

Handbook of Magma Functions

Part XIX: Geometry

Version 2.30-1

Sydney

November 10, 2026

Editors

John Cannon *Wieb Bosma* *Claus Fieker* *Allan Steel*

Handbook Contributors

Eran Assaf *Geoff Bailey* *Wieb Bosma* *Gavin Brown* *Nils Bruin* *John Cannon* *Jon Carlson* *Scott Contini* *Edgar Costa* *Bruce Cox* *Brendan Creutz* *Steve Donnelly* *Tim Dokchitser* *Willem de Graaf* *Andreas-Stephan Elsenhans* *Claus Fieker* *Damien Fisher* *Volker Gebhardt* *Sergei Haller* *Michael Harrison* *Florian Hess* *Derek Holt* *David Howden* *Al Kasprzyk* *Markus Kirschmer* *David Kohel* *Axel Kohnert* *Dimitri Leemans* *Paulette Lieby* *Graham Matthews* *Scott Murray* *Eamonn O'Brien* *Dan Roozemond* *Ben Smith* *Bernd Souvignier* *William Stein* *Allan Steel* *Damien Stehlé* *Nicole Sutherland* *Don Taylor* *Bill Unger* *Alexa van der Waall* *Paul van Wamelen* *Helena Verrill* *John Voight* *Mark Watkins* *Greg White*

Production Editors

Wieb Bosma *Claus Fieker* *Allan Steel* *Nicole Sutherland*

HTML Production

Claus Fieker *Allan Steel*

Copyright © 1994–2026 The University of Sydney. All rights reserved.

Contents

154 Finite Planes **1**

154.1 Introduction . . . 1
154.2 Construction of a Plane . . . 1
154.3 The Point-Set and Line-Set of a Plane . . . 3
154.4 The Set of Points and Set of Lines . . . 7
154.5 The Defining Points of a Plane . . . 7
154.6 Subplanes . . . 8
154.7 Structures Associated with a Plane . . . 9
154.8 Numerical Invariants of a Plane . . . 10
154.9 Properties of Planes . . . 10
154.10 Identity and Isomorphism . . . 11
154.11 The Connection between Projective and Affine Planes . . . 11
154.12 Operations on Points and Lines . . . 12
154.13 Arcs . . . 16
154.14 Unitals . . . 18
154.15 The Collineation Group of a Plane . . . 19
154.16 Translation Planes . . . 25
154.17 Planes and Designs . . . 25
154.18 Planes, Graphs, and Codes . . . 26

155 Incidence Geometry **28**

155.1 Introduction . . . 28
155.2 Construction of Incidence and Coset Geometries . . . 28
155.3 Elementary Invariants . . . 35
155.4 Conversion Functions . . . 36
155.5 Residues . . . 37
155.6 Truncations . . . 37
155.7 Shadows . . . 37
155.8 Shadow Spaces . . . 38
155.9 Automorphism Group and Correlation Group . . . 38
155.10 Properties of Incidence Geometries and Coset Geometries . . . 38
155.11 Intersection Properties of Coset Geometries . . . 39
155.12 Primitivity Properties on Coset Geometries . . . 40
155.13 Diagram of an Incidence Geometry . . . 41
155.14 C-Groups . . . 43
155.15 C^+-Groups . . . 45
155.16 Bibliography . . . 47

156 Convex Polytopes and Polyhedra **48**

156.1 Introduction and First Examples . . . 48
156.2 Polytopes, Cones, and Polyhedra . . . 52
156.3 Basic Combinatorics of Polytopes and Polyhedra . . . 56
156.4 The Combinatorics of Polytopes . . . 59
156.5 Cones and Polyhedra . . . 64
156.6 Toric Lattices . . . 72
156.7 Bibliography . . . 79

References **80**

Index **81**

Chapter 154

Finite Planes

154.1 Introduction

In this chapter we will present the categories of finite projective and affine planes.

The category names for projective and affine planes are `PlaneProj` and `PlaneAff` respectively. Within each of these categories we have what we will call *classical* planes — those which are defined by a vector space of dimension 2 (for affine planes) or 3 (for projective planes).

Some functions documented here apply to all types of planes, others are specific to projective, affine, or classical planes. It should be clear which is the case for each entry.

154.1.1 Planes in Magma

A point of a plane is considered to be a special object, and so points are given their own special type, `PlanePt`, in Magma. This allows the points of a plane to be defined over any type of Magma object, and also improves the efficiency of the code.

A special structure called the *point–set* acts as the parent structure for points. A point is created by coercing an appropriate Magma object into the point–set. It is also possible the get the i-th point, or a random point, from the point–set.

Similarly, lines of a plane have a special type `PlaneLn`, and the *line–set* acts as their parent structure. Lines can be created by coercing a suitable object into the line–set, or by asking for the i-th line, or a random line, from the line–set.

154.2 Construction of a Plane

All functions which create a plane return three values:

(i) the plane itself;

(ii) the point–set of the plane;

(iii) the line–set of the plane.

These "sets" ((ii) and (iii)) are used as the parent structures for points and lines respectively, and are explained more fully in the next section.

```
FiniteProjectivePlane< v | X : parameters >: RngIntElt, List -> PlaneProj
FiniteProjectivePlane< V | X : parameters >: SetIndx, List -> PlaneProj
```

```
Check: BoolElt                    Default: true
```

Construct the projective plane P having as point set the indexed set V (or $\{@1, 2, \ldots, v@\}$ if an integer v is given), and as line set $L = \{L_1, L_2, \ldots, L_b\}$ given by the list X. The value of X must be either:

(a) A list of subsets of the set V.

(b) A sequence, set, or indexed set of subsets of V.

(c) A list of lines of an existing plane.

(d) A sequence, set, or indexed set of lines of an existing plane.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be defined over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the plane P.

(g) A set of codewords of a linear code with length v. The line set of P is taken to be the set of supports of the codewords.

The optional boolean argument `Check` indicates whether or not to check that the given data satisfies the projective plane axioms.

`FiniteProjectivePlane(W): ModTupFld -> PlaneProj`

`FiniteProjectivePlane(F): FldFin -> PlaneProj`

`FiniteProjectivePlane(q): RngIntElt -> PlaneProj`

Given a 3–dimensional vector space W defined over the field $F = \mathbf{F}_q$, construct the classical projective plane defined by the one–dimensional and two–dimensional subspaces of W.

`FiniteAffinePlane< v | X : parameters >: RngIntElt, List -> PlaneAff`

`FiniteAffinePlane< V | X : parameters >: SetIndx, List -> PlaneAff`

Check: BoolElt	Default: true

Construct the affine plane P having as point set the indexed set V (or $\{@1, 2, \ldots, v@\}$ if an integer v is given), and as line set $L = \{L_1, L_2, \ldots, L_b\}$ given by the list X. The value of X must be either:

(a) A list of subsets of the set V.

(b) A sequence, set, or indexed set of subsets of V.

(c) A list of lines of an existing plane.

(d) A sequence, set, or indexed set of lines of an existing plane.

(e) A combination of the above.

(f) A $v \times b$ $(0,1)$-matrix A, where A may be defined over any coefficient ring. The matrix A will be interpreted as the incidence matrix for the plane P.

(g) A set of codewords of a linear code with length v. The line set of P is taken to be the set of supports of the codewords.

The optional boolean argument `Check` indicates whether or not to check that the given data satisfies the affine plane axioms.

`FiniteAffinePlane(W): ModFld -> PlaneAff`

`FiniteAffinePlane(F): FldFin -> PlaneProj`

`FiniteAffinePlane(q): RngIntElt -> PlaneProj`

Given a 2–dimensional vector space W defined over the field $F = \mathbf{F}_q$, construct the classical affine plane defined by the cosets of the subspaces of W.

Example: Classical Projective Plane of Order 3 (ex-9ba67e)

The classical projective plane of order 3 can be constructed by the following statement:

```
> P, V, L := FiniteProjectivePlane(3);
> P;
Projective Plane PG(2, 3)
> V;
Point-set of Projective Plane PG(2, 3)
> L;
Line-set of Projective Plane PG(2, 3)
```

A non-classical affine plane of order 2 can be constructed in the following way:

```
> A := FiniteAffinePlane< 4 | Setseq(Subsets({1, 2, 3, 4}, 2)) >;
> A: Maximal;
Affine Plane of order 2
Points: {@ 1, 2, 3, 4 @}
Lines:
    {1, 3},
    {1, 4},
    {2, 4},
    {2, 3},
    {1, 2},
    {3, 4}
```

To demonstrate the use of the `Check` argument, we recreate the classical projective plane of order 16 with `Check := true` (the default) and `Check := false`.

```
> P, V, L := FiniteProjectivePlane(16);
> time P2 := FiniteProjectivePlane<
>     Points(P) | {Set(l): l in L} : Check := true >;
Time: 10.769
> time P2 := FiniteProjectivePlane<
>     Points(P) | {Set(l): l in L} : Check := false >;
Time: 0.030
```

154.3 The Point-Set and Line-Set of a Plane

154.3.1 Introduction

An affine or projective plane in Magma consists of three objects: the plane P itself, the *point–set* V of P, and the *line–set* L of P.

Although called the point–set and line–set, V and L are not actual Magma sets. They simply act as the parent structures for the points and lines (respectively) of the plane P, enabling easy creation of these objects via the `!` and `.` operators.

The point–set V belongs to the Magma category `PlanePtSet`, and the line–set L to the category `PlaneLnSet`.

In this section, the functions used to create point–sets, line–sets and the points, and lines themselves are described.

154.3.2 Creating Point-Sets and Line-Sets

As mentioned above, the point–set and line–set are returned as the second and third arguments of any function which creates a plane. They can also be created via the following two functions.

`PointSet(P): Plane -> PlanePtSet`

Given a plane P, return the point–set V of P.

`LineSet(P): Plane -> PlaneLnSet`

Given a plane P, return the line–set L of P.

154.3.3 Using the Point-Set and Line-Set to Create Points and Lines

For efficiency and clarity, the points and lines of a plane are given special types in Magma. The category names for points and lines are `PlanePt` and `PlaneLn` respectively. They can be created in the following ways.

`V . i: PlanePtSet, RngIntElt -> PlanePt`

Given the point–set V of a plane P and an integer i, return the i-th point of P.

`V ! [a, b, c]: PlanePtSet, SeqEnum -> PlanePt`

Given the point–set V of a classical projective plane $P = PG_2(K)$, and elements a, b, c of the finite field K, create the projective point $(a : b : c)$ in the plane P.

`V ! [a, b]: PlanePtSet, SeqEnum -> PlanePt`

Given the point–set V of a classical affine plane $P = AG_2(K)$, and elements a, b of the finite field K, create the point (a, b) in the plane P.

`V ! x: PlanePtSet, Elt -> PlanePt`

Given the point–set V of a plane P, return the point of P corresponding to the element x, which should be coercible into the underlying point set for P. (In the case of classical planes, x should be coercible to a vector.)

`Representative(V): PlanePtSet -> PlanePt`

`Rep(V): PlanePtSet -> PlanePt`

Given the point–set V of a plane P, return a representative point of P.

`Random(V): PlanePtSet -> PlanePt`

Given the point–set V of a plane P, return a random point of P.

`L . i: PlanePtSet, RngIntElt -> PlanePt`

Given the line–set L of a plane P and an integer i, return the i-th line of P.

`L ! [a, b, c]: PlaneLnSet, SeqEnum -> PlaneLn`

Given the line set L of a classical plane P defined over a finite field K, and elements a, b, c of K, create the line $\langle a : b : c \rangle$ (i.e. the line given by the equation $ax + by + cz = 0$ if P is projective, or $ax + by + c = 0$ if P is affine).

`L ! [m, b]: PlaneLnSet, SeqEnum -> PlaneLn`

Given the line set L of a classical affine plane $P = AG_2(K)$, and elements m, b of the finite field K, create the affine line $y = mx + b$ in P.

`L ! S: PlaneLnSet, SetEnum -> PlaneLn`

`L ! S: PlaneLnSet, SeqEnum -> PlaneLn`

Given the line–set L of a plane P and a set or sequence S of collinear points of P, return the line containing the points of S.

`L ! l: PlaneLnSet, PlaneLn -> PlaneLn`

Given the line–set L of a plane P and a line l of a (possibly) different plane (generally a subplane of P), return the line of P corresponding to l.

`Representative(L): PlaneLnSet -> PlaneLn`

`Rep(L): PlaneLnSet -> PlaneLn`

Given the line–set L of a plane P, return a representative line of P.

`Random(L): PlaneLnSet -> PlaneLn`

Given the line–set L of a plane P, return a random line of P.

Example: Creating Points and Lines (ex-afb4c8)

The following example shows how points and lines of a plane can be created. First we study a classical projective plane.

```
> P, V, L := FiniteProjectivePlane(5);
> V;
Point-set of Projective Plane PG(2, 5)
> L;
Line-set of Projective Plane PG(2, 5)
```

Create the third point of P:

```
> V.3;
( 0 : 0 : 1 )
```

Create the point $(1 : 2 : 3)$ of P:

```
> V![1, 2, 3];
( 1 : 2 : 3 )
```

Choose a random point of P:

```
> Random(V);
( 1 : 0 : 0 )
> Random(V);
( 0 : 0 : 1 )
```

Create the sixth line of P:

```
> L.6;
< 1 : 1 : 3 >
```

Create the line of P given by the equation $4x + 3y + 2z = 0$:

```
> L![4, 3, 2];
< 1 : 2 : 3 >
```

Create the line of P containing the points $(0 : 0 : 1)$ and $(0 : 1 : 0)$:

```
> L![ V | [0, 0, 1], [0, 1, 0] ];
< 1 : 0 : 0 >
```

Get a representative from the line-set of P, and a random line:

```
> Rep(L);
< 1 : 0 : 0 >
> Random(L);
< 1 : 2 : 4 >
```

Now we look at a non-classical plane.

```
> V := {2, 4, 6, 8};
> A, P, L := FiniteAffinePlane< SetToIndexedSet(V) | Setseq(Subsets(V, 2)) >;
> A: Maximal;
Affine Plane of order 2
Points: {@ 2, 4, 6, 8 @}
```

(continues on next page)

(continued from previous page)

```
Lines:
    {6, 8},
    {2, 6},
    {2, 8},
    {2, 4},
    {4, 6},
    {4, 8}
> P;
Point-set of Affine Plane of order 2
> L;
Line-set of Affine Plane of order 2
```

Get the third point of A:

```
> P.3;
6
```

Create the point of A given by the integer 4:

```
> P!4;
4
```

Get a representative from the point–set of A:

```
> Rep(P);
2
```

Get the third line of A:

```
> L.3;
{2, 8}
```

Create the line of A containing the integers 2 and 6:

```
> L![2, 6];
{2, 6}
```

Choose a random line from A:

```
> Random(L);
{6, 8}
```

154.3.4 Retrieving the Plane from Points, Lines, Point-Sets, and Line-Sets

The `ParentPlane` function allows you to access the plane to which a point, line, point–set, or line–set belongs.

`ParentPlane(V): PlanePtSet -> Plane, PlanePtSet, PlaneLnSet`

The plane P for which V is the point–set.

`ParentPlane(L): PlaneLnSet -> Plane, PlanePtSet, PlaneLnSet`

The plane P for which L is the line–set.

`ParentPlane(p): PlanePt -> Plane, PlanePtSet, PlaneLnSet`

The plane P for which p is a point.

`ParentPlane(l): PlaneLn -> Plane, PlanePtSet, PlaneLnSet`

The plane P for which l is a line.

154.4 The Set of Points and Set of Lines

It may sometimes be desirable to have an explicitly enumerated set of the points or lines of a plane, as opposed to the point–set and line–set, which aren't true Magma sets. The following functions have been provided for this purpose.

`Points(P): Plane -> { PlanePt }`

An indexed set E whose elements are the points of the plane P. Note that this creates a standard indexed set and not the point–set of P, in contrast to the function `PointSet`.

`Lines(P): PlaneLnSet -> { PlaneLn}`

An indexed set containing the lines of the plane P. In contrast to the function `LineSet`, this function returns the collection of lines of P in the form of a standard indexed set.

154.5 The Defining Points of a Plane

Points of a plane have their own special type in Magma (`PlanePt`). However it may sometimes be necessary to return to the objects used to define the points. These are the elements of the indexed set originally used to create the plane, or, in the case of a classical plane, the vector space elements which define the points of the plane.

The functions listed in this section provide the means to do this.

`Support(P): Plane -> { Elt }`

An indexed set E which is the underlying point set of the plane P (i.e. the elements of the set have their "real" types; they are no longer a "PlanePt" of P).

`Support(l): PlaneLn -> SetEnum`

The set of underlying points contained in the line l of a plane P (i.e. the elements of the set have their "real" types; they are no longer a "PlanePt" of P).

`Support(P, p): Plane, PlanePt -> .`

`Support(p): PlanePt -> .`

The Magma object corresponding to the point p of a plane P.

Example: Points versus Support of a Plane (ex-d7a158)

The following code shows the difference between the `Points` and `Support` functions.

```
> A := FiniteAffinePlane< {@ 3, 4, 5, 6 @} | {3, 4}, {3, 5}, {3, 6},
>                                            {4, 5}, {4, 6}, {5, 6} >;
> Pts := Points(A);
> Supp := Support(A);
> Pts, Supp;
{@ 3, 4, 5, 6 @}
{@ 3, 4, 5, 6 @}
```

These sets look the same, but the elements have different types:

```
> Universe(Pts);
Point-set of Affine Plane of order 2
> Universe(Supp);
Integer Ring
```

The classical plane case is slightly different. Here the support is a set of vectors.

```
> P, V, L := FiniteProjectivePlane(2);
> Points(P);
{@ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 1 : 0 ),
```

(continues on next page)

(continued from previous page)

```
   ( 0 : 1 : 1 ), ( 1 : 1 : 1 ), ( 1 : 0 : 1 ) @}
> Support(P);
{@
    (1 0 0),
    (0 1 0),
    (0 0 1),
    (1 1 0),
    (0 1 1),
    (1 1 1),
    (1 0 1)
@}
> Universe(Points(P));
Point-set of Projective Plane PG(2, 2)
> Universe(Support(P));
Full Vector space of degree 3 over GF(2)
> l := Random(L);
> l;
< 0 : 0 : 1 >
> Support(l);
{
    (1 0 0),
    (0 1 0),
    (1 1 0)
}
```

154.6 Subplanes

The `sub` constructor allows subplanes of a projective or affine plane to be created. For classical planes, the `SubfieldSubplane` function is also provided.

`sub<P | L>: Plane, List -> Plane`

Given a plane P, construct the subplane of P generated by the points specified by L, where L is a list of one or more items of the following types:

(a) A point of P;

(b) A set or sequence of points of P;

(c) A subplane of P;

(d) A set or sequence of subplanes of P.

The set S of points defined by the list L must include a quadrangle if P is a projective plane and three non-collinear points if P is an affine plane. The function returns the smallest subplane of P containing S.

`SubfieldSubplane(P, F): Plane, FldFin -> Plane, PlanePtSet, PlaneLnSet`

The plane obtained from the classical plane P by taking only those points of P which have all coordinates lying in F, where F must be a subfield of `Field(P)`.

Example: Subplane Generated by a Quadrangle (ex-6d2e99)

In the plane $PG_2(4)$, the points $(1:0:0), (0:1:0), (0:0:1)$, and $(1:w:1)$, where w is a primitive element of $\mathbf{F}_4$, form a quadrangle. We form the subplane of $PG_2(4)$ generated by this quadrangle.

```
> K<w> := GF(4);
> P, V, L := FiniteProjectivePlane(K);
```

(continues on next page)

(continued from previous page)

```
> S := sub< P | [ V | [1, 0, 0], [0, 1, 0], [0, 0, 1], [1, w, 1] ] >;
> S: Maximal;
Projective Plane of order 2
Points: {@ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : w : 0 ),
( 1 : 0 : 1 ), ( 1 : w : 1 ), ( 0 : 1 : w^2 ) @}
Lines:
    {( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 0 : 1 : w^2 )},
    {( 1 : 0 : 0 ), ( 0 : 0 : 1 ), ( 1 : 0 : 1 )},
    {( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 1 : w : 0 )},
    {( 1 : 0 : 0 ), ( 1 : w : 1 ), ( 0 : 1 : w^2 )},
    {( 0 : 1 : 0 ), ( 1 : 0 : 1 ), ( 1 : w : 1 )},
    {( 0 : 0 : 1 ), ( 1 : w : 0 ), ( 1 : w : 1 )},
    {( 1 : w : 0 ), ( 1 : 0 : 1 ), ( 0 : 1 : w^2 )}
```

We next form the subplane of $AG_2(4)$ over $\mathbf{F}_2$.

```
> A := FiniteAffinePlane(4);
> S := SubfieldSubplane(A, GF(2));
> S: Maximal;
Affine Plane AG(2, 2)
> S subset A;
true
```

154.7 Structures Associated with a Plane

There are a number of structures naturally associated with a plane. This section lists some functions for accessing or creating them.

VectorSpace(P): Plane -> ModTupFld

The vector space underlying the classical plane P.

Field(P): Plane -> FldFin

The field over which the classical plane P is defined.

IncidenceMatrix(P): Plane -> AlgMatElt

The incidence matrix of the plane P.

Dual(P): Plane -> Plane, PlanePtSet, PlaneLnSet

The dual of the projective plane P.

Example: Vector Space and Field of a Plane (ex-6a749f)

The functions in this section are illustrated by the following example.

```
> A := FiniteAffinePlane(4);
> VectorSpace(A);
Full Vector space of degree 2 over GF(2^2)
> Field(A);
Finite field of size 2^2
>
> P := FiniteProjectivePlane< 7 | {1, 3, 5}, {1, 2, 7}, {1, 4, 6}, {2, 3, 6},
>                               {2, 4, 5}, {3, 4, 7}, {5, 6, 7} >;
> IP := IncidenceMatrix(P);
> IP;
[1 1 1 0 0 0 0]
[0 1 0 1 1 0 0]
[1 0 0 1 0 1 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 0 1 1 0]
[1 0 0 0 1 0 1]
[0 0 1 1 0 0 1]
[0 1 0 0 0 1 1]
> D := Dual(P);
> D;
Projective Plane of order 2
> IncidenceMatrix(D) eq Transpose(IP);
true
```

154.8 Numerical Invariants of a Plane

Order(P): Plane -> RngIntElt

The order of the plane P.

NumberOfPoints(P): Plane -> RngIntElt

 # V: PlanePtSet -> RngIntElt

The cardinality v of the point–set V of the plane P.

NumberOfLines(P): Plane -> RngIntElt

 # L: PlaneLnSet -> RngIntElt

The cardinality v of the line–set L of the plane P.

pRank(P): Plane -> RngIntElt

The p-rank of the plane P of order p^t.

pRank(P, p): Plane, RngIntElt -> RngIntElt

The p-rank of the plane P.

Example: Numerical Invariants of PG(2,8) (ex-1297f0)

We demonstrate some of the above functions with the plane $PG_2(8)$.

```
> P := FiniteProjectivePlane(8);
> NumberOfLines(P);
73
> Order(P);
8
> pRank(P);
28
> pRank(P, 2);
28
> pRank(P, 3);
72
> pRank(P, 5);
73
```

154.9 Properties of Planes

IsDesarguesian(P): Plane -> BoolElt

Returns **true** if and only if the plane P is a desarguesian plane.

IsSelfDual(P): PlaneProj -> BoolElt

Returns **true** if and only if the projective plane P is self–dual.

154.10 Identity and Isomorphism

Two planes are considered equal if their point sets are equal and they have the same lines.

`P eq Q: Plane, Plane -> BoolElt`

Return `true` if the planes P and Q are equal, otherwise `false`.

`P ne Q: Plane, Plane -> BoolElt`

Return `true` if the planes P and Q are not equal, otherwise `false`.

`IsIsomorphic(P, Q: parameters): Plane, Plane -> BoolElt, Map`

```
AutomorphismGroups: MonStgElt                    Default: "None"
```

Return `true` if the planes P and Q are isomorphic, otherwise `false`. If the planes are isomorphic, an isomorphism f from P onto Q is also returned. The function first computes none, one, or both of the automorphism groups of the left and right planes. This may assist the isomorphism testing. The parameter `AutomorphismGroups`, with valid string values `"Both"`, `"Left"`, `"Right"`, `"None"`, may be used to specify which of the automorphism groups should be constructed first if not already known. The default is `"None"`.

`P subset Q: Plane, Plane -> BoolElt`

Returns `true` if P is a subplane of Q, otherwise `false`.

154.11 The Connection between Projective and Affine Planes

There exist natural mathematical constructions to form a projective plane from an affine plane and vice versa. The functions in the this section provide a quick and easy way to do this in Magma.

`FiniteAffinePlane(P, l): PlaneProj, PlaneLn -> PlaneAff, PlanePtSet, PlaneLnSet, Map`

The affine plane obtained by removing the line l from the projective plane P, together with the point set and line set of the affine plane, plus the embedding map from the affine plane to P.

`ProjectiveEmbedding(P): PlaneAff -> PlaneProj, PlanePtSet, PlaneLnSet, Map`

The projective completion of the affine plane P, together with the point set and line set of the projective plane, plus the embedding map from P to the projective plane.

Example: Projective Embedding of an Affine Plane (ex-a0e5e5)

We begin with the classical affine plane A of order 3, and take the projective embedding P of A. We then remove a randomly selected line from P, and show that the affine plane produced by this action is isomorphic to the original affine plane A.

```
> A := FiniteAffinePlane(3);
> P := ProjectiveEmbedding(A);
> P;
Projective Plane of order 3
> A2 := FiniteAffinePlane(P, Random(LineSet(P)));
> A2;
Affine Plane of order 3
> iso, map := IsIsomorphic(A, A2);
> is_iso, map := IsIsomorphic(A, A2);
> is_iso;
true
> map;
Mapping from: PlaneAff: A to PlaneAff: A2
```

We demonstrate the use of the embedding map to get the correspondence between the points of the affine and projective planes.

```
> K<w> := GF(4);
> A, AP, AL := FiniteAffinePlane(K);
> P, PP, PL, f := ProjectiveEmbedding(A);
```

Now take a point of the affine plane and map it into the projective.

```
> AP.5;
( 1, w )
> AP.5 @ f;
5
```

Our point corresponds to `PP.5`, which in the affine plane is the pair `(1, w)`. The map `f` can be applied to any point or line of the affine plane to get the corresponding point or line of the projective plane. Given any point or line of the projective plane, provided that it is not on the adjoined line at infinity, the preimage in the affine plane can be found.

The line at infinity is always the last line in the line set of the projective plane created by `ProjectiveEmbedding`. We will call this line `linf`:

```
> linf := PL.#PL;
> linf;
{17, 18, 19, 20, 21}
> SetSeed(1, 3);
> p := Random(PP);
> p in linf;
false
> p @@ f;
( w, 1 )
> l := Random(PL);
> l eq linf;
false
> l @@ f;
< 1 : 1 : 0 >
> $1 @ f eq l;
true
```

Since neither `p` nor `l` were infinite we could find their preimages under `f`. Of course, when we map a line from `P` to `A` and back, we get the line we started with.

When an embedding is constructed by `FiniteAffinePlane(P, l)`, then `l` is the line at infinity for this embedding.

154.12 Operations on Points and Lines

All the usual equality, membership, and subset functions are provided along with a collection of deconstruction functions and others.

154.12.1 Elementary Operations

`p eq q: PlanePt, PlanePt -> BoolElt`

Returns `true` if the points p and q are equal, otherwise `false`.

`p ne q: PlanePt, PlanePt -> BoolElt`

Return `true` if the points p and q are not equal, otherwise `false`.

`l eq m: PlaneLn, PlaneLn -> BoolElt`

Return `true` if the lines l and m are equal, otherwise `false`.

`l ne m: PlaneLn, PlaneLn -> BoolElt`

Return `true` if the lines l and m are not equal, otherwise `false`.

`p in l: PlanePt, PlaneLn -> BoolElt`

Return `true` if point p lies on the line l, otherwise `false`.

`p notin l: PlanePt, PlaneLn -> BoolElt`

Return `true` if point p does not lie on the line l, otherwise `false`.

`S subset l: { PlanePt}, PlaneLn -> BoolElt`

Given a subset S of the point set of the plane P and a line l of P, return `true` if the subset S of points lies on the line l, otherwise `false`.

`S notsubset l: { PlanePt}, PlaneLn -> BoolElt`

Given a subset S of the point set of the plane P and a line l of P, return `true` if the subset S of points does not lie on the line l, otherwise `false`.

`l meet m: PlaneLn, PlaneLn -> PlanePt`

The unique point common to the lines l and m.

`Representative(l): PlaneLn -> PlanePt`

`Rep(l): PlaneLn -> PlanePt`

Given a line l of the plane P, return a representative point of P which is incident with l.

`Random(l): PlaneLn -> PlanePt`

Given a line l of the plane P, return a random point of P which is incident with l.

154.12.2 Deconstruction Functions

`Index(P, p): Plane, PlanePt -> RngIntElt`

Given a point p from the point–set V of a plane P, return the index of p, i.e. the integer i such that p is `V.i`.

`Index(P, l): Plane, PlaneLn -> RngIntElt`

Given a line l, return the index of l in the plane P, i.e. the integer i such that l is $L.i$ (where L is the line–set of P).

`p[i]: PlanePt, RngIntElt -> FldFinElt`

The i-th coordinate of the point p, which must be from a classical plane. If p is from a projective plane, then i must satisfy $1 \leq i \leq 3$; if p is from an affine plane, then i must satisfy $1 \leq i \leq 2$.

`l[i]: PlaneLn, RngIntElt -> FldFinElt`

The i-th coordinate of the line l, which must be from a classical plane. The integer i must satisfy $1 \leq i \leq 3$. Recall that in a classical plane $\langle a : b : c \rangle$ (where $a, b, c \in K$) represents the line given by the equation $ax + by + cz = 0$ in a projective plane or $ax + by + c = 0$ in an affine plane.

`Coordinates(P, p): Plane, PlanePt -> [ FldFinElt ]`

Given a point $p = (a : b : c)$ from a classical projective plane P (or $p = (a, b)$ from a classical affine plane P), return the sequence $[a, b, c]$ (or $[a, b]$ in the affine case) of coordinates of p.

`Coordinates(P, l): Plane, PlaneLn -> [ FldFinElt ]`

Given a line $l = \langle a : b : c \rangle$ from a classical plane P (projective or affine), return the sequence $[a, b, c]$ of coordinates of l.

`ElementToSequence(p): PlanePt -> [ FldFinElt ]`

`Eltseq(p): PlanePt -> [ FldFinElt ]`

Given a point $p = (a : b : c)$ from a classical projective plane P (or $p = (a, b)$ from a classical affine plane P), return the sequence $[a, b, c]$ (or $[a, b]$ in the affine case) of coordinates of p.

`ElementToSequence(l): PlaneLn -> [ FldFinElt ]`

Eltseq(l): PlaneLn -> [FldFinElt]

Given a line $l = \langle a : b : c \rangle$ from a classical plane P (projective or affine), return the sequence $[a, b, c]$ of coordinates of l.

Set(l): PlaneLn -> { PlanePt }

The set of points contained in the line l.

Example: Deconstructing Points and Lines (ex-b3b9f2)

The following example illustrates the use of some of the elementary and deconstruction functions on lines and points discussed in the previous two subsections.

```
> K<w> := GF(4);
> P, V, L := FiniteProjectivePlane(K);
```

Create the line $x + z = 0$:

```
> l := L![1, 0, 1];
> l;
< 1 : 0 : 1 >
```

Look at the points on the line l:

```
> Set(l);
{  ( 0 : 1 : 0 ), ( 1 : w^2 : 1 ), ( 1 : 0 : 1 ),
   ( 1 : w : 1 ), ( 1 : 1 : 1) }
```

Get the coordinates of the line l:

```
> Coordinates(P, l);
[ 1, 0, 1 ]
> l[1];
1
```

Find the index of the line l in the line–set L of P, and check it:

```
> Index(P, l);
8
> l eq L.8;
true
```

Test if a point is on the line l:

```
> V![1, 0, 1] in l;
true
```

Test a set of points for containment in l:

```
> S := {V.1, V.2};
> S;
{  ( 1 : 0 : 0 ), ( 0 : 1 : 0 ) }
> S subset l;
false
```

Create the line containing the points in S:

```
> l2 := L!S;
> l2;
< 0 : 0 : 1 >
```

(continues on next page)

(continued from previous page)

```
> S subset l2;
true
```

And finally, find the point common to the lines l and $l2$:

```
> p := l meet l2;
> p;
( 0 : 1 : 0 )
> p[3];
0
```

154.12.3 Other Point and Line Functions

`IsCollinear(P, S): Plane, { PlanePt} -> BoolElt, PlaneLn`

Return `true` if the set S of points of the plane P are collinear, otherwise `false`. If the points are collinear, the line which they define is also returned.

`IsConcurrent(P, R): Plane, { PlaneLn} -> BoolElt, PlanePt`

Return `true` if the set R of lines of the plane P are concurrent, otherwise `false`. If the lines are concurrent, their common point is returned as a second value.

`ContainsQuadrangle(P, S): Plane, { PlanePt } -> BoolElt`

Return `true` if the set S of points of a plane P contains a quadrangle.

`Pencil(P, p): Plane, PlanePt -> { PlaneLn }`

The pencil of lines passing through the point p in the plane P.

`Slope(l): PlaneLn -> FldFinElt`

The slope of the line l of a classical affine plane P.

`IsParallel(P, l, m): Plane, PlaneLn, PlaneLn -> BoolElt`

Return `true` if the line l is parallel to the line m in the affine plane P.

`ParallelClass(P, l): Plane, PlaneLn -> { PlaneLn }`

The parallel class containing the line l of an affine plane P.

`ParallelClasses(P): PlaneAff -> { { PlaneLn } }`

The partition into parallel classes of the lines of the affine plane P.

Example: Point and Line Functions in AG(2,3) `(ex-c0212f)`

We use the affine plane $AG_2(3)$ to demonstrate some of the above functions.

```
> A, V, L := FiniteAffinePlane(3);
```

Create the line $y = 2x + 1$ in A, and check its slope:

```
> l := L![2, 1];
> l;
< 1 : 1 : 2 >
> Slope(l);
2
```

Find the lines parallel to l:

```
> ParallelClass(l);
{
    < 1 : 1 : 0 >,
    < 1 : 1 : 1 >,
```

(continues on next page)

(continued from previous page)

```
    < 1 : 1 : 2 >
}
> [Slope(m): m in ParallelClass(l)];
[ 2, 2, 2 ]
```

Get the pencil of lines through a point of l:

```
> p := Rep(l);
> p;
( 1, 0 )
> Pencil(A, p);
{
    < 1 : 0 : 2 >,
    < 1 : 1 : 2 >,
    < 1 : 2 : 2 >,
    < 0 : 1 : 0 >
}
```

154.13 Arcs

A *k–arc* in a projective or affine plane P is a set of k points of P, no three of which are collinear. A k–arc is *complete* if it cannot be extended to a $(k+1)$–arc by the addition of another point. A *tangent* to an arc A is a line which meets A exactly once; a *secant* is a line which meets A exactly twice; and a *passant*, or *external line*, is a line which does not meet A at all.

kArc(P, k): Plane, RngIntElt -> SetEnum

Return a k–arc for the plane P.

CompleteKArc(P, k): Plane, RngIntElt -> SetEnum

Return a complete k–arc for the plane P (if one exists).

IsArc(P, A): Plane, { PlanePt } -> BoolElt

Returns `true` if the set of points A is an arc in the plane P, i.e. no three points of A are collinear.

IsComplete(P, A): Plane, { PlanePt } -> BoolElt

Returns `true` if the k–arc A is complete in the plane P.

Conic(P, S): Plane, { PlanePt } -> SetEnum

Given a set S of five points belonging to a classical projective plane P of order $n > 3$ and being in general position, construct the unique conic that passes through them.

QuadraticForm(S): { PlanePt } -> RngMPolElt

Given a set S of five points belonging to a classical projective plane of order $n > 3$ that are in general position, return the quadratic form defining the conic containing the five points.

Tangent(P, A, p): Plane, { PlanePt }, PlanePt -> PlaneLn

Given an arc A in the plane P, and a point p on A, return a tangent to A at p.

AllTangents(P, A): Plane, { PlanePt} -> { PlaneLn}

Given an arc A in the plane P, return the set of tangent lines to A.

AllSecants(P, A): Plane, { PlanePt} -> { PlaneLn}

Given an arc A in the plane P, return the set of secant lines to A.

ExternalLines(P, A): Plane, { PlanePt} -> { PlaneLn}

AllPassants(P, A): Plane, { PlanePt} -> { PlaneLn}

Given an arc A in the plane P, return the set of external lines to A.

`Knot(P, C): Plane, { PlanePt} -> PlanePt`

Given a conic C in the projective plane P of even order, return the knot of the conic C, i.e the intersection point of the tangents to C.

`Exterior(P, C): Plane, { PlanePt} -> { PlanePt}`

Given a conic C in the projective plane P of odd order, return the exterior points of C, i.e. the points of P that lie on two tangents of C.

`Interior(P, C): Plane, { PlanePt} -> { PlanePt}`

Given a conic C in the projective plane P of odd order, return the interior points of C, i.e. the points of P that do not lie on any tangent of C.

Example: Oval Design from an Arc `(ex-ca0adf)`

The following sequence of instructions constructs an oval design from $PG_2(16)$.

```
> P, V, L := FiniteProjectivePlane(16);
> oval := kArc(P, 18);
> pts := Points(P) diff oval;
> lns := ExternalLines(P, oval);
> I := IncidenceStructure< SetToIndexedSet(pts) | [l meet pts : l in lns] >;
> D := Design(Dual(I), 2);
> D;
2-(120, 8, 1) Design with 255 blocks
```

The next example uses various functions discussed so far, and shows the relationship between a plane and its subplanes.

```
> K<w> := GF(9);
> P, V, L := FiniteProjectivePlane(K);
> c := kArc(P, 5);
> c;
{  ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 2 : w^3 ),
   ( 1 : w : w) }
> C := Conic(P, c);
> C;
{  ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 2 : w^3 ),
   ( 1 : w : w), ( 1 : w^5 : w^6 ), ( 1 : w^7 : 2 ), ( 1 : w^2 : 1 ),
   ( 1 : w^3 : w^5 ), ( 1 : w^6 : w^2 ) }
> #C;
10
> #Interior(P, C);
36
>
> S, SV, SL := SubfieldSubplane(P, GF(3));
> S subset P;
true
> a := kArc(S, 4);
> IsArc(S, a);
true
> IsArc(P, a);
true
> IsComplete(S, a);
true
> IsComplete(P, a);
false
> a;
{ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 1 : 2 : 1 ) }
> S2 := sub< P | a >;
```

(continues on next page)

(continued from previous page)

```
> S2;
Projective Plane of order 3
> S2 eq S;
true
> p := Random(a);
> p;
( 1 : 2 : 1 )
> Tangent(S, a, p);
< 1 : 2 : 1 >
> AllTangents(S, a);
{
    < 1 : 2 : 0 >,
    < 0 : 1 : 2 >,
    < 1 : 2 : 1 >,
    < 1 : 0 : 1 >
}
> AllTangents(P, a);
{
    < 1 : w^5 : w^6 >,
    < 1 : 0 : w >,
    < 1 : w^2 : w >,
    < 1 : 0 : w^6 >,
    < 1 : 0 : 1 >,
    < 1 : w^6 : w^3 >,
    < 1 : w^6 : 0 >,
    < 0 : 1 : w^6 >,
    < 1 : 2 : 0 >,
    < 0 : 1 : 2 >,
    < 1 : 2 : 1 >,
    < 1 : w^3 : w^5 >,
    < 1 : 0 : w^7 >,
    < 1 : 0 : w^2 >,
    < 1 : w : 0 >,
    < 0 : 1 : w >,
    < 1 : 0 : w^5 >,
    < 1 : 0 : w^3 >,
    < 1 : w : w^7 >,
    < 1 : w^3 : 0 >,
    < 0 : 1 : w^3 >,
    < 1 : w^7 : w^2 >,
    < 1 : w^5 : 0 >,
    < 0 : 1 : w^5 >,
    < 1 : w^7 : 0 >,
    < 0 : 1 : w^7 >,
    < 1 : w^2 : 0 >,
    < 0 : 1 : w^2 >
}
```

154.14 Unitals

A unital in the classical projective plane $PG_2(q^2)$ is a set of $q^3 + 1$ points such that every line meeting two of these points meets exactly $q + 1$ of them.

IsUnital(P, U): Plane, { PlanePt} -> BoolElt

Given a set of points U belonging to a projective plane P defined over a field of cardinality q^2, return `true` if U is a unital.

AllTangents(P, U): Plane, { PlanePt} -> { PlaneLn}

Given a unital set of points U in the projective plane P, return the set of tangents to the points of U.

UnitalFeet(P, U, p): Plane, { PlanePt}, PlanePt -> { PlanePt}

The set of intersections of the unital set of points U with the tangents to U in the plane P which pass through the point p.

Example: Hermitian Unital in PG(2,9) (ex-fe7f50)

The following code computes the Hermitian unital given by the equation $x^{q+1} + y^{q+1} + z^{q+1} = 0$ in $PG_2(q^2)$ for $q = 3$.

```
> q := 3;
> F<w> := GaloisField(q ^ 2);
> P, V, L := FiniteProjectivePlane(F);
>
> hu := { V | [x,y,z] : x, y, z in F |
>                x^(q+1) + y^(q+1) + z^(q+1) eq 0 and {x, y, z} ne {0} };
>
> IsUnital(P, hu);
true
> UnitalFeet(P, hu, V.1);
{  ( 0 : 1 : w ), ( 0 : 1 : w^3 ), ( 0 : 1 : w^5 ), ( 0 : 1 : w^7 ) }
```

Since this set has more than one element, `V.1` must not be in hu:

```
> V.1 in hu;
false
```

For a point in hu:

```
> UnitalFeet(P, hu, Rep(hu));
{  ( 1 : 0 : w^7 ) }
```

Now we construct the design given by hu.

```
> blks := [blk : lin in L | #blk eq (q+1) where blk is lin meet hu ];
> D := Design< 2, SetToIndexedSet(hu) | blks >;
> D;
2-(28, 4, 1) Design with 63 blocks
```

154.15 The Collineation Group of a Plane

The automorphism group (or collineation group) A of a plane P is always presented as a permutation group G acting on the standard support $\{1, ..v\}$, where v is the number of points of P. The reasons for this include the fact that if the group is represented as acting on a set of objects relating to the plane, printed permutations are often unreadable.

So the collineation group G of P does not act directly on P. Instead, G–sets are used to transfer the action of G to various sets associated with P. The two most important G–sets, corresponding to action of G on the point set and on the line set, are returned by each of the functions provided for constructing the collineation group or some specified subgroup of it.

In some circumstances, rather than viewing collineations as group elements, it is desirable to view them is as mappings of P into itself. Associated with each incidence structure is a mapping structure, Aut(P), which denotes the set of collineations of P. Note that Aut(P) is the *parent* of the collineations of G so that the function `Aut(P)` simply creates a shell structure rather than the actual collineation group of P. A transfer map is provided to convert a permutation of the collineation group G into a mapping belonging to $Aut(P)$.

154.15.1 The Collineation Group Function

`CollineationGroup(P): Plane -> GrpPerm, GSet, GSet, PowMap, Map`

`AutomorphismGroup(P): Plane -> GrpPerm, GSet, GSet, PowMap, Map`

`PointGroup(P): Plane -> GrpPerm, GSet, GSet, PowMap, Map`

Construct the collineation group G of the plane P. The group G is returned as a permutation group on the standard support $\{1 \dots v\}$, where v is the number of points of P. The function also returns: a G–set Y being the point set of P acted on by G; a G–set W being the line set of P acted on by G; a power structure S; a transfer map t. Given a permutation g from G, one can create a map $f = t(g)$ which represents the automorphism g as a mapping in S from P to P (which maps both point sets and line sets). The G–sets Y and W should be used if one wishes to compute stabilizers or similar such subgroups of G so that the appropriate action is used.

`LineGroup(P): Plane -> GrpPerm, PowMap, Map`

Construct the collineation group G of the plane P in its action on the lines of P. The group G is returned as a permutation group on the standard support $\{1 \dots l\}$, where l is the number of lines of P. A power structure S and transfer map t are also returned, so that, given a permutation g from G, one can create a map $f = t(g)$ which represents the automorphism g as a mapping in S from L to L, where L is the line set of P.

`CollineationGroupStabilizer(P, k): Plane, RngIntElt -> GrpPerm, GSet, GSet, PowMap, Map`

A subgroup G of the collineation group of the plane P which stabilizes the first k base points, together with the points G–set, the lines G–set, the power structure A of all automorphisms of P, and the transfer map t from G into A.

`CollineationSubgroup(P): Plane -> GrpPerm, GSet, GSet, PowMap, Map`

A subgroup G of the collineation group of the plane P generated by one element, together with the points G–set, the lines G–set, the power structure A of all automorphisms of P, and the transfer map t from G into A.

154.15.2 General Action of Collineations

The collineation group G of a plane P is given in its action on the standard support. This support may be regarded as the indices of the points of P. The action of G on P is obtained using the G–set mechanism. The two basic G–sets associated with P correspond to the action of G on the set of points V and the set of lines L of P. These two G–sets are given as return values of the function `AutomorphismGroup`. Additional G–sets associated with P may be built using the G–set constructors. Given a G–set Y for G, the action of G on Y may be studied using the permutation group functions that allow a G–set as an argument. In this section, only a few of the available functions are described: see the chapter on permutation groups for a complete list.

The action of the collineation group on the plane may also be obtained using the ^ operator.

`y ^ g: Elt, GrpPermElt -> Elt`

Let G be a subgroup of the collineation group for the plane P and suppose $g \in G$. Given an element y that is either a point or line of P, return the image of y under g.

`y ^ G: Elt, GrpPerm -> GSet`

Let G be a subgroup of the collineation group for the plane P. Given an element y that is either a point or line of P, return the orbit of y under G.

`Image(g, Y, y): GrpPermElt, GSet, Elt -> Elt`

Let G be a subgroup of the collineation group for the plane P and let Y be a G–set for G. Given an element y belonging either to Y or to a G–set derived from Y, find the image of y under G.

`Orbit(G, Y, y): GrpPerm, GSet, Elt -> GSet`

Let G be a subgroup of the collineation group for the plane P and let Y be a G–set for G. Given an element y belonging either to Y or to a G–set derived from Y, construct the orbit of y under G.

`Orbits(G, Y): GrpPerm, GSet -> [ GSet ]`

Let G be a subgroup of the collineation group for the plane P and let Y be a G–set for G. This function constructs the orbits of the action of G on Y.

`Stabilizer(G, Y, y): GrpPerm, GSet, Elt -> GrpPerm`

Let G be a subgroup of the collineation group for the plane P and let Y be a G–set for G. Given an element y belonging either to Y or to a G–set derived from Y, construct the stabilizer of y in G.

`Action(G, Y): GrpPerm, GSet -> Hom(Grp), GrpPerm, GrpPerm`

Given a subgroup G of the collineation group of the plane P, and a G–set Y for G, construct the homomorphism $\phi : G \to L$, where the permutation group L gives the action of G on the set Y. The function returns:

(a) The natural homomorphism $\phi : G \to L$;

(b) The induced group L;

(c) The kernel of the action (a subgroup of G).

`ActionImage(G, Y): GrpPerm, GSet -> GrpPerm`

Given a subgroup G of the collineation group of the plane P, and a G–set Y for G, construct the permutation group L giving the action of G on the set Y.

`ActionKernel(G, Y): GrpPerm, GSet -> GrpPerm`

Given a subgroup G of the collineation group of the plane P, and a G–set Y for G, construct the kernel of the action of G on the set Y.

Example: G-Sets of the Collineation Group (ex-8ca8e9)

We illustrate the use of the G–sets returned by the `CollineationGroup` function.

```
> P := FiniteProjectivePlane(3);
> G, Y, W := CollineationGroup(P);
```

Compute the stabilizer of the first point of P:

```
> H := Stabilizer(G, Y, Points(P)[1]);
> H;
Permutation group H acting on a set of cardinality 13
    (3, 9)(5, 7)(6, 11)(8, 12)
    (2, 8, 12)(3, 9)(4, 6, 7, 10, 5, 11)
    (2, 9)(3, 4)(5, 8)(10, 13)
    (2, 8)(3, 9)(4, 5)(6, 10)
    (2, 3)(4, 13)(6, 8)(9, 10)
    (2, 13, 8)(3, 6, 4)(5, 10, 9)
    (2, 3)(4, 9)(7, 12)(10, 13)
> H eq CollineationGroupStabilizer(P, 1);
true
> lines := {m : m in Lines(P) | Points(P)[1] in m};
> l := Random(lines);
> l ^ H;
GSet{
  < 0 : 1 : 0 >,
  < 0 : 0 : 1 >,
  < 0 : 1 : 2 >,
  < 0 : 1 : 1 >
}
```

Compute the stabilizer of the first line of P:

```
> Stabilizer(G, W, Lines(P)[1]);
Permutation group acting on a set of cardinality 13
    (1, 8, 12, 4, 7, 9)(2, 3, 5)(6, 13)
    (3, 5, 11)(6, 7, 9)(8, 12, 13)
    (4, 10)(5, 11)(6, 7)(8, 12)
    (1, 10, 6, 7)(2, 11)(3, 5)(4, 13, 9, 12)
    (5, 11)(6, 12)(7, 8)(9, 13)
    (1, 10)(3, 5)(6, 7)(12, 13)
    (1, 4, 10)(3, 11)(6, 13, 9, 8, 7, 12)
```

Example: Projective Bundle in PG(2,q) (ex-d21993)

The following function `Bundle` returns a projective bundle in $PG_2(q)$.

```
> Bundle := function(q)
>     K<w> := GF(q^3);
>     P, V, L := FiniteProjectivePlane(q^3);
>     G, Y := CollineationGroup(P);
>     S := Support(P);  // normalized vectors
>     sig := sub< G |
>               [Index(P, V ! [S[i][1], S[i][2]^q, S[i][3]^q]) : i in [1..#V]]>;
>                       // group of planar collineations of order 3
>     p := V ! [1, w^2, w];
>     T := Orbit(sig, Y, p);
>     e2 := V![0, 1, 0];
>     e3 := V![0, 0, 1];
>     S := Points(SubfieldSubplane(P, GF(q)));
>     c23 := Conic(P, T join {e2, e3}) meet S;
>     e1 := Rep(S diff c23);
>     c12 := (Conic(P, T join {e1, e2}) meet S) diff { e1 };
>     c13 := (Conic(P, T join {e1, e3}) meet S) diff { e1 };
>     bundle := [ Conic(P, T join {e1, e}) meet S : e in c23 ] cat
>               [ Conic(P, T join {v1, v2}) meet S : v2 in c13, v1 in c12 ];
>     return FiniteProjectivePlane< S | bundle >;
>
> end function;
>
> PB := Bundle(3);
> PB;
Projective Plane of order 3
```

Example: Baer Derivation of an Affine Plane (ex-d549e1)

The function `BaerDerivation` below uses a Baer subplane to construct an affine plane.

```
> BaerDerivation := function(q)
> //--------------------------
> // Construct an affine plane by the technique of derivation using
> // Baer subplanes
>
>    Fq2< w > := FiniteField(q^2);
>    V := VectorSpace(Fq2, 3);
>    Plane, Pts, Lns := FiniteProjectivePlane(V);
>    G, Y := CollineationGroup(Plane);
```

Construct a Baer subplane:

```
>    Subplane := SubfieldSubplane(Plane, GF(q));
```

The Baer segment consists of those points of the Baer subplane that lie on the line at infinity. Take the line $x = 0$ as the line at infinity.

```
>    LineInf := Lns![1, 0, 0];
>
>    BaerSeg := Points(Subplane) meet LineInf;
```

We now find the subgroup of the collineation group that fixes the Baer segment. The translates of the Baer subplane under this subgroup will give us those Baer subplanes that contain the set `BaerSeg`. We use the G–set `Y` to specify the action of `G` on the points of `Plane`.

```
>    StabSeg := Stabilizer(G, Y, BaerSeg);
```

Rather than computing the translates of the entire Baer subplane, we compute the translates of `Subplane - BaerSeg` so that we get exactly those sets which become new affine lines.

```
>    BaerLines := Orbit(StabSeg, Y, Points(Subplane) diff BaerSeg);
```

We complete the new plane by taking those lines of $PG(2, q^2)$ which intersect the line at infinity at points other than those in the Baer segment. Upon removing the intersection point with `LineInf`, each such line becomes a line of the new affine plane.

```
>    AffLines := BaerLines join { Set(l) diff LineInf : l in Lns |
>                                 (BaerSeg meet l) eq {} };

>    return FiniteAffinePlane< SetToIndexedSet(&join(AffLines)) |␣
↪Setseq(AffLines)
>                                            : Check := false >;

> end function; /*BaerDerivation*/
```

154.15.3 Central Collineations

Let p be a point and l a line of a projective plane P. A (p, l)–*central collineation* is a collineation α of P which fixes l pointwise and p linewise. The line l is called the *axis* of α and the point p is called the *centre* of α.

`CentralCollineationGroup(P, p, l): Plane, PlanePt, PlaneLn -> GrpPerm, PowMap, Map`

The group G of (p, l)–central collineations of a projective plane P. A power structure S and transfer map t are also returned, so that, given a permutation g from G, one can create a map $f = t(g)$ which represents the permutation g as a mapping in S from P to P. (which maps both point sets and line sets).

`CentralCollineationGroup(P, p): Plane, PlanePt -> GrpPerm, PowMap, Map`

The group of central collineations with centre p of a projective plane P. A power structure S and transfer map t are also returned, so that, given a permutation g from G, one can create a map $f = t(g)$ which represents the permutation g as a mapping in S from P to P. (which maps both point sets and line sets).

`CentralCollineationGroup(P, l): Plane, PlaneLn -> GrpPerm, PowMap, Map`

The group of central collineations with axis l of a projective plane P. A power structure S and transfer map t are also returned, so that, given a permutation g from G, one can create a map $f = t(g)$ which represents the permutation g as a mapping in S from P to P. (which maps both point sets and line sets).

`IsCentralCollineation(P, g): Plane, GrpPermElt -> BoolElt, PlanePt, PlaneLn`

Returns `true` iff the collineation g of the projective plane P is a central collineation; if `true`, also returns the centre and axis of g. The support of the parent of g must be the point set of P or the standard support $\{1 \dots v\}$, where v is the number of points of P.

Example: Central Collineations of a Plane (ex-3873ae)

We find a group of central collineations of a plane P, and check that a generator of the group is in fact a central collineation and has the correct axis and centre.

```
> P, V, L := FiniteProjectivePlane< 13 |
>     {1, 2, 3, 4}, {1, 5, 6, 7}, {1, 8, 9, 10},
>     {1, 11, 12, 13}, {2, 5, 8, 11}, {2, 6, 9, 12},
>     {2, 7, 10, 13}, {3, 5, 9, 13}, {3, 6, 10, 11},
>     {3, 7, 8, 12}, {4, 5, 10, 12}, {4, 6, 8, 13},
>     {4, 7, 9, 11} >;
> p := V!3;
> l := L.1;
> G := CentralCollineationGroup(P, p, l);
> G;
Permutation group G acting on a set of cardinality 13
Order = 3
    (5, 13, 9)(6, 11, 10)(7, 12, 8)
> g := G.1;
> g;
(5, 9, 13)(6, 10, 11)(7, 8, 12)
> is_cent_coll, centre, axis := IsCentralCollineation(P, g);
> is_cent_coll;
true
> centre eq p;
true
> axis eq l;
true
```

Any line through the centre of a central collineation must be fixed by the collineation.

```
> lines := {m : m in Lines(P) | p in m};
> m := Random(lines);
> m;
{3, 7, 8, 12}
> m ^ G;
GSet{
  {3, 7, 8, 12}
}
```

154.15.4 Transitivity Properties

`IsPointTransitive(P): Plane -> BoolElt`

`IsTransitive(P): Plane -> BoolElt`

Return `true` iff the collineation group of the plane P acts transitively on the points of P.

`IsLineTransitive(P): Plane -> BoolElt`

Return `true` iff the collineation group of the plane P acts transitively on the lines of P.

Example: Transitivity of a Collineation Group (ex-7fe88b)

We check the transitivity of the collineation group of $AG_2(4)$.

```
> P := FiniteAffinePlane(4);
> IsPointTransitive(P);
true
> IsLineTransitive(P);
true
```

154.16 Translation Planes

The following functions may be used to construct translation planes by derivation. The original code is due to Jenny Key.

BaerDerivation(q2): RngIntElt -> PlaneAff, PlanePtSet, PlaneLnSet

An affine plane constructed by the technique of derivation with respect to a Baer subplane, where $q2$ is an even power of a prime.

BaerSubplane(P): PlaneProj -> PlaneProj, PlanePtSet, PlaneLnSet

A Baer subplane of the projective plane P.

OvalDerivation(q: parameters): RngIntElt -> PlaneAff, PlanePtSet, PlaneLnSet

```
HallOval: BoolElt                    Default: false
Print   : BoolElt                    Default: false
```

A translation plane from $\mathrm{PG}(2, q)$, where q is a power of 2, computed by derivation with respect to an oval. By default, the oval chosen is that defined by the points of the conic $y^2 = xz$ together with the nucleus, in $\mathrm{PG}(2, q)$. In the case $q = 16$, a Hall oval in the plane may be requested by assigning the parameter `HallOval` the value `true`. If the parameter `Print` is assigned the value `true`, then some information is printed during the computation.

154.17 Planes and Designs

Projective and affine planes can be viewed as special kinds of designs. The following functions convert between designs and planes.

Design(P): Plane -> Dsgn, SetIncPt, SetIncBlk

The design corresponding to the points and lines of the plane P.

FiniteAffinePlane(D): Inc -> Plane, PlanePtSet, PlaneLnSet

The affine plane corresponding to the incidence structure D.

FiniteProjectivePlane(D): Inc -> Plane, PlanePtSet, PlaneLnSet

The projective plane corresponding to the incidence structure D.

Example: Projective Plane from a Singer Difference Set (ex-55665c)

The development of a Singer difference set provides a design which satisfies the projective plane axioms, and thus can be converted to a projective plane in Magma.

```
> sds := SingerDifferenceSet(2, 3);
> sds;
{ 0, 1, 3, 9 }
> sdv := Development(sds);
> sdv;
2-(13, 4, 1) Design with 13 blocks
> spp := FiniteProjectivePlane(sdv);
> spp: Maximal;
Projective Plane of order 3
Points: {@ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 @}
Lines:
    {0, 1, 3, 9},
    {1, 2, 4, 10},
    {2, 3, 5, 11},
    {3, 4, 6, 12},
    {0, 4, 5, 7},
    {1, 5, 6, 8},
```

(continues on next page)

(continued from previous page)

```
    {2, 6, 7, 9},
    {3, 7, 8, 10},
    {4, 8, 9, 11},
    {5, 9, 10, 12},
    {0, 6, 10, 11},
    {1, 7, 11, 12},
    {0, 2, 8, 12}
> Universe(Support(spp));
Residue class ring of integers modulo 13
```

154.18 Planes, Graphs, and Codes

Magma provides the following functions to create the graphs and codes naturally associated to a projective or affine plane.

LineGraph(P): Plane -> Grph

The line graph of the plane P.

IncidenceGraph(P): Plane -> Grph

The incidence graph of the plane P. This bipartite graph has as vertex set the union of the point set V and line set L of P. A vertex $p \in V$ is adjacent to a vertex $l \in L$ whenever $p \in l$.

LinearCode(P, K): Plane, FldFin -> Code

Given a plane P with v points and a finite field K, this function returns the linear code C of length v generated by the characteristic functions of the lines of P considered as vectors of the K-space $K^{(v)}$.

Example: Incidence Graph and Code of a Plane (ex-43be98)

```
> P, V, L := FiniteAffinePlane(3);
> #V, #L;
9 12
> IncidenceGraph(P);
Graph
Vertex  Neighbours

1       10 11 12 19 ;
2       16 17 18 19 ;
3       13 14 15 19 ;
4       10 15 17 21 ;
5       12 14 16 21 ;
6       11 13 18 21 ;
7       10 14 18 20 ;
8       11 15 16 20 ;
9       12 13 17 20 ;
10      1 4 7 ;
11      1 6 8 ;
12      1 5 9 ;
13      3 6 9 ;
14      3 5 7 ;
15      3 4 8 ;
16      2 5 8 ;
17      2 4 9 ;
18      2 6 7 ;
19      1 2 3 ;
20      7 8 9 ;
```

(continues on next page)

(continued from previous page)

```
21      4 5 6 ;

> LinearCode(P, Field(P));
[9, 6, 3] Linear Code over GF(3)
Generator matrix:
[1 0 0 0 0 1 0 1 0]
[0 1 0 0 0 1 0 2 2]
[0 0 1 0 0 1 0 0 1]
[0 0 0 1 0 2 0 1 2]
[0 0 0 0 1 2 0 2 1]
[0 0 0 0 0 0 1 1 1]
```

Chapter 155

Incidence Geometry

155.1 Introduction

This chapter presents the functions designed for constructing and computing with incidence geometries and coset geometries.

We recall the basic definitions and notation in the field of Incidence Geometry. We refer to the *Handbook of Incidence Geometry* [Bue95], edited by Francis Buekenhout, or to Antonio Pasini's book *Diagram Geometries* [Pas94], for a more detailed overview of the subject.

Let X and I be two finite sets. Let $t : X \to I$ be a mapping from X onto I. Let $\sim$ be a reflexive and symmetric relation such that $\forall x, y \in X$, $x \sim y$ and $t(x) = t(y) \Rightarrow x = y$. The four-tuple $\Gamma(X, \sim, t, I)$ is what we call an *Incidence Geometry* in Magma. Remark that it is not a geometry in the sense of Buekenhout since we do not impose that every flag (i.e. clique of the incidence graph) of Γ must be contained in a chamber (i.e. a clique containing one element of each type). If the latter condition is satisfied, then an incidence geometry is a geometry in the sense of Buekenhout. The set X contains the *elements* of the geometry, while I is called the set of *types*. The function t is called the *type function* and $\sim$ is called the *incidence relation* of Γ. The cardinality of I is the *rank* of Γ.

It is possible to construct incidence geometries from a group and some of its subgroups using an algorithm first introduced by Jacques Tits in 1962 [Tit62]. Let G be a group and let I be a finite set. Let $\{G_i, i \in I\}$ be a set of subgroups of G. Define $X = \{G_i g, g \in G, i \in I\}$ to be the set of elements of Γ. Define the type function as $t : X \to I : G_i g \to i$ and the incidence relation as follows: $G_i g \sim G_j h$ iff $G_i g \cap G_j h \neq \emptyset$. The subgroups $\{G_i, i \in I\}$ are called the *maximal parabolic subgroups*. The subgroup $\cap_{i \in I} G_i$ is called the *Borel subgroup*. Finally, the subgroups $\{\cap_{j \in I \setminus \{i\}} G_j, i \in I\}$ are called the *minimal parabolic subgroups*. These geometries are called *Coset Geometries* in Magma to remind the user that they are constructed from a group. Again, a coset geometry is not a geometry in the sense of Buekenhout. If every flag of the coset geometry is contained in a chamber, then it is a Buekenhout geometry. We will see that, using coset geometries, it is easy to build huge incidence geometries by giving very little data.

The category names for the incidence geometries and coset geometries are:

- Incidence Geometry : `IncGeom`
- Coset Geometry : `CosetGeom`

155.2 Construction of Incidence and Coset Geometries

155.2.1 Construction of an Incidence Geometry

`IncidenceGeometry(G): GrphUnd -> IncGeom`

Construct the incidence geometry IG having as incidence graph the (labelled) graph G.

If G is an unlabelled graph, then the set of elements of IG is the set of vertices and edges of G, the set of types is $\{@1, 2@\}$ where elements of type 1 are vertices of G and elements of type 2 are

edges of G. An element x of type 1 is incident to an element y of type 2 iff the vertex x in G is on the edge y of G.

If G is a labelled graph, i.e. a graph whose vertices have labels, then the set of elements of IG is the set of vertices of G, the set of types is the set of labels of the vertices of G and the incidence graph is G.

The function returns one value: the incidence geometry IG.

Example: Petersen Graph as an Incidence Geometry (ex-a2453c)

The Petersen graph, considered as a rank two incidence geometry, may be constructed by the following statement:

```
> gr := Graph<10|
>   {1,2},{1,5},{1,6},{2,7},{2,3},{3,8},{3,4},{4,9},{4,5},{5,10},
>   {6,8},{8,10},{10,7},{7,9},{9,6}>;
> ig := IncidenceGeometry(gr);
> ig;
Incidence geometry ig with 2 types and with underlying graph:
Graph
Vertex  Neighbours

1       11 12 13 ;
2       11 14 15 ;
3       14 16 17 ;
4       16 18 19 ;
5       12 18 20 ;
6       13 21 22 ;
7       15 23 24 ;
8       17 21 25 ;
9       19 22 23 ;
10      20 24 25 ;
11      1 2 ;
12      1 5 ;
13      1 6 ;
14      2 3 ;
15      2 7 ;
16      3 4 ;
17      3 8 ;
18      4 5 ;
19      4 9 ;
20      5 10 ;
21      6 8 ;
22      6 9 ;
23      7 9 ;
24      7 10 ;
25      8 10 ;
```

Example: Cube as an Incidence Geometry (ex-2bc35f)

The rank three geometry consisting of the vertices, edges, and faces of the cube, with symmetric inclusion as incidence might be encoded in the following way. The first 8 vertices of the graph correspond to the vertices of the cube, the next 12 vertices correspond to the edges of the cube and the last 6 are the faces of the cube.

```
> g := Graph<26|
>    {1,9},{1,12},{1,13},{1,21},{1,22},{1,25},
>    {2,9},{2,10},{2,14},{2,21},{2,22},{2,23},
>    {3,10},{3,11},{3,15},{3,21},{3,23},{3,24},
```

(continues on next page)

(continued from previous page)

```
>    {4,11},{4,12},{4,16},{4,21},{4,24},{4,25},
>    {5,13},{5,18},{5,17},{5,22},{5,25},{5,26},
>    {6,14},{6,19},{6,18},{6,23},{6,22},{6,26},
>    {7,15},{7,19},{7,20},{7,24},{7,26},{7,23},
>    {8,17},{8,20},{8,16},{8,26},{8,24},{8,25},
>    {9,21},{9,22},{10,21},{10,23},{11,21},{11,24},
>    {12,21},{12,25},{13,22},{13,25},{14,22},{14,23},
>    {15,23},{15,24},{16,24},{16,25},{17,25},{17,26},
>    {18,22},{18,26},{19,23},{19,26},{20,24},{20,26}>;
> v := VertexSet(g);
> AssignLabels(v,[1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,3,3,3,3,3,3]);
> cube := IncidenceGeometry(g);
> cube;
Incidence geometry cube with 3 types and with underlying graph:
Graph
Vertex  Neighbours

1       9 12 13 21 22 25 ;
2       9 10 14 21 22 23 ;
3       10 11 15 21 23 24 ;
4       11 12 16 21 24 25 ;
5       13 17 18 22 25 26 ;
6       14 18 19 22 23 26 ;
7       15 19 20 23 24 26 ;
8       16 17 20 24 25 26 ;
9       1 2 21 22 ;
10      2 3 21 23 ;
11      3 4 21 24 ;
12      1 4 21 25 ;
13      1 5 22 25 ;
14      2 6 22 23 ;
15      3 7 23 24 ;
16      4 8 24 25 ;
17      5 8 25 26 ;
18      5 6 22 26 ;
19      6 7 23 26 ;
20      7 8 24 26 ;
21      1 2 3 4 9 10 11 12 ;
22      1 2 5 6 9 13 14 18 ;
23      2 3 6 7 10 14 15 19 ;
24      3 4 7 8 11 15 16 20 ;
25      1 4 5 8 12 13 16 17 ;
26      5 6 7 8 17 18 19 20 ;
```

Example: Constructing the Hoffman-Singleton Graph (ex-65148f)

The Hoffman-Singleton graph $HoSi$: the following description is taken from the book *Diagram Geometry*, by A. Cohen and F. Buekenhout [BCon].

Start with the set $X_1 = \binom{7}{3}$ of all triples from 7. There are 30 collections of 7 triples that form the lines of a *Fano plane* with point set 7. The group $Sym(7)$ acts transitively on this set of 30 elements, but the alternating group $Alt(7)$ has two orbits, of cardinality 15 each. Select one and call it X_2. Now consider the following graph Ω constructed from the incidence graph $(X_1 \cup X_2, *)$ by additionally joining two elements of X_1 whenever they have empty intersection. The following lines of Magma-code implement the Hoffman-Singleton graph as an incidence geometry of rank 2. We will study it more later.

```
> HoffmanSingletonGraph := function()
>   pentagram := Graph< 5 | { {1,3}, {3,5}, {5,2}, {2,4}, {4,1} } >;
>   pentagon := PolygonGraph(5);
>   PP := pentagram join pentagon;
>   HS := &join [ PP : i in [1..5] ];
>   return HS + { { Vertices(HS) | i + j*10, k*10 + 6 + (i+j*k) mod 5 } :
>                                  i in [1..5], j in [0..4], k in [0..4] };
> end function;
> ig := IncidenceGeometry(HoffmanSingletonGraph());
```

Example: Rank Four Geometry from Hoffman-Singleton (ex-666366)

A rank four geometry constructed from the Hoffman-Singleton graph : the following description is taken from the book *Diagram Geometry*, by A. Cohen and F. Buekenhout.

Take X_1 and $X_{1'}$ to be two copies of the vertex set of $HoSi$. The group $Sym(7)$ acts transitively on the set of maximal cocliques of $HoSi$. The subgroup $Alt(7)$ has two orbits of size 15 each. Let X_2 be one of them and $X_{2'}$ be a copy of X_2. Define incidence $\sim$ for $a \in X_1$, $a' \in X_{1'}$, $b \in X_2$, and $b' \in X_{2'}$, by

$a \sim a' \Leftrightarrow \{a, a'\}$ is an edge of $HoSi$

$a \sim b \Leftrightarrow a \notin b$

$a \sim b' \Leftrightarrow a \in b'$

$a' \sim b \Leftrightarrow a' \notin b$

$a' \sim b' \Leftrightarrow a' \notin b'$

$b \sim b' \Leftrightarrow b \cap b' = \emptyset$

The geometry $\Gamma(X_1 \cup X_{1'} \cup X_2 \cup X_{2'}, \sim)$ is the *Neumaier geometry*. The following lines of Magma-code implement this geometry as a rank four incidence geometry. We assume that ig is the incidence geometry corresponding to the Hoffman-Singleton graph, as described in the previous example.

```
> gr := HoffmanSingletonGraph();
> ig := IncidenceGeometry(gr);
> aut := AutomorphismGroup(ig);
> n := NormalSubgroups(aut);
> X1 := VertexSet(gr);
> g := CompleteGraph(50);
> e := EdgeSet(gr);
> ebis := [];
> for x in EdgeSet(gr) do
>     for i := 1 to 50 do
>         for j := i+1 to 50 do
>             if VertexSet(gr)!i in x then
>                 if VertexSet(gr)!j in x then
>                     Append(~ebis,[i,j]);
>                 end if;
>             end if;
>         end for;
>     end for;
> end for;
> for i := 1 to #ebis do
>     RemoveEdge(~g,ebis[i][1],ebis[i][2]);
> end for;
> c := MaximumClique(g);
> cbis := [];
> for i := 1 to 50 do
```

(continues on next page)

(continued from previous page)

```
>     if (VertexSet(gr)!i in c) then Append(~cbis,i); end if;
> end for;
> c := Set(cbis);
> X2 := Orbit(n[2]`subgroup,c);
> X2 := SetToSequence(X2);
> e := [];
> for i := 1 to 50 do
>     for j := 1 to 50 do
>         if X1!i in Neighbours(X1!j) then Append(~e,{i,j+50}); end if;
>     end for;
> end for;
> for i :=1 to 50 do
>     for j := 1 to 50 do
>         if not(i in X2[j]) then Append(~e,{i,j+100});end if;
>     end for;
> end for;
> for i :=1 to 50 do
>     for j := 1 to 50 do
>         if i in  X2[j] then Append(~e,{i,j+150});end if;
>     end for;
> end for;
> for i :=1 to 50 do
>     for j := 1 to 50 do
>         if i in  X2[j] then Append(~e,{i+50,j+100});end if;
>     end for;
> end for;
> for i :=1 to 50 do
>     for j := 1 to 50 do
>         if not(i in  X2[j]) then Append(~e,{i+50,j+150});end if;
>     end for;
> end for;
> for i :=1 to 50 do
>     for j := 1 to 50 do
>         if X2[i] meet X2[j] eq {} then Append(~e,{i+100,j+150});end if;
>     end for;
> end for;
> gr2 := Graph<200|Set(e)>;
> v := VertexSet(gr2);
> labs := [i: j in {1..50}, i in {1..4}];
> AssignLabels(v,labs);
> neumaier := IncidenceGeometry(gr2);
```

155.2.2 Construction of a Coset Geometry

`CosetGeometry(G, S, I): GrpPerm, Set, Set -> CosetGeom`

Construct the coset geometry CG with set of types I, obtained from the group G and the set S of subgroups of G. The sets S and I must have same cardinality.

If G is a permutation group and S is a set of subgroups of G, the corresponding coset geometry, obtained by using Tits' algorithm (see above) is constructed. If S and I are indexed sets, then the cosets of the subgroup $S[i]$ are elements of type $I[i]$, for all $i \in \{0, \ldots, n-1\}$ where n is the cardinality of S (and I). The function returns one value: the coset geometry CG.

`CosetGeometry(G, S): GrpPerm, Set -> CosetGeom`

Construct the coset geometry CG obtained from the group G and the set S of subgroups of G.

If G is a permutation group and S is a set of subgroups of G, the corresponding coset geometry,

obtained by using Tits' algorithm (see above) is constructed. To every subgroup in S is associated a number from 0 to $n-1$ where n is the cardinality of S. The function returns one value: the coset geometry CG.

Example: Petersen Graph as a Coset Geometry (ex-a1eb9c)

The Petersen graph, considered as a rank two coset geometry, can be implemented in the following way :

```
> g := Sym(5);
> g0 := Stabilizer(g,{1,2});
> g01 := Stabilizer(g,[{1,2},{3,4}]);
> g1 := sub<g|g01,(1,3)(2,4)>;
> cg := CosetGeometry(g,{g0,g1});
> cg;
Coset geometry cg with 2 types

Group:
Symmetric group g acting on a set of cardinality 5
Order = 120 = 2^3 * 3 * 5
    (1, 2, 3, 4, 5)
    (1, 2)

Maximal Parabolic Subgroups:
Permutation group g0 acting on a set of cardinality 5
Order = 12 = 2^2 * 3
    (3, 4)
    (1, 2)
    (4, 5)
Permutation group g1 acting on a set of cardinality 5
Order = 8 = 2^3
    (1, 2)
    (3, 4)
    (1, 3)(2, 4)

Type Set:
{@ 0, 1 @}
```

Example: Cube as a Coset Geometry (ex-4a5ca3)

The rank three geometry consisting of the vertices, edges, and faces of the cube, can be implemented as a coset geometry in the following way, which is much easier than having to give the full incidence graph :

```
> g := sub<Sym(8)|
>     (1,2,3,4)(5,6,7,8), (1,5)(2,6)(3,7)(4,8), (2,4,5)(3,8,6)>;
> g0 := Stabilizer(g,1);
> g1 := Stabilizer(g,{1,2});
> g2 := Stabilizer(g,{1,2,3,4});
> cg := CosetGeometry(g,{g0,g1,g2});
> cg;
Coset geometry cg with 3 types

Group:
Permutation group g acting on a set of cardinality 8
Order = 48 = 2^4 * 3
    (1, 2, 3, 4)(5, 6, 7, 8)
    (1, 5)(2, 6)(3, 7)(4, 8)
    (2, 4, 5)(3, 8, 6)
```

(continues on next page)

(continued from previous page)

```
Maximal Parabolic Subgroups:
Permutation group acting on a set of cardinality 8
Order = 8 = 2^3
    (1, 2)(3, 4)(5, 6)(7, 8)
    (2, 4)(6, 8)
Permutation group acting on a set of cardinality 8
Order = 6 = 2 * 3
    (2, 4, 5)(3, 8, 6)
    (3, 6)(4, 5)
Permutation group acting on a set of cardinality 8
Order = 4 = 2^2
    (1, 2)(3, 5)(4, 6)(7, 8)
    (1, 2)(3, 4)(5, 6)(7, 8)

Type Set:
{@ 0, 1, 2 @}
```

Example: Coset Geometry for Sym(6) (ex-16511c)

The following lines of Magma-code construct a coset geometry of rank 6 for the group $Sym(6)$.

```
> g := Sym(6);
> s := [Stabilizer(g,{1..j}) : j in {1..5}];
> cg := CosetGeometry(g,Set(s));
> cg;
Coset geometry cg with 5 types

Group:
Symmetric group g acting on a set of cardinality 6
Order = 720 = 2^4 * 3^2 * 5
    (1, 2, 3, 4, 5, 6)
    (1, 2)

Maximal Parabolic Subgroups:
Permutation group acting on a set of cardinality 6
Order = 48 = 2^4 * 3
    (3, 4)
    (4, 5)
    (5, 6)
    (1, 2)
Permutation group acting on a set of cardinality 6
Order = 48 = 2^4 * 3
    (1, 2)
    (2, 3)
    (3, 4)
    (5, 6)
Permutation group acting on a set of cardinality 6
Order = 36 = 2^2 * 3^2
    (1, 2)
    (4, 5)
    (5, 6)
    (2, 3)
Permutation group acting on a set of cardinality 6
Order = 120 = 2^3 * 3 * 5
    (2, 3)
```

(continues on next page)

(continued from previous page)

```
    (3, 4)
    (4, 5)
    (5, 6)
Permutation group acting on a set of cardinality 6
Order = 120 = 2^3 * 3 * 5
    (1, 2)
    (2, 3)
    (3, 4)
    (4, 5)

Type Set:
{@ 0, 1, 2, 3, 4 @}
```

155.3 Elementary Invariants

Points(D): IncGeom -> SetIndx

Elements(D): IncGeom -> SetIndx

Given an incidence geometry D, return the set of elements of D. These elements are the points of the incidence graph of D.

Types(D): IncGeom -> SetIndx

Given an incidence geometry D, return the set of types of D.

Types(C): CosetGeom -> SetIndx

Given a coset geometry C, return the set of types of C.

Rank(D): IncGeom -> RngIntElt

Given an incidence geometry D, return the rank of D, i.e. the cardinality of the set of types.

Rank(C): CosetGeom -> RngIntElt

Given a coset geometry C, return the rank of C.

IncidenceGraph(D): IncGeom -> GrphUnd, GrphVertSet, GrphEdgeSet

Given an incidence geometry D, return the incidence graph of D, its vertex set, and its edge set.

We remark that this function is not implemented for coset geometries but we may convert a coset geometry into an incidence geometry using **IncidenceGeometry** and then compute its incidence graph.

Group(C): CosetGeom -> GrpPerm

Given a coset geometry C, return the group from which C is constructed.

MaxParabolics(C): CosetGeom -> SetIndx

MaximalParabolics(C): CosetGeom -> SetIndx

Given a coset geometry C, return an indexed set containing the maximal parabolics of C.

MinParabolics(C): CosetGeom -> SetIndx

MinimalParabolics(C): CosetGeom -> SetIndx

Given a coset geometry C, return an indexed set containing the minimal parabolics of C.

Borel(C): CosetGeom -> GrpPerm

BorelSubgroup(C): CosetGeom -> GrpPerm

Given a coset geometry C, return the *Borel subgroup* of C, i.e. the intersection of all maximal parabolic subgroups of C.

Kernel(C): CosetGeom -> GrpPerm

Given a coset geometry C, return a permutation group which is its kernel, i.e. the subgroup of the Borel subgroup of C that fixes all elements of the geometry C.

`Kernels(C): CosetGeom -> SeqEnum`

Given a coset geometry C, return a sequence containing the i-kernel K_i of each maximal parabolic subgroup G_i of C. The i-kernel of the subgroup G_i is the subgroup consisting of all the elements of G_i that fix all the elements of the residue of G_i.

`Quotient(C, K): CosetGeom, GrpPerm -> CosetGeom`

Given a coset geometry $C = (G; (G_i)_{i \in I})$ and a permutation group K, return the coset geometry $(G/K; (G_i/K)_{i \in I})$ provided that K is a normal subgroup of G and of all the maximal parabolic subgroups of C.

155.4 Conversion Functions

In this section we describe the functions that are available to convert incidence geometries and coset geometries in other objects.

`IncidenceGeometry(C): CosetGeom -> IncGeom`

Construct the incidence geometry IG from the coset geometry C. This is done using Tits' algorithm described in the introduction of this chapter . The function returns one value: the incidence geometry IG.

`CosetGeometry(D): IncGeom -> BoolElt, CosetGeom`

Convert the incidence geometry D into a coset geometry.

If D is an incidence geometry that can be converted into a coset geometry, the coset geometry isomorphic to it is constructed in the following way. The group G of the coset geometry CG is the automorphism group of D. Magma determines a chamber C of D, that is a clique of the incidence graph of D containing one element of each type. To every element x in C, Magma associates a subgroup G_x which is the stabilizer of x in G. The subgroups $(G_x, x \in C)$ are the maximal parabolic subgroups of CG. In order to obtain a coset geometry combinatorially isomorphic to the incidence geometry we started with, the group G must be transitive on every rank two truncation of D. If this condition is satisfied, the function returns a boolean set to the value `true` and the coset geometry CG. Otherwise, the function returns `false`.

`Graph(D): IncGeom -> GrphUnd`

If `IsGraph(D)` returns `true`, this function construct the undirected graph corresponding to the incidence geometry D.

`Graph(C): CosetGeom -> GrphUnd`

If `IsGraph(C)` returns `true`, this function construct the undirected graph corresponding to the coset geometry C.

Example: Neumaier Geometry as a Coset Geometry (ex-cfd076)

Taking back the last example for incidence geometries, we can convert the Neumaier geometry into a coset geometry by typing the following command (`neumaier` is the Neumaier geometry constructed above):

```
> ok,cg := CosetGeometry(neumaier);
> ok;
true
```

This means the conversion has been done successfully. So *cg* is the coset geometry corresponding to `neumaier`.

```
> cg;
Coset geometry cg with 4 types
Group:
Permutation group acting on a set of cardinality 200
Order = 126000 = 2^4 * 3^2 * 5^3 * 7
```

(continues on next page)

(continued from previous page)

```
Maximal Parabolic Subgroups:
Permutation group acting on a set of cardinality 200
Order = 2520 = 2^3 * 3^2 * 5 * 7
Permutation group acting on a set of cardinality 200
Order = 2520 = 2^3 * 3^2 * 5 * 7
Permutation group acting on a set of cardinality 200
Order = 2520 = 2^3 * 3^2 * 5 * 7
Permutation group acting on a set of cardinality 200
Order = 2520 = 2^3 * 3^2 * 5 * 7

Type Set:
{@ 1, 2, 3, 4 @}
```

155.5 Residues

Let $\Gamma(X, \sim, t, I)$ be an incidence geometry and let F be a flag of Γ (i.e. a clique of the incidence graph of Γ).

We say that an element $x \in F$ is incident to the flag F if and only if x is incident to all elements in F, and we denote it $x \sim F$.

The *residue* Γ_F of the flag F in Γ is the geometry whose set of elements is $\{x \in X : x \sim F\} \backslash F$ and whose set of types is $I \backslash t(F)$, together with the restricted type function and incidence relation.

Let $\Gamma(G; (G_i)_{i \in I})$ be a coset geometry and assume that G acts flag–transitively on Γ. Let F be a flag of Γ. The *residue* of F is the coset geometry $\Gamma_F = \Gamma(\cap_{j \in F} G_j; (G_i \cap (\cap_{j \in F} G_j))_{i \in I \backslash t(F)})$.

`Residue(D, f): IncGeom, Set -> IncGeom`

Given an incidence geometry D and a flag f of D, return the residue of the flag f as an incidence geometry.

`Residue(C, f): CosetGeom, Set -> CosetGeom`

Given a coset geometry C and a subset f of the set of types of C, return the residue of the flag consisting in the maximal parabolics of C whose type is in f.

155.6 Truncations

Let $\Gamma(X, \sim, t, I)$ be an incidence geometry and let J be a subset of I. Then the *J–truncation* of Γ is the geometry whose set of elements is $t^{-1}(J)$, together with the restricted type function and incidence relation.

Let $J \subseteq I$. The *J–truncation* of the coset geometry $\Gamma(G; (G_i)_{i \in I})$ is the coset geometry $\Gamma(G; (G_j)_{j \in J})$.

`Truncation(D, t): IncGeom, Set -> IncGeom`

Given an incidence geometry D and t a subset of the set of types of D, return the *t–truncation* of D as an incidence geometry.

`Truncation(C, t): CosetGeom, Set -> CosetGeom`

Given a coset geometry C and t a subset of the set of types of C, return the *t–truncation* of C as a coset geometry.

155.7 Shadows

Let $\Gamma(X, \sim, t, I)$ be an incidence geometry. Let J be a subset of I and let F be a flag of Γ such that $t(F) \cap J = \emptyset$. The *J-shadow* of the flag F, denoted by $\sigma_J(F)$, is the set of flags of type J in the residue of the flag F.

`Shadow(D, I, F): IncGeom, Set, Set -> SetIndx`

Given an incidence geometry D, a subset I of the set of types of D and a flag F of D, return the I-shadow of the flag F as an indexed set of subsets of points of D.

155.8 Shadow Spaces

Let $\Gamma(X, \sim, t, I)$ be an incidence geometry. Let J be a subset of I. The *shadow space* $\Gamma(J)$ of Γ is the incidence structure whose point-set is the set of flags of type J of Γ and whose blocks are the $J-shadows$ of the flags of Γ.

`ShadowSpace(D, I): IncGeom, Set -> Inc`

Given an incidence geometry D and a subset I of the set of types of D, return the shadow space $D(I)$ as an incidence structure.

155.9 Automorphism Group and Correlation Group

These function are currently only available for incidence geometries.

An *automorphism* α of an incidence geometry $\Gamma(X, \sim, t, I)$ is an automorphism of the incidence graph of Γ such that for all $x \in X$, $t(\alpha(x)) = t(x)$. In other words, an automorphism cannot change the type of an element. The *automorphism group* of Γ, denoted $Aut(\Gamma)$, is the group of all automorphisms of Γ.

A *correlation* α of an incidence geometry $\Gamma(X, \sim, t, I)$ is an automorphism of the incidence graph of Γ such that for all $x, y \in X$, $t(x) = t(y) \Rightarrow t(\alpha(x)) = t(\alpha(y))$. The *correlation group* of Γ, denoted $Cor(\Gamma)$, is the group of all correlations of Γ.

It is obvious that $Aut(\Gamma)$ is a subgroup of $Cor(\Gamma)$.

For an incidence geometry Γ, we can compute $Aut(\Gamma)$ and $Cor(\Gamma)$ using the commands described below.

`AutomorphismGroup(D): IncGeom -> GrpPerm`

Given an incidence geometry D, return the group of type–preserving automorphisms of D as a permutation group of type `GrpPerm` acting on the set of elements of D.

`CorrelationGroup(D): IncGeom -> GrpPerm`

Given an incidence geometry D, return the group of automorphisms of D as a permutation group of type `GrpPerm` acting of the set on elements of D.

155.10 Properties of Incidence Geometries and Coset Geometries

Let us recall definitions of the properties described in this section.

An incidence geometry Γ is *flag–transitive* if for every two flags x, y of the same type of Γ, there exists an element g of $Aut(\Gamma)$ such that $g(x) = y$. We also say that $Aut(\Gamma)$ acts flag–transitively in this case.

Moreover, it is a flag–transitive geometry if it contains at least one chamber.

A coset geometry $\Gamma(G; (G_i)_{i \in I})$ is *flag–transitive* if for every two flags x, y of the same type of Γ, there exists an element g of G such that $g(x) = y$. It is then a flag–transitive geometry since the set $\{(G_i)_{i \in I}\}$ is a chamber of Γ.

`IsFTGeometry(D): IncGeom -> BoolElt`

Given an incidence geometry D, return `true` if and only if the automorphism group of D acts flag–transitively on D and D has at least one chamber.

`IsFTGeometry(C): CosetGeom -> BoolElt`

Given a coset geometry C, return `true` if and only if the group of C acts flag–transitively on C.

`IsFirm(X): IncGeom -> BoolElt`

`IsFirm(X): CosetGeom -> BoolElt`

Given either a coset geometry or an incidence geometry X that is flag transitive, return `true` if and only if every flag of X is contained in at least two chambers.

`IsThin(X): CosetGeom -> BoolElt`

`IsThin(X): IncGeom -> BoolElt`

Given either a coset geometry or an incidence geometry X that is flag transitive, return `true` if and only if every flag of X is contained in exactly two chambers.

`IsThick(X): CosetGeom -> BoolElt`

`IsThick(X): IncGeom -> BoolElt`

Given either a coset geometry or an incidence geometry X that is flag transitive, return `true` if and only if every flag of the geometry is contained in exactly three chambers.

`IsResiduallyConnected(X): IncGeom -> BoolElt`

`IsRC(X): IncGeom -> BoolElt`

`IsResiduallyConnected(X): CosetGeom -> BoolElt`

`IsRC(X): CosetGeom -> BoolElt`

Given either a coset geometry or an incidence geometry X that is flag transitive, return `true` if and only if every residue of rank at least two of X has a connected incidence graph.

`IsGraph(D): IncGeom -> GrphUnd`

Given an incidence geometry D, tests if this incidence geometry corresponds to a graph: D must be of rank two and such that for one of the two types, say e, all elements of this type are incident with exactly two elements of the other type. Elements of type e then correspond to edges of an undirected graph and elements of the other type to the vertices of that graph.

`IsGraph(C): CosetGeom -> GrphUnd`

Given a coset geometry C, tests if this geometry corresponds to a graph: C must be of rank two and one of the two maximal parabolic subgroups, say G_e, must contain the Borel subgroup as a subgroup of index 2. In that case, the cosets of G_e correspond to edges of a graph and the cosets of the other maximal parabolic subgroup correspond to the vertices of this graph.

155.11 Intersection Properties of Coset Geometries

Let $\Gamma(X, *, t, I)$ be an incidence geometry. Given a type $i \in I$, for any flag F of Γ, we define the *i-shadow* $\sigma_i(F)$ as the set of elements of type i incident with F.

We define the intersection property (IP) as it appears in [Bue79]:

(IP) for every type i, the intersection of the i-shadows of a variety x and a flag F is empty or it is the i-shadow of a flag incident to x and F. The same holds on the residues.

In earlier works, the second author, together with other persons, among whom are Francis Buekenhout, Michel Dehon, and Philippe Cara, imposed a condition called $(IP)_2$. This condition asks that all rank two residues of Γ satisfy (IP).

If Γ is a geometry of rank n, we could define a property $(IP)_k$ in the following way (for $k = 2, \ldots, n$) as suggested by Francis Buekenhout:

$(IP)_k$ for every residue R of rank k of Γ, for every type i in the set of types of R, the intersection of the i-shadows of a variety x and a flag F is empty or it is the i-shadow of a flag incident to x and F.

In [JL04], Pascale Jacobs and Dimitri Leemans have designed good algorithms to test these intersection properties. The Magma implementation is available with the following functions.

`HasIntersectionPropertyN(C, n): CosetGeom, RngIntElt -> BoolElt, BoolElt`

Given a coset geometry C and a positive integer n, this function determines firstly, whether C satisfies the intersection property of rank n, and secondly, whether C satisfies the weak intersection property of rank n. A boolean value corresponding to each of these cases is returned.

`HasIntersectionProperty(C): CosetGeom -> BoolElt`

Given a coset geometry C, this function returns the boolean value `true` if and only if C satisfies the intersection property.

`HasWeakIntersectionProperty(C): CosetGeom -> BoolElt`

Given a coset geometry C, this function returns the boolean value `true` if and only if C satisfies the weak intersection property.

155.12 Primitivity Properties on Coset Geometries

`IsPrimitive(C): CosetGeom -> BoolElt`

`IsPRI(C): CosetGeom -> BoolElt`

`IsPrimitive(C): CosetGeom -> BoolElt`

`IsPRI(C): CosetGeom -> BoolElt`

Given a coset geometry C, this function returns the boolean value `true` if and only if C is a primitive geometry, i.e. all of its maximal parabolic subgroups are maximal subgroups of its group.

`IsWeaklyPrimitive(C): CosetGeom -> BoolElt`

`IsWPRI(C): CosetGeom -> BoolElt`

`IsWeaklyPrimitive(C): CosetGeom -> BoolElt`

`IsWPRI(C): CosetGeom -> BoolElt`

Given a coset geometry C, this function returns the boolean value `true` if and only if C is a weakly primitive geometry, i.e. at least one of its maximal parabolic subgroups is a maximal subgroup of its group.

`IsResiduallyPrimitive(C): CosetGeom -> BoolElt`

`IsRPRI(C): CosetGeom -> BoolElt`

`IsResiduallyPrimitive(C): CosetGeom -> BoolElt`

`IsRPRI(C): CosetGeom -> BoolElt`

Given a coset geometry C, this function returns the boolean value `true` if and only if C is a primitive geometry and all of its residues are as well.

`IsResiduallyWealyPrimitive(C): CosetGeom -> BoolElt`

`IsRWPRI(C): CosetGeom -> BoolElt`

`IsRWP(C): CosetGeom -> BoolElt`

`IsResiduallyWeaklyPrimitive(C): CosetGeom -> BoolElt`

`IsRWPRI(C): CosetGeom -> BoolElt`

`IsRWP(C): CosetGeom -> BoolElt`

Given a coset geometry C, this function returns the boolean value `true` if and only if C is a weakly primitive geometry and all of its residues are as well.

`IsLocallyTwoTransitive(C): CosetGeom -> BoolElt`

`Is2T1(C): CosetGeom -> BoolElt`

Given a coset geometry C, this function returns the boolean value `true` if and only if C is locally two-transitive, i.e. all of its minimal parabolic subgroups have a two-transitive action on the cosets of the Borel subgroup.

`LocallySArcTransitive(C): CosetGeom -> RngIntElt`

Given a coset geometry C, this function returns the largest integer s such that C is locally s-arc-transitive but not locally $(s+1)$-arc-transitive. If the function returns 10, it means C is a coset geometry corresponding to a polygon and hence this value must be considered as infinity.

155.13 Diagram of an Incidence Geometry

As Francis Buekenhout defined it in [Bue79], the diagram of a firm, residually connected, flag–transitive geometry Γ is a complete graph K, whose vertices are the elements of the set of types I of Γ, provided with some additional structure which is further described as follows. To each vertex $i \in I$, we attach the order s_i which is $|\Gamma_F| - 1$ where F is any flag of type $I \backslash \{i\}$, and the number n_i of elements of type i of Γ. To every edge $\{i, j\}$ of K, we associate three positive integers d_{ij}, g_{ij}, and d_{ji} where g_{ij} (the gonality) is equal to half the girth of the incidence graph of a residue Γ_F of type $\{i, j\}$, and d_{ij} (resp. d_{ji}), the *i–diameter* (resp. *j–diameter*) is the greatest distance from some fixed i–element (resp. j–element) to any other element in Γ_F.

On a picture of the diagram, this structure will often be depicted as follows.

```
i  d_ij  g_ij  d_ji  j
O-------------------O
s_i-1             s_j-1
n_i                 n_j
```

Moreover, when $g_{ij} = d_{ij} = d_{ji} = n \neq 2$, we only write g_{ij} and if $n = 2$, we do not draw the edge $\{i, j\}$.

Diagram(D): IncGeom -> GrphUnd, GrphVertSet, GrphEdgeSet

Diagram(C): CosetGeom -> GrphUnd, GrphVertSet, GrphEdgeSet

Given a firm, residually connected, and flag–transitive incidence geometry D or a coset geometry C, return a complete graph K whose vertices and edges are labelled in the following way. Every vertex i of K is labelled with a sequence of two positive integers, the first one being s_i and the second one being the number of elements of type i in D. Every edge $\{i, j\}$ is labelled with a sequence of three positive integers which are respectively d_{ij}, g_{ij}, and d_{ji}.

Observe that both these functions do the same but the second one works much faster than the first one since it uses groups to compute the parameters of the diagram. So when the user has to compute the diagram of an incidence geometry, it is strongly advised that he first converts it into a coset geometry and then computes the diagram of the corresponding coset geometry.

Example: Diagram of the Petersen Geometry (ex-5b021d)

Back to the Petersen graph : let us now explore a little more the first incidence geometry we constructed. Suppose that *ig* is the rank two incidence geometry corresponding to the Petersen graph. We may test whether it is firm, residually connected, and flag–transitive.

```
> IsFirm(ig);
true
> IsRC(ig);
true
> IsFTGeometry(ig);
true
```

Since it satisfies these three properties, we can compute its diagram.

```
> d, v, e := Diagram(ig);
> d;
Graph
Vertex  Neighbours

1       2 ;
2       1 ;
> for x in v do print x, Label(x);end for;
1 [ 1, 10 ]
2 [ 2, 15 ]
> for x in e do print x, Label(x);end for;
{1, 2} [ 5, 5, 6 ]
```

We thus see that the diagram of ig can be drawn as follows.

```
1      5  5  6      2
0------------------0
1                  2
10                15
```

Example: Diagram of the Cube (ex-ee5e4d)

Back to the cube: let ig be the rank three incidence geometry of the cube as constructed above and compute its diagram.

```
> cube := IncidenceGeometry(g);
> d, v, e := Diagram(cube);
> d;
Graph
Vertex  Neighbours

1       2 3 ;
2       1 3 ;
3       1 2 ;

> for x in v do x, Label(x); end for;
1 [ 1, 8 ]
2 [ 1, 12 ]
3 [ 1, 6 ]
> for x in e do x, Label(x); end for;
{1, 2} [ 4, 4, 4 ]
{1, 3} [ 2, 2, 2 ]
{2, 3} [ 3, 3, 3 ]
```

So the diagram can be depicted as follows.

```
1         4         2         3         3
0------------------0------------------0
1                  1                  1
8                 12                  6
```

Example: Diagram of the Hoffman-Singleton Geometry (ex-cde392)

Back to the Hoffman-Singleton graph:

```
> ig := IncidenceGeometry(HoffmanSingletonGraph());
> d, v, e := Diagram(ig); d;
Graph
Vertex  Neighbours

1       2 ;
2       1 ;

> for x in v do print x, Label(x); end for;
1 [ 1, 50 ]
2 [ 6, 175 ]
> for x in e do print x, Label(x); end for;
{1, 2} [ 5, 5, 6 ]
```

So the diagram can be depicted as follows.

```
1      5  5  6      2
0-------------------0
1                    6
50                 175
```

Example: Diagram of Neumaier's Geometry (ex-3dd4a3)

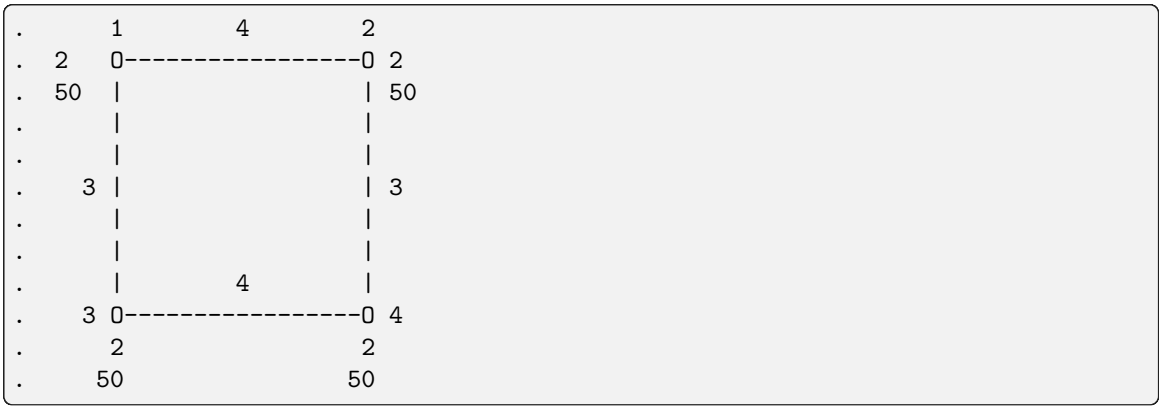
Back to Neumaier's geometry : let *ig* be the rank four incidence geometry of Neumaier, as described above and compute its diagram.

```
> d, v, e := Diagram(neumaier);
> d;
Graph
Vertex  Neighbours

1       2 3 4 ;
2       1 3 4 ;
3       1 2 4 ;
4       1 2 3 ;

> for x in v do print x, Label(x); end for;
1 [ 2, 50 ]
2 [ 2, 50 ]
3 [ 2, 50 ]
4 [ 2, 50 ]
> for x in e do print x, Label(x); end for;
{1, 2} [ 4, 4, 4 ]
{1, 3} [ 2, 2, 2 ]
{1, 4} [ 3, 3, 3 ]
{2, 3} [ 3, 3, 3 ]
{2, 4} [ 2, 2, 2 ]
{3, 4} [ 4, 4, 4 ]
```

So the diagram can be depicted as follows.



155.14 C-Groups

There is a strong link between C-groups and coset geometries.

Given a group G and a set S of elements of G, we say that (G, S) satisfies the *intersection property* if for every subsets I and J of S, we have $\langle I \rangle \cap \langle J \rangle = \langle I \cap J \rangle$.

We say that (G, S) satisfies the *string property* or *commuting property* if there is an ordering of the elements of S such that any two elements of S that are not one after the other commute.

A group G is a *C-group* if its set of generators is a set of involutions that satisfies the intersection property. Moreover G is a *string-C-group* if G is a C-group and satisfies the string property.

When G is a C-group, we associate to it a *Coxeter diagram*, that is a graph whose vertices are the generators of G. Two vertices are joined by an edge provided the corresponding two generators do not commute. Moreover, to each edge we put a label that is the order of the product of the two elements corresponding to the two vertices.

HasIntersectionProperty(G): GrpPerm -> BoolElt

Given a permutation group G, determines whether or not its set of generators $G.1, \ldots, G.n$ satisfy the intersection property.

HasStringProperty(G): GrpPerm -> BoolElt

Given a permutation group G, determines whether or not there exists an ordering $G.1, \ldots, G.n$ of the generators of G such that $G.iG.j = G.jG.i$ whenever $|i - j| > 1$.

IsCGroup(G): GrpPerm -> BoolElt

Given a permutation group G, determines whether or not its set of generators are involutions that satisfy the string property and the intersection property.

IsStringCGroup(G): GrpPerm -> BoolElt

Given a permutation group G, determines whether or not its set of generators are involutions that satisfy the string property and the intersection property.

CosetGeometryFromCGroup(G): GrpPerm -> CosetGeom

Given a permutation group G that is a C-group, construct the coset geometry D whose maximal parabolic subgroups are the subgroups generated by all but one generator of G. If G has n generators, D will be a coset geometry of rank n.

CosetGeometryToCGroup(D): CosetGeom -> GrpPerm

Given a coset geometry D that has Borel = 1 and with minimal parabolic subgroups being cyclic groups of order 2, construct G as the group generated by the involutions of the minimal parabolic subgroups and returns it provided it satisfies the intersection property.

CoxeterDiagram(G): GrpPerm -> GrphUnd

Given a C-group G, construct its Coxeter diagram.

Example: Converting the Cube to a C-Group (ex-306b2c)

Again back to the cube we constructed above as an incidence geometry. We can convert it into a coset geometry and then a string-C-group very easily using the following commands.

```
> cube := IncidenceGeometry(g);
> ok, cg := CosetGeometry(cube);
> G := CosetGeometryToCGroup(cg);
> G;
Permutation group G acting on a set of cardinality 26
    (1, 2)(3, 4)(5, 6)(7, 8)(10, 12)(13, 14)(15, 16)(17, 19)(23, 25)
    (2, 4)(6, 8)(9, 12)(10, 11)(14, 16)(17, 18)(19, 20)(22, 25)(23, 24)
    (3, 6)(4, 5)(10, 14)(11, 18)(12, 13)(15, 19)(16, 17)(21, 22)(24, 26)
> IsStringCGroup(G);
true
```

Example: 5-Simplex from a C-Group (ex-ffd840)

We also now have a way to construct a coset geometry (and hence the corresponding incidence geometry) from C-groups. For instance, using E. H. Moore's presentation for the symmetric groups, we can easily construct the 5-simplex as a String C-group.

```
> G:=sub<Sym(6)|(1,2),(2,3),(3,4),(4,5),(5,6)>;
> IsCGroup(G);
true
```

(continues on next page)

(continued from previous page)

```
> IsStringCGroup(G);
true
> d,v,e:=CoxeterDiagram(G);
> for x in e do print x,Label(x);end for;
{1, 2} 3
{2, 3} 3
{3, 4} 3
{4, 5} 3
> cg:=CosetGeometryFromCGroup(G);
> IsThin(cg);
true
> IsRC(cg);
true
> IsFTGeometry(cg);
true
> d,v,e := Diagram(cg);
> for x in v do print x,Label(x);end for;
1 [ 1, 20 ]
2 [ 1, 15 ]
3 [ 1, 6 ]
4 [ 1, 6 ]
5 [ 1, 15 ]
> for x in e do print x,Label(x);end for;
{1, 2} [ 3, 3, 3 ]
{1, 3} [ 2, 2, 2 ]
{1, 4} [ 2, 2, 2 ]
{1, 5} [ 3, 3, 3 ]
{2, 3} [ 3, 3, 3 ]
{2, 4} [ 2, 2, 2 ]
{2, 5} [ 2, 2, 2 ]
{3, 4} [ 2, 2, 2 ]
{3, 5} [ 2, 2, 2 ]
{4, 5} [ 3, 3, 3 ]
```

155.15 C^+-Groups

The concept of C^+-groups has been introduced by Fernandes, Leemans, and Weiss [MEFW] to generalise the study of rotational subgroups of Coxeter groups and study chiral geometries.

Let R be the set of generators of a group G. Suppose $R := \{\alpha_1, \ldots, \alpha_{r-1}\}$. Define $\alpha_0 := 1_{G^+}$ and $\alpha_{ij} := \alpha_i^{-1}\alpha_j$ for all $0 \leq i, j \leq r-1$. Let $G_I := \langle \alpha_{ij} \mid i, j \in I \rangle$ for $I \subseteq \{0, \ldots, r-1\}$.

If the pair (G, R) satisfies the condition that $\forall I, J \subseteq \{0, \ldots, r-1\}$, with $|I|, |J| \geq 2, G_I \cap G_J = G_{I \cap J}$, called the *intersection property* IP^+ (obtained in analogy with the intersection property of C-groups keeping only those equalities that involve subsets I and J of cardinality at least two), we say that (G, R) is a *C^+-group*.

It follows immediately from the intersection property IP^+, that R is an independent generating set for G, that means that $\alpha_i \notin \langle \alpha_j : j \neq i \rangle$.

A group G whose set of generators satisfies the intersection property IP^+ is called a *C^+-group*.

The *B-diagram* of G is a graph whose vertices are $\alpha_0 := Id(G)$ and $\alpha_i = G.i$ for each generator $G.i$ of G. Two vertices are joined by an edge provided the order of $\alpha_i^{-1}\alpha_j$ is at least 3. Moreover, this edge is labelled with the order of $\alpha_i^{-1}\alpha_j$.

`HasIntersectionPropertyPlus(G): GrpPerm -> BoolElt`

`IsCPlusGroup(G): GrpPerm -> BooElt`

Given a permutation group G, determines whether or not it is a C^+-group, that is whether or not its set of generators $G.1, \ldots, G.n$ satisfy the intersection property IP^+.

`CosetGeometryFromCPlusGroup(G): GrpPerm -> CosetGeom`

Given a C^+-group G, construct the coset geometry whose maximal parabolic subgroups are either generated by all but one generator of G, or the generators $G.1^{-1}G.i$ with $i = 2, \ldots, n$ where n is the number of generators of G. The constructed geometry is of rank $n + 1$.

`BDiagram(G): GrpPerm -> GrphUnd`

Given a C^+-group G, construct the B-diagram of G.

Example: 5-Simplex from Its Rotation Group (ex-4866f0)

C^+-groups can be used to construct coset geometries as well. The user has to be careful that the corresponding geometry will have its rotation group acting on it. Let us again construct the 5-simplex using its rotation subgroup that is the alternating group A_6.

```
> Gp := sub<Alt(6)|(1,2,3),(1,2)(3,4),(1,2)(4,5),(1,2)(5,6)>;
> IsCPlusGroup(Gp);
true
> d,v,e := BDiagram(Gp);
> for x in e do print x,Label(x); end for;
{1, 2} 3
{2, 3} 3
{3, 4} 3
{4, 5} 3
> cg:=CosetGeometryFromCPlusGroup(Gp);
> IsThin(cg);
false
```

Above we see that the coset geometry constructed is not considered thin by Magma in terms of groups. This is because the stabilizer of a flag of rank 4 is the identity in A_6. To get the group S_6 acting, we have to convert the coset geometry into an incidence geometry and convert that incidence geometry in a coset geometry again. The latter one will have the automorphism group of the underlying incidence geometry acting on it, that is S_6.

```
> ig:=IncidenceGeometry(cg);
> ok,cg2:=CosetGeometry(ig);
> ok;
true
> IsThin(cg2);
true
> IsRC(cg2);
true
> IsFTGeometry(cg2);
true
> d,v,e:=Diagram(cg2);
> for x in v do print x,Label(x); end for;
1 [ 1, 15 ]
2 [ 1, 6 ]
3 [ 1, 15 ]
4 [ 1, 6 ]
5 [ 1, 20 ]
> for x in e do print x,Label(x); end for;
{1, 2} [ 3, 3, 3 ]
{1, 3} [ 2, 2, 2 ]
{1, 4} [ 2, 2, 2 ]
{1, 5} [ 3, 3, 3 ]
```

(continues on next page)

(continued from previous page)

```
{2, 3} [ 2, 2, 2 ]
{2, 4} [ 2, 2, 2 ]
{2, 5} [ 2, 2, 2 ]
{3, 4} [ 3, 3, 3 ]
{3, 5} [ 3, 3, 3 ]
{4, 5} [ 2, 2, 2 ]
```

The interest of C^+-group is in that they permit also to construct chiral geometries. This is described in more details in [MEFW].

155.16 Bibliography

- [Bue79] Francis Buekenhout. Diagrams for geometries and groups. *J. Combin. Theory Ser. A*, 27(2):121–151, 1979.
- [Bue95] Francis Buekenhout. *Handbook of incidence geometry.* North-Holland, Amsterdam, 1995.
- [BCon] (no details available)
- [JL04] Pascale Jacobs and Dimitri Leemans. An algorithmic analysis of the intersection property. *LMS J. Comput. Math.*, 7:284–299 (electronic), 2004.
- [MEFW] D. Leemans M. E. Fernandes and A. I. Weiss. Highly symmetric hypertopes. Preprint, arXiv:1604.03162.
- [Pas94] Antonio Pasini. *Diagram geometries.* Oxford Science Publications. The Clarendon Press Oxford University Press, New York, 1994. ISBN 0-19-853497-3.
- [Tit62] Jacques Tits. Géométries polyédriques et groupes simples. *Atti 2a Riunione Groupem. Math. Express. Lat. Firenze*, pages 66–88, 1962.

Chapter 156

Convex Polytopes and Polyhedra

156.1 Introduction and First Examples

This is a package with tools to build and analyse finite-dimensional, finitely-generated, rational polyhedra. In general a polyhedron is the Minkowski sum of a rational polytope (the compact, convex hull of finitely many rational points) and a cone (the convex hull of finitely many rational half-lines emanating from the origin).

It is important to note that we are discussing only *rational* polyhedra: their vertices lie in some finite-dimensional rational vector space $L = \mathbb{Q}^n$ with the supporting hyperplanes represented in its dual (loosely speaking, all faces have rational gradients). There is a natural lattice in this setup: the integral sublattice $\mathbb{Z}^n$ of L. This is a lattice only in the sense of being a $\mathbb{Z}$-module; it does not come with a preferred definite quadratic form (or norm), and so, in particular, lengths of vectors are not defined. (We explain below how volumes are nevertheless defined relative to the integral lattice.) A point of L is called 'integral' if it lies in the integral sublattice, but there is no requirement that vertices of polyhedra be integral. Notwithstanding all these remarks, we refer to such ambient spaces L as 'lattices'.

This chapter is organised so that functions applying to compact polyhedra, namely polytopes, are collected together in coherent blocks as much as possible. Section *Polytopes, Cones, and Polyhedra* lists various ways of constructing polytopes, cones, and polyhedra, including Minkowski sum and other arithmetical and set-theoretical operations. Section *Basic Combinatorics of Polytopes and Polyhedra* presents the basic combinatorics associated to all polyhedra such as recovering their vertices or faces. The next section, Section *The Combinatorics of Polytopes*, is dedicated to polytopes. Most of the functions described here rely on compactness; they include enumeration of points and triangulation. Then Section *Cones and Polyhedra* details functions that apply to polyhedra more generally, although this includes some functions that apply to cones in particular: computing the integral generators of $C \cap \mathbb{Z}^n$ as a semigroup, for example, where C is a cone in L. Finally, there is Section *Toric Lattices* outlining the functions that operate on the ambient lattices. These functions are not usually required, since for the most part the ambient space is a book-keeping device that the package handles automatically, but they arise as domains and codomains of maps. This provides a mechanism for modifying the integral sublattice, which can be used to change the underlying notion of which points count as integral.

Before starting, we outline the capabilities of the package by extended examples. The first illustrates the basic machinery available for the analysis of polytopes—that is, the compact convex hulls of finitely many points.

Example: Polytope as a Convex Hull `(ex-422724)`

> A polytope can be constructed as the convex hull of finitely many points; the points can be denoted simply as sequences of integers (or even rational numbers), although they will be cast as points of an ambient lattice (that is, a $\mathbb{Q}$-vector space marked with a spanning lattice $\mathbb{Z}^n$). For example, we build a polytope P as the convex hull of the four points $(1, 0, 0)$, $(0, 1, 0)$, $(1, -3, 5)$, and $(-2, 2, -5)$.

```
> P := Polytope([ [0,0,0], [1,0,0], [0,1,0], [1,-3,5], [-2,2,-5] ]);
> P;
3-dimensional polytope P with 5 generators:
    ( 0,  0,  0),
    ( 1,  0,  0),
    ( 0,  1,  0),
    ( 1, -3,  5),
    (-2,  2, -5)
```

No analysis of P has been carried out in its construction. In fact, one of its defining 'generators' is not required, as can be seen from the vertices of P (which after this calculation will set as the default printing information).

```
> Vertices(P);
[
    (1, 0, 0),
    (0, 1, 0),
    (1, -3, 5),
    (-2, 2, -5)
]
> P;
3-dimensional polytope P with 4 vertices:
    ( 1,  0,  0),
    ( 0,  1,  0),
    ( 1, -3,  5),
    (-2,  2, -5)
```

One can extract the combinatorial components of P. Here we recover the facets of P, its faces of codimension 1, each of which is regarded as a polytope in its own right.

```
> Facets(P);
[
    2-dimensional polytope with 3 vertices:
        ( 1, -3,  5),
        (-2,  2, -5),
        ( 0,  1,  0),
    2-dimensional polytope with 3 vertices:
        ( 1, -3,  5),
        (-2,  2, -5),
        ( 1,  0,  0),
    2-dimensional polytope with 3 vertices:
        (1, -3,  5),
        (0,  1,  0),
        (1,  0,  0),
    2-dimensional polytope with 3 vertices:
        (-2,  2, -5),
        ( 0,  1,  0),
        ( 1,  0,  0)
]
```

Computing the (integral) points contained in a polytope (including points on its boundary) is one of the principal operations for polytopes.

```
> Points(P);
[
    (-2, 2, -5),
    (0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
    (0, 1, 0),
    (1, -3, 5),
    (1, 0, 0)
]
```

The interior points or boundary points can be retrieved separately using `InteriorPoints(P)` and `BoundaryPoint(P)`.

One can also compute the volume of a polytope.

```
> Volume(P);
20
```

The volume is the lattice-normalised volume: $\mathrm{Vol}(P) = n! \times \mathrm{vol}(P)$, where $\mathrm{vol}(P)$ is the Euclidean volume of P.

The polar, or dual, to P can be computed.

```
> D := Polar(P);
> D;
3-dimensional polytope D with 4 vertices:
    ( 3,  -1, -7/5),
    (-1,   3,  9/5),
    (-1,  -1,  1/5),
    (-1,  -1, -3/5)
```

The polar of P is again a polytope in this case (simply because the origin lies in the strict interior of P), although its vertices need not be integral.

The Ehrhart series,

$$\mathrm{Ehr}_D(t) = \Sigma_{n\geq 0} \#\left(nD \cap \mathbb{Z}^n\right) t^n,$$

computes the number of points in multiples of a polytope D.

```
> EhrhartCoefficients(D,10);
[ 1, 7, 33, 91, 193, 355, 585, 899, 1309, 1827, 2469 ]
```

The (infinite) Ehrhart series can be recovered as a rational function.

```
> E<t> := EhrhartSeries(D);
> E;
(t^7 + 4*t^6 + 15*t^5 + 12*t^4 + 12*t^3 + 15*t^2 + 4*t + 1)/(t^8 - 3*t^7 + 3*t^6
    - t^5 - t^3 + 3*t^2 - 3*t + 1)
```

It is possible to make maps of the underlying lattices and apply them to polyhedra.

```
> L := Ambient(P);
> K := Quotient(L![1,2,3]);
> K;
2-dimensional toric lattice K = (Z^3) / <1, 2, 3>
> f := LatticeMap(L, K, Matrix(3,2,[1,0,0,1,0,0]));
> Q := Image(f,P);
> Vertices(Q);
[
    (1, 0),
    (0, 1),
    (1, -3),
    (-2, 2)
]
```

In this case the image polytope $Q = f(P)$ is not a simplex. We can triangulate it.

```
> Triangulation(Q);
{
    2-dimensional polytope with 3 vertices:
        (1, 0),
        (1, -3),
        (0, 1)
    2-dimensional polytope with 3 vertices:
        (1, -3),
        (-2, 2),
        (0, 1)
}
```

Our second example illustrates the machinery available for cones and polyhedra more generally.

Example: Cones Generated by Rays (ex-bab5a6)

A ray is a rational half-line $\mathbb{Q}^+v$ from the origin passing through a point $v \in \mathbb{Q}^n$. A cone is the convex hull of finitely many rays.

```
> C := Cone([[2,2],[1,1/2]]);
> C;
Cone C with 2 generators:
    ( 2,   2),
    ( 1, 1/2)
```

Although any non-zero point on a ray is enough to define it, often the non-zero integral point nearest the origin is preferred. As with polytopes, there is no analysis of a cone C on its construction, but the minimal generators are displayed once some function has forced their calculation.

```
> Rays(C);
[
    (1, 1),
    (2, 1)
]
> C;
2-dimensional simplicial cone C with 2 minimal generators:
    (1, 1),
    (2, 1)
```

A polyhedron is the Minkowski sum (simply denoted by a plus sign) of a polytope and a cone.

```
> P := StandardSimplex(2);
> P + C;
2-dimensional polyhedron with 2-dimensional finite part with 3
vertices:
    (1, 0),
    (0, 1),
    (0, 0)
and 2-dimensional infinite part given by a cone with 2 minimal
generators:
    (2, 1),
    (1, 1)
```

The polyhedron $P + C$ is now subject to standard combinatorial analysis, such as computing its facets; some of these may be (compact) polytopes, while others may themselves be polyhedra described as the sum of a compact, or finite, part and a cone, or infinite part.

```
> Facets(P + C);
[
    1-dimensional polytope with 2 vertices:
        (1, 0),
        (0, 0),
    1-dimensional polyhedron with 0-dimensional finite part with one
    vertex:
        (0, 1)
    and 1-dimensional infinite part given by a cone with one minimal
    generator:
        (1, 1),
    1-dimensional polytope with 2 vertices:
        (0, 1),
        (0, 0),
    1-dimensional polyhedron with 0-dimensional finite part with one
    vertex:
        (1, 0)
    and 1-dimensional infinite part given by a cone with one minimal
    generator:
        (2, 1)
]
```

156.2 Polytopes, Cones, and Polyhedra

A *(rational) polytope* (in a lattice $L = \mathbb{Q}^n$) is the convex hull of finitely many points in L. (The points do not need to be integral points.)

A *cone* C (in a lattice L) is the convex hull of finitely many rays (or zero). Precisely, a *ray* is a half line emanating from the origin, but, as is common, we treat rays synonymously with the first integral lattice point on the ray outside the origin—thus, for example, an intrinsic that returns a ray will actually return an primitive integral point of the ambient lattice. Rays that are the intersection of a linear hyperplane with C are called *extreme rays* of C—these are the corner edges of C. A cone is *regular* if it is generated (as a semigroup in the $\mathbf{Z}$-module $\mathbb{Z}^n \subset L$) by its extreme rays (in other words, by the integral vectors on its extreme rays).

Combining all concepts so far, a *(rational) polyhedron* is the Minkowski sum of a polytope and a cone—the latter is the *tail cone* and is unique; the smallest possible polytope is unique but seems not to be used in the theory. We also use the expressions *compact part* and *infinite part* to indicate the polytope and the cone used to define a polyhedron.

156.2.1 Polytopes

`Polytope(Q): SeqEnum -> TorPol`

The polytope defined by taking the convex hull of the sequence of points Q, where the points can be specified as sequences of integers of rational numbers (of some fixed length) or as points of a common lattice.

`PolyhedronWithInequalities(A, c): SeqEnum, [RngIntElt] -> TorPol`

`PolyhedronWithInequalities(A, c): [TorLatElt], [RngIntElt] -> TorPol`

The polyhedron defined by the system of inequalities $A_i v \geq c_i$ for each vector A_i and integer c_i.

`RandomPolytope(L, n, k): TorLat, RngIntElt, RngIntElt -> TorPol`

`RandomPolytope(d, n, k): RngIntElt, RngIntElt, RngIntElt -> TorPol`

A polytope in the toric lattice L (or a toric lattice of dimension d) generated by n random integral points with coefficients of modulus at most the non-negative integer k.

`BoundingBox(P): TorPol -> TorPol, TorLatElt, TorLatElt`

The smallest box Q containing the polytope P. Also gives the bottom-left and top-right corners of Q.

`Polar(P): TorPol -> TorPol`

The polar dual polyhedron to the polyhedron P, namely

$$P^\star = \{u \in L^\vee \mid u \cdot v \geq -1 \ \forall v \in P\}\,.$$

`CrossPolytope(L): TorLat -> TorPol`

`CrossPolytope(d): RngIntElt -> TorPol`

The maximum dimensional cross-polytope in the lattice L, or of dimension d for an integer d.

`StandardSimplex(L): TorLat -> TorPol`

`StandardSimplex(d): RngIntElt -> TorPol`

The simplex given by the standard basis and the origin of the lattice L, or the d-dimensional simplex given by the standard basis and the origin for an integer d.

`CyclicPolytope(L, n): TorLat, RngIntElt -> TorPol`

`CyclicPolytope(d, n): RngIntElt, RngIntElt -> TorPol`

The cyclic polytope generated by n points in the lattice L, or by n points in d-dimensional space for an integer d.

`PolytopeOfProjectiveSpace(d): RngIntElt -> TorPol`

`PolytopeOfProjectiveSpace(L): TorLat -> TorPol`

`PolytopeOfWPS(d): RngIntElt -> TorPol`

A simplex whose spanning fan corresponds to projective space $\mathbb{P}^d$.

`PolytopeOfWPS(W): [RngIntElt] -> TorPol`

`PolytopeOfWPS(L, W): [RngIntElt] -> TorPol`

A simplex whose spanning fan corresponds to weighted projective space $\mathbb{P}(W)$.

156.2.2 Cones

Cones in lattices are of type `TorCon`.

`Cone(A): Seq -> TorCon`

The cone in a lattice generated by a sequence A of its elements (or simply of sequences of integer or rational coefficients).

`Cone(v): TorLatElt -> TorCon`

The cone in a lattice generated by a single element v of that lattice; namely the single ray $\mathbb{Q}^+ v$.

`ConeWithInequalities(B): Set -> TorCon`

The cone in a lattice L defined by the set B of elements of the dual lattice $L^\vee$ (or simply by a set of sequences of integer or rational coefficients). The cone is the intersection of half-spaces $v \cdot u \geq 0$ as v runs through B:

$$\bigcap_{v \in B} \{u \in L \mid v \cdot u \geq 0\}\,.$$

`FullCone(L): TorLat -> TorCon`

`FullCone(n): RngIntElt -> TorCon`

The cone which is the entire lattice L, or the lattice of dimension n for a positive integer n.

`PositiveQuadrant(L): TorLat -> TorCon`

`PositiveQuadrant(n): RngIntElt -> TorCon`

The cone in the lattice L generated by the standard basis vectors of L, or that of the lattice of dimension n for a positive integer n.

`ZeroCone(L): TorLat -> TorCon`

`ZeroCone(n): RngIntElt -> TorCon`

The cone which consists of only the origin of the lattice L, or of the lattice of dimension n for a positive integer n.

`RandomCone(d, n, k): RngIntElt, RngIntElt, RngIntElt -> TorCon`

`RandomCone(L, n, k): TorLat, RngIntElt, RngIntElt -> TorCon`

A random cone in a d-dimensional toric lattice – or in the toric lattice L – generated by n points whose coefficients lies between $-k$ and k.

`RandomPositiveCone(d, n, k): RngIntElt, RngIntElt, RngIntElt -> TorCon`

`RandomPositiveCone(L, n, k): TorLat, RngIntElt, RngIntElt -> TorCon`

A random cone in a d-dimensional toric lattice – or in the toric lattice L – generated by n points whose coefficients lies between 0 and k.

`Dual(C): TorCon -> TorCon`

The dual cone to the cone C, namely

$$C^\vee = \{u \in L^\vee \mid v \cdot u \geq 0 \text{ for all } v \in C\}\,.$$

where C lies in the lattice L.

`NormalisedCone(P): TorPol -> TorCon`

A cone C such that the polyhedron P is the intersection of C with a hyperplane at height one, together with the embedding of the ambient lattice of P into the ambient lattice of C.

`ConeInSublattice(C): TorCon -> TorCon, Map`

The cone of maximal dimension given by the intersection of the cone C with its linear span. Also gives the embedding of the sublattice in the ambient toric lattice.

`ConeQuotientByLinearSubspace(C): TorCon -> TorCon, Map, Map`

The strictly convex cone given by the quotient of the cone C by its maximal linear subspace. Also gives the quotient map.

`SimplicialSubcone(C): TorCon -> TorCon`

A simplicial cone of dimension equal to the dimension of C, contained in C.

`LatticeBasisInCone(C): TorCon -> [TorLatElt]`

Given a cone C of maximum dimension in the lattice L, returns a basis for L using elements which lie in C.

156.2.3 Polyhedra

Polytopes and polyhedra in lattices are of type `TorPol`.

`Polyhedron(C, H, h): TorCon, TorLatElt, FldRatElt -> TorPol`

`Polyhedron(C, H, h): TorCon, TorLatElt, RngIntElt -> TorPol`

The polyhedron constructed as the slice of the cone C by the hyperplane determined by the primitive dual vector H at height given by the integer or rational number h.

`Polyhedron(C): TorCon -> TorPol`

```
level: RngIntElt                    Default: 1
```

The polyhedron arising as the intersection of the cone C with the hyperplane at height one (can be changed via 'level').

`HalfspaceToPolyhedron(v, h): TorLatElt, FldRatElt -> TorPol`

HalfspaceToPolyhedron(Q, h): [FldRatElt], FldRatElt -> TorPol

The halfspace $\{u|v \cdot u \geq h\}$ as a polyhedron, where v is a point of a toric lattice (or a sequence of integral or rational numbers that are its coefficients) and $h \in \mathbb{Q}$.

HyperplaneToPolyhedron(v, h): TorLatElt, FldRatElt -> TorPol

HyperplaneToPolyhedron(Q, h): [FldRatElt], FldRatElt -> TorPol

The hyperplane $\{u|v \cdot u = h\}$ as a polyhedron, where v is a point of a toric lattice (or a sequence of integral or rational numbers that are its coefficients) and $h \in \mathbb{Q}$.

Polyhedron(C, f, v): TorCon, Map, TorLatElt -> TorPol

The polyhedron arising as the preimage of the cone C under the affine map $f + v$.

EmptyPolyhedron(L): TorLat -> TorPol

The empty polyhedron in the toric lattice L.

ConeToPolyhedron(C): TorCon -> TorPol

The cone C as a polyhedron.

PolyhedronInSublattice(P): TorPol -> TorPol, Map, TorLatElt

The polyhedron of maximal dimension given by the intersection of the toric lattice containing the polyhedron P with an affine sublattice of dimension equal to the dimension of P. Also gives the affine embedding as a lattice embedding and translation.

FixedSubspaceToPolyhedron(G): GrpMat -> TorPol

FixedSubspaceToPolyhedron(L, G): TorLat, GrpMat -> TorPol

The subspace (realised as a polyhedron) fixed by the action of the matrix group G on the toric lattice L. G should be a subgroup of either $GL(n, \mathbf{Z})$ or of $SL(n, \mathbf{Z})$, where n is the dimension of L.

Example: Polyhedra as Slices of a Quadrant (ex-d6dc93)

We construct some polyhedra by taking slices of the positive quadrant in 3-space.

```
> C := PositiveQuadrant(3);
> C;
3-dimensional simplicial cone C with 3 minimal generators:
    (1, 0, 0),
    (0, 1, 0),
    (0, 0, 1)
```

We slice using dual vectors, so we recover the dual lattice M to the ambient lattice of C. Some slices are compact, some are not.

```
> M := Dual(Ambient(C));
> P := Polyhedron(C, M ! [1,2,3], 1);
> IsPolytope(P);
true
> Q := Polyhedron(C, M ! [1,-2,3], 1 );
> IsPolytope(Q);
false
> Q;
2-dimensional polyhedron Q with 1-dimensional finite part with 2 vertices:
    ( 1,   -1),
    ( 0, -1/3)
and 2-dimensional infinite part given by a cone with 2 minimal generators:
    ( 2,   -1),
    ( 0,    1)
```

We can change the level, or height, at which the slice is taken.

```
> R := Polyhedron(C, M ! [1,2,3], -5);
> R;
Empty polyhedron
```

156.2.4 Arithmetic Operations on Polyhedra

`C eq D: TorCon, TorCon -> BoolElt`

Return `true` if and only if the cones C and D are equal, that is, they lie in the same toric lattice and have identical supports.

`P eq Q: TorPol, TorPol -> BoolElt`

Return `true` if and only if the polyhedra P and Q are equal, that is, they lie in the same toric lattice and have identical supports.

`C meet D: TorCon, TorCon -> TorCon`

The intersection of the cones C and D; the cones must lie in the same toric lattice otherwise an error is reported.

`P meet Q: TorPol, TorPol -> TorPol`

The intersection of the polyhedra P and Q; the polyhedra must lie in the same toric lattice otherwise an error is reported.

`P subset Q: TorPol, TorPol -> BoolElt`

Return `true` if and only if the support of the polyhedron P is contained in that of the polyhedron Q; the polyhedra must lie in the same toric lattice otherwise an error is reported.

`C + D: TorCon, TorCon -> TorCon`

`P + Q: TorPol, TorPol -> TorPol`

The Minkowski sum of the cones C and D or of the polyhedra P and Q.

`P + C: TorPol, TorCon -> TorPol`

`C + P: TorCon, TorPol -> TorPol`

The polyhedron constructed as the Minkowski sum of the polytope P and the cone C.

`P * Q: TorPol, TorPol -> TorPol`

The convex hull of the Cartesian product of the polyhedra P and Q.

`k * P: FldRatElt, TorPol -> TorPol`

The dilation of the polyhedron P by the rational number k.

`- P: TorPol -> TorPol`

The negation of the polyhedron P.

156.3 Basic Combinatorics of Polytopes and Polyhedra

Cones, polytopes, and polyhedra have similar combinatorial features: they are composed of faces of various dimensions that meet in faces of lower dimensions. Some functions apply to all of these geometrical objects, some do not.

A face of a cone C is the intersection of C with an affine hyperplane whose defining equation is non-negative on C. A facet of a cone C is a codimension 1 face of C. Definitions for polytopes and polyhedra are similar.

156.3.1 Vertices and Inequalities

`Vertices(P): TorPol -> SeqEnum[TorLatElt]`

A sequence containing the vertices of the polyhedron P.

`NumberOfVertices(P): TorPol -> RngIntElt`

The number of vertices of the polyhedron P.

`Rays(C): TorCon -> SeqEnum`

The sequence of generators of the rays of the cone C (returned as primitive lattice points). If C is strictly convex, this is the same as the minimal $\mathbf{R}$-generators.

`Ray(C, i): TorCon, RngIntElt -> TorLatElt`

The ith ray of the of the cone C (in the order returned by `Rays(C)`).

`LinearSpanEquations(C): TorCon -> SeqEnum`

`LinearSpanEquations(Q): [TorLatElt] -> SeqEnum`

A sequence of equations defining the minimal linear subspace containing the cone C (or the sequence of toric lattice points Q).

`LinearSpanGenerators(C): TorCon -> SeqEnum`

`LinearSpanGenerators(Q): [TorLatElt] -> SeqEnum`

A sequence of generators of the minimal linear subspace containing the cone C (or the sequence of toric lattice points Q).

`LinearSubspaceGenerators(C): TorCon -> SeqEnum`

A basis of the maximal linear subspace contained in the cone C.

`Inequalities(C): TorCon -> SeqEnum`

`Inequalities(P): TorPol -> SeqEnum, RngIntElt`

`MinimalInequalities(C): TorCon -> SeqEnum`

If the cone C or polyhedron P lies in a toric lattice L, then return a finite sequence of vectors in the dual lattice $\check{L}$ which define supporting hyperplanes of C or P. The list is forced to be minimal for `MinimalInequalities`, but might not be otherwise. If the argument is a polyhedron, then an integer k is returned as a second return value, so that the first k inequalities correspond to the facets of P while the remaining cut out the subspace containing P.

`MatrixOfInequalities(R, C): Rng, TorCon -> ModMatRngElt`

`MatrixOfInequalities(C): TorCon -> ModMatRngElt`

The inequalities of C arranged in a matrix, with each row representing an inequality.

Example: Inequalities Defining a Polytope (ex-b4cdf2)

Build a polytope and recover the inequalities that define it: the second return value, 3, says that all three inequalities define facets of P—we see an example below where this is not the case.

```
> P := Polytope([[1,0],[0,1],[-1,-1]]);
> Inequalities(P);
[
    <(2, -1), -1>,
    <(-1, -1), -1>,
    <(-1, 2), -1>
]
3
```

For each inequality use `HalfspaceToPolyhedron(m,h)` to define the corresponding halfspace, and then intersect them all to recover P—and finally recover the vertices we started with.

```
> PP := &meet  [HalfspaceToPolyhedron(H[1],H[2]) : H in Inequalities(P)];
> PP eq P;
true
> PP;
2-dimensional polytope PP with 3 vertices:
    ( 0,  1),
```

(continues on next page)

(continued from previous page)

```
    (-1, -1),
    ( 1,  0)
```

It can happen that a polytope does not span the ambient toric lattice in which it lies and that some of the inequalities are used to cut out the affine subspace that it does span.

```
> Q := Polytope([[1,0,2],[0,1,2],[-1,0,2],[0,-1,2]]);
> Inequalities(Q);
[
    <(1, -1, 0), -1>,
    <(-1, -1, 0), -1>,
    <(-1, 1, 0), -1>,
    <(1, 1, 0), -1>,
    <(0, 0, 1), 2>,
    <(0, 0, -1), -2>
]
4
```

In this case the final two inequalities are opposites of one another and cut out the affine hyperplane $z = 2$. The second return value 4 indicates that the first 4 inequalities are cutting out facets of the polytope while the remaining inequalities are cutting out the affine hyperplane.

`NormalCone(P, F): TorPol, TorPol -> TorCon`

The (outer) normal cone of the face F of the polyhedron P.

`NormalEdgeCones(P): TorPol -> [TorCon]`

The (outer) normal cones of the edges of the polyhedron P. The cones are presented in the same order as the edges of P; i.e. the i-th cone in the resulting sequence is the normal cone to the i-th edge returned by `Edges(P)` or `EdgeIndices(P)`.

`InnerNormal(C): TorCon -> TorLatElt`

`OuterNormal(C): TorCon -> TorLatElt`

For a Gorenstein cone C, i.e. a cone such that the primitive generators ρ_i of the rays of C lie in a common hyperplane, gives an element u in the dual lattice such that $u(\rho_i) = -1$ (inner normal) or $u(\rho_i) = 1$ (outer normal).

156.3.2 Facets and Faces

`fVector(C): TorCon -> SeqEnum[RngIntElt]`

`fVector(P): TorPol -> SeqEnum[RngIntElt]`

The f-vector of the polyhedron P or cone C.

`hVector(C): TorCon -> SeqEnum[RngIntElt]`

`hVector(P): TorPol -> SeqEnum[RngIntElt]`

The h-vector of the polyhedron P or cone C.

`Facets(C): TorCon -> SeqEnum`

`Facets(P): TorPol -> SeqEnum`

A sequence containing all facets of the toric cone C or polyhedron P.

`FacetIndices(P): TorPol -> SeqEnum`

A sequence of sets describing the facets of the polytope P. The jth set gives the indices of the vertices of P which define the jth facet of P.

`NumberOfFacets(P): TorPol -> RngIntElt`

The number of facets of the polyhedron P.

`Faces(C): TorCon -> SeqEnum`

`Faces(P): TorPol -> SeqEnum`

`Faces(C, i): TorCon, RngIntElt -> SeqEnum`

`Faces(P, i): TorPol, RngIntElt -> SeqEnum`

A sequence containing all face cones of the toric cone C or polyhedron P, or only those of dimension i if an integer i is also specified.

`FaceIndices(P, i): TorPol, RngIntElt -> SeqEnum`

A sequence of sets describing the i-dimensional faces of the polyhedron P. The jth set gives the indices of the vertices of P which define the jth i-dimensional face.

`NumberOfFaces(P, i): TorPol, RngIntElt -> RngIntElt`

The number of i-dimensional faces of the polyhedron P.

`Edges(P): TorPol -> SeqEnum`

A sequence containing all the edges of the polyhedron P.

`EdgeIndices(P): TorPol -> SeqEnum`

A sequence of sets describing the edges of the polyhedron P. The jth set gives the indices of the vertices of P which define the jth edge of P.

`NumberOfEdges(P): TorPol -> RngIntElt`

The number of edges of the polyhedron P.

`Graph(P): TorPol -> GrphUnd`

The graph of the face lattice of the polyhedron P. The vertices of the graph are labeled by the dimension of the corresponding face.

`FaceSupportedBy(C, H): TorCon, TorLatElt -> TorCon`

The face of the toric cone C supported by the toric lattice element H in the dual lattice to the one containing C (so H is a linear form on the ambient lattice of C).

`IsSupportingHyperplane(v, h, P): TorLatElt, FldRatElt, TorPol -> BoolElt, RngIntElt`

Return `true` if and only if the hyperplane defined by $v \cdot u = h$ is a supporting hyperplane of the polyhedron P, where v is a lattice point of the dual ambient lattice of P and h is a rational number. If so, also gives the sign τ such the hyperplane is a support of P (i.e. τ in $\{-1, 0, +1\}$ such that $\text{Sign}(v \cdot u - h)$ is either 0 or τ for all u in P). If P is contained within the hyperplane, then τ will be 0.

`SupportingCone(P, v): TorPol, TorLatElt -> TorCon`

The cone C such that $C + v$ is a supporting cone of the polyhedron P, where v is a vertex of P.

`IsFace(C, F): TorCon, TorCon -> BoolElt`

`IsFace(P, F): TorPol, TorPol -> BoolElt`

Return `true` if and only if F is a face of the cone C or polyhedron P.

156.4 The Combinatorics of Polytopes

156.4.1 Points in Polytopes and Polyhedra

Given a polygon, one might want to know how many integral points it contains, and one might want to list them. These are computed by different algorithms (although one can of course list the points and then count them). For point counting we use the methods of Barvinok and Pommersheim ([BP99]), as described in [DLHTY04]. More generally, one might want to know the number of points in integral dilations of a polytope: these numbers are the coefficients of a generating function, the Ehrhart series; this is discussed in a later section.

`v in C: TorLatElt, TorCon -> BoolElt`

`v in P: TorLatElt, TorPol -> BoolElt`

Returns `true` if and only if the point v is contained in the cone C or polyhedron P.

`IsInInterior(v, C): TorLatElt, TorCon -> BoolElt`

`IsInInterior(v, P): TorLatElt, TorPol -> BoolElt`

Returns `true` if and only if the point v lies in the strict interior of the cone C or polyhedron P.

`IsOnBoundary(v, C): TorLatElt, TorCon -> BoolElt`

`IsOnBoundary(v, P): TorLatElt, TorPol -> BoolElt`

Returns `true` if and only if the point v lies on the (relative) boundary of the cone C or polyhedron P.

`HasIntegralPoint(P): TorPol -> BoolElt`

Returns `true` if and only if the polyhedron P contains an integral lattice point.

`Points(P): TorPol -> SeqEnum[TorLatElt]`

`InteriorPoints(P): TorPol -> SeqEnum[TorLatElt]`

`BoundaryPoints(P): TorPol -> SeqEnum[TorLatElt]`

The integral toric lattice points, the strictly interior lattice points, or the boundary lattice points of the polytope P.

`NumberOfPoints(P): TorPol -> RngIntElt`

`NumberOfInteriorPoints(P): TorPol -> RngIntElt`

`NumberOfBoundaryPoints(P): TorPol -> RngIntElt`

The number of integral toric lattice points, the number of strictly interior lattice points, or the number of boundary lattice points of the polytope P.

`Volume(P): TorPol -> FldRatElt`

`VolumeOfBoundary(P): TorPol -> FldRatElt`

The normalised volume and normalised boundary volume of the polytope P.

156.4.2 Ehrhart Theory of Polytopes

`EhrhartSeries(P): TorPol -> FldFunRatUElt`

The rational generating function $\mathrm{Ehr}_P(t)$ of the Ehrhart series for the polytope P. The coefficient c_m of t^m in the power-series expansion of $\mathrm{Ehr}_P(t)$ is equal to the number of lattice points in mP.

`EhrhartDeltaVector(P): TorPol -> SeqEnum`

The Ehrhart δ-vector (also known as the h^*-vector) for the polytope P. This is a sequence of integers defining the numerator of the Ehrhart generating function $\mathrm{Ehr}_P(t)$ of P. More precisely, if $n = \dim(P)$ and $k \in \mathbb{Z}_{>0}$ is the smallest dilation of P such that kP is a lattice polytope, then $\mathrm{Ehr}_P(t) = (\delta_0 + \delta_1 t + \ldots + \delta_{k(n+1)-1} t^{k(n+1)-1})/(1-t^k)^{n+1}$.

`EhrhartPolynomial(P): TorPol -> [RngUPolElt]`

A sequence representing the Ehrhart (quasi-)polynomial for the polytope P. That is, a sequence of polynomials $[p_0, \ldots, p_{r-1}]$ of length r, the quasi-period of the Ehrhart polynomial, so that the number of lattice points of the dilation mP is the value of $p_s(k)$, where $m = kr + s$ is the Euclidean division of m by r; in other words, s is the least residue of m modulo r. Note that since Magma indexes sequences from 1, we have that $p_i =$ `EhrhartPolynomial(P)[i+1]`.

`EhrhartCoefficients(P, l): TorPol, RngIntElt -> [RngIntElt]`

The first $l+1$ coefficients of the Ehrhart series for the polytope P (starting with $0P$ up to and including lP).

`EhrhartCoefficient(P, k): TorPol, RngIntElt -> RngIntElt`

The number of lattice points in the (non-negative, integral) dilation kP of the polytope P.

156.4.3 Isomorphism Testing and Normal Forms for Polytopes

It is often necessary to determine whether two convex polytopes P and Q, embedded in a lattice Λ, are isomorphic with respect to a lattice automorphism; that is, whether there exists an element of $GL_n(\mathbb{Z})$ sending P to Q. For example, in toric geometry lattice polytopes form one of the key constructions of projective toric varieties, and any classification must somehow address the issue of whether there exists an automorphism of the underlying lattice sending P to Q.

In general, any isomorphism problem can be solved in one of two ways: on a case-by-case basis by constructing an explicit isomorphism between the two objects, or by determining a normal form for each isomorphism class. Both approaches are supported.

The first approach – dynamically constructing a lattice-preserving isomorphism in $GL_n(\mathbb{Z})$ between the two polytopes – has the advantage that it works equally well for rational polytopes and for polytopes of non-zero codimension. The algorithm works by reducing the problem to a graph isomorphism question to which well-developed tools such as Brendan McKay's Nauty software [McKb, McK81] can then be applied.

The second approach – to compute a normal form $\mathrm{NF}(P)$ for each isomorphism class – works only for lattice polytopes of codimension zero. The normal form algorithm used by Magma is based on one developed by Kreuzer and Skarke for their Palp software [KS04]. Briefly, row and column permutations are applied to the vertex–facet pairing matrix of P, placing it in a form that is maximal with respect to a certain ordering. This in turn defines an order in which to list the vertices of P; the choice of basis is fixed by taking the Hermite normal form.

Slightly more generally, one may wish to consider polytopes up to change of basis and translation, i.e. up to affine equivalence. Once again Magma offers two solutions. Affine equivalence between two (possibly rational) polytopes can be determined on a case-by-case basis, or an affine normal form can be calculated. All the algorithms are described in detail in [GK13].

`IsIsomorphic(P, Q): TorPol, TorPol -> BoolElt, Map`

Returns `true` iff the polytopes P and Q are isomorphic, i.e. if there exists an element in $GL_n(\mathbb{Z})$ sending P to Q. If true, also gives an isomorphism from P to Q. The polytopes may be rational, and need not be of maximum dimension in the ambient lattice.

Example: Isomorphism Testing for Polytopes `(ex-66ee55)`

We begin by creating a random rational 2-dimensional polytope in a 3-dimensional lattice. We shall then apply a random change of basis to P to generate an isomorphic polytope Q.

```
> P:=RandomPolytope(3,3,4) / 2;
> P;
2-dimensional polytope P with 3 generators:
    ( 0,  -2,   1),
    ( 0, 1/2,  -2),
    (-1, 3/2,   0)
> Q:=P * RandomGLnZ(3,3,3);
> Q;
2-dimensional polytope Q with 3 generators:
    ( -2,  -3,   -1),
    (1/2, 5/2, -3/2),
    (5/2, 3/2,  5/2)
```

Now we recover the isomorphism between these two polytopes:

```
> bool,phi:=IsIsomorphic(P,Q);
> bool;
true
> Image(phi,P) eq Q;
true
```

The lattice isomorphism can easily be recovered as a matrix in $GL_3(\mathbb{Z})$:

```
> M:=DefiningMatrix(phi);
> M;
[-1  0 -1]
[ 1  1  1]
[ 0 -1  1]
> P * M eq Q;
true
```

IsEquivalent(P, Q): TorPol, TorPol -> BoolElt, Map, TorLatElt

Returns **true** iff the polytopes P and Q are equivalent, i.e. if there exists an element in $GL_n(\mathbb{Z})$ and lattice translation sending P to Q. If true, also gives the map as a lattice map φ and translation v such that $\varphi(P) + v = Q$.

NormalForm(P): TorPol -> SeqEnum, GrpPermElt

PALPNormalForm(P): TorPol -> SeqEnum, GrpPermElt

The Palp normal form of the maximum dimensional lattice polytope P. This is given as a sequence of lattice points equal to the (ordered) vertices of $\mathrm{NF}(P)$. Also gives the permutation of the vertices of P used to calculate this normal form.

Example: Normal Form of a Lattice Polytope (ex-3bf045)

We start by computing the normal form of a lattice polytope.

```
> P:=Polytope([[62,23],[-8,-3],[-27,-10]]);
> NF:=NormalForm(P);
> NF;
[
    (1, 0),
    (0, 1),
    (-2, -1)
]
```

People familiar with toric geometry will recognise this as the weighted projective space $\mathbb{P}(1,1,2)$; we can verify this in one line of code by comparing the normal forms:

```
> NormalForm(PolytopeOfWPS([1,1,2])) eq NF;
true
```

Now we exhibit one way to recover the change of basis that sends P to its normal form $\mathrm{NF}(P)$. We need to capture the permutation of the vertices, so we redo the calculation and this time we save the second return value:

```
> NF,perm:=NormalForm(P);
> perm;
(1, 3)
```

The change of basis in $GL_2(\mathbb{Z})$ is now trivial to compute:

```
> vertsP:=PermuteSequence(Vertices(P),perm);
> VP:=Matrix(vertsP);
> VQ:=Matrix(NF);
> M:=Solution(Transpose(VP),Transpose(VQ));
> M:=Transpose(M);
> M;
[ -3  10]
[  8 -27]
> P * M;
2-dimensional polytope with 3 vertices:
```

(continues on next page)

(continued from previous page)

```
    (-2, -1),
    ( 0,  1),
    ( 1,  0)
```

AffineNormalForm(P): TorPol -> SeqEnum, GrpPermElt

The affine normal form of the maximum dimensional lattice polytope P. This is given as a sequence of lattice points equal to the (ordered) vertices of AffNF(P). Also gives the permutation of the vertices of P used to calculate this affine normal form.

Example: Classifying Polygons by Affine Normal Form (ex-7e87a2)

We wish to classify all polygons in an $[0, N] \times [0, N]$ box but not contained in a smaller $[0, N-1] \times [0, N-1]$ box, up to change of basis and translation. To do this we make use of the affine normal form. We will perform the classification for $N \leq 4$.

```
> procedure classify_polygons_recurse(P,~class,allclass)
>     for v in Vertices(P) do
>         Q:=Polytope(Exclude(Points(P),v));
>         if Dimension(Q) eq 2 then
>             AffNF:=AffineNormalForm(Q);
>             if not AffNF in class and
>                     not &or[AffNF in subclass : subclass in allclass] then
>                 Include(~class,AffNF);
>                 $$(Q,~class,allclass);
>             end if;
>         end if;
>     end for;
> end procedure;
>
> function classify_polygons(N)
>     allclass:=[];
>     for i in [1..N] do
>         B:=Polytope([[0,0],[i,0],[0,i],[i,i]]);
>         class:={AffineNormalForm(B)};
>         classify_polygons_recurse(B,~class,allclass);
>         Append(~allclass,class);
>     end for;
>     return allclass;
> end function;
>
> classification:=classify_polygons(4);
> [#class : class in classification];
[ 2, 15, 131, 1369 ]
```

MaximalVertexFacetHeightMatrix(P): TorPol -> AlgMatElt

This is the maximal vertex–facet pairing matrix of P constructed as part of the normal form algorithm. For the details of its construction, see [GK13, KS04].

156.4.4 Automorphisms of a Polytope

AutomorphismGroup(P): TorPol -> GrpMat

The subgroup of $GL(n, \mathbf{Z})$ (acting on the ambient lattice) which leaves the polyhedron P unchanged.

Example: Automorphism Group of a Polytope (ex-49a1bb)

```
> P:=CrossPolytope(2);
> P;
```

(continues on next page)

(continued from previous page)

```
2-dimensional polytope P with 4 vertices:
    ( 1,  0),
    ( 0,  1),
    (-1,  0),
    ( 0, -1)
> AutomorphismGroup(P);
MatrixGroup(2, Integer Ring)
Generators:
    [0 1]
    [1 0]

    [ 0  1]
    [-1  0]
```

156.4.5 Operations on Polytopes

`Triangulation(P): TorPol -> SetEnum`

A sequence of polytopes that make a triangulation of the polytope P.

`TriangulationOfBoundary(P): TorPol -> SetEnum`

A sequence of polytopes that make a triangulation of the boundary of the polytope P.

156.5 Cones and Polyhedra

156.5.1 Generators of Cones

`BoxElements(C): TorCon -> SetEnum`

Given a simplicial cone C, returns the coefficients of the points in the fundamental domain generated by the rays of C, with respect to the rays.

Example: Box Elements of a Simplicial Cone (ex-6a5b6c)

Let $C \subset L$ be a simplicial cone and consider the sublattice $L' \subset L$ generated by the rays of C. We shall construct a representative set of points of L contained in the fundamental domain of L'.

```
> C:=Cone([[1,-1,-3],[1,1,0],[1,0,1]]);
> rays:=Rays(C);
> Index(rays);
5
> pts:=BoxElements(C);
> pts;
{
    [ 0, 0, 0 ],
    [ 3/5, 3/5, 4/5 ],
    [ 2/5, 2/5, 1/5 ],
    [ 1/5, 1/5, 3/5 ],
    [ 4/5, 4/5, 2/5 ]
}
> {&+[rays[i] * c[i] : i in [1..3]] : c in pts};
{
    (2, 0, -2),
    (2, 0, -1),
    (1, 0, -1),
    (0, 0, 0),
    (1, 0, 0)
}
```

Using these points we can construct the rational Ehrhart generating function counting the number of points as successive heights in C:

```
> d:=Dimension(C);
> heights:=[&+c : c in pts];
> delta:=[#[i : i in heights | i eq h] : h in [0..d - 1]];
> R<x>:=RationalFunctionField(Rationals());
> f:=&+[delta[i] * x^(i - 1) : i in [1..d]] / (1 - x)^d;
> f;
(-2*x^2 - 2*x - 1)/(x^3 - 3*x^2 + 3*x - 1)
> K<t>:=PowerSeriesRing(Rationals(),10);
> K ! f;
1 + 5*t + 14*t^2 + 28*t^3 + 47*t^4 + 71*t^5 + 100*t^6 + 134*t^7 + \\
    173*t^8 + 217*t^9 + O(t^10)
> w:=Dual(Ambient(C)) ! [1,0,0];
> [NumberOfPoints(Polyhedron(C,w,i)) : i in [0..9]];
[ 1, 5, 14, 28, 47, 71, 100, 134, 173, 217 ]
```

HilbertBasis(C): TorCon -> SeqEnum

ZGenerators(C): TorCon -> SeqEnum

```
level: RngIntElt                    Default: Infinity()
```

If C lies in a toric lattice L, then return a finite sequence of elements of L that generate C as a monoid. If specified, the parameter `level` restricts the search for generators to that given level; this assumes that C is graded.

RGenerators(C): TorCon -> SeqEnum

MinimalRGenerators(C): TorCon -> SeqEnum

If C lies in a toric lattice L, then return a finite sequence of elements of L that generate C over $\mathbb{Q}_+$. The list is forced to be minimal for `MinimalRGenerators`, but might not be otherwise.

Points(C, H, h): TorCon, TorLatElt, FldRatElt -> SetEnum

The set of integral lattice points in the cone C contained in the hyperplane given by the section determined by the dual lattice element H at height given by the rational number h. The intersection of H with C is required to be compact.

Example: Cone Generators in a Sublattice (ex-7976dc)

Here we find the generators of a cone in the sublattice spanned by its defining lattice points. First build some points in a toric lattice and make the cone they generate.

```
> L := ToricLattice(4);
> B := [ L | [1,2,3,-3], [-1,0,1,0], [1,2,1,-2], [2,0,0,-1], [0,0,2,-1] ];
> L1,f := Sublattice(B);
> Dimension(L1);
3
> C := Cone(B);
> Points(Polytope(B));
{
    (1, 2, 1, -2),
    (2, 0, 0, -1),
    (-1, 0, 1, 0),
    (1, 2, 3, -3),
    (1, 0, 1, -1),
    (0, 0, 2, -1),
    (0, 1, 1, -1),
    (1, 1, 2, -2)
}
```

(continues on next page)

(continued from previous page)

```
> ZGenerators(C);
[
    (-1, 0, 1, 0),
    (0, 1, 1, -1),
    (1, 2, 1, -2),
    (2, 0, 0, -1)
]
```

Using the embedding map f, pull the cone back to the lattice span of C.

```
> C1 := C @@ f;
```

In the smaller lattice, this cone is nonsingular.

```
> IsNonsingular(C1);
true
```

We can find out which points generate the cone in the smaller lattice and compare them with the generators of the original cone.

```
> B1 := ZGenerators(C1);
> B1;
[
    (-1, 0, 1),
    (1, 1, 0),
    (2, 0, -1)
]
> [ Index(B,Image(f,b)) : b in B1 ];
[ 2, 3, 4 ]
```

`QuotientGenerators(C): TorCon -> SetEnum`

Let $C \subset L$ be a simplicial cone of maximum dimension in the lattice L. Let $L' \subset L$ be the sublattice generated by the rays of C. This intrinsic returns a sequence of generators for the torsion of L/L'; the generators are expressed in terms of the basis given by the rays of C.

Example: Cone of a Cyclic Quotient Singularity `(ex-badd91)`

Consider the cyclic quotient singularity $1/3(1,1,2)$. We will begin by constructing the corresponding cone C.

```
> L:=ToricLattice(3);
> v:=L ! [1/3,1/3,2/3];
> N,emb:=AddVectorToLattice(v);
> C:=Image(emb,PositiveQuadrant(L));
> C;
Cone C with 3 generators:
    (1, 0, 0),
    (0, 1, 0),
    (1, 1, 3)
> IsIsolated(C);
true
> IsQFactorial(C);
true
> IsTerminal(C);
true
```

Now we reverse this operation: we take the cone C and recognise that it corresponds to $1/3(1,1,2)$.

```
> QuotientGenerators(C);
{
    [ 2/3, 2/3, 1/3 ]
}
```

Of course, the generators are defined only up to multiplication by units in μ_3.

156.5.2 Properties of Polyhedra

`CompactPart(P): TorPol -> TorPol`

The polytope defined by taking the convex hull of the vertices of the polyhedron P. This is the smallest polytope which can occur as a factor in an expression of P as the Minkowski sum of a polytope and a cone.

`IntegralPart(P): TorPol -> TorPol`

The polyhedron defined by taking the convex hull of the integral points contained in the polyhedron P.

`InfinitePart(P): TorPol -> TorCon`

The tail cone of the polyhedron P.

`IsEmpty(P): TorPol -> BoolElt`

Return `true` if and only if the polyhedron P is empty.

Example: Polar Polyhedron of a Polytope (ex-a585a6)

One can build a polytope as the convex hull of a finite sequence of points—these may be elements of a toric lattice or simply sequences of integer or rational coefficients.

```
> P := Polytope([[-4,2,1],[0,3,4],[3,1,-3],[3,0,0],[2,-1,1]]);
```

The polar polyhedron comprises dual vectors that evaluate to at least -1 on P. It is compact if and only if the origin lies in the interior of P.

```
> D := Polar(P);
> D;
3-dimensional polyhedron D with 3-dimensional finite part with 5 vertices:
    ( 1/2,      1,    -1),
    (-1/3,    1/2,   1/6),
    (-1/3,   1/21,  -2/7),
    (-1/3,  -3/13, -1/13),
    (1/67, -37/67, 11/67)
and 3-dimensional infinite part given by a cone with 3 minimal generators:
    (   1,      2,     0),
    (   2,      9,     5),
    (   7,      9,    10)
```

Since D is infinite, there is no intrinsic to list its integral points, but one can determine whether it contains integral points at all.

```
> HasIntegralPoint(D);
true
```

Computing the span of all the integral points of D is possible.

```
> IntegralPart(D);
3-dimensional polyhedron with 3-dimensional finite part with 6 vertices:
    (0,  0,  0),
    (0,  1,  0),
```

(continues on next page)

(continued from previous page)

```
    (0,  2,  1),
    (1,  1,  1),
    (1,  2, -1),
    (2,  2,  3)
and 3-dimensional infinite part given by a cone with 3 minimal generators:
    (1,  2,  0),
    (2,  9,  5),
    (7,  9, 10)
```

The compact and infinite parts of D can be recovered.

```
> cD := CompactPart(D);
> cD;
3-dimensional polytope cD with 5 vertices:
    ( 1/2,      1,    -1),
    (-1/3,    1/2,   1/6),
    (-1/3,   1/21,  -2/7),
    (-1/3,  -3/13, -1/13),
    (1/67, -37/67, 11/67)
```

Polytopes are special cases of polyhedra (those whose infinite tail cone is the zero cone, or equivalently those that are compact), and the distinction can be determined.

```
> IsPolytope(cD);
true
> IsPolytope(D);
false
```

Example: Polyhedron from a Cone `(ex-d24d32)`

```
> C:=Cone([[0,1,0],[0,1,1],[1,1,2],[1,1,4]]);
> P:=Polyhedron(C);
> P;
2-dimensional polyhedron P with 1-dimensional finite part with 2
vertices:
    (1, 2),
    (1, 4)
and 2-dimensional infinite part given by a cone with 2 minimal
generators:
    (1, 0),
    (1, 1)
```

```
> CC:=Cone([[1,0],[1,1]]);
> QQ:=Polytope([[1,2],[1,4]]);
> PP:=CC + QQ;
> PP;
2-dimensional polyhedron PP with 1-dimensional finite part with 2
vertices:
    (1, 2),
    (1, 4)
and 2-dimensional infinite part given by a cone with 2 minimal
generators:
    (1, 0),
    (1, 1)
```

But there's a potential catch.

```
> PP eq P;
false
> Ambient(PP);
2-dimensional toric lattice Z^2
> Ambient(P);
2-dimensional toric lattice ker <1, 0, 0>
> P:=ChangeAmbient(P,Ambient(PP));
> PP eq P;
true
```

The ambients are different simply because of the method of construction of the two polyhedra, but they can be forced into the same space.

IsMaximumDimensional(C): TorCon -> BoolElt

IsMaximumDimensional(P): TorPol -> BoolElt

Return **true** if and only if the cone C or polyhedron P has dimension equal to that of its ambient lattice.

IsStrictlyConvex(C): TorCon -> BoolElt

Return **true** if and only if the cone C is strictly convex; that is, if there exists a hyperplane H such that C is contained on one side of H and C meets H in single point 0.

IsLinearSpace(C): TorCon -> BoolElt

IsLinearSpace(P): TorPol -> BoolElt

Return **true** if and only if the cone C or the polyhedron P is a linear space.

IsSimplicial(C): TorCon -> BoolElt

IsSimplicial(P): TorPol -> BoolElt

Return **true** if and only if the cone C or the polyhedron P is simplicial.

IsSimplex(P): TorPol -> BoolElt

Return **true** if and only if the polyhedron P is a simplex.

IsSimple(P): TorPol -> BoolElt

Return **true** if and only if the polyhedron P is simple.

IsAffineLinear(P): TorPol -> BoolElt

Return **true** if and only if the polyhedron P is an affine linear space.

IsZero(C): TorCon -> BoolElt

Return **true** if and only if the cone C is supported at the origin of its ambient toric lattice.

ContainsZero(P): TorPol -> BoolElt

Return **true** if and only if the polyhedron P contains the origin strictly in its interior.

IsPointed(P): TorPol -> BoolElt

Returns **true** iff the polyhedron P is pointed, i.e. if it is empty of has a vertex.

IsFlag(P): TorPol -> BoolElt

Returns **true** iff the simplicial polytope P is a flag polytope.

IsPerfectlyCentered(P): TorPol -> BoolElt

Returns **true** iff the polyhedron P is a polytope containing the origin strictly in its interior such that for any non-empty face F of P, the intersection relint(F) with the (outer) normal cone is non-empty.

IsIntegrallyClosed(P): TorPol -> BoolElt

Returns **true** iff the polytope P is integrally closed; i.e. if every lattice point in kP can be written as the sum of k lattice points in P, for all $k \in \mathbb{Z}_{\geq 0}$.

156.5.3 Attributes of Polyhedra

`IsPolytope(P): TorPol -> BoolElt`

Returns `true` if and only if the polyhedron P is a polytope; that is, if and only if P is compact.

`Dimension(C): TorCon -> RngIntElt`

`Dimension(P): TorPol -> RngIntElt`

The dimension of the toric cone C or polyhedron P.

`Degree(P): TorPol -> RngIntElt`

`Codegree(P): TorPol -> RngIntElt`

The degree and codegree of the integral polytope P. The codegree is equal to the smallest dilation $k \in \mathbb{Z}_{>0}$ such that kP contains an interior lattice point.

`Index(C): TorCon -> RngIntElt`

The index of the sublattice generated by the minimal $\mathbf{R}$-generators of the cone C in its linear span.

`Width(P): TorPol -> FldRatElt, SetEnum`

`Width(P, u): TorPol, TorLatElt -> FldRatElt`

The width of a polytope P in a lattice L with respect to an element $u \in L^\vee$ in the dual lattice is the different $\max\{u(v) : v \in P\} - \min\{u(v) : v \in P\}$. More generally, the width of P is defined to be the smallest possible width ranging over all non-zero u.

Example: Lattice Width of a Polytope `(ex-d2ed42)`

We calculate the (minimum) width of a polytope P, along with all forms realising this width.

```
> P:=Polytope([[1,-2,3],[2,-4,-1],[5,-4,1],[-3,-3,-3]]);
> width,us:=Width(P);
> width;
1
> us;
{
    (1, 3, -1),
    (-1, -3, 1)
}
> u:=Representative(us);
> [u * v : v in Vertices(P)];
[ -8, -9, -8, -9 ]
```

In particular, the fact that P is of width one implies that P contains no (strict) interior lattice points:

```
> NumberOfInteriorPoints(P);
0
```

We calculate a change of basis to make the fact that P is of width one immediately apparent.

```
> phi:=ChangeBasis(u);
> DefiningMatrix(phi);
[ 1  0  1]
[ 0  1  3]
[ 0  0 -1]
> Image(phi,P);
3-dimensional polytope with 4 vertices:
    ( 1, -2, -8),
    ( 2, -4, -9),
    ( 5, -4, -8),
    (-3, -3, -9)
```

`IsPyramid(P): TorPol -> BoolElt, TorLatElt, TorPol, Map, TorLatElt`

Returns `true` iff the polytope P is a pyramid, i.e. if there exists a facet F and an integral vertex $u \in P$ such that $P = \text{conv}(F \cup \{u\})$, where u is at height one relative to F. If P is a pyramid, then a choice of apex u, base facet F, an element φ in $GL_n(\mathbb{Z})$, and a lattice translation v are also returned. These are chosen such that $\varphi(P) + v$ sends u to $(1, 0, \ldots, 0)$ and embeds F in the $(n-1)$-dimensional sublattice generated by $(0, 1, \ldots, 0), \ldots, (0, 0, \ldots, 1)$.

`Pyramid(P): TorPol -> TorPol, Map, Map, Map, Map`

The pyramid given by embedding the polyhedron P in the lattice $\mathbb{Z} \times L$ (where L is the ambient lattice containing P) and adding the point $(1, 0, \ldots, 0)$ to the convex hull. Also gives the two embedding maps and the two projection maps of the underlying lattices.

Example: Recognising a Pyramid `(ex-0b2d83)`

We begin with a 3-dimensional polytope P living in a 4-dimensional lattice:

```
> P:=Polytope([
>     [ -3, -2, -1, 3 ],
>     [ 2, 2, 0, -3 ],
>     [ 3, 0, -2, 0 ],
>     [ 4, 1, 61, -121 ]
> ]);
> Dimension(P);
3
```

Next we see that P is a pyramid $P = \text{conv}(F \cup \{u\})$. We recover the 2-dimensional polytope F and the corresponding apex u.

```
> bool,u,F,phi,v:=IsPyramid(P);
> bool;
true
> F;
2-dimensional polytope F with 3 vertices:
    ( 2,   2,   0,   -3),
    ( 3,   0,  -2,    0),
    ( 4,   1,  61, -121)
> u;
(-3, -2, -1, 3)
```

We make this pyramid construction explicit by calculating $\varphi(P) + v$:

```
> Image(phi,P) + v;
3-dimensional polytope with 4 vertices:
    (1,  0,  0,  0),
    (0,  0,  0,  0),
    (0,  1, -2, -1),
    (0,  2, -1, -1)
```

Let us consider F as a simplex in a 2-dimensional lattice; we see that it is equivalent to the standard empty triangle S.

```
> F:=PolyhedronInSublattice(F);
> F;
2-dimensional polytope F with 3 vertices:
    (2,  0),
    (3, -1),
    (4, -1)
> NumberOfInteriorPoints(F);
0
```

(continues on next page)

(continued from previous page)

```
> S:=Polytope([
>     [ 0, 0 ],
>     [ 1, 0 ],
>     [ 0, 1 ]
> ]);
> bool:=IsEquivalent(F,S);
> bool;
true
```

Since P is equivalent to a pyramid over S, we have that the Ehrhart δ-vector (also known as the h^*-vector) of P must be equal to $(1, 0, 0, 0)$. We verify this fact:

```
> EhrhartDeltaVector(P);
[ 1, 0, 0, 0 ]
```

VertexEdgeIncidenceMatrix(P): TorPol -> ModMatRngElt

The vertex–edge incidence matrix of the polyhedron P, where the i, j-th entry is 1 if and only if the i-th vertex is contained in the j-th edge.

VertexFacetIncidenceMatrix(P): TorPol -> ModMatRngElt

The vertex–facet incidence matrix of the polyhedron P, where the i, j-th entry is 1 if and only if the i-th vertex is contained in the j-th facet.

VertexFacetHeightMatrix(P): TorPol -> AlgMatElt

The vertex–facet height matrix of the polyhedron P, where the i, j-th entry is equal to the lattice height of the i-th vertex over the j-th facet.

EdgeFacetIncidenceMatrix(P): TorPol -> ModMatRngElt

The edge–facet incidence matrix of the polyhedron P, where the i, j-th entry is 1 if and only if the i-th edge is contained in the j-th facet.

156.5.4 Combinatorics of Polyhedral Complexes

Ambient(C): TorCon -> TorLat

Ambient(P): TorPol -> TorLat

The ambient toric lattice of the toric cone C or polyhedron P.

ChangeAmbient(C, L): TorCon, TorLat -> TorCon

ChangeAmbient(P, L): TorPol, TorLat -> TorPol

Make a cone or polyhedron identical to the toric cone C or polyhedron P but lying in the toric lattice L; the identification of the existing ambient toric lattice with L is simply by identification of the two standard bases and it requires the dimensions of the two spaces to be equal.

156.6 Toric Lattices

One often begins to study toric geometry by considering a 'lattice' $L = \mathbf{Z}^n$ and discussing polygons or cones in its overlying rational (or real) vector space $L_{\mathbf{Q}} = L \otimes \mathbf{Q}$ whose vertices lie on L; these are examples of so-called 'lattice polytopes'. Anybody who has given a seminar on toric geometry will know the constant frustration of distinguishing between L and $L_{\mathbf{Q}}$ (or worse, a bit later in the story, the duals of these). In Magma, we bind these two spaces together in a single object, a *toric lattice*. These spaces lie underneath all the combinatorics. It is not often necessary to be explicit about them, since the package handles them invisibly in the background, but they are useful for the usual parent and data typing purposes: when one creates an empty sequence intended to house elements of a toric lattice, it should be properly typed from the outset, for instance. Toric lattices created as duals record that relationship.

Toric lattices are of type `TorLat` and their elements are of type `TorLatElt`.

156.6.1 Toric Lattices

A toric lattice is a finite-dimensional rational vector space with a distinguished free $\mathbb{Z}$-module L that spans it: it is the pair $L \otimes \mathbb{Q} \supset L$ where $L \cong \mathbb{Z}^n$. We usually refer to the toric lattice as L, and although the feeling of using L is that it is simply the given $\mathbb{Z}$-module, the covering vector space allows fluent working with non-integral points of L.

`ToricLattice(n): RngIntElt -> TorLat`

Create an n-dimensional toric lattice $\mathbb{Q}^n \supset \mathbb{Z}^n$. The integer n must be non-negative.

`ScalarLattice() -> TorLat`

The unique one-dimensional toric lattice of scalars $\mathbb{Q} \supset \mathbb{Z}$.

Example: Summing an Empty Sequence of Points (ex-baeb7b)

A properly typed sequence of points of a toric lattice can be summed with the usual convention.

```
> L := ToricLattice(3);
> nopoints := [ L | ];
> &+ nopoints;
(0, 0, 0)
```

A toric lattice is really a vector space marked with a finitely-generated **Z**-module that spans it. In particular, its points may well have rational coefficients, although those with integral coefficients are distinguished.

```
> somepoints := [ L | [1/2,2/3,3/4], [1,2,3] ];
> [ IsIntegral(v) : v in somepoints ];
[ false, true ]
```

`Dual(L): TorLat -> TorLat`

The dual lattice of the toric lattice L.

Example: Dual Toric Lattice (ex-81a499)

The dual of a toric lattice is also a toric lattice: integrality of dual points is clear.

```
> L := ToricLattice(2);
> L;
2-dimensional toric lattice L = Z^2
> M := Dual(L);
> M;
2-dimensional toric lattice M = (Z^2)^*
```

Magma's default printing for toric lattices tries to help keep track of which lattice is which—this becomes more useful when working with toric varieties. The relationship between L and M is preserved, and double-dual returns L.

```
> M eq L;
false
> L eq Dual(M);
true
```

`L + M: TorLat, TorLat -> TorLat, TorLatMap, TorLatMap, TorLatMap, TorLatMap`

`DirectSum(L, M): TorLat, TorLat -> TorLat, TorLatMap, TorLatMap, TorLatMap, TorLatMap`

`DirectSum(Q): SeqEnum -> TorLat, TorLatMap, TorLatMap, TorLatMap, TorLatMap`

The direct sum of the two toric lattices L and M. The following natural maps are also returned: the embedding of L into the sum, the embedding of M, the projection of the sum onto L, and the projection onto M.

The argument can also be a sequence Q of toric lattices. In this case, there are three return values: the direct sum of all lattices in Q, a sequence of inclusion maps of lattices in Q into the sum, and a sequence of projection maps from the sum onto each of the elements of Q in turn.

`L ^ n: TorLat, RngIntElt -> TorLat, SeqEnum, SeqEnum`

The direct sum of the toric lattice L with itself n times. Also return are a sequence of injections of L as factors into the sum and a sequence of projections of the sum onto its factors. This is identical to `DirectSum([L,L,...,L])` where the sequence has length n.

`Dimension(L): TorLat -> RngIntElt`

The dimension of the toric lattice L.

156.6.2 Points of Toric Lattices

Points of L are interpreted to mean points of $L \otimes \mathbb{Q}$. If the coefficients of a point are in fact integral, then this is recognised: so it is possible to tell whether a point is in fact a point of L, not just the rational span.

The usual rational vector space arithmetic operates on these lattice points.

`L ! [a,b,...]: TorLat, [RngIntElt] -> TorLatElt`

`LatticeVector(L, Q): TorLat, [RngIntElt] -> TorLatElt`

`LatticeVector(Q): [RngIntElt] -> TorLatElt`

The point $(a, b, \dots)$ as an element of the toric lattice L, where $Q = [a, b, \dots]$ is a sequence of rational numbers or integers. (If the toric lattice L is omitted, then it will be created.)

`L . i: TorLat, RngIntElt -> TorLatElt`

`Basis(L, i): TorLat, RngIntElt -> TorLatElt`

The ith standard basis element of the toric lattice L.

`Basis(L): TorLat -> TorLatElt`

The standard basis element of the toric lattice L as a sequence of points of L.

`Form(L, Q): TorLat, [RngIntElt] -> TorLatElt`

The point $(a, b, \dots)$ as an element of the dual of the toric lattice L, where $Q = [a, b, \dots]$ is a sequence of rational numbers or integers.

`Zero(L): TorLat -> TorLatElt`

The zero vector in the toric lattice L.

`P + Q: TorLatElt, TorLatElt -> TorLatElt`

`P - Q: TorLatElt, TorLatElt -> TorLatElt`

`n * P: FldRatElt, TorLatElt -> TorLatElt`

`P / n: TorLatElt, FldRatElt -> TorLatElt`

The sum and difference of toric lattice points P and Q (or points of the dual), the rational multiple nP and quotient P/n, where $n \in \mathbb{Q}$.

`P eq Q: TorLatElt, TorLatElt -> BoolElt`

Return `true` if and only if the toric lattice points P and Q are the same point of the same lattice.

`AreProportional(P, Q): TorLatElt, TorLatElt -> BoolElt, FldRatElt`

Return `true` if and only if the toric lattice points P and Q are rational multiples of one another; the factor Q/P is returned in that case.

`P / Q: TorLatElt, TorLatElt -> FldRatElt`

The rational factor P/Q in the case that the toric lattice points P and Q are rational multiples of one another.

Example: Arithmetic with Toric Lattice Points (ex-817cf9)

This simple example builds some points in a toric lattice and performs arithmetic on them.

```
> L := ToricLattice(3);
> a := L ! [1,2,3];
> a;
(1, 2, 3)
> L eq Parent(a);
true
> b := L ! [1/2,1,3/2];
> a + b;
(3/2, 3, 9/2)
> a eq b;
false
> a eq 2*b;
true
> b/a;
1/2
```

`v in L: TorLatElt, TorLat -> BoolElt`

Return `true` if and only if the toric lattice point v lies in the toric lattice L.

`Matrix(R, S): Rng, [TorLatElt] -> ModMatRngElt`

`Matrix(S): [TorLatElt] -> ModMatRngElt`

The matrix whose rows are given by the lattice points in the sequence S.

`Vector(v): TorLatElt -> ModTupFldElt`

The toric lattice point v realised as a vector.

`IsZero(v): TorLatElt -> BoolElt`

Return `true` if and only if the toric lattice point v is the zero vector.

`IsIntegral(v): TorLatElt -> BoolElt`

Return `true` if and only if the coefficients of the toric lattice point v are integral.

`IsPrimitive(v): TorLatElt -> BoolElt`

Return `true` if and only if the toric lattice point v is a primitive integral vector; that is, v is not divisible as an integral vector in its ambient toric lattice.

`PrimitiveLatticeVector(v): TorLatElt -> TorLatElt`

The first toric lattice point on the ray spanned by v.

Example: Primitive Points in a Sublattice (ex-db5829)

A point of a sublattice may be primitive in the sublattice even though it is not primitive in the bigger lattice: treating sublattices as the images of embeddings makes this point transparent. First build a sublattice K of L together with the embedding map.

```
> L := ToricLattice(2);
> K,emb := Sublattice([L | [2,0],[0,2]]);
```

Construct a point in L that is clearly not primitive.

```
> vL := L ! [2,2];
> IsPrimitive(vL);
false
```

Pulling this point back to K shows that it is obviously primitive in K.

```
> vK := vL @@ emb;
> vK;
(1, 1)
> IsPrimitive(vK);
true
```

But notice that coercion of the point of L into K is not the same thing: it simply works with the coefficients of the point and gives an answer that is not what is wanted in this case.

```
> K ! vL;
(2, 2)
> IsPrimitive(K ! vL);
false
```

156.6.3 Operations on Toric Lattices

`L eq K: TorLat, TorLat -> BoolElt`

Return `true` if and only if the toric lattices L and K are the same object in Magma (and not merely isomorphic).

`Sublattice(Q): [TorLatElt] -> TorLat, TorLatMap`

A toric lattice L_1 isomorphic to the sublattice of a toric lattice L generated by the sequence Q of elements of L together with an embedding map of L_1 in L. The Magma subobject constructor `sub< L | Q >` can also be used, although the sequence Q must be a sequence of elements of L and cannot be interpreted more broadly.

`ToricLattice(Q): [[RngIntElt]] -> TorLat, TorLatMap`

The toric sublattice of $\mathbb{Q}^n \supset \mathbb{Z}^n$ (regarded as a toric lattice) generated by the sequences of integers of length n of which the sequence Q comprises.

`Quotient(C): TorCon -> TorLat, TorLatMap`

`Quotient(Q): [TorLatElt] -> TorLat, TorLatMap`

`Quotient(v): TorLatElt -> TorLat, TorLatMap`

A toric lattice isomorphic to the toric lattice that is the quotient of a toric lattice L by the linear span of the cone C, or the elements $v \in L$ that comprise the sequence Q or the single primitive vector $v \in L$. The projection map of L to the quotient is also returned.

`AddVectorToLattice(v): TorLatElt -> TorLat, TorLatMap`

`AddVectorToLattice(Q): [TorLatElt] -> TorLat, TorLatMap`

A toric lattice L_1 isomorphic to the toric lattice generated by a toric lattice L together with the vector $v \in L$ (or by a sequence of vectors of L). The inclusion map of L in L_1 is also returned.

`AreGenerators(S): SetEnum -> BoolElt`

Returns `true` iff the set of toric lattice points $S \subset L$ generate the lattice L.

`IsSublattice(L): TorLat -> BoolElt`

Return `true` if and only if the toric lattice L was constructed as a sublattice of another toric lattice.

`IsSuperlattice(L): TorLat -> BoolElt`

Return `true` if and only if the toric lattice L was constructed as a superlattice of another toric lattice (by the addition of a rational vector).

`IsDirectSum(L): TorLat -> BoolElt`

Return `true` if and only if the toric lattice L was constructed as the direct sum of other toric lattices.

`IsQuotient(L): TorLat -> BoolElt`

Return `true` if and only if the toric lattice L was constructed as the quotient of another toric lattice.

Sublattice(L): TorLat -> TorLat, TorLatMap

If the toric lattice L was constructed as the extension of a toric lattice K by the addition of a vector, then return K.

Superlattice(L): TorLat -> TorLat, TorLatMap

If the toric lattice L was constructed as a sublattice of a toric lattice K, then return K.

Summands(L): TorLat -> SeqEnum, SeqEnum, SeqEnum

If the toric lattice L was constructed as a direct sum, return a sequence of the toric lattice summands used (and parallel sequences of their embedding maps in L and the projections from L to them).

Example: Basis of a Toric Sublattice (ex-2fbe11)

Construct a sublattice of a toric lattice L and compute its natural basis in the original coordinates of L; the user does not have control over the choice of coordinates and inclusion map.

```
> L := ToricLattice(2);
> L1,phi1 := Sublattice([L| [1,2],[2,1]]);
> L1;
2-dimensional toric lattice L1 = sub(Z^2)
> phi1(L1.1), phi1(L1.2);
(1, 2)
(0, 3)
```

A similar calculation for a quotient map.

```
> L2,phi2 := Quotient(L ! [3/2,2]);
> L2;
1-dimensional toric lattice L2 = (Z^2) / <3, 4>
> phi2(L.1), phi2(L.2);
(-4)
(3)
```

And again for an extension of L.

```
> L3,phi3 := AddVectorToLattice(L ! [1/5,2/5]);
> L3;
2-dimensional toric lattice L3 = (Z^2) + <1/5, 2/5>
> phi3(L1.1), phi3(L1.2);
(1, 0)
(2, 5)
```

156.6.4 Maps of Toric Lattices

Maps between toric lattices are of type `TorLatMap`. They can be constructed using the `hom< L -> K | M >` constructor, where M is the matrix of the desired linear map with respect to the standard bases of L and K. The usual evaluation operations `@` and `@@` can be applied, although there are image and preimage intrinsics too. Basic arithmetic of maps (sum, difference, and scalar multiple, each taken pointwise) is available.

ZeroMap(L, K): TorLat, TorLat -> TorLatMap

The zero map between toric lattices L and K.

IdentityMap(L): TorLat -> TorLatMap

The identity map on the toric lattice L.

hom< L -> K | M >: TorLat, TorLat, Mtrx -> TorLatMap

LatticeMap(L, K, M): TorLat, TorLat, Mtrx -> TorLatMap

The map between toric lattices L and K determined by the matrix M (with respect to the standard bases of L and M).

`LatticeMap(L, Q): TorLat, [TorLatElt] -> TorLatMap`

The toric lattice map from toric lattice L to the toric lattice M determined by the sequence Q of toric lattice points, where M is the toric lattice containing the points of Q.

`DefiningMatrix(f): TorLatMap -> ModMatRngElt`

The defining matrix of the lattice map f.

`Image(f, C): TorLatMap, TorCon -> TorCon`

`Image(f, P): TorLatMap, TorPol -> TorPol`

`Image(f, v): TorLatMap, TorLatElt -> TorLatElt`

The image of the cone C or polyhedron P or lattice element v under the toric lattice map f; the type of the object is preserved under the map.

`Preimage(f, C): TorLatMap, TorCon -> TorCon`

`Preimage(f, P): TorLatMap, TorPol -> TorPol`

`Preimage(f, v): TorLatMap, TorLatElt -> TorLatElt`

The preimage of the cone C or polyhedron P or lattice element v by the toric lattice map f; the type of the object is preserved.

`KernelEmbedding(f): TorLatMap -> Map`

`KernelEmbedding(v): TorLatElt -> Map`

The inclusion of the kernel of the toric lattice map f, regarded as a distinct toric lattice, into the domain of f; or the same for the dual subspace annihilated by lattice element v.

`KernelBasis(f): TorLatMap -> SeqEnum`

`KernelBasis(v): TorLatElt -> SeqEnum`

A basis for the kernel of the toric lattice map f or the dual of the sublattice annihilated by the lattice element v.

`ImageBasis(f): TorLatMap -> SeqEnum`

A basis for the image of the toric lattice map f.

`IsCokernelTorsionFree(f): TorLatMap -> BoolElt`

Return `true` if and only if the cokernel of the toric lattice map f is torsion free.

`ChangeBasis(v): TorLatElt -> Map`

Given a primitive form $v \in L^\vee$ in the lattice dual to L, returns a change of basis of L such that $\ker(v)$ maps to the standard codimension 1 lattice.

Example: Change of Basis Revealing Lattice Width `(ex-03b562)`

We start by constructing a polytope P of lattice width two:

```
> P:=Polytope([
>     [ 1, 0, -1 ],
>     [ 0, 0, 1 ],
>     [ 0, -1, 3 ],
>     [ -1, 1, 0 ],
>     [ 0, 0, -1 ]
> ]);
> width,ws:=Width(P);
> width;
2
```

We wish to construct a change of basis to make this explicit: we want to construct $Q \cong P$ such that Q is of width two with respect to $(0, 0, 1)$.

```
> phi:=ChangeBasis(Representative(ws));
> Q:=Image(phi,P);
> Q;
3-dimensional polytope Q with 5 vertices:
    ( 1,  0,  1),
    ( 0,  0,  1),
    ( 0, -1,  1),
    (-1,  1,  0),
    ( 0,  0, -1)
> w:=Dual(Ambient(Q)).3;
> [w * v : v in Vertices(Q)];
[ 1, 1, 1, 0, -1 ]
```

We conclude by taking slices through Q at each height:

```
> Polyhedron(Q,w,-1);
0-dimensional polytope with one vertex:
    (0,  0, -1)
> Polyhedron(Q,w,0);
2-dimensional polytope with 3 vertices:
    ( -1,    1,   0),
    (  0, -1/2,   0),
    (1/2,    0,   0)
> Polyhedron(Q,w,1);
2-dimensional polytope with 3 vertices:
    (0, -1,  1),
    (0,  0,  1),
    (1,  0,  1)
```

156.7 Bibliography

- [BP99] Alexander Barvinok and James E. Pommersheim. An algorithmic theory of lattice points in polyhedra. In *New perspectives in algebraic combinatorics (Berkeley, CA, 1996–97)*, volume 38 of Math. Sci. Res. Inst. Publ., pages 91–147. Cambridge Univ. Press, Cambridge, 1999.
- [DLHTY04] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symbolic Comput.*, 38(4):1273–1302, 2004. URL: http://dx.doi.org/10.1016/j.jsc.2003.04.003, doi:10.1016/j.jsc.2003.04.003.
- [GK13] Roland Grinis and Alexander M. Kasprzyk. Normal forms of convex lattice polytopes. Preprint; available at http://arxiv.org/abs/1301.6641, January 2013.
- [KS04] Maximilian Kreuzer and Harald Skarke. PALP, A package for analyzing lattice polytopes with applications to toric geometry. *Computer Phys. Comm.*, 157:87–106, 2004.
- [McKb] B. D. McKay. nauty User's Guide (Version 2.2). http://cs.anu.edu.au/ bdm/nauty/nug.pdf.
- [McK81] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

References

[BP99] Alexander Barvinok and James E. Pommersheim. An algorithmic theory of lattice points in polyhedra. In *New perspectives in algebraic combinatorics (Berkeley, CA, 1996–97)*, volume 38 of Math. Sci. Res. Inst. Publ., pages 91–147. Cambridge Univ. Press, Cambridge, 1999.

[Bue79] Francis Buekenhout. Diagrams for geometries and groups. *J. Combin. Theory Ser. A*, 27(2):121–151, 1979.

[Bue95] Francis Buekenhout. *Handbook of incidence geometry*. North-Holland, Amsterdam, 1995.

[BCon] (no details available)

[DLHTY04] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symbolic Comput.*, 38(4):1273–1302, 2004. URL: http://dx.doi.org/10.1016/j.jsc.2003.04.003, doi:10.1016/j.jsc.2003.04.003.

[GK13] Roland Grinis and Alexander M. Kasprzyk. Normal forms of convex lattice polytopes. Preprint; available at http://arxiv.org/abs/1301.6641, January 2013.

[JL04] Pascale Jacobs and Dimitri Leemans. An algorithmic analysis of the intersection property. *LMS J. Comput. Math.*, 7:284–299 (electronic), 2004.

[KS04] Maximilian Kreuzer and Harald Skarke. PALP, A package for analyzing lattice polytopes with applications to toric geometry. *Computer Phys. Comm.*, 157:87–106, 2004.

[MEFW] D. Leemans M. E. Fernandes and A. I. Weiss. Highly symmetric hypertopes. Preprint, arXiv:1604.03162.

[McKb] B. D. McKay. nauty User's Guide (Version 2.2). http://cs.anu.edu.au/ bdm/nauty/nug.pdf.

[McK81] B. D. McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[Pas94] Antonio Pasini. *Diagram geometries*. Oxford Science Publications. The Clarendon Press Oxford University Press, New York, 1994. ISBN 0-19-853497-3.

[Tit62] Jacques Tits. Géométries polyédriques et groupes simples. *Atti 2a Riunione Groupem. Math. Express. Lat. Firenze*, pages 66–88, 1962.

Index

Symbols

L (*operation*), 10
V (*operation*), 10
- P (*operation*), 56
5-Simplex from Its Rotation Group (*example*), 46
5-Simplex from a C-Group (*example*), 44

A

Action(G, Y) (*function*), 21
ActionImage(G, Y) (*function*), 21
ActionKernel(G, Y) (*function*), 21
AddVectorToLattice(Q) (*function*), 76
AddVectorToLattice(v) (*function*), 76
AffineNormalForm(P) (*function*), 63
AllPassants(P, A) (*function*), 16
AllSecants(P, A) (*function*), 16
AllTangents(P, A) (*function*), 16
AllTangents(P, U) (*function*), 18
Ambient(C) (*function*), 72
Ambient(P) (*function*), 72
AreGenerators(S) (*function*), 76
AreProportional(P,Q) (*function*), 74
Arithmetic with Toric Lattice Points (*example*), 74
Automorphism Group of a Polytope (*example*), 63
AutomorphismGroup(D) (*function*), 38
AutomorphismGroup(P) (*function*), 20, 63

B

Baer Derivation of an Affine Plane (*example*), 22
BaerDerivation(q2) (*function*), 25
BaerSubplane(P) (*function*), 25
Basis of a Toric Sublattice (*example*), 77
Basis(L) (*function*), 74
Basis(L,i) (*function*), 74
BDiagram(G) (*function*), 46
Borel(C) (*function*), 35
BorelSubgroup(C) (*function*), 35
BoundaryPoints(P) (*function*), 60
BoundingBox(P) (*function*), 52
Box Elements of a Simplicial Cone (*example*), 64
BoxElements(C) (*function*), 64

C

C + D (*operation*), 56
C + P (*operation*), 56
C eq D (*operation*), 56
C meet D (*operation*), 56
Central Collineations of a Plane (*example*), 23
CentralCollineationGroup(P, l) (*function*), 23
CentralCollineationGroup(P, p) (*function*), 23
CentralCollineationGroup(P, p, l) (*function*), 23
Change of Basis Revealing Lattice Width (*example*), 78
ChangeAmbient(C,L) (*function*), 72
ChangeAmbient(P,L) (*function*), 72
ChangeBasis(v) (*function*), 78
Classical Projective Plane of Order 3 (*example*), 2
Classifying Polygons by Affine Normal Form (*example*), 63
Codegree(P) (*function*), 70
CollineationGroup(P) (*function*), 20
CollineationGroupStabilizer(P, k) (*function*), 20
CollineationSubgroup(P) (*function*), 20
CompactPart(P) (*function*), 67
CompleteKArc(P, k) (*function*), 16
Cone Generators in a Sublattice (*example*), 65
Cone of a Cyclic Quotient Singularity (*example*), 66
Cone(A) (*function*), 53
Cone(v) (*function*), 53
ConeInSublattice(C) (*function*), 54
ConeQuotientByLinearSubspace(C) (*function*), 54
Cones Generated by Rays (*example*), 51
ConeToPolyhedron(C) (*function*), 55
ConeWithInequalities(B) (*function*), 53
Conic(P, S) (*function*), 16
Constructing the Hoffman-Singleton Graph (*example*), 30
ContainsQuadrangle(P, S) (*function*), 15
ContainsZero(P) (*function*), 69
Converting the Cube to a C-Group (*example*), 44
Coordinates(P, l) (*function*), 13
Coordinates(P, p) (*function*), 13
CorrelationGroup(D) (*function*), 38
Coset Geometry for Sym(6) (*example*), 34
CosetGeometry(D) (*function*), 36

CosetGeometry(G, S) (*function*), 32
CosetGeometry(G, S, I) (*function*), 32
CosetGeometryFromCGroup(G) (*function*), 44
CosetGeometryFromCPlusGroup(G) (*function*), 46
CosetGeometryToCGroup(D) (*function*), 44
CoxeterDiagram(G) (*function*), 44
Creating Points and Lines (*example*), 5
CrossPolytope(d) (*function*), 53
CrossPolytope(L) (*function*), 53
Cube as a Coset Geometry (*example*), 33
Cube as an Incidence Geometry (*example*), 29
CyclicPolytope(d,n) (*function*), 53
CyclicPolytope(L,n) (*function*), 53

D

Deconstructing Points and Lines (*example*), 14
DefiningMatrix(f) (*function*), 78
Degree(P) (*function*), 70
Design(P) (*function*), 25
Diagram of Neumaier's Geometry (*example*), 43
Diagram of the Cube (*example*), 42
Diagram of the Hoffman-Singleton Geometry (*example*), 42
Diagram of the Petersen Geometry (*example*), 41
Diagram(C) (*function*), 41
Diagram(D) (*function*), 41
Dimension(C) (*function*), 70
Dimension(L) (*function*), 74
Dimension(P) (*function*), 70
DirectSum(L,M) (*function*), 73
DirectSum(Q) (*function*), 73
Dual Toric Lattice (*example*), 73
Dual(C) (*function*), 54
Dual(L) (*function*), 73
Dual(P) (*function*), 9

E

EdgeFacetIncidenceMatrix(P) (*function*), 72
EdgeIndices(P) (*function*), 59
Edges(P) (*function*), 59
EhrhartCoefficient(P,k) (*function*), 60
EhrhartCoefficients(P,l) (*function*), 60
EhrhartDeltaVector(P) (*function*), 60
EhrhartPolynomial(P) (*function*), 60
EhrhartSeries(P) (*function*), 60
Elements(D) (*function*), 35
ElementToSequence(l) (*function*), 13
ElementToSequence(p) (*function*), 13
Eltseq(l) (*function*), 13
Eltseq(p) (*function*), 13
EmptyPolyhedron(L) (*function*), 55
Exterior(P, C) (*function*), 17
ExternalLines(P, A) (*function*), 16

F

FaceIndices(P,i) (*function*), 59
Faces(C) (*function*), 58
Faces(C,i) (*function*), 58
Faces(P) (*function*), 58
Faces(P,i) (*function*), 58
FaceSupportedBy(C,H) (*function*), 59
FacetIndices(P) (*function*), 58
Facets(C) (*function*), 58
Facets(P) (*function*), 58
Field(P) (*function*), 9
FiniteAffinePlane(D) (*function*), 25
FiniteAffinePlane(F) (*function*), 2
FiniteAffinePlane(P, l) (*function*), 11
FiniteAffinePlane(q) (*function*), 2
FiniteAffinePlane(W) (*function*), 2
FiniteAffinePlane< V | X : parameters > (*constructor*), 2
FiniteAffinePlane< v | X : parameters > (*constructor*), 2
FiniteProjectivePlane(D) (*function*), 25
FiniteProjectivePlane(F) (*function*), 2
FiniteProjectivePlane(q) (*function*), 2
FiniteProjectivePlane(W) (*function*), 2
FiniteProjectivePlane< V | X : parameters > (*constructor*), 1
FiniteProjectivePlane< v | X : parameters > (*constructor*), 1
FixedSubspaceToPolyhedron(G) (*function*), 55
FixedSubspaceToPolyhedron(L,G) (*function*), 55
Form(L,Q) (*function*), 74
FullCone(L) (*function*), 53
FullCone(n) (*function*), 53
fVector(C) (*function*), 58
fVector(P) (*function*), 58

G

G-Sets of the Collineation Group (*example*), 21
Graph(C) (*function*), 36
Graph(D) (*function*), 36
Graph(P) (*function*), 59
Group(C) (*function*), 35

H

HalfspaceToPolyhedron(Q,h) (*function*), 54
HalfspaceToPolyhedron(v,h) (*function*), 54
HasIntegralPoint(P) (*function*), 60
HasIntersectionProperty(C) (*function*), 39
HasIntersectionProperty(G) (*function*), 44
HasIntersectionPropertyN(C,n) (*function*), 39
HasIntersectionPropertyPlus(G) (*function*), 45
HasStringProperty(G) (*function*), 44
HasWeakIntersectionProperty(C) (*function*), 40
Hermitian Unital in PG(2,9) (*example*), 19
HilbertBasis(C) (*function*), 65
hom< L -> K | M > (*constructor*), 77
hVector(C) (*function*), 58
hVector(P) (*function*), 58
HyperplaneToPolyhedron(Q,h) (*function*), 55

HyperplaneToPolyhedron(v,h) (*function*), 55

I

IdentityMap(L) (*function*), 77
Image(f,C) (*function*), 78
Image(f,P) (*function*), 78
Image(f,v) (*function*), 78
Image(g, Y, y) (*function*), 20
ImageBasis(f) (*function*), 78
Incidence Graph and Code of a Plane (*example*), 26
IncidenceGeometry(C) (*function*), 36
IncidenceGeometry(G) (*function*), 28
IncidenceGraph(D) (*function*), 35
IncidenceGraph(P) (*function*), 26
IncidenceMatrix(P) (*function*), 9
Index(C) (*function*), 70
Index(P, l) (*function*), 13
Index(P, p) (*function*), 13
Inequalities Defining a Polytope (*example*), 57
Inequalities(C) (*function*), 57
Inequalities(P) (*function*), 57
InfinitePart(P) (*function*), 67
InnerNormal(C) (*function*), 58
IntegralPart(P) (*function*), 67
Interior(P, C) (*function*), 17
InteriorPoints(P) (*function*), 60
Is2T1(C) (*function*), 40
IsAffineLinear(P) (*function*), 69
IsArc(P, A) (*function*), 16
IsCentralCollineation(P, g) (*function*), 23
IsCGroup(G) (*function*), 44
IsCokernelTorsionFree(f) (*function*), 78
IsCollinear(P, S) (*function*), 15
IsComplete(P, A) (*function*), 16
IsConcurrent(P, R) (*function*), 15
IsCPlusGroup(G) (*function*), 45
IsDesarguesian(P) (*function*), 10
IsDirectSum(L) (*function*), 76
IsEmpty(P) (*function*), 67
IsEquivalent(P,Q) (*function*), 62
IsFace(C,F) (*function*), 59
IsFace(P,F) (*function*), 59
IsFirm(X) (*function*), 38
IsFlag(P) (*function*), 69
IsFTGeometry(C) (*function*), 38
IsFTGeometry(D) (*function*), 38
IsGraph(C) (*function*), 39
IsGraph(D) (*function*), 39
IsInInterior(v,C) (*function*), 60
IsInInterior(v,P) (*function*), 60
IsIntegral(v) (*function*), 75
IsIntegrallyClosed(P) (*function*), 69
IsIsomorphic(P, Q: parameters) (*function*), 11
IsIsomorphic(P,Q) (*function*), 61
IsLinearSpace(C) (*function*), 69
IsLinearSpace(P) (*function*), 69
IsLineTransitive(P) (*function*), 24
IsLocallyTwoTransitive(C) (*function*), 40
IsMaximumDimensional(C) (*function*), 69
IsMaximumDimensional(P) (*function*), 69
Isomorphism Testing for Polytopes (*example*), 61
IsOnBoundary(v,C) (*function*), 60
IsOnBoundary(v,P) (*function*), 60
IsParallel(P, l, m) (*function*), 15
IsPerfectlyCentered(P) (*function*), 69
IsPointed(P) (*function*), 69
IsPointTransitive(P) (*function*), 24
IsPolytope(P) (*function*), 70
IsPRI(C) (*function*), 40
IsPrimitive(C) (*function*), 40
IsPrimitive(v) (*function*), 75
IsPyramid(P) (*function*), 70
IsQuotient(L) (*function*), 76
IsRC(X) (*function*), 39
IsResiduallyConnected(X) (*function*), 39
IsResiduallyPrimitive(C) (*function*), 40
IsResiduallyWeaklyPrimitive(C) (*function*), 40
IsResiduallyWealyPrimitive(C) (*function*), 40
IsRPRI(C) (*function*), 40
IsRWP(C) (*function*), 40
IsRWPRI(C) (*function*), 40
IsSelfDual(P) (*function*), 10
IsSimple(P) (*function*), 69
IsSimplex(P) (*function*), 69
IsSimplicial(C) (*function*), 69
IsSimplicial(P) (*function*), 69
IsStrictlyConvex(C) (*function*), 69
IsStringCGroup(G) (*function*), 44
IsSublattice(L) (*function*), 76
IsSuperlattice(L) (*function*), 76
IsSupportingHyperplane(v,h,P) (*function*), 59
IsThick(X) (*function*), 39
IsThin(X) (*function*), 39
IsTransitive(P) (*function*), 24
IsUnital(P, U) (*function*), 18
IsWeaklyPrimitive(C) (*function*), 40
IsWPRI(C) (*function*), 40
IsZero(C) (*function*), 69
IsZero(v) (*function*), 75

K

k * P (*operation*), 56
kArc(P, k) (*function*), 16
Kernel(C) (*function*), 35
KernelBasis(f) (*function*), 78
KernelBasis(v) (*function*), 78
KernelEmbedding(f) (*function*), 78
KernelEmbedding(v) (*function*), 78
Kernels(C) (*function*), 35
Knot(P, C) (*function*), 16

L

L ! l (*operation*), 4

L ! S (*operation*), 4
L ! [a, b, c] (*operation*), 4
L ! [a,b,...] (*operation*), 74
L ! [m, b] (*operation*), 4
L + M (*operation*), 73
L . i (*operation*), 4, 74
L ^ n (*operation*), 74
L eq K (*operation*), 76
l eq m (*operation*), 12
l meet m (*operation*), 13
l ne m (*operation*), 12
Lattice Width of a Polytope (*example*), 70
LatticeBasisInCone(C) (*function*), 54
LatticeMap(L,K,M) (*function*), 77
LatticeMap(L,Q) (*function*), 77
LatticeVector(L,Q) (*function*), 74
LatticeVector(Q) (*function*), 74
LinearCode(P, K) (*function*), 26
LinearSpanEquations(C) (*function*), 57
LinearSpanEquations(Q) (*function*), 57
LinearSpanGenerators(C) (*function*), 57
LinearSpanGenerators(Q) (*function*), 57
LinearSubspaceGenerators(C) (*function*), 57
LineGraph(P) (*function*), 26
LineGroup(P) (*function*), 20
Lines(P) (*function*), 7
LineSet(P) (*function*), 4
LocallySArcTransitive(C) (*function*), 40
l[i] (*literal*), 13

M

Matrix(R,S) (*function*), 75
Matrix(S) (*function*), 75
MatrixOfInequalities(C) (*function*), 57
MatrixOfInequalities(R,C) (*function*), 57
MaximalParabolics(C) (*function*), 35
MaximalVertexFacetHeightMatrix(P) (*function*), 63
MaxParabolics(C) (*function*), 35
MinimalInequalities(C) (*function*), 57
MinimalParabolics(C) (*function*), 35
MinimalRGenerators(C) (*function*), 65
MinParabolics(C) (*function*), 35

N

n * P (*operation*), 74
Neumaier Geometry as a Coset Geometry (*example*), 36
Normal Form of a Lattice Polytope (*example*), 62
NormalCone(P,F) (*function*), 58
NormalEdgeCones(P) (*function*), 58
NormalForm(P) (*function*), 62
NormalisedCone(P) (*function*), 54
NumberOfBoundaryPoints(P) (*function*), 60
NumberOfEdges(P) (*function*), 59
NumberOfFaces(P,i) (*function*), 59
NumberOfFacets(P) (*function*), 58
NumberOfInteriorPoints(P) (*function*), 60
NumberOfLines(P) (*function*), 10
NumberOfPoints(P) (*function*), 10, 60
NumberOfVertices(P) (*function*), 56
Numerical Invariants of PG(2,8) (*example*), 10

O

Orbit(G, Y, y) (*function*), 20
Orbits(G, Y) (*function*), 20
Order(P) (*function*), 10
OuterNormal(C) (*function*), 58
Oval Design from an Arc (*example*), 17
OvalDerivation(q: parameters) (*function*), 25

P

P * Q (*operation*), 56
P + C (*operation*), 56
P + Q (*operation*), 56, 74
P / n (*operation*), 74
P / Q (*operation*), 74
P - Q (*operation*), 74
P eq Q (*operation*), 11, 56, 74
p eq q (*operation*), 12
p in l (*operation*), 13
P meet Q (*operation*), 56
P ne Q (*operation*), 11
p ne q (*operation*), 12
p notin l (*operation*), 13
P subset Q (*operation*), 11, 56
PALPNormalForm(P) (*function*), 62
ParallelClass(P, l) (*function*), 15
ParallelClasses(P) (*function*), 15
ParentPlane(L) (*function*), 6
ParentPlane(l) (*function*), 6
ParentPlane(p) (*function*), 6
ParentPlane(V) (*function*), 6
Pencil(P, p) (*function*), 15
Petersen Graph as a Coset Geometry (*example*), 33
Petersen Graph as an Incidence Geometry (*example*), 29
Point and Line Functions in AG(2,3) (*example*), 15
PointGroup(P) (*function*), 20
Points versus Support of a Plane (*example*), 7
Points(C,H,h) (*function*), 65
Points(D) (*function*), 35
Points(P) (*function*), 7, 60
PointSet(P) (*function*), 3
Polar Polyhedron of a Polytope (*example*), 67
Polar(P) (*function*), 53
Polyhedra as Slices of a Quadrant (*example*), 55
Polyhedron from a Cone (*example*), 68
Polyhedron(C) (*function*), 54
Polyhedron(C,f,v) (*function*), 55
Polyhedron(C,H,h) (*function*), 54

PolyhedronInSublattice(P) (*function*), 55
PolyhedronWithInequalities(A,c) (*function*), 52
Polytope as a Convex Hull (*example*), 48
Polytope(Q) (*function*), 52
PolytopeOfProjectiveSpace(d) (*function*), 53
PolytopeOfProjectiveSpace(L) (*function*), 53
PolytopeOfWPS(d) (*function*), 53
PolytopeOfWPS(L,W) (*function*), 53
PolytopeOfWPS(W) (*function*), 53
PositiveQuadrant(L) (*function*), 53
PositiveQuadrant(n) (*function*), 53
pRank(P) (*function*), 10
pRank(P, p) (*function*), 10
Preimage(f,C) (*function*), 78
Preimage(f,P) (*function*), 78
Preimage(f,v) (*function*), 78
Primitive Points in a Sublattice (*example*), 75
PrimitiveLatticeVector(v) (*function*), 75
Projective Bundle in PG(2,q) (*example*), 22
Projective Embedding of an Affine Plane (*example*), 11
Projective Plane from a Singer Difference Set (*example*), 25
ProjectiveEmbedding(P) (*function*), 11
Pyramid(P) (*function*), 71
p[i] (*literal*), 13

Q

QuadraticForm(S) (*function*), 16
Quotient(C) (*function*), 76
Quotient(C, K) (*function*), 36
Quotient(Q) (*function*), 76
Quotient(v) (*function*), 76
QuotientGenerators(C) (*function*), 66

R

Random(L) (*function*), 4
Random(l) (*function*), 13
Random(V) (*function*), 4
RandomCone(d,n,k) (*function*), 54
RandomCone(L,n,k) (*function*), 54
RandomPolytope(d,n,k) (*function*), 52
RandomPolytope(L,n,k) (*function*), 52
RandomPositiveCone(d,n,k) (*function*), 54
RandomPositiveCone(L,n,k) (*function*), 54
Rank Four Geometry from Hoffman-Singleton (*example*), 31
Rank(C) (*function*), 35
Rank(D) (*function*), 35
Ray(C,i) (*function*), 57
Rays(C) (*function*), 57
Recognising a Pyramid (*example*), 71
Rep(L) (*function*), 4
Rep(l) (*function*), 13
Rep(V) (*function*), 4
Representative(L) (*function*), 4
Representative(l) (*function*), 13
Representative(V) (*function*), 4
Residue(C, f) (*function*), 37
Residue(D, f) (*function*), 37
RGenerators(C) (*function*), 65

S

S notsubset l (*operation*), 13
S subset l (*operation*), 13
ScalarLattice() (*function*), 73
Set(l) (*function*), 14
Shadow(D, I, F) (*function*), 37
ShadowSpace(D,I) (*function*), 38
SimplicialSubcone(C) (*function*), 54
Slope(l) (*function*), 15
Stabilizer(G, Y, y) (*function*), 21
StandardSimplex(d) (*function*), 53
StandardSimplex(L) (*function*), 53
sub<P | L> (*constructor*), 8
SubfieldSubplane(P, F) (*function*), 8
Sublattice(L) (*function*), 76
Sublattice(Q) (*function*), 76
Subplane Generated by a Quadrangle (*example*), 8
Summands(L) (*function*), 77
Summing an Empty Sequence of Points (*example*), 73
Superlattice(L) (*function*), 77
Support(l) (*function*), 7
Support(P) (*function*), 7
Support(p) (*function*), 7
Support(P, p) (*function*), 7
SupportingCone(P,v) (*function*), 59

T

Tangent(P, A, p) (*function*), 16
ToricLattice(n) (*function*), 73
ToricLattice(Q) (*function*), 76
Transitivity of a Collineation Group (*example*), 24
Triangulation(P) (*function*), 64
TriangulationOfBoundary(P) (*function*), 64
Truncation(C, t) (*function*), 37
Truncation(D, t) (*function*), 37
Types(C) (*function*), 35
Types(D) (*function*), 35

U

UnitalFeet(P, U, p) (*function*), 19

V

V ! x (*operation*), 4
V ! [a, b, c] (*operation*), 4
V ! [a, b] (*operation*), 4
V . i (*operation*), 4
v in C (*operation*), 59
v in L (*operation*), 75
v in P (*operation*), 59

`Vector Space and Field of a Plane` (*example*), 9
`Vector(v)` (*function*), 75
`VectorSpace(P)` (*function*), 9
`VertexEdgeIncidenceMatrix(P)` (*function*), 72
`VertexFacetHeightMatrix(P)` (*function*), 72
`VertexFacetIncidenceMatrix(P)` (*function*), 72
`Vertices(P)` (*function*), 56
`Volume(P)` (*function*), 60
`VolumeOfBoundary(P)` (*function*), 60

W

`Width(P)` (*function*), 70
`Width(P,u)` (*function*), 70

Y

`y ^ G` (*operation*), 20
`y ^ g` (*operation*), 20

Z

`Zero(L)` (*function*), 74
`ZeroCone(L)` (*function*), 54
`ZeroCone(n)` (*function*), 54
`ZeroMap(L,K)` (*function*), 77
`ZGenerators(C)` (*function*), 65