=0,1,\ldots,h-1$. Thus a collineation of $\mathrm{P\Gamma L}(n+1,q)$ is uniquely defined by a non-singular $(n+1)\times(n+1)$-matrix and an integer i with $0\le i\le h-1$.

4.1 The collineation group

1 ▶ `PGammaL( n, q )` F

Given a positive integer n and a prime power q, this function returns the collineation group $\mathrm{P\Gamma L}(n,q)$.

2 ▶ `Size( G )` A

Given a collineation group G, this function returns the number of elements of G.

```
gap> G := PGammaL(4,5);
PGammaL( 4, 5 )
gap> Size(G);
29016000000
```

As collineation groups are constructed as usual groups in GAP, some existing basic group functions are available:

3 ▶ `GeneratorsOfGroup( G )` A

returns the generators of the collineation group G.

4 ▶ `One( G )` A

returns the identity of the collineation group G.

```
gap> G := PGammaL(4,25);
PGammaL( 4, 25 )
gap> GeneratorsOfGroup(G);
[ 0 ; [ [ Z(5)^0, 0*Z(5), 0*Z(5), 0*Z(5) ],
      [ 0*Z(5), Z(5^2)^23, 0*Z(5), 0*Z(5) ],
      [ 0*Z(5), 0*Z(5), Z(5^2)^23, 0*Z(5) ],
      [ 0*Z(5), 0*Z(5), 0*Z(5), Z(5^2)^23 ] ],
  0 ; [ [ Z(5)^0, 0*Z(5), 0*Z(5), Z(5)^2 ], [ 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) ] ],
  1 ; [ [ 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> One(G);
0 ; [ [ 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 ] ]
```

4.2 Collineations

1 ▶ `Collineation( G, i, A )` O

Suppose G is the collineation group $\mathrm{P\Gamma L}(n, q)$, $q = p^h$, $0 \leq i \leq h-1$, suppose A is a non-singular $(n \times n)$-matrix over $\mathrm{GF}(q)$. Then `Collineation` returns the collineation from G that is defined by the matrix A and the field automorphism of $\mathrm{GF}(q)$ defined by $x \mapsto x^{p^i}$.

2 ▶ *c1* `=` *c2*
▶ `\=(` *c1*`,` *c2* `)` O

returns `true` if and only if the collineation *c1* is equal to collineation *c2*.

3 ▶ *c1* `*` *c2*
▶ `\*(` *c1*`,` *c2* `)` O

returns *c1*·*c2*, i.e. the collineation that is the concatenation of *c1* and *c2* by applying *c1* first.

```
gap> G := PGammaL(3,16);
PGammaL( 3, 16 )
gap> A := [[1,0,1],[0,1,0],[0,1,1]]*Z(16)^0;
[ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, Z(2)^0 ] ]
gap> c1 := Collineation(G,0,A);
0 ; [ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, Z(2)^0 ] ]
gap> c2 := Collineation(G,2,A);
2 ; [ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), Z(2)^0, Z(2)^0 ] ]
gap> c1=c2;
false
gap> c1*c2;
2 ; [ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> c2*c2;
0 ; [ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0 ] ]
```

4 ▶ `Inverse( c )` A

returns the collineation that is the inverse of the collineation c.

```
gap> G := PGammaL(3,32);
PGammaL( 3, 32 )
gap> A := [[1,1,1],[0,1,1],[0,0,1]]*Z(16)^0;
[ [ Z(2)^0, Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0, Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> c := Collineation(G,2,A);
2 ; [ [ Z(2)^0, Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0, Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> Inverse(c);
3 ; [ [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, Z(2)^0 ],
  [ 0*Z(2), 0*Z(2), Z(2)^0 ] ]
```

4.3 Actions on basic objects

1► *s*^*c*
► \^(*s*, *c*) O

This operation returns the image of *s* under the collineation *c*, where *s* can be a projective point, a hyperplane or in general be a subspace of a projective geometry.

```
gap> P := ProjectiveGeometry(4,27);
PG( 4, 27 )
gap> G := PGammaL(5,27);
PGammaL( 5, 27 )
gap> p := ProjectivePoint(P,[Z(3)^0,Z(3),Z(3)^2,Z(27),Z(27)]);
[ Z(3)^0, Z(3), Z(3)^0, Z(3^3), Z(3^3) ]
gap> q := ProjectivePoint(P,[Z(3)^0,0*Z(3),0*Z(3),Z(27)^8,Z(27)]);
[ Z(3)^0, 0*Z(3), 0*Z(3), Z(3^3)^8, Z(3^3) ]
gap> A := [[1,1,1,1,1],[0,0,0,0,1],[0,0,1,1,1],[0,1,1,1,1],[0,0,0,1,1]]*Z(3)^0;
[ [ Z(3)^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)^0 ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0 ],
  [ 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ],
  [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, Z(3)^0 ] ]
gap> c := Collineation(G,2,A);
2 ; [ [ Z(3)^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)^0 ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0 ],
  [ 0*Z(3), Z(3)^0, Z(3)^0, Z(3)^0, Z(3)^0 ],
  [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, Z(3)^0 ] ]
gap> p^c;
[ Z(3)^0, Z(3^3)^3, Z(3^3), Z(3^3)^16, Z(3^3)^14 ]
gap> q^c;
[ Z(3)^0, Z(3^3)^5, Z(3^3)^5, Z(3^3)^23, Z(3^3)^23 ]
gap> H := Hyperplane(P,[1,0,1,0,1]*Z(3)^0);
[ Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ]
gap> H^c;
[ Z(3)^0, Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ]
gap> S := SubspaceOfPG([p,q]);
Subspace of PG( 4, 27 )
gap> S^c;
Subspace of PG( 4, 27 )
gap> Print(last,"\n");
Subspace of PG( 4, 27 ) spanned by:
[ ProjectivePoint( PG( 4, 27 ), [ Z(3)^0, Z(3^3)^3, Z(3^3), Z(3^3)^16,
      Z(3^3)^14 ] ),
  ProjectivePoint( PG( 4, 27 ), [ 0*Z(3), Z(3)^0, Z(3^3)^19, Z(3^3)^19,
      Z(3)^0 ] ) ]
gap> R := Intersection(S,SubspaceOfPG(H));
Subspace of PG( 4, 27 )
gap> Print(R,"\n");
Subspace of PG( 4, 27 ) by intersection of:
[ Hyperplane( PG( 4, 27 ), [ 0*Z(3), Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3) ] ),
  Hyperplane( PG( 4, 27 ), [ Z(3)^0, Z(3^3)^11, 0*Z(3), Z(3^3)^5, 0*Z(3) ] ),
  Hyperplane( PG( 4, 27 ), [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3), Z(3^3)^12 ] ),
```

```
    Hyperplane( PG( 4, 27 ), [ Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ] ) ]
gap> R^c;
Subspace of PG( 4, 27 )
gap> Print(last,"\n");
Subspace of PG( 4, 27 ) by intersection of:
[ Hyperplane( PG( 4, 27 ), [ 0*Z(3), Z(3)^0, Z(3), Z(3)^0, Z(3) ] ),
  Hyperplane( PG( 4, 27 ), [ Z(3)^0, Z(3^3)^3, Z(3^3)^23, 0*Z(3), 0*Z(3) ] ),
  Hyperplane( PG( 4, 27 ), [ Z(3)^0, Z(3^3)^8, Z(3^3)^16, Z(3^3)^11, 0*Z(3)
     ] ),
  Hyperplane( PG( 4, 27 ), [ Z(3)^0, Z(3), Z(3^3)^17, Z(3^3)^4, 0*Z(3) ] ) ]
```

4.4 The permutation group derived from a collineation group acting on the projective points

We note that the use of the functions in this section may take a lot of time since is it necessary to calculate all the projective points of the projective geometry in which you work.

1 ▶ `AsPermutationGroup( G )` A

returns a permutation group which is computed as follows. Suppose P =PG(n, q). Remember that we are able to assign a number to every projective point of P. As we can compute the image of every point under a collineation of G = PΓL$(n + 1, q)$, we can determine the number of the image point. If we do so for every projective point of P, we get for every generator of G a permutation of the set $\{1, 2, \ldots, \frac{q^{n+1}-1}{q-1}\}$. We take these permutations to generate a permutation group, and it is that group that is returned by `AsPermutationGroup`.

```
gap> G := PGammaL(3,8);
PGammaL( 3, 8 )
gap> AsPermutationGroup(G);
<permutation group with 3 generators>
gap> GeneratorsOfGroup(last);
[ ( 4,44,56,64, 6,70,45)( 5,68,48, 8,10,49,60)( 7,54,38,55,16,11,66)
    ( 9,24,46,62,19,15,63)(12,20,33,73,57,26,30)(13,21,58,34,31,36,27)
    (17,47,50,52,29,43,18)(23,69,40,65,72,51,39)(25,61,41,71,42,53,67),
  ( 1,42,72,59,58, 3, 2)( 4,40, 5,30,22,21,25)( 6,37,36,41,68,65,20)
    ( 7,12,55,38,24,10,56)( 8,14,13,71,26,23,70)( 9,29,73,48,63,44,19)
    (11,47,28,27,53,62,39)(15,66,32,31,61,43,51)(16,45,33,52,49,50,17)
    (18,46,35,34,67,54,69), ( 4,26,68)( 5,70,20)( 6,30, 8)( 7,52,63)( 9,38,17)
    (10,45,73)(11,18,15)(12,49,44)(13,36,21)(14,37,22)(16,29,24)(19,55,50)
    (23,65,40)(25,71,41)(27,34,31)(28,35,32)(33,48,56)(39,69,51)(43,62,54)
    (46,66,47)(53,67,61)(57,60,64) ]
```

2 ▶ `PermutationByCollineation( c )` O

returns the permutation that is generated by the collineation c as described above. If c is a collineation of PΓL$(n + 1, q)$, then a permutation of $\{1, 2, \ldots, \frac{q^{n+1}-1}{q-1}\}$ is returned.

3 ▶ `PermutationToCollineation( G, perm )` O

returns the collineation in G that is induced by the permutation *perm*. There are some conditions that must be satisfied, otherwise this function will give an error message. If G = PΓL$(n + 1, q)$, a necessary condition is that *perm* must be a permutation on the set $\{1, 2, \ldots, \frac{q^{n+1}-1}{q-1}\}$. This condition is however not sufficient. There are permutations of $\{1, 2, \ldots, \frac{q^{n+1}-1}{q-1}\}$ which do not induce collineations of G.

```
gap> G := PGammaL(3,4);
PGammaL( 3, 4 )
gap> H := AsPermutationGroup(G);
Group([ ( 4,14,21)( 5,15,17)( 7,10,20)( 8, 9,12)(11,13,19),
  ( 1, 8,21,16,15, 3, 2)( 4,10,20,18,17, 9, 7)( 5,12,11,14,19,13, 6),
  ( 4,14)( 5,17)( 6,18)( 7,11)( 9,12)(10,19)(13,20) ])
gap> gens := GeneratorsOfGroup(G);
[ 0 ; [ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2^2)^2, 0*Z(2) ],
      [ 0*Z(2), 0*Z(2), Z(2^2)^2 ] ],
  0 ; [ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
      [ 0*Z(2), Z(2)^0, 0*Z(2) ] ],
  1 ; [ [ 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 ] ] ]
gap> g := gens[1]^5*gens[2]^7*gens[3]^9;
1 ; [ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2^2)^2, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2^2)^2 ] ]
gap> PermutationByCollineation(g);
( 4,21)( 6,18)( 7,13)( 8, 9)(10,11)(15,17)(19,20)
gap> gens := GeneratorsOfGroup(H);
[ ( 4,14,21)( 5,15,17)( 7,10,20)( 8, 9,12)(11,13,19),
  ( 1, 8,21,16,15, 3, 2)( 4,10,20,18,17, 9, 7)( 5,12,11,14,19,13, 6),
  ( 4,14)( 5,17)( 6,18)( 7,11)( 9,12)(10,19)(13,20) ]
gap> perm := gens[1]^10*gens[2]*gens[3]^4;
( 1, 8, 7,20, 4,19,14,16,15, 9,11, 6, 5, 3, 2)(10,18,17,12,21)
gap> PermutationToCollineation(G,perm);
0 ; [ [ Z(2)^0, 0*Z(2), Z(2)^0 ], [ Z(2^2)^2, 0*Z(2), 0*Z(2) ],
  [ 0*Z(2), Z(2^2)^2, 0*Z(2) ] ]
```

5 Quadrics and Hermitian varieties

5.1 Definitions and introduction

Suppose $F(X_0, X_1, \ldots, X_n)$ is a homogeneous polynomial of degree m, $m \geq 1$, over the finite field $\mathrm{GF}(q)$. By $V(F)$, we denote the set of points of $\mathrm{PG}(n, q)$ whose coordinate vectors satisfy F, i.e. $p(x_0, x_1, \ldots, x_n)$ is a point of $V(F)$ if and only if $F(x_0, x_1, \ldots, x_n) = 0$.

A *quadric* in $\mathrm{PG}(n, q)$, $n \geq 1$, is each set of points $V(F)$ in $\mathrm{PG}(n, q)$ with $F = \sum_{i,j=0}^{n} a_{ij} X_i X_j$. In the summation we suppose the extra condition $i \leq j$.

Suppose we are now working in $\mathrm{PG}(n, q)$, with q a square. A *Hermitian variety* in $\mathrm{PG}(n, q)$, $n \geq 1$, is each set of points $V(F)$ in $\mathrm{PG}(n, q)$ with $F = \sum_{i,j=0}^{n} a_{ij} X_i X_j^{\sqrt{q}}$. For a_{ij}, the condition $a_{ij}^{\sqrt{q}} = a_{ji}$ is required.

It is well known that for every quadric Q in $\mathrm{PG}(n, q)$, there exists a base transition of $\mathrm{PG}(n, q)$ so that the equation of Q becomes (with respect to the new base) $F = X_0X_1 + X_2X_3 + \ldots + X_{r-1}X_r$ or $F = X_0^2 + X_1X_2 + \ldots + X_{r-1}X_r$ or $F = g(X_0, X_1) + X_2X_3 + \ldots + X_{r-1}X_r$, with $g(X_0, X_1)$ an irreducible quadratic polynomial over $\mathrm{GF}(q)$. These three forms are called respectively hyperbolic, parabolic and elliptic. We denote these forms by the name "standard forms". Furthermore, Q is non-singular if and only if $r = n$ in the three cases. The *nature* of a quadric is the couple $[r, w]$ wich has the following meaning. The number r is one less than the number of used indeterminates in the standard form of the quadric and $w = 0, 1, 2$ if the quadric is respectively elliptic, parabolic or hyperbolic.

Hermitian varieties have an analogous but simplified property. For every Hermitian variety H of $\mathrm{PG}(n, q)$, q a square, there exists a base transition of $\mathrm{PG}(n, q)$ so that the equation of H becomes (with respect to the new base) $F = X_0^{\sqrt{q}+1} + X_1^{\sqrt{q}+1} + \ldots + X_r^{\sqrt{q}+1}$. We also denote this equation with the name "Standard form". Similarly, H is non-singular if and only if $r = n$. The *nature* of a Hermitian variety is defined as r, which is one less than the number of indeterminates that is used in its standard form.

We will often use polynomials in GAP. Suppose we are working in $\mathrm{PG}(n, q)$. In that case we need polynomials in at most $n + 1$ indeterminates. It is necessary that we use the same indeterminates for all quadrics and Hermitian varieties in $\mathrm{PG}(n, q)$.

1 ▶ `Indets( P )` A

returns a list of $n + 1$ indeterminates over the finite field $\mathrm{GF}(q)$ when P=$\mathrm{PG}(n, q)$. If we need polynomials to construct quadrics or Hermitian varieties, we will always use the indeterminates returned by `Indets`.

For the next sections, we suppose always q being a square when dealing with Hermitian varieties in $\mathrm{PG}(n, q)$.

5.2 Construction of quadrics and Hermitian varieties

1 ▶ `Quadric( P, A )` O
▶ `Quadric( P, F )` O

The first variant returns a quadric in the projective geometry P using the matrix A. If $A = (a_{ij})$, then the quadric $V(F)$ in P is returned with $F = \sum_{i,j=0}^{n} a_{ij} X_i X_j$, $i \leq j$. Only the elements of (a_{ij}) with $i \leq j$ are needed and used. So A is regarded as an upper triangle matrix.

The second variant returns a quadric in the projective geometry P using the polynomial F. The polynomial F must be a homogeneous polynomial of degree 2 over GF(q), then the quadric $V(F)$ is returned.

```
gap> P := ProjectiveGeometry(3,5);
PG( 3, 5 )
gap> A := [[1,0,1,0],[0,1,1,1],[0,0,1,0],[0,0,0,1]]*Z(5)^0;
[ [ Z(5)^0, 0*Z(5), Z(5)^0, 0*Z(5) ], [ 0*Z(5), Z(5)^0, Z(5)^0, Z(5)^0 ],
  [ 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> Q := Quadric(P,A);
Quadric of PG( 3, 5 )
gap> Print(Q,"\n");
Quadric of PG( 3, 5 )
x_0^2+x_0*x_2+x_1^2+x_1*x_2+x_1*x_3+x_2^2+x_3^2
gap> x := Indets(P);
[ x_0, x_1, x_2, x_3 ]
gap> F := x[1]*x[2]+x[3]*x[4];
x_0*x_1+x_2*x_3
gap> Q := Quadric(P,F);
Quadric of PG( 3, 5 )
gap> Print(Q,"\n");
Quadric of PG( 3, 5 )
x_0*x_1+x_2*x_3
```

2 ▶ `HermitianVariety( P, A )` O

▶ `HermitianVariety( P, F )` O

The first variant returns a Hermitian variety in the projective geometry P using the matrix A. If $A = (a_{ij})$ then the Hermitian variety $V(F)$ in P is returned with $F = \sum_{i,j=0}^{n} a_{ij} X_i X_j^{\sqrt{q}}$. The condition $A = (A^T)^{\sqrt{q}}$ is required.

The second variant returns a Hermitian variety in the projective geometry P using the polynomial F. The following condition must be fulfilled. There must exist a Hermitian matrix A (that is a matrix for which $A = (A^T)^{\sqrt{q}}$) such that $F = X \cdot A \cdot (X^T)^{\sqrt{q}}$ with $X = (X_0, X_1, \ldots, X_n)$.

```
gap> P := ProjectiveGeometry(2,16);
PG( 2, 16 )
gap> A :=
[[Z(2)^0,Z(16),Z(2)^0],[Z(16)^4,0*Z(2),0*Z(2)],[Z(2)^0,0*Z(2),Z(2)]];
[ [ Z(2)^0, Z(2^4), Z(2)^0 ], [ Z(2^4)^4, 0*Z(2), 0*Z(2) ],
  <a GF2 vector of length 3> ]
gap> H := HermitianVariety(P,A);
Hermitian Variety of PG( 2, 16 )
gap> Print(H,"\n");
Hermitian Variety of PG( 2, 16 )
x_0^5+Z(2^4)^4*x_0^4*x_1+x_0^4*x_2+Z(2^4)*x_0*x_1^4+x_0*x_2^4+x_2^5
gap> x := Indets(P);
[ x_0, x_1, x_2 ]
gap> F := x[1]^5+x[2]^5+x[3]^5;
x_0^5+x_1^5+x_2^5
gap> H := HermitianVariety(P,F);
Hermitian Variety of PG( 2, 16 )
gap> Print(H,"\n");
Hermitian Variety of PG( 2, 16 )
x_0^5+x_1^5+x_2^5
```

5.3 Construction of standard forms

1 ▶ `Quadric( P, nat)` O

returns the standard quadric in P with given nature nat. The standard form is completely defined by the nature of the quadric.

2 ▶ `StandardHyperbolicQuadric( P )` A
▶ `StandardEllipticQuadric( P )` A
▶ `StandardParabolicQuadric( P )` A

Suppose that we are working in P=PG$(2s+1, q)$. The first attribute returns the standard quadric in P with nature $[2s+1, 2]$, the second attribute returns the standard quadric in P with nature $[2s+1, 0]$. The third function will give an error message because a non-singular parabolic quadric does not exist in PG$(2s+1, q)$.

Suppose that we are working in P=PG$(2s, q)$. The first two attributes will give an error message, because a non-singular elliptic and a non-singular hyperbolic quadric does not exist in PG$(2s, q)$. The third attribute returns the standard quadric in P with nature $[2s, 1]$.

```
gap> P := ProjectiveGeometry(3,4);
PG( 3, 4 )
gap> Q := Quadric(P,[2,1]);
Quadric of PG( 3, 4 )
gap> Print(Q,"\n");
Quadric of PG( 3, 4 )
x_0^2+x_1*x_2
gap> Q := Quadric(P,[3,0]);
Quadric of PG( 3, 4 )
gap> Print(Q,"\n");
Quadric of PG( 3, 4 )
x_0^2+x_0*x_1+Z(2^2)*x_1^2+x_2*x_3
gap> Print(StandardHyperbolicQuadric(P),"\n");
Quadric of PG( 3, 4 )
x_0*x_1+x_2*x_3
gap> Print(StandardEllipticQuadric(P),"\n");
Quadric of PG( 3, 4 )
x_0^2+x_0*x_1+Z(2^2)*x_1^2+x_2*x_3
gap> P := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> Q := StandardParabolicQuadric(P);
Quadric of PG( 2, 5 )
gap> Print(Q,"\n");
Quadric of PG( 2, 5 )
x_0^2+x_1*x_2
```

3 ▶ `HermitianVariety( P, r )` O

returns the standard Hermitian variety in P with nature r. The standard form is completely defined by the nature r.

4 ▶ `StandardHermitianVariety( P )` O

Suppose that we are working in PG(n, q). This attribute returns the standard Hermitian variety in P with nature n.

```
gap> P := ProjectiveGeometry(4,49);
PG( 4, 49 )
gap> H := HermitianVariety(P,3);
Hermitian Variety of PG( 4, 49 )
gap> Print(H,"\n");
Hermitian Variety of PG( 4, 49 )
x_0^8+x_1^8+x_2^8+x_3^8
gap> Print(StandardHermitianVariety(P),"\n");
Hermitian Variety of PG( 4, 49 )
x_0^8+x_1^8+x_2^8+x_3^8+x_4^8
```

5.4 Some properties of quadrics and Hermitian varieties

1► `IsSingular( K )` P

returns true if and only if the quadric or Hermitian variety K is singular.

2► `IsCanonical( K )` P

returns true if and only if the quadric or Hermitian variety has a standard equation.

3► `StandardForm( K )` A

returns the standard equation of the quadric or Hermitian variety K.

4► `Nature( K )` A

returns the nature of the quadric or Hermitian variety K.

```
gap> P := ProjectiveGeometry(3,8);
PG( 3, 8 )
gap> x := Indets(P);
[ x_0, x_1, x_2, x_3 ]
gap> F := (x[1]+x[2])*(x[3]+x[4])+x[4]^2;
x_0*x_2+x_0*x_3+x_1*x_2+x_1*x_3+x_3^2
gap> Q := Quadric(P,F);
Quadric of PG( 3, 8 )
gap> IsSingular(Q);
true
gap> IsCanonical(Q);
false
gap> StandardForm(Q);
x_0^2+x_1*x_2
gap> Nature(Q);
[ 2, 1 ]
gap> F := x[1]^2+x[2]^2+x[3]*x[4]+x[1]*x[2];
x_0^2+x_0*x_1+x_1^2+x_2*x_3
gap> Q := Quadric(P,F);
Quadric of PG( 3, 8 )
gap> IsSingular(Q);
false
gap> IsCanonical(Q);
true
gap> StandardForm(Q);
x_0^2+x_0*x_1+x_1^2+x_2*x_3
gap> Nature(Q);
```

```
[ 3, 0 ]
gap> P := ProjectiveGeometry(4,81);
PG( 4, 81 )
gap> x := Indets(P);
[ x_0, x_1, x_2, x_3, x_4 ]
gap> F := x[1]^9*x[2]+x[1]*x[2]^9+x[3]^9*x[4]+x[3]*x[4]^9+x[5]^10;
x_0^9*x_1+x_0*x_1^9+x_2^9*x_3+x_2*x_3^9+x_4^10
gap> H := HermitianVariety(P,F);
Hermitian Variety of PG( 4, 81 )
gap> IsSingular(H);
false
gap> IsCanonical(H);
false
gap> StandardForm(H);
x_0^10+x_1^10+x_2^10+x_3^10+x_4^10
gap> Nature(H);
4
gap> F := x[1]^10+x[2]^10+x[3]^9*x[4]+x[3]*x[4]^9+x[4]^9*x[5]+x[4]*x[5]^9;
x_0^10+x_1^10+x_2^9*x_3+x_2*x_3^9+x_3^9*x_4+x_3*x_4^9
gap> H := HermitianVariety(P,F);
Hermitian Variety of PG( 4, 81 )
gap> IsSingular(H);
true
gap> IsCanonical(H);
false
gap> StandardForm(H);
x_0^10+x_1^10+x_2^10+x_3^10
gap> Nature(H);
3
```

5.5 Base transition

1 ▶ `BaseTransition( K )` A

returns the matrix D of the base transition $(x_0, x_1, \ldots, x_n) = (y_0, y_1, \ldots, y_n)D$ for which the equation of the quadric or Hermitian variety K becomes (with respect to the new base corresponding to the coordinates $(y_0, y_1, \ldots, y_n)$) a standard form.

```
gap> P := ProjectiveGeometry(2,5);
PG( 2, 5 )
gap> x := Indets(P);
[ x_0, x_1, x_2 ]
gap> F := x[2]^2-x[1]*x[3];
-x_0*x_2+x_1^2
gap> Q := Quadric(P,F);
Quadric of PG( 2, 5 )
gap> BaseTransition(Q);
[ [ 0*Z(5), Z(5)^3, 0*Z(5) ], [ Z(5)^2, 0*Z(5), 0*Z(5) ],
  [ 0*Z(5), 0*Z(5), Z(5)^2 ] ]
gap> P := ProjectiveGeometry(3,9);
PG( 3, 9)
gap> x := Indets(P);
[ x_0, x_1, x_2, x_3 ]
```

```
gap> F := x[1]^3*x[2]+x[1]*x[2]^3+x[3]^3*x[4]+x[3]*x[4]^3;
x_0^3*x_1+x_0*x_1^3+x_2^3*x_3+x_2*x_3^3
gap> H := HermitianVariety(P,F);
Hermitian Variety of PG( 3, 9 )
gap> BaseTransition(H);
[ [ Z(3)^0, Z(3^2), 0*Z(3), 0*Z(3) ], [ Z(3^2)^3, Z(3^2)^2, 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, Z(3^2) ], [ 0*Z(3), 0*Z(3), Z(3^2)^3, Z(3^2)^2 ] ]
```

5.6 Singular space, singular points

1 ▶ `SingularSpace( K )` A

This function returns the subspace spanned by the singular points of the quadric or Hermitian variety K. If K is non-singular, the empty subspace is returned. If the function returns a subspace S, you can find all singular points of K by `ProjectivePoints(`S`)`.

```
gap> P := ProjectiveGeometry(4,7);
PG( 4, 7 )
gap> x := Indets(P);
[ x_0, x_1, x_2, x_3, x_4 ]
gap> F := x[1]*x[2]+x[5]^2;
x_0*x_1+x_4^2
gap> Q := Quadric(P,F);
Quadric of PG( 4, 7 )
gap> S := SingularSpace(Q);
Subspace of PG( 4, 7 )
gap> Dimension(S);
1
gap> ProjectivePoints(S);
[ [ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7)^0, Z(7)^0, 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7)^0, Z(7), 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7)^0, Z(7)^2, 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7)^0, Z(7)^3, 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7)^0, Z(7)^4, 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7)^0, Z(7)^5, 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7) ] ]
gap> S := SingularSpace(StandardParabolicQuadric(P));
Subspace of PG( 4, 7 )
gap> Dimension(S);
-1
gap> IsEmpty(S);
true
gap> P := ProjectiveGeometry(3,9);
PG( 3, 9 )
gap> H := HermitianVariety(P,2);
Hermitian Variety of PG( 3, 9 )
gap> S := SingularSpace(H);
Subspace of PG( 3, 9 )
gap> Dimension(S);
0
gap> ProjectivePoint(S);
[ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ]
```

```
gap> IsEmpty(SingularSpace(StandardHermitianVariety(P)));
true
```

5.7 Kernel of quadrics

Suppose Q is a quadric in $\mathrm{PG}(n,q)$, q even. The *kernel space* of Q is the intersection of the tangent hyperplanes in all non-singular points of Q. If n is odd, one can easily prove that the kernel space is empty when Q is non-singular. If n is even, the kernel space is non-empty even when Q is non-singular. It is also easy to prove that the singular space is always a subspace of the kernel space. So we come to the next definition. The *kernel points* of Q are all the points in the kernelspace not in the singular space of Q.

1 ▶ `KernelSpace( Q )` A

returns the kernel space of the quadric Q in $\mathrm{PG}(n,q)$, q even.

2 ▶ `KernelPoints( Q )` A

returns a list of points that are in the kernel space but not in the singular space of Q.

```
gap> P := ProjectiveGeometry(2,2);
PG( 2, 2 )
gap> Q := StandardParabolicQuadric(P);
Quadric of PG( 2, 2 )
gap> Print(Q,"\n");
Quadric of PG( 2, 2 )
x_0^2+x_1*x_2
gap> S := KernelSpace(Q);
Subspace of PG( 2, 2 )
gap> Dimension(S);
0
gap> ProjectivePoint(S);
[ Z(2)^0, 0*Z(2), 0*Z(2) ]
gap> IsSingular(Q);
false
gap> KernelPoints(Q);
[ [ Z(2)^0, 0*Z(2), 0*Z(2) ] ]
gap> P := ProjectiveGeometry(3,4);
PG( 3, 4 )
gap> Q := Quadric(P,[3,2]);
Quadric of PG( 3, 4 )
gap> Print(Q,"\n");
Quadric of PG( 3, 4 )
x_0*x_1+x_2*x_3
gap> Dimension(KernelSpace(Q));
-1
gap> Q := Quadric(P,[2,1]);
Quadric of PG( 3, 4 )
gap> Print(Q,"\n");
Quadric of PG( 3, 4 )
x_0^2+x_1*x_2
gap> S := KernelSpace(Q);
Subspace of PG( 3, 4 )
gap> ProjectivePoints(S);
[ [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ],
```

```
  [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2^2) ], [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2^2)^2 ],
  [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ] ]
gap> ProjectivePoints(SingularSpace(Q));
[ [ 0*Z(2), 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> KernelPoints(Q);
[ [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2^2) ],
  [ Z(2)^0, 0*Z(2), 0*Z(2), Z(2^2)^2 ], [ Z(2)^0, 0*Z(2), 0*Z(2), 0*Z(2) ] ]
```

5.8 Projective points and base points

1 ▶ `ProjectivePoints( `K` )` A

returns a list with all the projective points of the quadric or Hermitian variety K.

```
gap> P := ProjectiveGeometry(3,4);
PG( 3, 4 )
gap> Q := Quadric(P,[3,0]);
Quadric of PG( 3, 4 )
gap> list := ProjectivePoints(Q);;
gap> Length(list);
17
gap> list;
[ [ 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), Z(2)^0, Z(2^2), Z(2)^0 ], [ 0*Z(2), Z(2)^0, Z(2)^0, Z(2^2) ],
  [ 0*Z(2), Z(2)^0, Z(2^2)^2, Z(2^2)^2 ], [ Z(2)^0, Z(2)^0, Z(2^2), Z(2)^0 ],
  [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2^2) ], [ Z(2)^0, Z(2)^0, Z(2^2)^2, Z(2^2)^2 ],
  [ Z(2)^0, Z(2^2), Z(2^2), Z(2)^0 ], [ Z(2)^0, Z(2^2), Z(2)^0, Z(2^2) ],
  [ Z(2)^0, Z(2^2), Z(2^2)^2, Z(2^2)^2 ], [ Z(2)^0, Z(2^2)^2, Z(2)^0, Z(2)^0 ]
    , [ Z(2)^0, Z(2^2)^2, Z(2^2)^2, Z(2^2) ],
  [ Z(2)^0, Z(2^2)^2, Z(2^2), Z(2^2)^2 ], [ Z(2)^0, 0*Z(2), Z(2)^0, Z(2)^0 ],
  [ Z(2)^0, 0*Z(2), Z(2^2)^2, Z(2^2) ], [ Z(2)^0, 0*Z(2), Z(2^2), Z(2^2)^2 ] ]
gap> Q := Quadric(P,[2,1]);
Quadric of PG( 3, 4 )
gap> ProjectivePoints(Q);
[ [ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, Z(2^2)^2, Z(2^2), 0*Z(2) ], [ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ],
  [ Z(2)^0, Z(2^2), Z(2^2)^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, Z(2)^0 ], [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2^2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0, Z(2^2)^2 ], [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2^2) ], [ 0*Z(2), Z(2)^0, 0*Z(2), Z(2^2)^2 ],
  [ Z(2)^0, Z(2^2)^2, Z(2^2), Z(2)^0 ], [ Z(2)^0, Z(2^2)^2, Z(2^2), Z(2^2) ],
  [ Z(2)^0, Z(2^2)^2, Z(2^2), Z(2^2)^2 ], [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
  [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2^2) ], [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2^2)^2 ],
  [ Z(2)^0, Z(2^2), Z(2^2)^2, Z(2)^0 ], [ Z(2)^0, Z(2^2), Z(2^2)^2, Z(2^2) ],
  [ Z(2)^0, Z(2^2), Z(2^2)^2, Z(2^2)^2 ] ]
gap> P := ProjectiveGeometry(4,4);
PG( 4, 4 )
gap> H := HermitianVariety(P,4);
Hermitian Variety of PG( 4, 4 )
gap> list := ProjectivePoints(H);;
gap> Length(list);
165
```

It is well known and easy to prove that every singular quadric or Hermitian variety F is a cone $\pi_s F'$ with vertex the singular space π_s and base F', a non-singular quadric or Hermitian variety in a subspace π_r with π_s and π_r having an empty intersection and spanning the whole space. The quadric F' may be empty.

2 ► `BasePoints( K )` A

returns the projective points of the non-singular base K'. If K is non-singular, then `BasePoints` will return all projective points.

```
gap> P := ProjectiveGeometry(3,3);
PG( 3, 3 )
gap> Q := Quadric(P,[2,1]);
Quadric of PG( 3, 3 )
gap> BasePoints(Q);
[ [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0, 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), 0*Z(3) ] ]
```

3 ► `BaseVec( K )` A

returns the base points of K as vectors.

5.9 Miscellaneous functions

1 ► `p in K`
► `\in( p, K )` O

tests whether the projective point p is a point of the quadric or Hermitian variety K.

```
gap> P := ProjectiveGeometry(3,5);
PG( 3, 5 )
gap> Q := Quadric(P,[3,2]);
Quadric of PG( 3, 5 )
gap> p := ProjectivePoint(P,[1,-1,-1,1]*Z(5)^0);
[ Z(5)^0, Z(5)^2, Z(5)^2, Z(5)^0 ]
gap> p in Q;
false
gap> p := ProjectivePoint(P,[1,1,-1,1]*Z(5)^0);
[ Z(5)^0, Z(5)^0, Z(5)^2, Z(5)^0 ]
gap> p in Q;
true
```

2 ► `IsSingularPoint( K, p )` O

returns true if p is a singular point of the quadric or Hermitian variety K.

```
gap> P := ProjectiveGeometry(3,9);
PG( 3, 9 )
gap> H := HermitianVariety(P,2);
Hermitian Variety of PG( 3, 9 )
gap> p := ProjectivePoint(P,[0,0,0,1]*Z(9)^0);
[ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ]
gap> IsSingularPoint(H,p);
true
```

3 ► `LinesThroughPoint( K, p )` O

returns the list of lines through the point p of the non-singular quadric or non-singular Hermitian variety K that lie completely on K.

```
gap> P := ProjectiveGeometry(3,2);
PG( 3, 2 )
gap> Q := Quadric(P,[3,2]);
Quadric of PG( 3, 2 )
gap> p := ProjectivePoint(P,[1,1,-1,1]*Z(2)^0);
[ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ]
gap> LinesThroughPoint(Q,p);
[ Subspace of PG( 3, 2 ), Subspace of PG( 3, 2 ) ]
```

5.10 Stabilizer groups

1 ▶ `StabilizerGroup( K )` A

Suppose we are working in P=PG(n, q), with K a quadric or Hermitian variety in P. This function returns the subgroup of PΓL$(n+1, q)$ stabilizing the **set** of points of K. So the set of points of K is invariant for the action of the returned group. Furthermore the group is returned as subgroup of a collineation group, so it is not a permutation group. Nevertheless the same functions as for the group PΓL$(n+1, q)$ are available. (`Size`, `GeneratorsOfGroup` and `AsPermutationGroup` are the most important).

```
gap> P := ProjectiveGeometry(2,9);
PG( 2, 9 )
gap> x := Indets(P);
[ x_0, x_1, x_2 ]
gap> F := x[2]^2-x[1]*x[3];
-x_0*x_2+x_1^2
gap> Q := Quadric(P,F);
Quadric of PG( 2, 9 )
gap> G := StabilizerGroup(Q);
<group of size 1440 with 4 generators>
gap> GeneratorsOfGroup(G);
[ 1 ; [ [ 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 ] ],
  1 ; [ [ Z(3)^0, 0*Z(3), 0*Z(3) ], [ Z(3^2)^5, Z(3^2)^2, 0*Z(3) ],
      [ Z(3^2)^2, Z(3^2)^3, Z(3) ] ],
  0 ; [ [ Z(3)^0, 0*Z(3), 0*Z(3) ], [ Z(3)^0, Z(3^2)^7, 0*Z(3) ],
      [ Z(3)^0, Z(3^2)^3, Z(3^2)^6 ] ],
  1 ; [ [ 0*Z(3), 0*Z(3), Z(3)^0 ], [ 0*Z(3), Z(3^2)^2, Z(3^2)^6 ],
      [ Z(3), Z(3), Z(3) ] ] ]
gap> IsSubgroup(PGammaL(3,9),G);
true
gap> H := AsPermutationGroup(G);
<permutation group of size 1440 with 4 generators>
gap> P := ProjectiveGeometry(4,4);
PG( 4, 4 )
gap> H := HermitianVariety(P,3);
Hermitian Variety of PG( 4, 4 )
gap> Print(H,"\n");
Hermitian Variety of PG( 4, 4 )
x_0^3+x_1^3+x_2^3+x_3^3
gap> G := StabilizerGroup(H);
<group of size 39813120 with 13 generators>
gap> IsSubgroup(PGammaL(5,4),G);
true
```

6 Orthogonal and Unitary groups

6.1 Orthogonal groups

1 ▶ `PGammaOPlus( n, q )` F
▶ `PGammaOMinus( n, q )` F

Suppose n even, $n \geq 2$. The first function returns the stabilizer group of the non-singular standard hyperbolic quadric in $\mathrm{PG}(n-1,q)$. The returned group is a subgroup of $\mathrm{P\Gamma L}(n,q)$. Again all functions available for collineation groups are available for the returned group.

The second function is analogous to the first but returns the stabilizer group of the non-singular standard elliptic quadric in $\mathrm{PG}(n-1,q)$.

2 ▶ `PGammaO( n, q )` F

Suppose n odd, $n \geq 2$. This function returns the stabilizer group of the non-singular standard parabolic quadric in $\mathrm{PG}(n-1,q)$ This function is also analogous to the functions mentioned above.

```
gap> G := PGammaO(3,5);
PGammaO( 3, 5 )
gap> Size(G);
120
gap> GeneratorsOfGroup(G);
[ 0 ; [ [ Z(5)^0, Z(5)^2, 0*Z(5) ], [ 0*Z(5), Z(5)^2, 0*Z(5) ],
      [ Z(5), Z(5)^2, Z(5)^2 ] ],
  0 ; [ [ Z(5)^0, Z(5)^0, 0*Z(5) ], [ 0*Z(5), Z(5)^3, 0*Z(5) ],
      [ Z(5)^2, Z(5), Z(5) ] ],
  0 ; [ [ Z(5)^0, Z(5), 0*Z(5) ], [ 0*Z(5), Z(5)^2, 0*Z(5) ],
      [ Z(5)^0, Z(5)^0, Z(5)^2 ] ],
  0 ; [ [ Z(5)^0, 0*Z(5), Z(5)^3 ], [ 0*Z(5), 0*Z(5), Z(5)^0 ],
      [ Z(5)^0, Z(5)^0, Z(5)^2 ] ] ]
gap> AsPermutationGroup(G);
<permutation group of size 120 with 4 generators>
gap> G := PGammaOPlus(4,8);
PGammaO+( 4, 8 )
gap> Size(G);
1524096
gap> Length(GeneratorsOfGroup(G));
6
gap> G := PGammaOMinus(4,3);
PGammaO-( 4, 3 )
gap> Size(G);
1440
gap> GeneratorsOfGroup(G);
[ 0 ; [ [ 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), Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3) ] ],
  0 ; [ [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3), 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ], [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ]
    ,
  0 ; [ [ Z(3)^0, 0*Z(3), Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ]
        , [ 0*Z(3), 0*Z(3), Z(3), 0*Z(3) ], [ Z(3), 0*Z(3), Z(3)^0, Z(3) ] ],
  0 ; [ [ 0*Z(3), Z(3)^0, Z(3)^0, 0*Z(3) ], [ Z(3)^0, 0*Z(3), Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ], [ Z(3), Z(3)^0, Z(3)^0, Z(3)^0 ] ],
  0 ; [ [ Z(3)^0, Z(3), Z(3)^0, 0*Z(3) ], [ Z(3)^0, Z(3)^0, Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), Z(3), 0*Z(3) ], [ 0*Z(3), Z(3), Z(3), Z(3)^0 ] ],
  0 ; [ [ 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ], [ 0*Z(3), Z(3)^0, Z(3)^0, Z(3) ] ] ]
```

6.2 Unitary Group

1 ▶ `PGammaU( `n`, `q` )` F

This function returns the stabilizer group of the non-singular standard Hermitian variety in $\mathrm{PG}(n-1,q)$. The returned group is a subgroup of $\mathrm{P\Gamma L}(n,q)$. Again all functions available for collineation groups are available for the returned group.

```
gap> G := PGammaU(3,9);
PGammaU( 3, 9 )
gap> Size(G);
12096
gap> GeneratorsOfGroup(G);
[ 1 ; [ [ 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) ] ],
  1 ; [ [ 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(3^2)^6 ] ],
  1 ; [ [ 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(3^2)^2 ] ],
  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)^0 ] ],
  0 ; [ [ Z(3)^0, Z(3^2)^6, 0*Z(3) ], [ Z(3^2)^6, Z(3)^0, 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), Z(3^2)^5 ] ],
  1 ; [ [ Z(3)^0, Z(3), Z(3^2)^7 ], [ Z(3), Z(3)^0, Z(3^2)^7 ],
      [ Z(3^2)^3, Z(3^2)^3, 0*Z(3) ] ],
  0 ; [ [ Z(3)^0, Z(3^2)^3, Z(3^2)^3 ], [ Z(3^2)^7, Z(3^2)^7, Z(3) ],
      [ Z(3^2), Z(3^2)^6, Z(3^2) ] ] ]
gap> AsPermutationGroup(G);
<permutation group of size 12096 with 7 generators>
gap> IsSubgroup(PGammaL(3,9),G);
true
```

Bibliography

[Hir98] J.W.P. Hirschfeld. *Projective Geometries over finite fields (Second Edition)*. Clarendon Press - Oxford, 1998.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "`PermutationCharacter`" comes before "permutation group".