The PL package

Version 2.7.0

February, 2015

Nurlan M. Sadikov
Alexey I. Korepanov
Igor G. Korepanov

Nurlan M. Sadikov Email: ancleCharly@gmail.com

Copyright

Contents

Chapter 1

Ball complexes

Package PL deals with piecewise-linear manifolds. They are described combinatorially as *ball complexes*. A ball complex is, simply speaking, such kind of a cell complex where all *closed* cells (= balls) are *embedded*. In particular, their boundaries are genuine spheres, not crumpled/folded. The formal definition of PL ball complex reads: A PL ball complex is a pair (X,U), where X is a compact Euclidean polyhedron and U is a covering of X by closed PL-balls such that the following axioms are satisfied:

- the relative interiors of balls from U form a partition of X,
- the boundary of each ball from U is a union of balls from U.

We also call PL ball complexes "polytopes", for brevity, hence prefix "Pol" in the names of some of our functions.

1.1 Representation of a PL ball complex

A PL ball complex is determined up to a PL homeomorphism only by the combinatorics of adjunctions of its balls. Due to this, we represent them combinatorially in the following way. First, we assume that all vertices in the complex are numbered from 1 to their total number N_0. Hence, in this sense, the 0-skeleton of the complex is described. Next, assuming that the k-skeleton is already given, which implies (in particular) the numeration of all k-cells, we describe the $(k+1)$-skeleton as the list of all $(k+1)$-cells, each of which, in its turn, is the set of numbers of k-cells in its boundary. Then we compose the list of length n, where n is the dimension of the complex, whose elements are lists of 1-,..., n-cells. Thus, a three-dimensional ball B^3 may be represented by the following PL ball complex with two vertices 1 and 2:

Example

```
[
  [ [1,2], [1,2] ], # two one-dimensional simplexes, each with
                    # ends 1 and 2, of which the first is referred to
                    # in the next line as 1, the second - as 2;
  [ [1,2], [1,2] ], # two disks - bigons - bounded each by
                    # one-dimensional simplexes 1 and 2;
  [ [1,2] ]         # the three-ball bounded by bigons 1 and 2
]
```

Actually, we add a list of vertices with their names, or numbers, or something like that in the beginning of the above ball complex representation. For instance, our function `ballAB`(n) calls them `"A"` and `"B"`. So, our GAP representation of the mentioned ball in is the following record:

Example

```
gap> ballAB(3);
rec( vertices := [ "A", "B" ],
  faces := [ [ [ 1, 2 ], [ 1, 2 ] ], [ [ 1, 2 ], [ 1, 2 ] ], [ [ 1, 2 ] ] ]
 )
```

Also, other useful information may be added in this record, describing, for instance, the *symmetries* of the polytope.

1.1.1 IsPolytope

▷ `IsPolytope(`*pol*`)` (function)

This function checks that *pol* satisfies the formal rules describing how our polytopes must be built. It returns `true` if no error has been found, and `false` otherwise.

The formal rules are as follows:

- the presence of fields `.vertices` and `.faces`,
- the absence of dead links: each cell is a list of actually existing cells making its boundary,
- also, the Euler characteristic of each cell is checked.

In the following example, `T2` is a two-dimensional torus.

Example

```
gap> IsPolytope(T2);
true
```

As there is no algorithm for recognizing an n-sphere (at least, for an arbitrary n), the verification of whether each cell is indeed a sphere has been dropped. This of course may lead to a mistake in the case where the data satisfies the formal rules except that some cells are not spheres: the function will still return `true` in this case.

1.2 Classical invariants

Here are functions computing some classical manifold invariants, including the fundamental group.

1.2.1 EulerNumber

▷ `EulerNumber(`*pol*`)` (function)

Computes the Euler number for a ball complex. This function is polymorphic and can accept as its input data in the polytope format (`IsPolytope`), or a named list whose elements are the numbers of cells of all dimensions, as is returned by the function `LengthPol`.

Example

```
gap> EulerNumber(T2);
0
gap> EulerNumber(sphereAB(4));
2
gap> EulerNumber(sphereAB(3));
0
```

1.2.2 FundGroup

▷ `FundGroup(pol)` (function)

Computes a corepresentation of the fundamental group of a given polytope. Uses GAP's algorithm to simplify it.

1.3 General information about a ball complex

1.3.1 LengthPol

▷ `LengthPol(pol)` (function)

Returns the following record: `LengthPol.d` is the cardinality of the d-skeleton in polytope *pol*.

Example

```
gap> LengthPol(T2);
total16
rec( 0 := 4, 1 := 8, 2 := 4 )
```

1.3.2 PolBoundary

▷ `PolBoundary(pol)` (function)

Computes the boundary of polytope *pol*. Returns the list of boundary $(n-1)$-cells (n is the dimension of *pol*.

Example

```
gap> PolBoundary(T2);
[ ]
gap> s1:=sphereAB(1);;
gap> d2:=ballAB(2);;
gap> ft2:=PolProduct(d2,s1);;
gap> PolBoundary(ft2);
[ 1, 2, 3, 4 ]
```

Here `T2` is a 2-torus, `s1` - a 1-sphere, `d2` - a 2-ball, and `ft2` - a solid torus obtained as the Cartesian product of `d2` and `s1`.

1.3.3 PolInnerFaces

▷ PolInnerFaces(*pol*) (function)

Builds the list whose i-th entry is the list of inner faces of polytope *pol* of dimension $(i-1)$. Here i ranges from 1 to the polytope dimension. Any face is outer if it has at most one adjacent face of a higher dimension, or if it lies in the boundary of an outer face. Inner faces are, of course, the faces that are not outer.

Example

```
gap> PolInnerFaces(T2);
[ [ 1 .. 4 ], [ 1 .. 8 ] ]
gap> PolInnerFaces(ft2);
[ [  ], [  ], [ 5, 6 ] ]
gap> PolInnerFaces(ballAB(5));
[ [  ], [  ], [  ], [  ], [  ] ]
```

1.3.4 MaxTree

▷ MaxTree(*pol*) (function)

finds a maximal tree in the 1-skeleton of a polytope as a list of edges.

Let cells a and b have dimensions n and $n-1$, respectively, and let b lie in the boundary of a. We introduce incidence numbers, or *relative orientations* $\varepsilon(b|a) = \pm 1$, satisfying $\varepsilon(b_1|a)\varepsilon(c|b_1) = -\varepsilon(b_2|a)\varepsilon(c|b_2)$, where cell c is of dimension $n-2$ and $b_1, b_2 \subset a$, $c \subset b_1 \cap b_2$. If a is an edge, then the relative orientation is negative for its vertex with the smaller number, and positive for its vertex with the bigger number.

1.3.5 CellOrient

▷ CellOrient(*pol*) (function)

Computes inductively relative orientations for cells of dimensions $1..n = dim(pol)$ of polytope *pol*.

Example

```
gap> b1 := ballAB(1);
rec( faces := [ [ [ 1, 2 ] ] ], vertices := [ "A", "B" ] )
gap> square := PolProduct(b1,b1);
rec(
  faces := [ [ [ 1, 2 ], [ 3, 4 ], [ 1, 3 ], [ 2, 4 ] ], [ [ 1, 2, 3, 4 ] ] ],
  vertices := [ [ "A", "A" ], [ "A", "B" ], [ "B", "A" ], [ "B", "B" ] ] )
gap> CellOrient(square);
[ [ [ -1, 1 ], [ -1, 1 ], [ -1, 1 ], [ -1, 1 ] ], [ [ -1, 1, 1, -1 ] ] ]
```

1.3.6 PolOrient

▷ PolOrient(*pol*) (function)

If *pol* is orientable, gives a consistent orientation of n-faces ($n = \dim(pol)$), otherwise returns $fail$.

1.3.7 OrientTriangulated

▷ `OrientTriangulated(pol)` (function)

Function computes a consistent orientation for the simplices of the greatest dimension of a given *triangulated* complex *pol*. Returns array of $-1, 1$-s corresponding to the orientation of simplices of greatest dimension of *pol*.

1.3.8 dataPachner

▷ `dataPachner(dim, k)` (function)

Returns the description of the Pachner move in dimension *dim*, whose initial configuration, or *left-hand side* (l.h.s.), consists of k simplices (hence, the final configuration, or *right-hand side* (r.h.s.), consists of $dim - k + 2$ simplices). The description is returned in the form of a record, where `.l` describes the polytope (cluster of simplices) in the l.h.s., while `.r` describes the polytope in the r.h.s.

Example

```
gap> Print(dataPachner(3,2));
rec(
  l := rec(
      pol := rec(
          faces :=
           [
              [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ], [ 1, 4 ], [ 2, 4 ], [ 3, 4 ],
                  [ 1, 5 ], [ 2, 5 ], [ 3, 5 ] ],
              [ [ 1, 4, 5 ], [ 1, 7, 8 ], [ 2, 4, 6 ], [ 2, 7, 9 ],
                  [ 3, 5, 6 ], [ 3, 8, 9 ], [ 1, 2, 3 ] ],
              [ [ 1, 3, 5, 7 ], [ 2, 4, 6, 7 ] ] ],
          vertices := [ 1, 2, 3, 4, 5 ] ),
      sim := [ [ 1, 2, 3, 4 ], [ 1, 2, 3, 5 ] ],
      vnut := [ 7 ] ),
  r := rec(
      pol := rec(
          faces :=
           [
              [ [ 1, 2 ], [ 1, 4 ], [ 2, 4 ], [ 1, 5 ], [ 2, 5 ], [ 4, 5 ],
                  [ 1, 3 ], [ 3, 4 ], [ 3, 5 ], [ 2, 3 ] ],
              [ [ 1, 2, 3 ], [ 1, 4, 5 ], [ 2, 7, 8 ], [ 4, 7, 9 ],
                  [ 3, 8, 10 ], [ 5, 9, 10 ], [ 2, 4, 6 ], [ 3, 5, 6 ],
                  [ 6, 8, 9 ] ],
              [ [ 1, 2, 7, 8 ], [ 3, 4, 7, 9 ], [ 5, 6, 8, 9 ] ] ],
          vertices := [ 1, 2, 3, 4, 5 ] ),
      sim := [ [ 1, 2, 4, 5 ], [ 1, 3, 4, 5 ], [ 2, 3, 4, 5 ] ],
      vnut := [ 7, 8, 9 ] ) )
```

Both `.l` and `.r` are, in their turn, also records. Each of them contains an entry `.pol` representing the corresponding polytope in the form of a triangulated ball complex, an entry `.sim` representing the

same polytope as a list of *dim*-simplices each given *as the list of its vertices*, and an entry `.vnut` - the list of the (numbers of the) *inner* $(n-1)$-cells.

Simplicial complex are often understood in a restricted sense, namely, it is requried that no two simplices have the same sets of vertices. In this case, a simplicial complex can be given in the form of a list of simplices, each of those specified by the list of its vertices. For an n-dimensional manifold with boundary, it is enough to list just its n-simplices. For instance, the boundary of 4-simplex 12345 can be written as

Example

```
gap> sim := [ [ 1, 2, 3, 4 ], [ 1, 2, 3, 5 ], [ 1, 2, 4, 5 ],
                                [ 1, 3, 4, 5 ], [ 2, 3, 4, 5 ] ];
```

1.3.9 FromSimplexToPolytope

▷ `FromSimplexToPolytope(`*sim*`)` (function)

turns a list *sim* of simplices into a ball complex.

Example

```
gap> FromSimplexToPolytope(sim);
rec(
  faces :=
    [ [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ], [ 1, 4 ], [ 2, 4 ], [ 3, 4 ], [ 1, 5 ],
          [ 2, 5 ], [ 3, 5 ], [ 4, 5 ] ],
      [ [ 1, 2, 3 ], [ 1, 4, 5 ], [ 2, 4, 6 ], [ 3, 5, 6 ], [ 1, 7, 8 ],
          [ 2, 7, 9 ], [ 3, 8, 9 ], [ 4, 7, 10 ], [ 5, 8, 10 ], [ 6, 9, 10 ] ]
        , [ [ 1, 2, 3, 4 ], [ 1, 5, 6, 7 ], [ 2, 5, 8, 9 ], [ 3, 6, 8, 10 ],
          [ 4, 7, 9, 10 ] ] ], vertices := [ 1, 2, 3, 4, 5 ] )
```

1.4 Working with cells

In this Section, we describe the functions providing information about a cell in a triangulation of a PL manifold. The cell is determined by the pair of numbers $[dim, ind]$, where *dim* is its dimension, and *ind* - its position in the list `pol.faces[`*dim*`]`. Such a pair will be called below the *address* of the cell, or simply "cell", and denoted $adr = [dim, ind]$.

1.4.1 PolBnd

▷ `PolBnd(`*pol, adr*`)` (function)

Creating an index of boundary faces of face $adr = [dim, ind]$ of complex *pol* in the form of a list. Its *i*'th entry is the list of $(i-1)$-dimensional faces of *pol* which are in the boundary of *adr*.

Example

```
gap> PolBnd(T2,[2,3]);
[ [ 1, 2, 3, 4 ], [ 2, 4, 5, 7 ] ]
```

1.4.2 FaceComp

▷ `FaceComp(pol, adr)` (function)

A function similar to `PolBnd`, but the output is in the form of a record with entries *.d*, where *d* is the cell dimension. Also, the input cell is included.

Example

```
gap> FaceComp(T2,[2,3]);
rec( 0 := [ 1, 2, 3, 4 ], 1 := [ 2, 4, 5, 7 ], 2 := [ 3 ] )
```

1.4.3 StarFace

▷ `StarFace(pol, adr)` (function)

Computes the star of cell *adr*, that is, all cells of higher dimensions containing *adr*, in the ball complex *pol*, in the form of a record. The entry *.i* contains the numbers of *i*-cells contained in the star.

Example

```
gap> StarFace(T2,[0,3]);
rec( 1 := [ 2, 3, 6, 7 ], 2 := [ 1, 2, 3, 4 ] )
```

In this example, 1-cells indexed 2, 3, 6 and 7, and 2-cells indexed 1, 2, 3 and 4, form the star of vertex no. 3 of the ball complex `T2`.

1.4.4 PolCheckComb

▷ `PolCheckComb(pol, adr)` (function)

Checks whether face *adr* of ball complex *pol* is "strictly combinatorial", that is, whether every its subface of lower dimension is uniquely determined by its vertices (and dimension).

1.5 Changing the ball complex representation of the same manifold

1.5.1 PolTriangulate

▷ `PolTriangulate(pol)` (function)

Triangulates polytope *pol*. Triangulation is understood in a broader sense, as a ball complex made only of simplices, but a simplex may *not* be determined by its boundary (not to say its vertices).

Example

```
gap> PolTriangulate(sphereAB(2));
rec(
  faces := [ [ [ 1, 2 ], [ 1, 2 ], [ 1, 3 ], [ 2, 3 ], [ 1, 4 ], [ 2, 4 ] ],
      [ [ 1, 3, 4 ], [ 2, 3, 4 ], [ 1, 5, 6 ], [ 2, 5, 6 ] ] ],
  vertices := [ "A", "B", "V1", "V2" ] )
```

1.5.2 PolCanonicalOrder

▷ `PolCanonicalOrder(`*pol*`)` (function)

Produces the canonical order of cells in a triangulated polytope. The canonical order is the lexicographical order of cells when these are represented by their vertices.

1.5.3 FirstBoundary

▷ `FirstBoundary(`*pol*`)` (function)

Changes the order of polytope *pol* cells in such way that the boundary cells go first in the lists *pol*`.faces[`*dim*`]`. The order of cells within the boundary remains unchanged.

1.5.4 PermFaces

▷ `PermFaces(`*pol*`, `*perm*`, `*dim*`)` (function)

Re-orders of *dim*-cells in polytope *pol* according to permutation *perm*. Important note: works correctly on *pol*`.vertices` ($dim = 0$) and *pol*`.faces`, but *ignores* additional entries (if any), such as symmetries (*pol*`.syms`).

We call ad-cell in a ball complex *minimal* if its boundary consists of only two $(d-1)$-cells.

1.5.5 ContractMiniFace

▷ `ContractMiniFace(`*pol*`, `*adr*`)` (function)

Collapses minimal cell *adr* in polytope *pol*. That is, withdraws *adr* from the complex, and its two boundary cells (of dimension less by 1 than *adr*) are, accordingly, glued together into one cell.

Example

```
gap> s1:=sphereTriangul(1); # triangulated sphere of dimension 1
rec( faces := [ [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ] ], vertices := [ 1 .. 3 ] )
gap> s1:=ContractMiniFace(s1,[1,1]);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ] ] ], vertices := [ 1, 3 ] )
```

In some cases, the result of collapsing a minimal cell may *not* be a ball complex. This function does not check this, this is left to the user! For instance, if we try to collapse one more edge in the above circle $s1$, we get

Example

```
gap> s1:=ContractMiniFace(s1,[1,1]);
rec( faces := [ [ [ 1 ] ] ], vertices := [ 1 ] )
gap> IsPolytope(s1);
false
```

1.5.6 DivideFace

▷ `DivideFace(pol, adr, set)` (function)

Let there be a cell *adr* of dimension $d = adr[1]$ in ball complex *pol*, and let *set* be a set (in the GAP sense) of cells that lie in the boundary of cell *adr*, and form together a $(d-2)$-sphere S^{d-2}.

The function breaks the cell *adr* into two parts by spanning a disk D^{d-1} over S^{d-2} and inside cell *adr*.

Warning: the function does *not* check whether *set* really makes a sphere in the boundary of *adr*.

Example

```
gap> octahedron;
rec(
  faces :=
        [ [ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 1, 4 ], [ 1, 5 ], [ 2, 5 ], [ 3, 5 ],
                  [ 4, 5 ], [ 1, 6 ], [ 2, 6 ], [ 3, 6 ], [ 4, 6 ] ],
          [ [ 1, 5, 6 ], [ 2, 6, 7 ], [ 3, 7, 8 ], [ 4, 5, 8 ], [ 1, 9, 10 ],
                  [ 2, 10, 11 ], [ 3, 11, 12 ], [ 4, 9, 12 ] ], [ [ 1 .. 8 ] ] ],
        vertices := [ 1 .. 6 ] )
gap> DivideFace(octahedron,[3,1],[1,2,3,4]);
rec(
  faces :=
         [ [ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 1, 4 ], [ 1, 5 ], [ 2, 5 ], [ 3, 5 ],
                  [ 4, 5 ], [ 1, 6 ], [ 2, 6 ], [ 3, 6 ], [ 4, 6 ] ],
          [ [ 1, 5, 6 ], [ 2, 6, 7 ], [ 3, 7, 8 ], [ 4, 5, 8 ], [ 1, 9, 10 ],
            [ 2, 10, 11 ], [ 3, 11, 12 ], [ 4, 9, 12 ], [ 1, 2, 3, 4 ] ],
          [ [ 1, 4, 2, 3, 9 ], [ 5, 6, 7, 8, 9 ] ] ],
               vertices := [ 1 .. 6 ] )
```

There is also a special case where a 1-cell *adr* is divided, by adding a vertex in its middle. In this case, the name of this new vertex is indicated instead of the set *set*.

Example

```
gap> s1:=sphereAB(1);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ] ] ], vertices := [ "A", "B" ] )
gap> DivideFace(s1,[1,1],3);
rec( faces := [ [ [ 1, 3 ], [ 1, 2 ], [ 2, 3 ] ] ], vertices := [ "A", "B", 3 ] )
```

1.5.7 UnionFaces

▷ `UnionFaces(pol, cell1, cell2)` (function)

The input is ball compex *pol* and two cells *cell*1 and *cell*2 of the same dimension, whose intersection - the *partition* between them - is just one (closed) cell of dimension less by one. The function unites these cells into one and deletes the partition cell. The details are as follows.

First, we assume that *pol* is a correctly defined ball complex. Then, the function checks that the intersection of *cell*1 and *cell*2 is indeed exactly one closed cell, as described above. This implies, of course, the checking of *all* cells of *all* dimensions less than that of *cell*1 and *cell*2.

Moreover, one more check is performed within the *star* (see `StarFace`) of the partition being deleted: the absence of dead links after the deletion. Namely, the partition cell must not enter in the boundary of any *other* cell of the same dimension as *cell*1 and *cell*2.

If the result of any of these checks is negative, then the function returns the *initial* polytope. ****What is "p3" below?****

Example

```
gap> p3 := dataPachner(3,3).l.pol; # this gives a cluster of 3 tetrahedra
                                   # around an edge, as in the l.h.s.
                                   # of Pachner move 3-2
rec(
  faces :=
    [ [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ], [ 1, 4 ], [ 2, 4 ], [ 3, 4 ], [ 1, 5 ],
          [ 2, 5 ], [ 3, 5 ], [ 4, 5 ] ],
      [ [ 2, 4, 6 ], [ 2, 7, 9 ], [ 4, 7, 10 ], [ 3, 5, 6 ], [ 3, 8, 9 ],
          [ 5, 8, 10 ], [ 1, 2, 3 ], [ 1, 4, 5 ], [ 1, 7, 8 ] ],
      [ [ 1, 4, 7, 8 ], [ 2, 5, 7, 9 ], [ 3, 6, 8, 9 ] ] ],
  vertices := [ 1, 2, 3, 4, 5 ] )
gap> IsPolytope(p3);
true
gap> p3 := UnionFaces(p3,[3,1],[3,2]); # in the new p3, the first two
                                       # tetrahedra are united into one cell
rec(
  faces :=
    [ [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ], [ 1, 4 ], [ 2, 4 ], [ 3, 4 ], [ 1, 5 ],
          [ 2, 5 ], [ 3, 5 ], [ 4, 5 ] ],
      [ [ 2, 4, 6 ], [ 2, 7, 9 ], [ 4, 7, 10 ], [ 3, 5, 6 ], [ 3, 8, 9 ],
          [ 5, 8, 10 ], [ 1, 4, 5 ], [ 1, 7, 8 ] ],
      [ [ 1, 2, 4, 5, 7, 8 ], [ 3, 6, 7, 8 ] ] ],
  vertices := [ 1, 2, 3, 4, 5 ] )
gap> IsPolytope(p3);
true
```

Example

```
gap> UnionFaces(T2,[2,1],[2,3]) = T2;   # T2 is a 2-torus made of four squares
true
```

1.5.8 PolSimplify

▷ `PolSimplify(`*pol*`)` (function)

Simplifies polytope *pol* using `UnionFaces`. The function searches through all possibilities, starting from the cells of the maximal dimension. Note, however, that, after applying this function, new possibilities may occur. So, the function can be applied several times, in order to simplify the polytope as far as possible.

Example

```
gap> a:=ballTriangul(3); # triangulated 3-ball (tetrahedron)
rec(
  faces := [ [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ],
      [ [ 1, 2, 4 ], [ 1, 3, 5 ], [ 2, 3, 6 ], [ 4, 5, 6 ] ], [ [ 1 .. 4 ] ] ]
    , vertices := [ 1 .. 4 ] )
gap> PolSimplify(a);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ] ], [ [ 1, 2 ], [ 1, 2 ] ], [ [ 1, 2 ] ] ]
```

```
    , vertices := [ 1, 2 ] )
```

1.6 Topological operations

1.6.1 FreeUnionPol

▷ `FreeUnionPol(`*pol1, pol2*`)` (function)

Free union of polytopes

Example

```
gap> FreeUnionPol(s1,s1);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ], [ 3, 4 ], [ 3, 4 ] ] ],
  vertices := [ [ 1, "A" ], [ 1, "B" ], [ 2, "A" ], [ 2, "B" ] ] )
```

The free union is performed by concatenating the lists of faces. The indices (numbers) of faces belonging to the second polytope are shifted - increased by the number of corresponding faces in the first polytope.

1.6.2 PolDoubleCone

▷ `PolDoubleCone(`*pol*`)` (function)

Makes a double cone with vertices V1 and V2 over the given polytope *pol*.

Example

```
gap> PolDoubleCone(s1);
rec(
  faces := [ [ [ 1, 2 ], [ 1, 2 ], [ 1, 3 ], [ 2, 3 ], [ 1, 4 ], [ 2, 4 ] ],
      [ [ 1, 3, 4 ], [ 2, 3, 4 ], [ 1, 5, 6 ], [ 2, 5, 6 ] ] ],
  vertices := [ "A", "B", "V1", "V2" ] )
```

1.6.3 ConnectedSum

▷ `ConnectedSum(`*N, M*`)` (function)

Makes a connected sum of two polytopes *N* and *M* of the same dimension.

Currently, the function does not take care of orientations. An improved function is expected to appear in this package, where the result will depend on the orientations prescribed by the user.

Example

```
gap> s2 := sphereAB(2);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ] ], [ [ 1, 2 ], [ 1, 2 ] ] ],
  vertices := [ "A", "B" ] )
gap> s2s2 := ConnectedSum(s2,s2);
rec(
  faces := [ [ [ 1, 2 ], [ 1, 2 ], [ 1, 2 ], [ 1, 2 ] ],
      [ [ 1, 3 ], [ 1, 2 ], [ 3, 4 ], [ 2, 4 ] ] ],
  vertices := [ [ 1, "A" ], [ 1, "B" ] ] )
```

```
gap> s2s2 := PolSimplify(s2s2);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ] ], [ [ 1, 2 ], [ 1, 2 ] ] ],
  vertices := [ [ 1, "A" ], [ 1, "B" ] ] )
```

We see also that here `PolSimplify` returns essentially the same `s2` from which we started.

1.6.4 PolProduct

▷ `PolProduct(`*pol1, pol2*`)` (function)

Calculates the Cartesian product of two ball complexes *pol*1 and *pol*2. This is the ball complex made of all balls $D_i^s \times D_j^t$, where D_i^s and D_j^t are cells in M and N, respectively.

Example

```
gap> PolProduct(s1,s1);
rec(
  faces :=
    [ [ [ 1, 2 ], [ 1, 2 ], [ 3, 4 ], [ 3, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ],
          [ 2, 4 ] ],
      [ [ 1, 3, 5, 6 ], [ 2, 4, 5, 6 ], [ 1, 3, 7, 8 ], [ 2, 4, 7, 8 ] ] ],
  vertices := [ [ "A", "A" ], [ "A", "B" ], [ "B", "A" ], [ "B", "B" ] ] )
```

1.6.5 ImageInPolProduct

▷ `ImageInPolProduct(`*pol1, pol2, cells*`)` (function)

Auxiliary function for calculating the Cartesian product of polytopes *pol*1 and *pol*2. The last argument *cells* is the list of two cells *kl*1 and *kl*2 belonging to these two respective polytopes. Computes the address of cell $kl1 \times kl2$ in the mentioned Cartesian product.

Example

```
gap> s1:=sphereAB(1);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ] ] ], vertices := [ "A", "B" ] )
gap> ImageInPolProduct(s1,s1,[[1,2],[1,1]]);
[ 2, 3 ]
```

One more possibility for the input is named lists instead of polytopes *pol*1 and *pol*2. In these, to each dimension k corresponds the cardinality of the set of k-cells. This possibility helps if the function is frequently called for one and the same Cartesian product.

Example

```
gap> ls1:=rec(0:=2, 1:=2);
rec( 0 := 2, 1 := 2 )
gap> ImageInPolProduct(ls1,ls1,[[1,2],[1,1]]);
[ 2, 3 ]
```

Note that the function is valid only for the complex obtained by `PolProduct`. If the complex has changed somehow, the address may become incorrect.

1.6.6 PreimageInPolProduct

▷ `PreimageInPolProduct(`*pol1, pol2, imageface*`)` (function)

For a given cell *imageface* in the Cartesian product of polytopes *pol*1 and *pol*2, returns the list of two cells whose Cartesian product is *imageface*. The Cartesian product is of course made according to `PolProduct`.

1.6.7 PolProductSyms

▷ `PolProductSyms()` (function)

Cartesian product of two polytopes with symmetries of multipliers transferred to it. First go the symmetries of the first multiplier, then - the second.

1.6.8 PolProductSymsDict

▷ `PolProductSymsDict()` (function)

Cartesian product of two polytopes with symmetries of multipliers transferred to it. First go the symmetries of the first multiplier, then - the second. Also returns the face dictionary.

1.6.9 PolFactorInvolution

▷ `PolFactorInvolution(`*pol, invol*`)` (function)

pol is a polytope with symmetries, *invol* is such a list of some of its symmetries (repetitions possible) that it is known that the product of symmetries in *invol* is an involution. Returns the factored polytope.

The function is used in `KummerSurface()`.

1.7 Main principles of ball complex re-buildings

Here, we expalin the main principles of ball complex re-buildings that are related to the change of cell indexation (numbers of cells in the lists). Change of cell indexation is one of the main difficulties for writing our programs in a fast and easy way. In the second version of package PL, we used the following idea. As far as it is feasible, the addresses of cells not taking part in an operation must remain unchanged. This is, however, not always possible. For instance, this is impossible if a cell is removed from a polytope. If shifting indices cannot be avoided, then we must try to change the minimal amount of indices. The fact that all the polytope construction is based upon the cell numbers in the lists *pol*`.faces[`*k*`]`, complicates the work related to complex re-buildings. Starting from the third version of package PL, a possibility will be implemented for a rigid cell indexation, where the position of a cell in *pol*`.faces[`*k*`]` will be also the name of this cell. This will mean, in particular, that empty entries may appear in the lists *pol*`.faces[`*k*`]`. At this moment, there are the following functions that permit to change the polytope structure and attached information.

1.7.1 DelFace

▷ `DelFace(pol, adr)` (function)

Cell *adr* is removed from polytope *pol*. Note that the result may is not guaranteed to be a correct polytope:

Example

```
gap> s1 := sphereAB(1);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ] ] ], vertices := [ "A", "B" ] )
gap> t2 := PolProduct(s1,s1);
rec(
  faces :=
    [ [ [ 1, 2 ], [ 1, 2 ], [ 3, 4 ], [ 3, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ],
          [ 2, 4 ] ],
      [ [ 1, 3, 5, 6 ], [ 2, 4, 5, 6 ], [ 1, 3, 7, 8 ], [ 2, 4, 7, 8 ] ] ],
  vertices := [ [ "A", "A" ], [ "A", "B" ], [ "B", "A" ], [ "B", "B" ] ] )
gap> pol := DelFace(t2,[1,2]);
rec(
  faces :=
    [ [ [ 1, 2 ], [ 3, 4 ], [ 3, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ], [ 2, 4 ] ]
      , [ [ 1, 2, 4, 5 ], [ 3, 4, 5 ], [ 1, 2, 6, 7 ], [ 3, 6, 7 ] ] ],
  vertices := [ [ "A", "A" ], [ "A", "B" ], [ "B", "A" ], [ "B", "B" ] ] )
gap> IsPolytope(pol);
false
```

This function contains the most frequently used code that permits to remove all mentionings of the given cell and prepare the data for the further work of the algorithm. Namely, it does the following:

- the cell $adr = [d,k]$ is removed,
- references to this cell are removed,
- the indexation of d-cells is changed,
- accordingly, the references to d-cells in $(d+1)$-cells are changed.

1.7.2 wasDelFace

▷ `wasDelFace(pol, adr)` (function)

Corrects the information attached to polytope *pol* that may have changed after removing cell *adr*. The indices of the cells of the same dimension as *adr* and that go after *adr* must be decreased by 1.

Currently, this function works only for the information defining a 2-knot in a polytope. Namely, in the case if a 2-cell is removed, the relevant message will be displayed, the index of the 2-cell will be removed from the list `.2knot.sheets` and, if there is a reference to this 2-cell in `.2knot.dpoints.(1kl)`, then the corresponding position will be cleared.

In order to simplify polytope re-buildings, we will stick to the following principle in the following releases of PL. The index of a k-cell in the list `pol.faces[k]` is simultaneously its identifier. This will allow us to create lists (skeletons) `pol.faces[k]` with empty positions. If a cell is removed from a skeleton, the corresponding position in `pol.faces[k]` is emptied. This will alow us to simplify

the calculations related to the re-enumerating of cells. This principle will be supported starting from the third version of Package PL. Starting from the fourth version, *only* this way of re-building will be supported.

1.8 The library of polytopes

Here we list some ball complex realizations of "standard" manifolds. There are the following functions for the disks and spheres of a given dimension.

1.8.1 ballAB

▷ `ballAB(`*dim*`)` (function)

Creates the minimal ball complex decomposition of a *dim*-ball, with just two vertices "A" and "B".

Example

```
gap> ballAB(3);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ] ], [ [ 1, 2 ], [ 1, 2 ] ], [ [ 1, 2 ] ] ]
    , vertices := [ "A", "B" ] )
```

1.8.2 sphereAB

▷ `sphereAB(`*dim*`)` (function)

Creates the minimal ball complex decomposition of a *dim*-sphere, with just two vertices "A" and "B".

Example

```
gap> sphereAB(3);
rec(
faces := [ [ [ 1, 2 ], [ 1, 2 ] ], [ [ 1, 2 ], [ 1, 2 ] ],
  [ [ 1, 2 ], [ 1, 2 ] ] ], vertices := [ "A", "B" ] )
```

There are also *triangulated* balls and spheres.

1.8.3 ballTriangul

▷ `ballTriangul(`*dim*`)` (function)

Creates a simplex of dimension *dim*.

Example

```
gap> ballTriangul(2);
rec( faces := [ [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ], [ [ 1 .. 3 ] ] ],
vertices := [ 1 .. 3 ] )
```

1.8.4 sphereTriangul

▷ `sphereTriangul(dim)` (function)

Creates a triangulated sphere in the form of the boundary of a $(dim+1)$simplex.

Example

```
gap> sphereTriangul(1);
rec( faces := [ [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ] ], vertices := [ 1 .. 3 ] )
```

1.8.5 projectivePlane

▷ `projectivePlane(dim)` (function)

Real projective space of dimension *dim*

1.8.6 s2s2twisted

▷ `s2s2twisted` (global variable)

Twisted product of two spheres S^2

1.8.7 cp2

▷ `cp2` (global variable)

Complex projective plane

1.8.8 PoincareSphere

▷ `PoincareSphere` (global variable)

Poincare sphere

1.8.9 TorTwist

▷ `TorTwist(n)` (function)

Twisted 3-torus - the torus bundle over S^1, with the monodromy matrix `[ [ 1, -n ], [ 0, 1 ]` The algorithm works for *natural* $n = 1,2,3,\ldots$.

1.8.10 KummerSurface

▷ `KummerSurface()` (function)

Creates the Kummer surface, with singularities resolved - i.e., a true 4-manifold.

1.8.11 Lens

▷ `Lens(n, k)` (function)

Three-dimensional lens space $L(n,k)$.

Chapter 2

Subpolytopes and embeddings

Subpolytope *subpol* will be understood as a set of cells of a given dimension k if the union of these cells (cells are understood as *closed*) forms a PL submanifold $N \subset M$. The simplest example of a subpolytope is a disk represented as the list of just one element of any dimension of a polytope *pol*. We will not, however, define subpolytope as any rigid construction. It will be clear from the context what we mean each time.

2.1 Subpolytopes, boundaries, gluing, removing neighborhoods...

2.1.1 SetOfFacesBoundary

▷ `SetOfFacesBoundary(pol, subpol, dim)` (function)

Returns the boundary of subpolytope *subpol* of dimension *dim* of polytope *pol*. To be exact, *subpol* here is a list of some *dim*-cells in *pol*. The returned boundary is, accordingly, a list of $(dim-1)$-cells.

Example

```
gap> s1 := sphereAB(1);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ] ] ], vertices := [ "A", "B" ] )
gap> t2 := PolProduct(s1,s1);
rec(
  faces :=
    [ [ [ 1, 2 ], [ 1, 2 ], [ 3, 4 ], [ 3, 4 ], [ 1, 3 ], [ 2, 4 ], [ 1, 3 ],
          [ 2, 4 ] ],
      [ [ 1, 3, 5, 6 ], [ 2, 4, 5, 6 ], [ 1, 3, 7, 8 ], [ 2, 4, 7, 8 ] ] ],
  vertices := [ [ "A", "A" ], [ "A", "B" ], [ "B", "A" ], [ "B", "B" ] ] )
gap> SetOfFacesBoundary(t2,[1],2);
[ 1, 3, 5, 6 ]
gap> SetOfFacesBoundary(t2,[2],2);
[ 2, 4, 5, 6 ]
gap> SetOfFacesBoundary(t2,[2,1],2);
[ 1, 2, 3, 4 ]
```

2.1.2 SubPolytope

▷ `SubPolytope(`*pol, subpol, dim*`)` (function)

subpol is a list of some *dim*-cells in polytope *pol*. The function returns (the closure of) *subpol* as a separate polytope.

Example

```
gap> SubPolytope(t2,[1,2],1);
rec( faces := [ [ [ 1, 2 ], [ 1, 2 ] ] ],
  vertices := [ [ "A", "A" ], [ "A", "B" ] ] )
```

2.1.3 ParallelSimplify

▷ `ParallelSimplify(`*pol, subpol, dim*`)` (function)

Polytope *pol* is simplified using function `UnionFaces`, along with its subpolytope *subpol* of dimension *dim* which is simplified simultaneously. The result is in the form of polytope but with an additional named list `.subpol` containing the numbers of *dim*-cells of the simplified polytope that belong to the simplified subpolytope. The function works only for subpolytopes whose dimension *dim* is strictly less than the dimension of *pol*.

2.1.4 PolMinusFace

▷ `PolMinusFace(`*pol, adr*`)` (function)

Cuts out a neighborhood of a face with adress *adr* from polytope *pol*. In this way, new cells may appear, while some of the old cells are returned in several copies. The function is organized in such way that the olds cells retain the same positions in lists *pol*`.faces[`*i*`]` and *pol*`.vertices` as they they had in the input polytope *pol*; for a multiplicated cell, one of its copies stays in that position (while the other copies are added of course somewhere on the right of the corresponding list).

2.1.5 PolMinusFaceDoublingMethod

▷ `PolMinusFaceDoublingMethod(`*pol, adr*`)` (function)

Suppose there is a neighborhood of a k-cell *adr* in a ball complex *pol* homeomorphic to several n-disks D^n_i glued along the cell *adr*. Then, the withdrawal of *adr* can be performed more economically, namely by creating, for each n-disk, its own copy of *adr*.

The following simple example deals with a bouquet of line segments glued together in vertex $adr = [0,1]$.

Example

```
gap> buket:=rec( vertices := [ 1, 2, 3, 4 ],
faces := [ [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ] ] ] );;
gap> PolMinusFaceDoublingMethod(buket,[0,1]);
rec( vertices := [ 1, 2, 3, 4, 5, 6 ],
 faces:=[ [ [ 1, 2 ], [ 3, 5 ], [ 4, 6 ] ] ] )
```

2.1.6 PolMinusPol

▷ `PolMinusPol(`*pol, subpol, dim*`)` (function)

Cuts out subpolytope *subpol* of dimension *dim* from polytope *pol*. As far as possible, the most economical way is chosen for doing this.

2.1.7 GlueFaces

▷ `GlueFaces(`*pol, pair, dim*`)` (function)

Glues together two cells in polytope *pol*, *assuming that their boundaries coincide*. The cells have dimension *dim*, and *pair* is the list of their two numbers. Remark: the function does *not* check whether the cells have indeed the same boundary. Neither does it check that the result is a valid ball complex. Here is a simple example where the result is *not* a ball complex:

Example

```
gap>d2:=ballAB(2);;
gap>GlueFaces(d2,[1,2],1);
rec(faces:=[ [ [1,2] ], [ [1] ] ],
        vertices:= ["A","B"]
```

2.1.8 VerticesRullGlueFace

▷ `VerticesRullGlueFace(`*pol, pair, dim*`)` (function)

Glues together two cells in polytope *pol*. The cells have dimension *dim*, and *pair* is the list of their two numbers.

The function looks at the *names* of vertices of two cells in *pair*, and glues them in such way that the *same-name* vertices are identified.

2.1.9 VerticesRullGluePol

▷ `VerticesRullGluePol(`*pol, subpol1, subpol2, dim*`)` (function)

Identical subpolytopes *subpol*1 and *subpol*2 of polytope *pol* and of dimension *dim* are glued into one.

Each of these subpolytopes must have at least one *dim*-cell with at least $dim + 1$ vertices and, moreover, be *combinatorial* in the following sense: both this cell and all its subcells must be determined uniquely by the sets of their vertices.

The gluing rules are determined by the *vertex names* in the list *pol*`.vertices`. The function begins with gluing together the two identical combinatorial *dim*-cells described above, starting with gluing each pair of same-named vertices into one. Then the process continues inductively on all cells of *subpol*1 and *subpol*2. The presence of the two mentioned identical combinatorial *dim*-cells ensures that the gluing is performed in a unique way.

If *two* polytopes are to be glued together by identifying their identical subpolytopes, then `FreeUnionPol` can be used first.

2.1.10 GlueIndenticalSubpolitops

▷ `GlueIndenticalSubpolitops(`*pol, sub1, sub2, dim*`)` (function)

Glues together two identical subpolytopes *sub*1 and *sub*2 of polytope *pol*. Faces with identical indices are identified. ***A more detailed description is required.***

Chapter 3

Immersions and knots

Package PL provides the possibility to work with usual one-dimensional knots and with two-dimensional knotted surfaces.

3.1 Classical knots

There are two different ways of describing a knot diagram in package PL. Both of these have their own advantages.

3.1.1 Describing a knot diagram

The first way of describing a knot diagram is as follows. Assign a name to each double point, call these names "generators", choose arbitrarily the initial point (not being a double point of the diagram) and the positive direction on the knot. Go along the knot in the positive direction and make the *word* according to the following rule: at the beginning, the word is empty, then each double point is written on the right of the word and in the degree -1 if we are moving along the underpass, or 1 if we are moving along the overpass. The word is ready when we have returned to the initial point.

Additionally, each double point has its *orientation*. If the pair "positive directions of the overpass and underpass" at a double point determine the positive orientation of the plane, then the orientation is 1, otherwise -1.

The mirror image of a knot corresponds to changing all orientations.

For the trefoil knot, the simplest diagram yields the word $ac^{-1}ba^{-1}cb^{-1}$, and all orientations are negative. Here is how this diagram is represented in the package PL library.

Example

```
gap> Trefoil;
rec( kod := [ [ "a", 1 ], [ "c", -1 ], [ "b", 1 ],
              [ "a", -1 ], [ "c", 1 ], [ "b", -1 ] ],
     orient := [ [ "a", -1 ], [ "b", -1 ], [ "c", -1 ] ] )
```

The word and the orientations are thus represented as `.kod` and `.orient`, and the example makes it clear how exactly this is done.

To describe *links* in this way is slightly more complicated, because the link components must be identified additionally. For instance, `TorusKnot(2,4);` produces a two-component *link* as follows:

Example

```
gap> TorusKnot(2,4);
rec(
  kod := rec( 1 := [ [ 1, 1 ], [ 2, -1 ], [ 3, 1 ], [ 4, -1 ] ],
      2 := [ [ 1, -1 ], [ 2, 1 ], [ 3, -1 ], [ 4, 1 ] ] ),
  orient := [ [ 1, 1 ], [ 2, 1 ], [ 3, 1 ], [ 4, 1 ] ] )
```

The second way of describing a knot diagram consists in building the ball complex decomposition of sphere S^2 corresponding naturally to the diagram. In this description, the knot goes along the 1-cells of the complex.

The details can be seen on the following example.

Example

```
rec(
  1knot :=
    rec(
      dpoints := rec( 1 := [ 6, 1, 3, 4 ], 2 := [ 2, 3, 5, 6 ],
          3 := [ 4, 5, 1, 2 ] ), sheets := [ 1 .. 6 ] ),
  faces := [ [ [ 1, 3 ], [ 2, 3 ], [ 1, 2 ], [ 1, 3 ], [ 2, 3 ], [ 1, 2 ] ],
      [ [ 1, 3, 5 ], [ 1, 4 ], [ 2, 4, 6 ], [ 2, 5 ], [ 3, 6 ] ] ],
  vertices := [ "a", "b", "c" ] )
```

This example shows a trefoil diagram made as a ball complex decomposition of S^2 with the additional information concerning the knot itself, and included in the named list `.1knot` (meaning "one-dimensional knot", in contrast with knotted 2-surfaces). This latter contains the list `.sheets` (remark: the word "sheets" may look a bit strange here, but the point is that there is also a parallel construction for 2-knots!) of all 1-cells along which the knot diagram goes, and the named list `.dpoints` which assigns 4-lists to the names - double point indices. The first two entries in each of these 4-lists are the indices of two 1-cells forming the overpass at the double point, while the second two entries are the indices of two 1-cells forming the underpass.

This desription contains redundant information and is not really convenient for writing it manually looking at a knot diagram. It has, nevertheless, some advantages. First, it allows to work directly with link diagrams. Second, this way of description is easily generalized to two-dimensional knotted surfaces, which will be explained below.

3.1.2 Knot1OnSphere2

▷ `Knot1OnSphere2(`*knot*`)` (function)

Creates a ball complex decomposition of sphere S^2 with all information about the *knot* diagram, as explained above.

Example

```
gap> Knot1OnSphere2(Figure8);
rec(
  1knot :=
    rec(
      dpoints := rec( 1 := [ 8, 1, 4, 3 ], 2 := [ 4, 5, 8, 7 ],
          3 := [ 2, 3, 5, 6 ], 4 := [ 6, 7, 1, 2 ] ), sheets := [ 1 .. 8 ] ),
  faces :=
```

```
    [ [ [ 1, 4 ], [ 3, 4 ], [ 1, 3 ], [ 1, 2 ], [ 2, 3 ], [ 3, 4 ], [ 2, 4 ],
          [ 1, 2 ] ],
      [ [ 1, 3, 6 ], [ 1, 4, 7 ], [ 2, 5, 7 ], [ 2, 6 ], [ 3, 5, 8 ], [ 4, 8 ]
        ] ], vertices := [ "a", "b", "c", "d" ] )
```

3.1.3 KnotInS3

▷ `KnotInS3(`*knot*`)` (function)

Creates a ball complex decomposition of *three*-dimensional sphere S^3. The 1-cells along which the *knot* goes are shown in the named list `.knot`.

Example

```
gap> KnotInS3(Figure8);
rec(
  faces :=
    [ [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ], [ 7, 8 ], [ 1, 7 ], [ 2, 8 ], [ 5, 7 ],
          [ 6, 8 ], [ 1, 5 ], [ 2, 6 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ],
          [ 4, 6 ], [ 5, 7 ], [ 6, 8 ], [ 3, 7 ], [ 4, 8 ], [ 1, 3 ],
          [ 2, 4 ], [ 1, 8 ], [ 5, 8 ], [ 2, 5 ], [ 2, 3 ], [ 3, 6 ],
          [ 6, 7 ], [ 4, 7 ], [ 1, 4 ] ],
      [ [ 1, 6, 21 ], [ 3, 8, 22 ], [ 1, 9, 23 ], [ 1, 11, 24 ],
          [ 2, 14, 25 ], [ 3, 15, 26 ], [ 2, 17, 27 ], [ 1, 20, 28 ],
          [ 5, 11, 17 ], [ 6, 12, 18 ], [ 7, 13, 17 ], [ 8, 14, 18 ],
          [ 7, 15 ], [ 8, 16 ], [ 9, 13, 19 ], [ 10, 14, 20 ], [ 11, 19 ],
          [ 12, 20 ], [ 4, 5, 21 ], [ 4, 7, 22 ], [ 3, 10, 23 ],
          [ 2, 12, 24 ], [ 3, 13, 25 ], [ 4, 16, 26 ], [ 4, 18, 27 ],
          [ 2, 19, 28 ] ],
      [ [ 1, 4, 7, 9, 10, 19, 22, 25 ], [ 2, 5, 7, 11, 12, 20, 23, 25 ],
          [ 2, 6, 13, 14, 20, 24 ], [ 3, 5, 8, 15, 16, 21, 23, 26 ],
          [ 4, 8, 17, 18, 22, 26 ],
          [ 1, 3, 6, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 24 ] ] ],
  knot := [ 21, 22, 23, 24, 25, 26, 27, 28 ],
  vertices := [ [ "a", "1" ], [ "a", "0" ], [ "b", "1" ], [ "b", "0" ],
      [ "c", "1" ], [ "c", "0" ], [ "d", "1" ], [ "d", "0" ] ] )
```

3.1.4 Reidemeister10Everywhere

▷ `Reidemeister10Everywhere(`*knot*`)` (function)

Checks the *knot* diagram for presence of free loops that can be removed by the first Reidemeister move R_0^{-1}, and removes them.

The numbers 10 in the function name are because such a Reidemeister takes 1 vertex and returns (locally) 0 vertices.

3.1.5 ZeroLinkFromKnot

▷ `ZeroLinkFromKnot(`*knot*`)` (function)

Given a *knot* diagram, creates a link, adding to *knot* its copy having zero linking number with it.

First, a tubular neighborhood of *knot* in the three-dimensional space is made. The copy mentioned above is the projection of *knot* along the z axis onto the boundary of this tubular neighborhood.

3.1.6 KnotGroup

▷ `KnotGroup(knot)` (function)

Calculates the knot group, i.e., the fundamental group of the knot complement in S^3. Wirtinger relations are used.

3.1.7 TorusKnot

▷ `TorusKnot(q, p)` (function)

Creates a diagram of the torus knot (q, p), if q and p are coprime, or otherwise of the corresponding link. Parameter $q > 0$ is the number of threads, while p is the number of revolutions.

Example

```
gap> k:=TorusKnot(2,3);
rec( kod := [ [ 1, 1 ], [ 2, -1 ], [ 3, 1 ], [ 1, -1 ], [ 2, 1 ], [ 3, -1 ] ],
  orient := [ [ 1, 1 ], [ 2, 1 ], [ 3, 1 ] ] )
```

3.1.8 ZeifertSurface

▷ `ZeifertSurface(knot)` (function)

Creates a Seifert surface of the *knot*, embedded in the 3-sphere. All 2-cells belonging to the Seifert surface are listed in the list `.zeifert`.

Example

```
gap> pol:=ZeifertSurface(Knot7_7);;
gap> zeif:=SubPolytope(pol, pol.zeifert, 2);;
gap> PolOrient(zeif);
[ 1, 1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, 1 ]
```

Here we have created a sphere with a Seifert surface in it, then singled out this Seifert surface as a separate polytope, and checked that this latter is orientable by constructing an explicit orientation of its 2-cells.

3.1.9 ZeifertSurfaceWithSimplyBoundary

▷ `ZeifertSurfaceWithSimplyBoundary(knot)` (function)

Creates a Seifert surface of the *knot* whose boundary consists of only two 1-cells (and the knot goes, accordingly, along these two cells).

Example

```
gap> pol:=ZeifertSurfaceWithSimplyBoundary(Knot7_7);;
gap> SetOfFacesBoundary(pol,pol.zeifert,2);
```

```
[ 49, 159 ]
```

3.2 Knotted surfaces

Here we understand a knotted surface, or surface-knot, as a 2-surface, connected or not, and embedded in the four-dimensional Euclidean space $\mathbb{R}^4$. Projection $\pi:: \mathbb{R}^4 \to \mathbb{R}^3$ is *generic* for knotted surface T if the image $\pi(T)$ is $\mathbb{R}^3$ is locally homeomorphic to either (1) 2-disk, (2) two 2-disks intersecting transversally, (3) three 2-disks intersecting transversally, or (4) Whitney umbrella. Points having neighborhoods of types (2), (3) or (4) are called *double points*, *triple points*, or *branching points*, respectively. The set of all points (2), (3) and (4) is called *singularity set*. The *diagram* of T is its image $\pi(T)$ together with the information on which sheet is "higher" or "lower" near all double points, w.r.t. to the projection π.

3.2.1 Defining knotted surfaces

By a one-point compactification, we pass from $\mathbb{R}^3$ to the 3-sphere S^3. We pepresent a knotted surface T diagram as a ball decomposition of S^3 such that all sheets of T lie in its 2-skeleton. All information concerning the diagram within the polytope is contained in the attached named list `.2knot`. This list contains, in its turn, named list `.sheets` where the indiced of all 2-cells are collected belonging to the surface knot, and named list `.dpoints` where the names are the double point edges, and to each edge, a four-element list is attached of the 2-cells lying in the diagram and in the star of this edge. The first two entries in this list are the 2-cells lying in the upper sheet, while the two last - the 2-cells lying in the lower sheet. This information is of course enough to identify also the triple and branching points.

3.2.2 PolSimplifyWith2Knot

▷ `PolSimplifyWith2Knot(`*pol*`)` (function)

Simplifies polytope *pol* containing a knotted surface. Uses function `UnionFaces` which unites pairs of balls in the polytope.

This function, similarly to `PolSimplify`, does not check whether further simplifications of the obtained polytope are possible. So, its makes sense to apply this function several times.

3.2.3 SingularitySet2Knot

▷ `SingularitySet2Knot(`*pol*`)` (function)

Returns the singularity set, i.e., the double point graph (actually containing also triple and branching points) for the knotted surfaces diagram which is supposed to be contained in the record *pol*. The list `.graf` contains all 1-cells of the ambient polytope *pol* that enter in the double point graph. Also, there is the named list `.order` where the names are the indices of vertices, and the corresponding entry describes the vertex star in the following format. For a triple point, the first two elements are the "upper" edges - those formed by the intersection of the upper and the middle sheets; the second two elements are the "middle" edges - those formed by the intersection of the upper and the lower sheets; and the last two elements are the "lower" edges - those formed by the intersection of the middle and

the lower sheets. For a double point, the list contains just two edges, and for a branching point - just one edge.

Example

```
gap> SingularitySet2Knot(TurnKnot(Trefoil,2));
        ...
rec( graf := [ 7, 4, 5, 6, 19, 20, 21, 22, 17, 18, 23, 8, 9, 10, 12, 45, 32,
      31, 25, 26, 51, 54, 55, 64, 112, 113, 114, 115, 134, 136, 137, 138,
      131, 130, 129, 119, 109, 118, 127, 106, 124, 105, 123, 104, 122, 102,
      99, 98, 79, 97, 78, 53, 94, 52, 59, 93, 92, 57, 73, 56, 72, 90, 70, 69,
      68, 67, 66, 65 ],
  order := rec( 1 := [ 137, 92 ], 10 := [ 12, 102 ], 11 := [ 10, 12 ],
      13 := [ 4 ], 14 := [ 22, 17, 19, 20, 104, 97 ],
      15 := [ 19, 18, 21, 22, 105, 98 ], 16 := [ 20, 21, 18, 23, 106, 99 ],
      19 := [ 25 ], 2 := [ 138, 94 ], 20 := [ 32, 31, 25, 26, 109, 102 ],
      21 := [ 23, 26 ], 23 := [ 17, 32 ], 24 := [ 31, 112 ],
      25 := [ 113, 104 ], 26 := [ 114, 105 ], 27 := [ 115, 106 ],
      30 := [ 45, 109 ], 34 := [ 45, 112 ], 36 := [ 113, 118 ],
      37 := [ 115, 119 ], 40 := [ 51, 56, 54, 53, 118, 122 ],
      41 := [ 53, 52, 55, 56, 114, 123 ], 42 := [ 54, 55, 52, 57, 119, 124 ],
      45 := [ 127, 59 ], 46 := [ 59, 57 ], 48 := [ 51 ],
      49 := [ 64, 69, 67, 66, 129, 122 ], 5 := [ 4, 9, 7, 6, 97, 92 ],
      50 := [ 66, 65, 69, 68, 130, 123 ], 51 := [ 68, 67, 70, 65, 131, 124 ],
      54 := [ 72 ], 55 := [ 79, 78, 73, 72, 134, 127 ], 56 := [ 73, 70 ],
      58 := [ 64, 79 ], 59 := [ 136, 78 ], 6 := [ 5, 6, 8, 9, 98, 93 ],
      60 := [ 137, 129 ], 61 := [ 130, 93 ], 62 := [ 138, 131 ],
      64 := [ 134, 90 ], 67 := [ 136, 90 ], 7 := [ 7, 8, 5, 10, 99, 94 ] ) )
```

3.2.4 TripleDoubleBranchPoints

▷ `TripleDoubleBranchPoints(`*pol*`)` (function)

Returns, for a ball complex *pol* with a knotted surface diagram inside it, a named list (record) containing, in its turn, named lists `.triple`, `.double` and `.branch` where the names are those polytope vertices which are triple, double or branching points of the diagram, respectively. In `.triple`, for each vertex v again a named list is returned, containing the fields `.u`, `.m` and `.d` being the lists of the 2-cells belonging to the upper, middle, and lower sheets and lying in the star of v. In `.double`, there are only lists `.u` and `.d`. As for the named list `.branch`, it simply returns, for each branching vertex v, the list of 2-cells belonging to the knotted surface and lying in the star of v.

Example

```
 gap> pol:=TurnKnot(Trefoil,1);;
 ...
 gap> TripleDoubleBranchPoints(pol);
rec( branch := rec( 13 := [ 2, 6, 22, 30 ], 19 := [ 13, 16, 37, 44 ] ),
  double :=
    rec( 1 := rec( d := [ 24, 25, 41, 57 ], u := [ 22, 57, 51, 27 ] ),
      10 := rec( d := [ 28, 37, 38 ], u := [ 2, 30, 7 ] ),
      11 := rec( d := [ 28, 36, 38 ], u := [ 2, 6, 7 ] ),
      2 := rec( d := [ 23, 28, 55, 52, 55 ], u := [ 25, 41, 26 ] ),
      21 := rec( d := [ 36, 38, 43 ], u := [ 8, 12, 14 ] ),
      23 := rec( d := [ 30, 39, 48 ], u := [ 8, 12, 16 ] ),
```

```
      24 := rec( d := [ 7, 46, 47 ], u := [ 13, 44, 14 ] ),
      25 := rec( d := [ 24, 40, 41 ], u := [ 27, 39, 51 ] ),
      26 := rec( d := [ 26, 42, 27 ], u := [ 23, 24, 40 ] ),
      27 := rec( d := [ 23, 43, 52 ], u := [ 26, 42, 41 ] ),
      30 := rec( d := [ 47, 48, 51, 57 ], u := [ 19, 44, 20, 43 ] ),
      34 := rec( d := [ 46, 54, 57, 47 ], u := [ 14, 20, 19, 44 ] ) ),
  triple :=
    rec(
      14 := rec( d := [ 32, 33, 40, 41 ], m := [ 27, 30, 35, 39 ],
          u := [ 8, 9, 11, 12 ] ),
      15 := rec( d := [ 27, 34, 35, 42 ], m := [ 23, 31, 32, 40 ],
          u := [ 9, 10, 11, 12 ] ),
      16 := rec( d := [ 23, 31, 36, 43 ], m := [ 33, 34, 41, 42 ],
          u := [ 8, 9, 10, 12 ] ),
      20 := rec( d := [ 37, 38, 43, 44 ], m := [ 7, 30, 47, 48 ],
          u := [ 8, 13, 14, 16 ] ),
      5 := rec( d := [ 24, 25, 32, 33 ], m := [ 22, 27, 30, 35 ],
          u := [ 2, 3, 5, 6 ] ),
      6 := rec( d := [ 26, 27, 34, 35 ], m := [ 23, 24, 31, 32 ],
          u := [ 3, 4, 5, 6 ] ),
      7 := rec( d := [ 23, 28, 31, 36 ], m := [ 25, 26, 33, 34 ],
          u := [ 2, 3, 4, 6 ] ) ) )
```

3.2.5 IsDiagrammOf2Knot

▷ `IsDiagrammOf2Knot(`*pol*`)` (function)

Check a 2-knot diagram in a 3-manifold. Namely: (1) the correctness of all references to polytope cells, (2) the double point graph, (3) the absence of self-osculation points.

Example

```
gap> pol:=TurnKnot(Figure8,-1);;

 All good!

gap> IsDiagrammOf2Knot(pol);
true
```

3.2.6 SurfaceOf2Knot

▷ `SurfaceOf2Knot(`*M3*`)` (function)

For a 3-manifold M^3 equipped with a knotted surface diagram, the preimage of the surface is returned, that is, the 2-manifold, where also the preimages of double, triple, and branching points are indicated. The named list `.preimages`contains lists `.1` and `.0`.. List `.1` consists of pairs of preimages of double point edges; the first entry is the edge lying in the lower sheet, and the second entry is the edge lying in the upper sheet. List `.0` consists of lists of length either 1 or 3; these are the lists of preimages of branching and triple points, respectively. For a triple points, the preimages go in the following order: the vertex on the lower sheet first, then the middle, and then the upper vertex.

Example

3.2.7 TurnKnot

▷ TurnKnot(*knot, number*) (function)

Returns a diagram, embedded in the 3-sphere S^3, of the *twist-spun* 2-knot obtained from the 1-knot diagram *knot*. The *number* is the number of twists; it can be positive, negative or zero. In the last case, we get just a spun-diagram.

Example

```
gap> TurnKnot(Trefoil,0);
I'm trying simplify a polytope.

  ...

 All good!

rec(
  2knot :=
    rec( dpoints := rec( 11 := [ 12, 7, 9, 10 ], 12 := [ 8, 9, 11, 12 ],
          13 := [ 10, 11, 13, 8 ], 14 := [ 18, 7, 15, 16 ],
          15 := [ 14, 15, 17, 18 ], 16 := [ 16, 17, 13, 14 ] ),
      sheets := [ 7, 8, 14, 9, 15, 10, 16, 11, 17, 12, 18, 13 ] ),
  faces :=
    [ [ [ 2, 3 ], [ 1, 2 ], [ 1, 3 ], [ 2, 3 ], [ 1, 2 ], [ 5, 6 ], [ 4, 5 ],
          [ 4, 6 ], [ 5, 6 ], [ 4, 5 ], [ 1, 4 ], [ 2, 5 ], [ 3, 6 ],
          [ 1, 4 ], [ 2, 5 ], [ 3, 6 ] ],
      [ [ 1, 3, 5 ], [ 1, 4 ], [ 2, 5 ], [ 6, 8, 10 ], [ 6, 9 ], [ 7, 10 ],
          [ 11, 14 ], [ 1, 6, 12, 13 ], [ 2, 7, 11, 12 ], [ 3, 8, 11, 13 ],
          [ 4, 9, 12, 13 ], [ 5, 10, 11, 12 ], [ 13, 16 ], [ 1, 6, 15, 16 ],
          [ 2, 7, 14, 15 ], [ 3, 8, 14, 16 ], [ 4, 9, 15, 16 ],
          [ 5, 10, 14, 15 ] ],
      [ [ 7, 10, 13, 16 ], [ 1, 4, 8, 10, 12 ], [ 2, 5, 8, 11 ],
          [ 3, 6, 9, 12 ], [ 1, 4, 14, 16, 18 ], [ 2, 5, 14, 17 ],
          [ 3, 6, 15, 18 ] ] ],
  vertices := [ [ 1, "a" ], [ 1, "b" ], [ 1, "c" ], [ 2, "a" ], [ 2, "b" ],
      [ 2, "c" ] ] )
```

3.2.8 2KnotInS4

▷ 2KnotInS4(*pol*) (function)

Given a three-dimensional polytope *pol* with a surface knot T diagram in it, returns an embedding of T into the 4-sphere S^4. The result is a ball complex decomposition of S^4 with the additional list .2knot containing the 2-faces that form the surface knot.

3.2.9 OrientBrockenDiagramm

▷ `OrientBrockenDiagramm(s3)` (function)

Returns a broken diagram for an orientable connected knotted surface.

For an orientable 2-knot in 3-sphere $s3$, its broken diagram and double point graph are computed. Returns a named list, as explained below.

Example

```
gap> s3:=TurnKnot(Trefoil,2);;
gap> rez:=OrientBrockenDiagramm(s3);;
gap> RecNames(date);
[ "images", "preimages", "manifold", "colines", "cosheets", "cofaces",
  "coorient" ]
```

The list `rez.manifold` contains the preimage of the 2-knot, that is, a 2-manifold.

Example

```
gap> RecNames(rez.manifold);
[ "vertices", "faces" ]
```

The list `rez.images` contains the images of the singular points in the 2-knot diagram.

Example

```
gap> PrintObj(rez.images);
rec(
  0 := [ 4, 5, 6, 11, 12, 13, 14, 17, 18, 38, 39, 40, 45, 46, 47, 48, 51, 52
     ],
  1 := [ 7, 4, 5, 6, 19, 20, 21, 22, 24, 16, 17, 18, 8, 9, 10, 11, 25, 54,
                 . . .
```

The list `rez.images.0` contains the vertex indices in polytope $s3$ for the triple and branching points. The list `rez.images.1` contains the edge indices in $s3$ for the double point edges.

The list `rez.preimages` contains the preimages of singular points in the 2-knot diagram.

Example

```
gap> PrintObj(rez.preimages);
rec(
  0 := [ [ 69, 4, 70 ], [ 71, 5, 72 ], [ 6, 73, 74 ], [ 11 ],
                 . . .
  1 := [ [ 7, 143 ], [ 4, 144 ], [ 5, 145 ], [ 6, 146 ], [ 19, 147 ],
                 . . .
```

The list `rez.preimages.0` contains the indices of the triple and branching points preimages, and the list `rez.preimages.1` contains the indices of the double edges. These indices correspond to the record `rez.manifold`. The numerations in `rez.images.i` and `rez.preimages.i` coincide in the sense that the preimage of element `rez.images.`$i[k]$ consists of the elements in `rez.preimages.`$i[k]$. For instance, the preimage of vertex 14 - a triple point - consists of three points in `rez.manifold`. Vertex 11 in polytope $s3$ is a branching point in the 2-knot diagram, so we get just one point as its preimage in polytope `rez.manifold`

Example

```
gap> pos:=Position(rez.images.0,14);;
gap> rez.preimages.0[pos];
[ 82, 14, 81 ]
gap> pos:=Position(rez.images.0,11);;
gap> rez.preimages.0[pos];
[ 11 ]
```

And finally, each double edge has exactly two preimages.

The singular point graph has the triple and branching points as its vertices, and the diagram's double edges as its edges. An edge of the singular point graph may pass through several edges of the ball complex. The singular point graph edges are numbered in some arbitrary way. The list `rez.colines` assigns to each i-th double edge from `rez.images.1` the singular point graph edges that passes through it.

The list `rez.cosheets` shows, for an i-th 2-cell from `s3.2knot`, to which 2-cell of the broken diagram it belongs.

As it is assumed that the preimage of the 2-knot is orientable, it follows that the double edges can also be oriented, and orientations can be assigned to the graph vertices as well. Namely, the direction of a double edge is positive if it makes a positive vector triple together with the normal vectors to the corresponding upper and lower sheets. Triple point orientation is the sign of the triple of the outgoing double edges, and branching point orientation tells whether the oriented double edge points at or from it. These orientations are contained in the lists `rez.coorient.`dim, where $dim = 0$ or 1.

Note that, as all edges are written as 2-lists of their vertices going in the *increasing* order, the orientation of an edge is written as either 1 or -1 depending on whether or not this increasing order orientation coincides with the double edge orientation described above.

The list `rez.cofaces.0` contains, in its i-th position, the list of double point graph edges that contain the i-th vertex of the graph. If this vertex is a triple point, then there are six such edges. The first three in their list `rez.cofaces.0[`i`]` are the edges pointing into the ith vertex, while the last three point out of the vertex. Moreover, the edges in the lists `rez.cofaces.0`, for triple points, are strictly ordered. The 1st and 4th entries are the upper edges, the 2nd and 5th entries are the middle edges, and the 3rd and 6th entries are the lower edges.

The list `rez.cofaces.1` gives, for each double point graph edge, the list of three entries, of which the first is the index of the upper sheet, the second is the index of that lower sheet that lies in the direction of the positive normal vector to the upper sheet, and the third is the index of that lower sheet that lies against the direction of the positive normal vector to the upper sheet.

Chapter 4

Related structures, auxiliary functions

4.1 Some useful lists

4.1.1 ConnectedSubset

▷ `ConnectedSubset(`*list*`)` (function)

Let *list* be a list of lists. We say that its entries $list[i]$ and $list[k]$ *can be joined* if there exist such chain of entries $list[i] = list[j_0], list[j_1], \ldots, list[j_n] = list[k]$ where every two neighbors have a non-empty intersection. The function returns all indiced i of those entries that can be joined with $list[1]$.

Example

```
gap> list:=[[2,3],[4,5],[6,7],[1,4],[2,1]];;
gap> ConnectedSubset(list);
[ 1, 5, 4, 2 ]
gap> ConnectedSubset(T2.faces[1]);
[ 1, 4, 5, 8, 2, 6, 3, 7 ]
```

This function is applied to finding the connected components of a manifold, hence its name.

4.1.2 SortCircle

▷ `SortCircle(`*list*`)` (function)

The input *list* must consist of two-element lists; these can be thought of as edges of a graph. The function sorts *list*, assuming that this graph is a cycle.

Example

```
gap> list:=T2.faces[1]{T2.faces[2][3]};
[ [ 2, 3 ], [ 1, 4 ], [ 1, 2 ], [ 3, 4 ] ]
gap> SortCircle(list);
[ [ 2, 3 ], [ 1, 2 ], [ 1, 4 ], [ 3, 4 ] ]
gap> list;
[  ]
```

4.1.3 LineOrdering

▷ `LineOrdering(list)` (function)

The input *list* must consist of two-element lists; these can be thought of as edges of a graph. Assuming that the graph is just a chain of n edges joining $n+1$ vertices, the function returns one of the two corresponding linear orderings of the edges as list `.order`, and their consistent orientations as `.orient`.

Example

```
gap> list:=[ [ 2, 3 ], [ 4, 1 ], [ 2, 4 ], [ 15, 1 ], [ 15, 5 ] ];;
gap> LineOrdering(list);
rec( order := [ 5, 4, 2, 3, 1 ], orient := [ 1, -1, 1, -1, 1 ] )
```

If the graph is not such a chain, the fuction finds some (random) subgraph in it that *is* a chain.

Example

```
gap> cycle:=[[1,2],[3,4],[4,2],[1,3]];;
gap> LineOrdering(cycle);
rec( order := [ 3, 1, 4, 2 ], orient := [ 1, -1, 1, 1 ] )
```

4.2 Rational functions

Given a polynomial f, it can be factored, using the GAP's `Factors` algorithm, as $f = af_1^{k_1}\cdots f_n^{k_n}$. We will write such factorization as $[a, f_1, k_1, ..., f_n, k_n]$. The first element in this list is the coefficient a, and then each even element is a factor, and each odd element is the multiplicity of the preceding factor. We call such way of describing a polynomial RatFunc format.

A *rational function* r is represented in RatFunc format as a two-element list. The first element is its numerator, and the second - its denominator. If one of these elements is empty, this means that the numerator or denominator is 1.

4.2.1 ConvertPolynomeToRatFunc

▷ `ConvertPolynomeToRatFunc(f)` (function)

Converts polynomial f into the RatFunc format.

4.2.2 ConvertToRatFunc

▷ `ConvertToRatFunc(f)` (function)

Converts rational function f into the RatFunc format.

4.2.3 ConvertFromRatFuncToPolynom

▷ `ConvertFromRatFuncToPolynom(f)` (function)

Converts polynomial f from RatFunc to usual format.

4.2.4 ConvertFromRatFunc

▷ `ConvertFromRatFunc(f)` (function)

Converts rational function f from RatFunc to usual format.

4.2.5 SimplifyRatFunc

▷ `SimplifyRatFunc(f)` (function)

Simplifies a rational function f. It can be given either in the usual or RatFunc format.

Example

```
gap> x:=Indeterminate(Rationals,"x");;
gap> y:=Indeterminate(Rationals,"y");;
gap> f:=(x^2-y^2)/(x+y)^2;;
gap> f1:=ConvertToRatFunc(f);
[ [ 1, x-y, 1, x+y, 1 ], [ 1, x+y, 2 ] ]
gap> SimplifyRatFunc(f1);
[ [ 1, x-y, 1 ], [ 1, x+y, 1 ] ]
```

Example

```
gap> g:=[[2,x-y^2,0],[]];;
gap> SimplifyRatFunc(g);
[ [ 2 ], [ 1 ] ]
```

Example

```
gap> x:=Indeterminate(GF(2),"x");
gap> y:=Indeterminate(GF(2),"y");
gap> z:=Indeterminate(GF(2),"z");
gap> f:=x^2+y^2+z^2;;
gap> ConvertToRatFunc(f);
[ [ Z(2)^0, x+y+z, 2 ], [ Z(2)^0 ] ]
```

4.2.6 SumRatFunc

▷ `SumRatFunc(f, g)` (function)

Calculates the sum of rational functions f and g given in the RatFunc format.

4.2.7 ProdRatFunc

▷ `ProdRatFunc(f, g)` (function)

Calculates the product of rational functions f and g given in the RatFunc format.

4.2.8 GcdPolynomial

▷ `GcdPolynomial(f, g)` (function)

Calculates the greatest common divisor of polynomials f and g. These can be entered either in RatFunc or usual, format; the result is in RatFunc format.

This calculation does not involve the overall coefficients of f and g, their gcd must be calculated separately.

4.2.9 LcmPolynomial

▷ `LcmPolynomial(f, g)` (function)

Calculates the least common multiple of polynomials f and g. These can be entered either in RatFunc or usual, format; the result is in RatFunc format.

This calculation does not involve the overall coefficients of f and g, their lcm must be calculated separately.

4.2.10 DerivativePolynomRatFunc

▷ `DerivativePolynomRatFunc(f, x)` (function)

Calculates the derivative of polynomial f given in RatFunc format.

4.2.11 DerivativeRatFunc

▷ `DerivativeRatFunc(f, x)` (function)

Calculates the derivative of rational function f given in RatFunc format.

4.2.12 JacobiMatRatFunc

▷ `JacobiMatRatFunc(listf, listx)` (function)

For list *listf* of rational functions, and list *listx* of variables, the Jacobian matrix is calculated. The data in *listf* can be given in a mixed format: some entries in the usual, and others in the RatFunc format.

4.3 Matrices

4.3.1 Pfaffian

▷ `Pfaffian(mat)` (function)

Calculates the Pfaffian of a skew-symmetric matrix *mat*. Uses an analogue of the Gauss algorithm.

Example

```
gap> mat:=[[0,-3,-1,1],[3,0,-1,2],[1,1,0,3],[-1,-2,-3,0]];;
gap> PrintArray(mat);
[ [   0,  -3,  -1,   1 ],
```

```
  [   3,   0,  -1,   2 ],
  [   1,   1,   0,   3 ],
  [  -1,  -2,  -3,   0 ] ]
gap> Pfaffian(mat);
-8
```

4.4 Grassmann algebras

At the moment, we don't have any realization of Grassmann algebra structures that could allow to declare variables as 'Grassmann'. Nevertheless, here is how we can operate with Grassmann monomials and their sums - arbitrary Grassmann algebra elements.

We assume that all Grassmann variables are numbered. The default ordering of Grassmann variables is lexicographical: for instance, the result of multiplication of two Grassmann polynomials (see below) will have its monomials ordered lexicographically.

A Grassmann monomial - product of some Grassmann variables and a scalar - is represented as a two-element list whose first element is the list of indices of the Grassmann variables, and whose second element is the scalar (element of a field, or commutative ring or algebra).

A Grassmann algebra element (= function of Grassmann variables) $f(x_1,x_2,\ldots,x_k) = \sum a_{i_1,\ldots,i_s} x_{i_1}\cdots x_{i_s}$ is represented as a named list with two fields `f.monomials` and `f.coeffs`. The first of these contains, at its j-th position, a word of Grassmann variables, while the second contains, at the same position, the coefficient corresponding to this word (that is, `f.monomials[j]` and `f.coeffs[j]` describe together one monomial entering in f.

Note that a monomial may contain *repeated variables*, which would mean its vanishing, of course. Using `GrassmannProduct` with unity (represented as `rec(monomials := [[]], coeffs := [1])`) brings a Grassmann polynomial into the canonical form, deleting, in particular, such monomials.

4.4.1 GrassmannMonomialsProduct

▷ `GrassmannMonomialsProduct(`*mon1, mon2*`)` (function)

Calculate the product of two Grassmann monomials *mon*1 and *mon*2. Here is how this works for the following two products: $(2\ a_1a_2)*(3\ a_5a_4a_6) = -6\ a_1a_2a_3a_4a_5a_6$ and $(2\ a_1a_2)*(3\ a_5a_2a_6) = 0$.

Example

```
gap> GrassmannMonomialsProduct([[1,2],2],[[5,4,6],3]);
[ [ 1, 2, 4, 5, 6 ], -6 ]
gap> GrassmannMonomialsProduct([[1,2],2],[[5,2,6],3]);
[ [  ], 0 ]
```

4.4.2 GrassmannProduct

▷ `GrassmannProduct(`*f, g*`)` (function)

Calculates the product of Grassmann algebra elements f and g.

4.4.3 GrassmannSum

▷ `GrassmannSum(f, g)` (function)

Calculates the sum of Grassmann algebra elements f and g.

4.4.4 BerezinIntegral

▷ `BerezinIntegral(f, a)` (function)

Calculates the Berezin integral of function f w.r.t. Grassmann variable (indexed as) a.

4.4.5 BerezinMultipleIntegral

▷ `BerezinMultipleIntegral(f, list)` (function)

Calculates the multiple Berezin integral of function f w.r.t. those Grassmann variables whose indices are in *list*. The order of integration is determined by the order of variables in *list*.

4.5 Four-manifold invariants from hexagon cohomology

Hexagon cohomologies produce "discrete action densities" for piecewise linear TQFT's, see https://arxiv.org/abs/1707.02847. These action densities are polynomials over a field of finite characteristic. Currently, this package provides tools for calculating manifold invariants only in characteristic two, although some functions can already work in different characteristics.

4.5.1 actionlib

▷ `actionlib` (global variable)

A data structure containing nontrivial discrete action densities for different characteristics. It consists of lists `actionlib.`*char.deg*, each containing (some) action densities that are *homogeneous* polynomials of degree *deg* over the prime field of characteristic *char*. Our discrete action densities are actually actions for a single pentachoron, and they are written in `actionlib` for pentachoron 12345 with the standard orientation.

The nontriviality of a discrete action density means here that it corresponds to a nontrivial (not being a coboundary) hexagon cocycle.

Example

```
gap> actionlib.2.3[1];
rec( coeffs := [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
  monoms := [ [ 1, 2, 4 ], [ 1, 3, 4 ], [ 1, 3, 5 ], [ 2, 3, 5 ], [ 2, 4, 5 ] ] )
```

As we see, the action density is represented as a dictionary with keys `coeffs` and `monoms`. The list `.monoms` consists of the monomials, while the list `.coeffs` consists of their coefficients, making together the polynomial in question. So, the action density in the above example is $x_1x_2x_4 + x_1x_3x_4 + x_1x_3x_5 + x_2x_3x_5 + x_2x_4x_5$. The variables $x_1, \ldots, x_5$ correspond to the 3-faces of pentachoron 12345, going in the lexicographic order: 1234, ..., 2345.

4.5.2 getLsystem

▷ `getLsystem(pol, action)` (function)

Produces a simplified form of the action for the whole polytope *pol* from a given input *action* for pentachoron $u = 12345$ - i.e., here *action* is an action density as stored in `actionlib`. Also, produces the "rough invariant" of the manifold, as introduced in https://arxiv.org/abs/1707.02847.

The simplified form means the following. The action for the whole polytope is the sum over all pentachora (with proper signs determined by their orientations), hence it depends on a humongous lot of variables x_i. But actually there exists a huge subspace W in the linear space of all variables x_i, such that the action is always constant along W, see again the above mentioned paper. This means that the action depends in fact only on a few linear combinations of x_i.

So, the result contains field `.dim` - number of these linear combinations now thought of as independent variables on which the (polynomial) action depends, field `.monomes` - the monomials entering in the action, and field `.coeffs` - their coefficients. There is also field `.rough_invariant` containing the "rough invariant" of the manifold.

Example

```
gap> s1:=sphereAB(1);;
gap> t2:=PolProduct(s1,s1);;
gap> t4:=PolProduct(t2,t2);; # we made thus a 4-torus
gap> action := actionlib.2.3[1];;
gap> t4syst := getLsystem(t4, action);
rec( coeffs := [ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ],
  dim := 6, monomes := [ [ 1, 1, 6 ], [ 1, 6, 6 ], [ 5, 6, 6 ], [ 5, 5, 6 ],
      [ 2, 2, 5 ], [ 3, 3, 4 ], [ 3, 4, 4 ], [ 2, 5, 5 ] ], rough_invariant := 6 )
```

4.5.3 lookover

▷ `lookover(lsystem, k)` (function)

Here *lsystem* is a polynomial given by its monomials `.monomes`, their corresponding coefficients `.coeffs`, and the number of variables `.dim`, like produced by the `getLsystem` command (but `rough_invariant` is not needed here). When the variables take all the values in the field of degree k (note that the characteristic is automatically taken from the polynomial), the polynomials takes values in the same field with some *frequencies* (numbers of times).

The command produces these frequencies for all field elements, written in the form of a list. The order of corresponding field elements can be seen using function `List(field, x -> x)` (exactly this function!).

Example

```
gap> lookover(t4syst, 2);
[ 2080, 0, 2016, 0 ]
gap> List(GF(4), x -> x);
[ 0*Z(2), Z(2^2), Z(2)^0, Z(2^2)^2 ]
```

As we have computed `t4syst` over the field `GF(2)`, and we put $k = 2$ in the example, the result contains the frequencies of elements from field `GF(4)`.

References

Index