convex 1.0 alpha

This worksheet describes the new features of 'convex' and differences to versions <1.
We suppose that the new version is stored as "convex.m" in a directory appearing in the libname variable!

```
> with(convex);
convex 1.0 as of Thu Mar 20 10:02:32 CET 2003
Copyright (C) 1999-2003 Matthias Franz
Distributed under the GNU General Public License
Warning, the name polar has been redefined
```

[*affinehull*, *ambientdim*, *arecompatible*, *boundary*, *codim*, *contains*, *containsrelint*, *convhull*, *corank*, *crosspolytope*, *cube*, *cyclicpolytope*, *delaunay*, *dim*, *directsum*, *distance*, *domain*, *dotprod*, *draw*, *dual*, *edges*, *emptypcomplex*, *emptypolyhedron*, *emptypolytope*, *facechain*, *facefan*, *faces*, *facets*, *fan*, *flagf*, *flagh*, *fullcone*, *fullpolyhedron*, *furthestdelaunay*, *fvector*, *genhvector*, *hilbertbasis*, *homology*, *hplanes*, *hspacenos*, *hspaces*, *image*, *incidencematrix*, *incidentfacets*, *incidentrays*, *intersection*, *isaffine*, *isbounded*, *iscomplete*, *iscontained*, *isempty*, *isface*, *isfulldim*, *islinear*, *ispointed*, *ispolytopal*, *isquasipolytopal*, *isregular*, *issimple*, *issimplicial*, *issimplicial1*, *join*, *lensspace*, *lineality*, *linearhull*, *lines*, *maximal*, *maximum*, *minimal*, *minimum*, *minkowskisum*, *normalfan*, *pcomplex*, *permutahedron*, *plotdata*, *polar*, *poshull*, *posorthant*, *preimage*, *projspace*, *proximum*, *randompolytope*, *rank*, *raynos*, *rays*, *readpoly*, *recession*, *regularpart*, *regularsubdiv*, *relint*, *simplicialsubdiv*, *skeleton*, *stdsimplex*, *stellarsubdiv*, *support*, *surface*, *torsion*, *transversalfan*, *traverse*, *traverse2*, *vertices*, *volume*, *voronoi*, *wprojspace*, *writepoly*, *zerocone*, *zerofan*]

What's cool?

- New test whether a given fan is (quasi-)polytopal.
- New type PCOMPLEX for polyhedral complexes.
- Functions for Voronoi diagrams and Delaunay "triangulations".
- The function 'homology' computes Betti numbers of fans and polyhedral complexes.
- Functions to compute flag f-vectors, flag h-vectors and generalised h-vectors.
- The function 'traverse' applies a given function to all elements of a fan or polyhedral complex.

infolevel

now works for each type separately. For example, if you want to have information about the processing of fans, say

```
> infolevel[FAN] := 3:
```

In addition, you may use the following to control information about the Motzkin elemination procedure:

```
> infolevel[CONVEX] := 0:
```

vec

is the type formerly called 'ray'. It now also accepts the new Vector type.

ray

is the type formerly called 'affray'.

mat

now also accepts the new Matrix type.

CONE

CONE1

is the type of non-zero cones

CONE0

is the type of pointed cones.

poshull(arg1:{vec, line, CONE, POLYHEDRON}, ...)::CONE

is the new name of 'hull'. It now also accepts polyhedra as arguments. It still returns the cone (with apex the origin) generated by all its arguments:

```
> P := stdsimplex(2);
  vertices(P);
```

$$P := \mathrm{POLYTOPE}(2, 2, 3, 3)$$

$$[[0, 0], [1, 0], [0, 1]]$$

```
> C := poshull(P);
```

$$C := \mathrm{CONE}(2, 2, 0, 2, 2)$$

```
> C &= posorthant(2);
```

true

```
> poshull(P, [-1, -1]) &= fullcone(2);
```

true

```
> poshull(emptypolyhedron(3)) &= zerocone(3);
```

true

lines
hplanes

are now in some canonical form due to Gaussian elimination.

```
> C := poshull([1,1,1,0], line([2,1,0,2]), line([0,0,0,3]));
```

$$C := \mathrm{CONE}(4, 3, 2, 1, 1)$$

```
> lines(C);
```

$$[[2, 1, 0, 0], [0, 0, 0, 1]]$$

```
> hplanes(C);
```

$$[[1, -2, 1, 0]]$$

rays
hspaces

also undergo Gaussian elimination with respect to lines and hyperplanes, respectively. As a consequence, they are not necessarily orthogonal to the lines and hyperplanes any more:

```
> rays(C);
```

$$[[0, 1, 2, 0]]$$

```
> hspaces(C);
```

$$[[0, 0, 1, 0]]$$

image(C::CONE, A::{mat, rational, real_infinity})::CONE
preimage(C::CONE, A::{mat, rational, real_infinity})::CONE

now also accept a second argument of type rational or 'infinity' or '-infinity':

```
> C := posorthant(2);
```

$$C := \mathrm{CONE}(2, 2, 0, 2, 2)$$

```
> rays(image(C, -2));
```

$$[[-1, 0], [0, -1]]$$

```
> image(C, infinity) &= fullcone(2);
```

true

polar(C::CONE)::CONE

is the negative of the dual.

maximal(C::CONE)::CFACE
minimal(C::CONE)::CFACE

are the new names of maxface(C) and minface(C). The same applies to polyhedra.

```
> maximal(C);
```

CFACE(2, 0)

```
> minimal(C);
```

CFACE(0, 2)

support(C::CONE, x1, ...)::CFACE

The calling format has changed. Lists are not longer allowed except if they represent rays. The same applies to the other variants of 'support'.

```
> support(C, [1, 2]);
```

CFACE(2, 0)

```
> support(C, [2, -1]);
```

FAIL

```
> support(C, [[1, 2], [2, -1]]);
Error, (in CONE[support]) illegal argument type(s)
```

The reason for this change is that the following statement is now unambiguous:

```
> support(fullpolyhedron(0), []);
```

CFACE(1, 0)

The following makes sense:

```
> f := support(C);
  f &= minimal(C);
```

$$f := \mathrm{CFACE}(0, 2)$$

true

POLYHEDRON

POLYHEDRON1

is the type of non-empty polyhedra.

POLYHEDRON0

POLYHEDRON_0
POLYHEDRON__0

are the types of polyhedra containing the origin, cantaining it in their relative interiors and in their interiors, repsectively.

convhull(arg1::{vec, ray, line, POLYHEDRON, CONE}, ...)::POLYHEDRON

is the new name of 'convexhull'. It now accepts also cones as arguments, which are automatically converted to affine cones:

```
> P := convhull(cube(2), posorthant(2));
```

P := POLYHEDRON(2, 2, 0, [1, 2], [2])

```
> vertices(P);
  rays(P);
```

[[-1, -1]]

[[0, 1], [1, 0]]

polar(P::POLYHEDRON)::POLYHEDRON0

is again the negative of the dual, i.e., equal to image(dual(P), -1).

emptypolytope(n::nonnegint)::POLYTOPE

is a synonym for 'emptypolyhedron'.

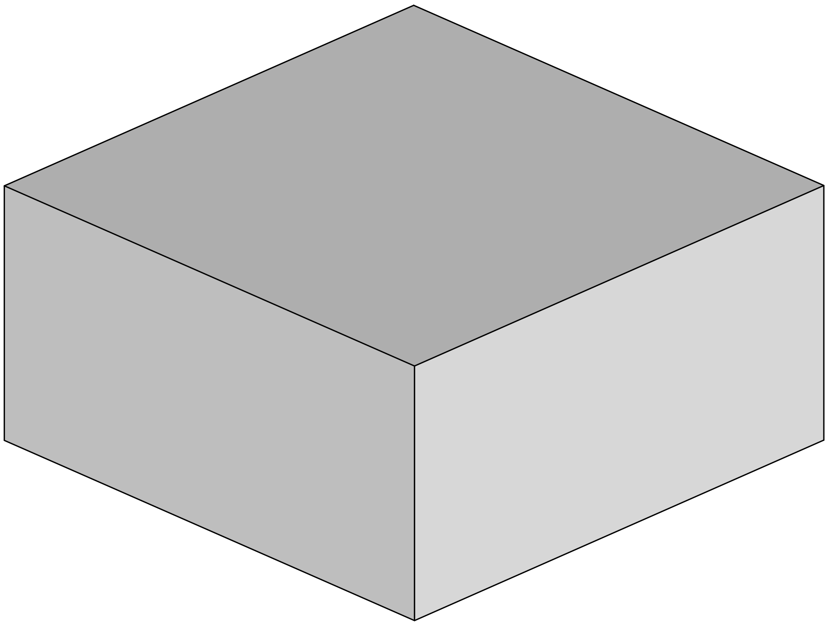
draw(P::POLYTOPE)
plotdata(P::POLYTOPE)

are used for graphics. 'plotdata' just gives the data structures to be used inside PLOT or PLOT3D,while 'draw' directly plots the object:

```
> plotdata(cube(2));
```

POLYGONS([[-1, -1], [-1, 1], [1, 1], [1, -1]])

```
> draw(cube(3));
```



genhvector(P::POLYHEDRON1)::Array

is the same as genhvector(normalfan(P)), see below.

FAN

FAN0

is the type of pointed fans.

codim(F::FAN)::nonnegint

is the codimension of the linear hull of the elements in F.

dim(F::FAN)::nonnegint

now is the maximal dimension of elements of F, not the dimension of their linear hull. As a consequence, dimension and codimension do not necessarily add up to the ambient dimension:

```
> F := fan(poshull([1, 0]), poshull([0, 1]));
```

$$F := \mathrm{FAN}(2, 2, 0, 2, [2, 0])$$

```
> ambientdim(F);
  dim(F), codim(F);
```

$$2$$

$$1, 0$$

fan(arg1:{CONE, FAN}, ...)::FAN

now also accepts other fans as arguments:

```
> F1 := facefan(convhull([1, 0], [-1, 0]));
```

$$F1 := \mathrm{FAN}(2, 1, 0, 2, [2, 0])$$

```
> F2 := fan(F1, poshull([1, 1], [-1, 2]));
```

$$F2 := \mathrm{FAN}(2, 2, 0, 4, [2, 1])$$

maximal(F::FAN)::list(FANCONE)
maximal(F::FAN, list)::list(FANCONE)
maximal(F::FAN, array)::array(list(FANCONE))

The function 'maximal' replaces 'maxcones' and 'maxconeslist':

```
> maximal(F2);
```

$$[\mathrm{FANCONE}(2, 2, 0, 2, 2), \mathrm{FANCONE}(2, 1, 0, 1, 1), \mathrm{FANCONE}(2, 1, 0, 1, 1)]$$

```
> maximal(F2, list);
```

$$[\mathrm{FANCONE}(2, 2, 0, 2, 2), \mathrm{FANCONE}(2, 1, 0, 1, 1), \mathrm{FANCONE}(2, 1, 0, 1, 1)]$$

```
> maximal(F2, array);
```

array(0 .. 2, [

$(0) = [\,]$

$(1) = [\mathrm{FANCONE}(2, 1, 0, 1, 1), \mathrm{FANCONE}(2, 1, 0, 1, 1)]$

$(2) = [\mathrm{FANCONE}(2, 2, 0, 2, 2)]$

])

minimal(F::FAN)::list(FANCONE)

is there for the sake of completeness:

```
> minimal(F2);
```

$$[\mathrm{FANCONE}(2, 0, 0, 0, 0)]$$

image & preimage

work as for cones. The preimage of a fan is always a fan, but not necessarily the image.

```
> F := normalfan(cube(2));
```

$$F := \mathrm{FAN}(2, 2, 0, 4, [0, 4])$$

```
> preimage(F, [[1], [0]]);
```

$$\mathrm{FAN}(1, 1, 0, 2, [2])$$

```
> image(F, [[1, 1]]);
Error, (in FAN[image]) arguments must be compatible
```

fvector(F::FAN)::Array

returns the number of cones in each dimension:

```
> fvector(F);
```

Array(0 .. 2, {(0) = 1, (1) = 4, (2) = 4}, *datatype* = *anything*, *storage* = *rectangular*, *order* = *Fortran_order*)

flagf(F::{FAN, CONE}, S::set(integer))::nonnegint

returns the number of chains of cones of the given dimensions in F:

```
> flagf(F, {1, 2});
```

$$8$$

```
> flagf(F, {0, 1, 2});
```

$$8$$

```
> flagf(F, {});
```

$$1$$

This is by definition.

flagh(F::{FAN, CONE}, S::set(integer))::integer

returns the corresponding flag h-number, i.e., the alternating sum over all flagf(F, T) where T runs over all subsets of S:

```
> flagh(F, {1, 2});
```

$$1$$

genhvector(F::{FAN0, CONE0})::Array

computes Stanley's generalised h-vector (which equals for, say, complete or affine F the intersection homology Betti numbers of the associated toric variety). For simplicial F this is the ordinary h-vector.

```
> genhvector(F);
```

Array(0 .. 2, {(0) = 1, (1) = 2, (2) = 1 }, *datatype* = *anything*, *storage* = *rectangular*, *order* = *Fortran_order*)

```
> genhvector(facefan(cube(3)));
```

Array(0 .. 3, {(0) = 1, (1) = 5, (2) = 5, (3) = 1 }, *datatype* = *anything*, *storage* = *rectangular*, *order* = *Fortran_order*)

skeleton(F::{FAN, CONE}, k::nonnegint)::FAN

returns the k-dimensional skeleton of F:

```
> skeleton(F2, 2);
```

FAN(2, 2, 0, 4, [2, 1])

```
> skeleton(F2, 1);
```

FAN(2, 2, 0, 4, [4, 0])

```
> skeleton(F2, 0);
```

FAN(2, 0, 0, 0, [0, 0])

regularpart(F::FAN0)::FAN0

returns the subfan of regular cones.

```
> regularpart(F2);
```

FAN(2, 2, 0, 4, [4, 0])

simplicialsubdiv(F::FAN0)::FAN0

returns some simplicial subdivison of F. The subdivision is made without introducing new rays. The simplicial part of F is not changed.

```
> F0 := facefan(cube(3));
  F1 := simplicialsubdiv(F0);
```

F0 := FAN(3, 3, 0, 8, [0, 0, 6])

F1 := FAN(3, 3, 0, 8, [0, 0, 12])

regularsubdiv(F::FAN0)::FAN0

returns some regular subdivision of F. The regular part of F is not changed.

```
> isregular(F0);
```

false

```
> F2 := regularsubdiv(F0);
```

F2 := FAN(3, 3, 0, 26, [0, 0, 48])

```
> isregular(F2);
```

true

ispolytopal(F::FAN)::boolean
ispolytopal(F::FAN, P::name)::boolean
isquasipolytopal(F::FAN)::boolean
isquasipolytopal(F::FAN, P::name)::boolean

A fan is called polytopal if it is the normal fan of a polytope, and quasi-polytopal if it is a subfan of a polytopal fan. These functions test these properties. If a second argument is given, it receives a polyhedron (not necessarily a polytope) such that F is a subfan of normalfan(P).

```
> ispolytopal(normalfan(cube(2)), 'P');
```

true

```
> P;
  vertices(P);
```

POLYHEDRON(2, 2, 0, [1, 2], [2])

[[0, 0]]

A fan is polytopal iff it is quasi-polytopal and complete.

```
> ispolytopal(fan(posorthant(2)), 'P');
  isquasipolytopal(fan(posorthant(2)), 'P');
```

false

true

```
> P;
  vertices(P);
  rays(P);
```

POLYHEDRON(2, 2, 0, [1, 2], [2])

[[0, 0]]

[[1, 0], [0, 1]]

Polytopal fans need not be pointed:

```
> ispolytopal(fan(fullcone(3)), 'P');
```

true

```
> P;
  vertices(P);
```

POLYTOPE(3, 0, 1, 1)

[[0, 0, 0]]

An example of a regular, complete, non-polytopal fan (Demazure, cf. Fulton, p. 71):

```
> v := [[-1, 0, 0], [ 0,-1, 0], [ 0, 0,-1],
        [ 1, 1, 1], [ 1, 1, 0], [ 0, 1, 1], [ 1, 0, 1]];
  F := fan(op(map(x -> poshull(v[x[1]], v[x[2]], v[x[3]]),
  [[1,2,3],
           [1,2,6], [2,6,7], [6,7,4],
           [2,3,7], [3,7,5], [7,5,4],
           [3,1,5], [1,5,6], [5,6,4]])));
```

v := [[-1, 0, 0], [0, -1, 0], [0, 0, -1], [1, 1, 1], [1, 1, 0], [0, 1, 1], [1, 0, 1]]

F := FAN(3, 3, 0, 7, [0, 0, 10])

```
> isregular(F);
```

true

```
> iscomplete(F);
```

true

```
> isquasipolytopal(F);
```

$$false$$

Strange enough, the following seems to be too hard:

```
> F := facefan(cube(3));
  isquasipolytopal(skeleton(F, 2));
```

$$F := \mathrm{FAN}(3, 3, 0, 8, [0, 0, 6])$$

```
Warning, computation interrupted
```

homology(F::FAN)::Array

computes the Betti numbers of F, see description under PCOMPLEX. These are essentially the Betti numbers of the intersection of F with a sphere around the origin.

```
> C := posorthant(2);
```

$$C := \mathrm{CONE}(2, 2, 0, 2, 2)$$

```
> homology(fan(C));
```

$$\mathrm{Array}(0\,..\,2, \{\ \}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> homology(boundary(C));
```

$$\mathrm{Array}(0\,..\,2, \{(1) = 1\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> homology(fan(fullcone(3)));
```

$$\mathrm{Array}(0\,..\,3, \{(3) = 1\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

traverse(F::{FAN, CONE}, g::procedure)

This is some kind of 'map' for fans instead of lists. It calls g(f, convert(rays(f), set)) for every element f of F. (Whether or not the entire fan is traversed is actually controlled by the return value of g.) For performance reasons, f is a face of some (maximal) cone in F, hence of type CFACE, not FANCONE. The result of g must evaluate to 'true' or 'false'. If it is 'false', then the faces of f are skipped. The function traverse returns a set. Before we describe it in detail, let us give a (silly) example:

```
> g := proc(f)
     userinfo(1, g, rays(f));
     dim(f) > 2
  end;
```

$$g := \textbf{proc}(f)\ \mathrm{userinfo}(1, g, \mathrm{rays}(f));\ 2 < \dim(f)\ \textbf{end proc}$$

(Note that we could use the second argument of g instead of calling rays(f).)

```
> infolevel[g] := 1:
```

```
> F := facefan(cube(3));
```

$$F := \mathrm{FAN}(3, 3, 0, 8, [0, 0, 6])$$

The following call lists all faces of dimension at least 2. (Think about the strict inequality in the definition of g!)

```
> A := traverse(F, g):
g:   [[1, -1, -1], [1, 1, -1], [1, -1, 1], [1, 1, 1]]
g:   [[1, -1, 1], [1, 1, 1]]
g:   [[1, 1, -1], [1, 1, 1]]
g:   [[1, -1, -1], [1, -1, 1]]
g:   [[1, -1, -1], [1, 1, -1]]
g:   [[-1, -1, 1], [1, -1, 1], [-1, 1, 1], [1, 1, 1]]
g:   [[-1, 1, 1], [1, 1, 1]]
g:   [[-1, -1, 1], [-1, 1, 1]]
g:   [[-1, -1, 1], [1, -1, 1]]
g:   [[-1, 1, -1], [1, 1, -1], [-1, 1, 1], [1, 1, 1]]
g:   [[-1, 1, -1], [-1, 1, 1]]
g:   [[-1, 1, -1], [1, 1, -1]]
```

```
g:   [[-1, -1, -1], [-1, 1, -1], [-1, -1, 1], [-1, 1, 1]]
g:   [[-1, -1, -1], [-1, -1, 1]]
g:   [[-1, -1, -1], [-1, 1, -1]]
g:   [[-1, -1, -1], [1, -1, -1], [-1, -1, 1], [1, -1, 1]]
g:   [[-1, -1, -1], [1, -1, -1]]
g:   [[-1, -1, -1], [1, -1, -1], [-1, 1, -1], [1, 1, -1]]
```

The format of the result of traverse is as follows: It is a set containing the sets of rays of all cones for which g was called:

```
> A;
```

{{[-1, -1, -1], [-1, 1, -1], [-1, -1, 1], [-1, 1, 1]}, {[-1, 1, -1], [1, 1, -1]},
{[-1, -1, -1], [-1, 1, -1]}, {[-1, -1, -1], [1, -1, -1], [-1, -1, 1], [1, -1, 1]},
{[1, -1, -1], [1, 1, -1], [1, -1, 1], [1, 1, 1]}, {[-1, -1, -1], [1, -1, -1], [-1, 1, -1], [1, 1, -1]},
{[-1, -1, -1], [1, -1, -1]}, {[1, -1, 1], [1, 1, 1]}, {[1, 1, -1], [1, 1, 1]},
{[1, -1, -1], [1, -1, 1]}, {[-1, -1, 1], [1, -1, 1], [-1, 1, 1], [1, 1, 1]}, {[1, -1, -1], [1, 1, -1]},
{[-1, -1, -1], [-1, -1, 1]}, {[-1, 1, 1], [1, 1, 1]}, {[-1, -1, 1], [-1, 1, 1]},
{[-1, -1, 1], [1, -1, 1]}, {[-1, 1, -1], [1, 1, -1], [-1, 1, 1], [1, 1, 1]}, {[-1, 1, -1], [-1, 1, 1]}
}

The following statements just produce the f-vector of F (and this is how fvector is actually implemented):

```
> fv := Array(0..ambientdim(F)):
  traverse(F, proc(f) global d, fv; d := dim(f); fv[d] :=
  fv[d]+1; true end):
  fv;
```

Array(0 .. 3, {(0) = 1, (1) = 8, (2) = 12, (3) = 6}, *datatype* = *anything*,
storage = *rectangular*, *order* = *Fortran_order*)

The functions skeleton and regularpart also use traverse.

traverse2(F::{FAN, CONE}, g2::procedure)
traverse2(F::FAN, g1::procedure, g2::procedure)

The first variant calls g2(f0::CFACE, f1::CFACE, convert(rays(f0), set), convert(rays(f1), set)) for every pair (f0, f1) such that f0 is a facet of f1. Both faces have the same domain. The second variant combines traverse and traverse2: It calls g1(f1::CFACE, convert(rays(f1), set)) for every element f1 of F. If g1 retuns true, then for all facets f0 of f1 it calls g2(f0, f1, ...) and does the same recursively for f0 instead of f1. The result is in both cases like that of traverse.

```
> g2 := proc(f0, f1)
    userinfo(1, g, rays(f0), rays(f1))
  end;
```

$g2$:= **proc**($f0$, $f1$) userinfo(1, g, rays($f0$), rays($f1$)) **end proc**

```
> F := facefan(cube(2));
```

F := FAN(2, 2, 0, 4, [0, 4])

```
> traverse2(F, g2);
g2:   [[-1, -1]]   [[-1, -1], [-1, 1]]
g2:   []   [[-1, -1]]
g2:   [[-1, 1]]   [[-1, -1], [-1, 1]]
g2:   []   [[-1, 1]]
g2:   [[-1, -1]]   [[-1, -1], [1, -1]]
g2:   [[1, -1]]   [[-1, -1], [1, -1]]
g2:   []   [[1, -1]]
g2:   [[-1, 1]]   [[-1, 1], [1, 1]]
g2:   [[1, 1]]   [[-1, 1], [1, 1]]
g2:   []   [[1, 1]]
g2:   [[1, -1]]   [[1, -1], [1, 1]]
g2:   [[1, 1]]   [[1, -1], [1, 1]]
```

{{ }, {[-1, -1]}, {[-1, 1], [-1, -1]}, {[-1, -1], [1, -1]}, {[-1, 1]}, {[1, 1], [1, -1]}, {[1, 1]}, {[-1, 1], [1, 1]}, {[1, -1]}}

```
> traverse2(F, g, g2);
g:   [[-1, -1], [-1, 1]]
g:   [[-1, -1], [1, -1]]
g:   [[-1, 1], [1, 1]]
g:   [[1, -1], [1, 1]]
```

{{[-1, 1], [-1, -1]}, {[-1, -1], [1, -1]}, {[1, 1], [1, -1]}, {[-1, 1], [1, 1]}}

The function homology is implemented with the help of traverse2.

FANCONE

maximum(C1::FANCONE, ...)::FANCONE
minimum(C1::FANCONE, ...)::FANCONE

are the new names for 'maxcone' and 'mincone'.

preimage(C::FANCONE, A)::FANCONE

is again a FANCONE, namely of the preimage of the domain of C.

PCOMPLEX

is a polyhedral complex, i.e., a non-empty finite collection of compatible polyhedra. Hence, they are to polyhedra what fans are to cones.
Note that a PCOMPLEX may contain an empty polhedron as sole element. Most functions for fans or polyhedra also apply to polyhedral complexes. The elements of a PCOMPLEX are polyhedra of the special type CELL.

pcomplex(x1::{POLYHEDRON, PCOMPLEX}, ...)::PCOMPLEX

is the equivalent of 'fan':

```
> PC := pcomplex(convhull(cube(3), ray([1, 1, 0])));
```

PC := PCOMPLEX(3, 3, 0, [6, 1], [0, 0, 0, 1])

```
> L := maximal(PC);
```

L := [CELL(3, 3, 0, [6, 1], 6)]

```
> type(L[1], POLYHEDRON);
```

true

```
> domain(L[1]);
```

PCOMPLEX(3, 3, 0, [6, 1], [0, 0, 0, 1])

```
> vertices(PC);
```

{[-1, -1, 1], [1, -1, 1], [-1, 1, 1], [-1, -1, -1], [1, -1, -1], [-1, 1, -1]}

```
> rays(PC);
```

{[1, 1, 0]}

```
> isbounded(PC);
```

false

```
> P := support(PC, [-1, 1, 0]);
```

P := CELL(3, 1, 2, 2)

```
> isbounded(P);
```

true

```
> domain(P);
```

PCOMPLEX(3, 3, 0, [6, 1], [0, 0, 0, 1])

```
> convert(P, POLYHEDRON);
```

POLYTOPE(3, 1, 2, 2)

lines(PC::PCOMPLEX)::list(ray)

One major difference is that the elements of a PCOMPLEX need not have the same lineality space in order to be compatible:

```
> P := cube(1) &x fullpolyhedron(1);
```

P := POLYHEDRON(2, 2, 1, [2, 0], [1])

```
> PC := pcomplex(P, image(P, [2, 0]));
  lines(PC);
```

PC := PCOMPLEX(2, 2, 1, [3, 0], [0, 0, 2])

[[0, 1]]

```
> PC := pcomplex(P, convhull([3, 4]));
  lines(PC);
```

PC := PCOMPLEX(2, 2, *FAIL*, [3, 0], [1, 0, 1])

FAIL

```
> PC := pcomplex(P, convhull([1, 4]));
Error, (in CONVEX[pcomplex]) arguments must be compatible
```

convert(F::FAN, affine)::PCOMPLEX
convert(F::FAN, affine, v::vec)::PCOMPLEX

convert the fan F to a polyhedral complex representing the same collection of cones, but now considered as polyhedra. The second form moves the origin to the point v.

```
> PC := convert(facefan(cube(2)), affine, [1, 3]);
```

PC := PCOMPLEX(2, 2, 0, [1, 4], [0, 0, 4])

```
> vertices(PC);
  rays(PC);
```

{[1, 3]}

{[-1, -1], [1, -1], [-1, 1], [1, 1]}

```
> iscomplete(PC);
```

true

voronoi(v1::vec, ..., vn::vec)::PCOMPLEX

gives the Voronoi complex associated to the point set {v1, ..., vn}. This is a decomposition of the whole space into subsets of points being closest to one of given points. The resulting PCOMPLEX is always complete:

```
> L := vertices(cube(3));
  PC := voronoi(op(L));
```

L := [[-1, -1, -1], [1, -1, -1], [-1, 1, -1], [1, 1, -1], [-1, -1, 1], [1, -1, 1], [-1, 1, 1], [1, 1, 1]]

PC := PCOMPLEX(3, 3, 0, [1, 6], [0, 0, 0, 8])

```
> isbounded(PC);
  iscomplete(PC);
```

false

true

delaunay(v1::vec, ..., vn::vec)::PCOMPLEX

gives the Delaunay "triangulation" corresponding to the point set {v1, ..., vn}.

```
> P := cube(3);
  PC := delaunay(seq(op(vertices(image(P, i))), i = 1..3));
```

$$P := \mathrm{POLYTOPE}(3, 3, 8, 6)$$

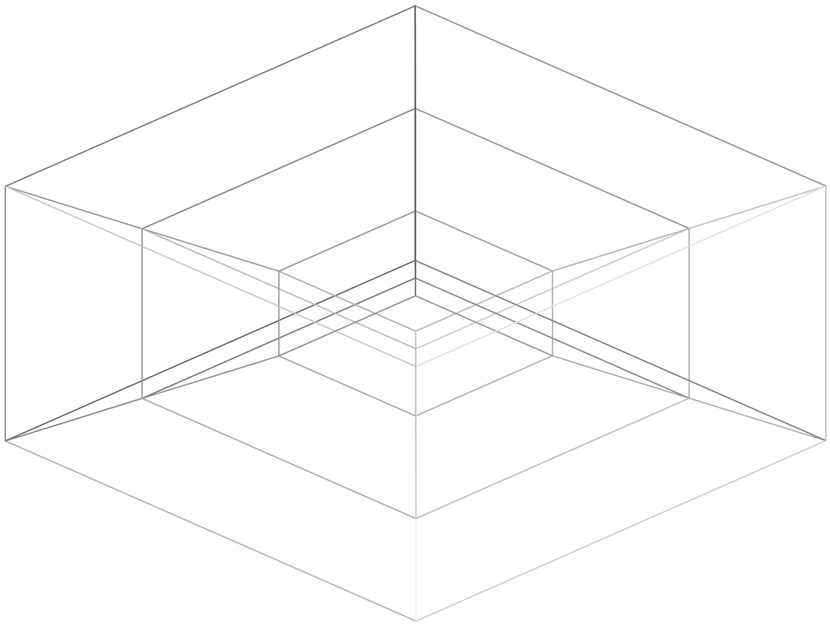
$$PC := \mathrm{PCOMPLEX}(3, 3, 24, [0, 0, 0, 13])$$

We will see in a minute what this looks like.

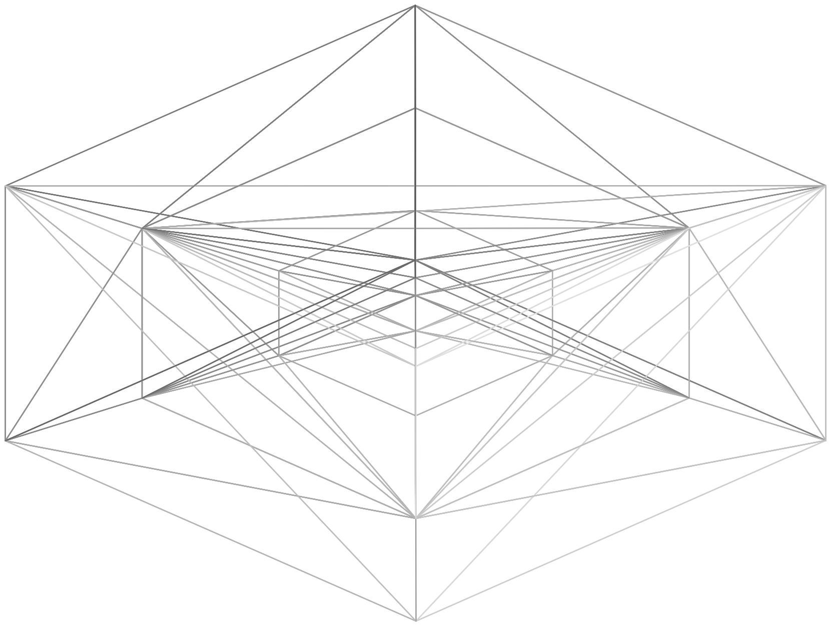
draw
plotdata

work as for polyhedra. One usually wants to give the plot option 'style = line'.

```
> draw(PC, style = line);
```



```
> draw(simplicialsubdiv(PC), style = line);
```



homology(PC::PCOMPLEX)::Array

computes the Betti numbers of PC (with respect to homology with closed support):

```
> PC := boundary(cube(3));
  homology(PC);
```

$$PC := \mathrm{PCOMPLEX}(3, 3, 8, [0, 0, 6, 0])$$

Array(0 .. 3, {(0) = 1, (2) = 1}, *datatype* = *anything*, *storage* = *rectangular*, *order* = *Fortran_order*)

```
> PC := pcomplex(PC, image(PC, [2, 0, 0]));
  homology(PC);
```

$$PC := \mathrm{PCOMPLEX}(3, 3, 12, [0, 0, 11, 0])$$

Array(0 .. 3, {(0) = 1, (2) = 2}, *datatype* = *anything*, *storage* = *rectangular*, *order* = *Fortran_order*)

CELL

Everything regarding FANCONE applies mutatis mutandis to CELL.

CFACE
PFACE

are the new names for 'face'. A face of a CONE has type 'CFACE', one of a polyhedron "PFACE'. The following applies equally to type PFACE.

rank(f::CFACE)::nonnegint
corank(f::CFACE)::nonnegint

are the rank and the corank of f in the face lattice of domain(f). By definition, the rank of the minimal face of a cone is 0, regardless of the dimension of the lineality space.

```
> C := poshull([1, 0, 0, 0], line([1, 0, -1, 0]));
```

C := CONE(4, 2, 1, 1, 1)

```
> f := minimal(C);
  dim(f), codim(f);
  rank(f), corank(f);
```

f := CFACE(0, 1)

1, 3

0, 1

```
> f := maximal(C);
  dim(f), codim(f);
  rank(f), corank(f);
```

f := CFACE(1, 0)

2, 2

1, 0

For polyhedra this is subtler:

```
> f := minimal(cube(2));
  dim(f), codim(f);
  rank(f), corank(f);
```

f := PFACE(0, 4)

-1, 3

0, 3

maximum(f1::CFACE, ...)::CFACE
minimum(f1::CFACE, ...)::CFACE

are the new names of 'maxface(f1, ...' and 'minface(f1, ...)'.

preimage(f::CFACE, A)::CFACE

If domain(f) is of type FANCONE, then so is the domain of the result:

```
> F := facefan(cube(1));
  C := support(F, [-1]);
  f := maximal(C);
```

F := FAN(1, 1, 0, 2, [2])

C := FANCONE(1, 1, 0, 1, 1)

f := CFACE(1, 0)

```
> A := [[1, 3]];
  f2 := preimage(f, A);
```

A := [[1, 3]]

$f2$:= CFACE(1, 0)

```
> C2 := domain(f2);
```

$C2$:= FANCONE(2, 2, 1, 1, 1)

```
> F2 := domain(C2);
```

$F2$:= FAN(2, 2, 1, 2, [0, 2])

```
> C2 &= preimage(C, A);
  F2 &= preimage(F, A);
```

true

true

convert(f::CFACE, CONE)::CONE

returns a FANCONE if the domain of f is so.

```
> C1 := convert(f, CONE);
```

$C1$:= FANCONE(1, 1, 0, 1, 1)

```
> evalb(domain(C1) = F);
```

true

isregular(f::CFACE)::boolean
isregular(f::CFACE, U::name)::boolean

is the same as isregular(convert(f, CONE)) and isregular(convert(f, CONE), U), respectively. This of course does not apply to PFACE.

Conversions

convert(C::CONE, affine)::POLYHEDRON1
convert(C::CONE, v::vec, affine)::POLYHEDRON1

convert a cone to a polyhedron representing the same point set (in the first case) or shifted by v (in the second). This was formerly achieved by the calls convert(C, affcone) or convert(C, affcone, v). The type 'affcone' does not exist any more.

convert(P::POLYHEDRON, CONE)::CONE

converts P to its homogenisation, by which it is internally represented.

```
> P := cube(2);
```

P := POLYTOPE(2, 2, 4, 4)

```
> C := convert(P, CONE);
```

C := CONE(3, 3, 0, 4, 4)

```
> rays(C);
```

[[-1, -1, 1], [1, -1, 1], [-1, 1, 1], [1, 1, 1]]

If P is a cell, then the domain will be converted, too, if this is possible (see below).

```
> PC := pcomplex(P);
  P2 := support(PC, [1, 0]);
  C2 := convert(P2, CONE);
```

PC := PCOMPLEX(2, 2, 4, [0, 0, 1])

$P2$:= CELL(2, 1, 2, 2)

$C2$:= FANCONE(3, 2, 0, 2, 2)

```
> domain(C2);
```

FAN(3, 3, 0, 4, [0, 0, 1])

```
> domain(C2) &= convert(PC, FAN);
```

true

convert(C::CONE, POLYHEDRON)::POLYHEDRON

converts C to a polyhedron in the ambient space of dimension one less. The ambient dimension of C must be at least 1. Since this involves intersection with the hyperplane at infinity, this operation is not injective:

```
> C := fullcone(2);
```

C := CONE(2, 2, 2, 0, 0)

$$[\,]$$

```
> P := convert(C, POLYHEDRON);
```

$$P := \mathrm{POLYHEDRON}(1, 1, 1, [1, 0], [0])$$

```
> P &= fullpolyhedron(1);
```

true

```
> C2 := convert(P, CONE);
```

$$C2 := \mathrm{CONE}(2, 2, 1, 1, 1)$$

```
> hspaces(C2);
```

$$[[0, 1]]$$

```
> C3 := convert(convert(C2, POLYHEDRON), CONE);
```

$$C3 := \mathrm{CONE}(2, 2, 1, 1, 1)$$

```
> C3 &= C2;
```

true

If C belongs to a fan, then the whole fan is converted, too (see below):

```
> F := fan(C);
  C1 := maximal(F)[1];
```

$$F := \mathrm{FAN}(2, 2, 2, 0, [0, 1])$$

$$C1 := \mathrm{FANCONE}(2, 2, 2, 0, 0)$$

```
> P1 := convert(C1, POLYHEDRON);
```

$$P1 := \mathrm{CELL}(1, 1, 1, [1, 0], [0])$$

```
> domain(P1);
```

$$\mathrm{PCOMPLEX}(1, 1, 1, [1, 0], [0, 1])$$

convert(PC::PCOMPLEX, FAN)::FAN

converts the polyhedral complex to a fan by converting all cells to fancones. This requires that the polyhedral complex has a well-defined lineality space:

```
> P := convhull([0, 0], line([1, 0]));
```

$$P := \mathrm{POLYHEDRON}(2, 1, 1, [1, 0], [0])$$

```
> PC := pcomplex(image(P, [0, 1]), image(P, [0, -1]));
  lines(PC);
```

$$PC := \mathrm{PCOMPLEX}(2, 2, 1, [2, 0], [0, 2, 0])$$

$$[[1, 0]]$$

```
> convert(PC, FAN);
```

$$\mathrm{FAN}(3, 3, 1, 2, [0, 2, 0])$$

```
> PC := pcomplex(PC, convhull([0, 0]));
  lines(PC);
```

$$PC := \mathrm{PCOMPLEX}(2, 2, \mathit{FAIL}, [3, 0], [1, 2, 0])$$

FAIL

```
> convert(PC, FAN);
Error, (in PCOMPLEX[`convert/FAN`]) argument has no lineality space
```

convert(F::FAN, PCOMPLEX)::PCOMPLEX

The ambient dimension of F must be at least 1.

convert(f::PFACE, CFACE)::CFACE

converts the face of a polyhedron P = domain(f) to the corresponding face of convert(P, CONE). If the domain of f is a cell, then the domain of the converted face is a fancone, if the polyhedral complex can be converted.

convert(f::CFACE, PFACE)::PFACE

onverts the face of a cone C = domain(f) to the face of the polyhedron convert(C, POLYHEDRON) which is cut out by the affine hyperplanes corresponding to the linear hyperplanes cutting out f. The result might be a smaller face than expected for two reasons:

The new face might by empty because the true "corresponding face" contains no affine point:

```
> C := posorthant(2);
```

$$C := \mathrm{CONE}(2, 2, 0, 2, 2)$$

```
> f1 := support(C, [0, 1]);
  f2 := support(C, [1, 0]);
```

$$f1 := \mathrm{CFACE}(1, 1)$$

$$f2 := \mathrm{CFACE}(1, 1)$$

```
> convert(f1, PFACE);
  convert(f2, PFACE);
```

$$\mathrm{PFACE}(1, 1)$$

$$\mathrm{PFACE}(0, 2)$$

```
> dim(f2);
```

$$-1$$

There might be no true "corresponding face" because the doain is intersected with the halfspace at infinity during conversion:

```
> C := poshull([1, -1]);
```

$$C := \mathrm{CONE}(2, 1, 0, 1, 1)$$

```
> f1 := minimal(C);
  f2 := maximal(C);
```

$$f1 := \mathrm{CFACE}(0, 1)$$

$$f2 := \mathrm{CFACE}(1, 0)$$

```
> convert(f1, PFACE);
  convert(f2, PFACE);
```

$$\mathrm{PFACE}(0, 0)$$

$$\mathrm{PFACE}(0, 0)$$

If the domain of f is a cell, then the domain of the converted face is a fancone if the polyhedral complex can be converted.

Known Bugs

- The Gauss elimination does currently not work for large matrices.
- ’issimplicial’ for fans is not implemented. In fact, I don’t know how to call functions like ’issimplicial’, ’issimplex’ etc. systematically.