simpcomp

A GAP toolbox for simplicial complexes

Version 2.1.2

December 2015

Felix Effenberger
Jonathan Spreer

Felix Effenberger Email: felix.effenberger@mis.mpg.de
Address: Max Planck Institute for Mathematics in the Sciences
Inselstr. 22
D-04103 Leipzig, Germany

Jonathan Spreer Email: j.spreer@uq.edu.au
Address: University of Queensland
School of Mathematics and Physics
Brisbane QLD 4072 Australia

Abstract

simpcomp is an extension (a so called package) to GAP for working with simplicial complexes in the context of combinatorial topology. The package enables the user to compute numerous properties of (abstract) simplicial complexes (such as the f-, g- and h-vectors, the face lattice, the fundamental group, the automorphism group, (co-)homology with explicit basis computation, etc.). It provides functions to generate simplicial complexes from facet lists, orbit representatives or difference cycles. Moreover, a variety of infinite series of combinatorial manifolds and pseudomanifolds (such as the simplex, the cross polytope, transitive handle bodies and sphere bundles, etc.) is given and it is possible to create new complexes from existing ones (links and stars, connected sums, simplicial cartesian products, handle additions, bistellar flips, etc.). simpcomp ships with an extensive library of known triangulations of manifolds and a census of all combinatorial 3-manifolds with transitive cyclic symmetry up to 22 vertices. Furthermore, it provides the user with the possibility to create own complex libraries. In addition, functions related to slicings and polyhedral Morse theory as well as a combinatorial version of algebraic blowups and the possibility to resolve isolated singularities of 4-manifolds are implemented.

simpcomp caches computed properties of a simplicial complex, thus avoiding unnecessary computations, internally handles the vertex labeling of the complexes and insures the consistency of a simplicial complex throughout all operations.

If possible, simpcomp makes use of the GAP package homology [DHSW11] for its homology computation but also provides the user with own (co-)homology algorithms. For automorphism group computation the GAP package GRAPE [Soi12] is used, which in turn uses the program `nauty` by Brendan McKay [MP14]. An internal automorphism group calculation algorithm is used as fallback if the GRAPE package is not available.

Copyright

Acknowledgements

A few functions of simpcomp are based on code from other authors. The bistellar flips implementation, the algorithm to collapse bounded simplicial complexes as well as the classification algorithm for transitive triangulations is based upon work of Frank Lutz (see [Lut03] and the GAP programs BISTELLAR and MANIFOLD_VT from [Lut]). Some functions were carried over from the homology package by Dumas et al. [DHSW11] – these functions are marked in the documentation and the source code. The internal (co-)homology algorithms were implemented by Armin Weiss.

Most of the complexes in the simplicial complex library are taken from the "Manifold Page" by Frank Lutz [Lut].

The authors acknowledge support by the Deutsche Forschungsgemeinschaft (DFG): simpcomp has been developed within the DFG projects Ku 1203/5-2 and Ku 1203/5-3.

Contents

Chapter 1

Introduction

simpcomp is a GAP package that provides the user with functions to do calculations and constructions with simplicial complexes in the context of combinatorial topology (see abstract). If possible, it makes use of the GAP packages homology [DHSW11] by J.-G. Dumas et al. and GRAPE [Soi12] by L. Soicher.

Most parts of this manual can be accessed directly from within GAP using its internal help system.

1.1 What is new

simpcomp is a package for working with simplicial complexes. It claims to provide the user with a broad spectrum of functionality regarding simplicial constructions.

simpcomp allows the user to interactively construct complexes and to compute their properties in the GAP shell. Furthermore, it makes use of GAP's expertise in groups and group operations. For example, automorphism groups and fundamental groups of complexes can be computed and examined further within the GAP system. Apart from supplying a facet list, the user can as well construct simplicial complexes from a set of generators and a prescribed automorphism group – the latter form being the common in which a complex is presented in a publication. This feature is to our knowledge unique to simpcomp. Furthermore, simpcomp as of Version 1.3.0 supports the construction of simplicial complexes of prescribed dimension, vertex number and transitive automorphism group as described in [Lut03], [CK01] and a number of functions (function prefix `SCSeries...`) provide infinite series of combinatorial manifolds with transitive automorphism group.

As of Version 1.4.0, simpcomp provides the possibility to perform a combinatorial version of algebraic blowups, so-called simplicial blowups, for combinatorial 4-manfolds as described in [SK11] and [Spr11a]. The implementation can be used as well to resolve isolated singularities of combinatorial 4-pseudomanifolds. It seems that this feature, too, is unique to simpcomp.

Starting from Version 1.5.4, simpcomp comes with more efficient code to perform bistellar moves implemented in `C` (see function `SCReduceComplexFast` (9.2.15)). However, this feature is completely optional.

1.2 simpcomp benefits

The origin of simpcomp is a collection of scripts of the two authors [Eff11a], [Spr11a] that provide basic and often-needed functions and operations for working with simplicial complexes. Apart from some optional code dealing with bistellar moves (see Section 9 and in particular

SCReduceComplexFast (9.2.15)), it is written entirely in the GAP scripting language, thus giving the user the possibility to see behind the scenes and to customize or alter simpcomp functions if needed.

The main benefit when working with simpcomp over implementing the needed functions from scratch is that simpcomp encapsulates all methods and properties of a simplicial complex in a new GAP object type (as an abstract data type). This way, among other things, simpcomp can transparently cache properties already calculated, thus preventing unnecessary double calculations. It also takes care of the error-prone vertex labeling of a complex. As of Version 1.5, simpcomp makes use of GAP's caching mechanism (as described in [BL98]) to cache all known properties of a simplicial complex. In addition, a customized data structure is provided to organize the complex library and to cache temporary information about a complex.

simpcomp provides the user with functions to save and load the simplicial complexes to and from files and to import and export a complex in various formats (e.g. from and to polymake/TOPAZ [GJ00], SnapPea [Wee99] and Regina [BBP+13] (via the SnapPea file format), Macaulay2 [GS], LaTeX, etc.).

In contrast to the software package polymake [GJ00] providing the most efficient algorithms for each task in form of a heterogeneous package (where algorithms are implemented in various languages), the primary goal when developing simpcomp was not efficiency (this is already limited by the GAP scripting language), but rather ease of use and ease of extensibility by the user in the GAP language with all its mathematical and algebraic capabilities. Extending simpcomp is possible directly from within GAP, without having to compile anything, see Chapter 18.

1.3 How to save time reading this document

The core component in simpcomp is the newly defined object types SCPropertyObject and its derived subtype SCSimplicialComplex. When working with this package it is important to understand how objects of these types can be created, accessed and modified. The reader is therefore advised to first skim over the Chapters 3 and 5.

The impatient reader may then directly skip to Chapter 17 to see simpcomp in action.

The next advised step is to have a look at the functions for creating objects of type SCSimplicialComplex, see the first section of Chapter 6.

The rest of Chapter 6 contains most of the functions that simpcomp provides, except for the functions related to (co-)homology, bistellar flips, simplicial blowups, polyhedral Morse theory, slicings (discrete normal surfaces) and the simplicial complex library that are described in the Chapters 8 to 13. Functions for the more general GAP object type SCPolyhedralComplex are described in Chapter 4 .

1.4 Organization of this document

This manual accompanying simpcomp is organized as follows.

- Chapter 2 provides a short introduction into the theory of simplicial complexes and PL-topology.
- Chapter 3 gives a short overview about the newly defined GAP object types simpcomp is working with.

- Chapter 4 is devoted to the description of the GAP object type SCPolyhedralComplex that is defined by simpcomp.
- Chapter 5 introduce the GAP object types SCSimplicialComplex and SCNormalSurface which are both derived from SCPolyhedralComplex.
- In Chapter 6 functions for working with simplicial complexes are described.
- Chapter 7 gives an overview over functions related to slicings / discrete normal surfaces.
- Chapter 8 describes the homology- and cohomology-related functions of simpcomp.
- Chapter 9 contains a description of the functions related to bistellar flips provided by simpcomp.
- In Chapter 10 simplicial blowups and resolutions of singularities of combinatorial 4-pseudomanifolds are explained.
- In Chapter 11 polyhedral Morse theory is discussed.
- In Chapter 13 the simplicial complex library and the input output functionality that simpcomp provides is described in detail.
- Chapter 15 contains descriptions of functions not fitting in the other chapters, such as the error handling and the email notification system of simpcomp.
- Chapter 16 contains a list of all property handlers allowing to access properties of a SCSimplicialComplex object, a SCNormalSurface object or a SCLibRepository object via the dot operator (pseudo object orientation).
- Chapter 17 contains the transcript of a demo session with simpcomp showing some of the constructions and calculations with simplicial complexes that can also be used as a first overview of things possible with this package.
- Finally, Chapter 18 focuses on the description of the internal structure of simpcomp and deals with aspects of extending the functionality of the package.

1.5 How to assure simpcomp works correctly

As with all software, it is important to test whether simpcomp functions correctly on your system after installing it. GAP has an internal testing mechanism and simpcomp ships with a short testing file that does some sample computations and verifies that the results are correct.

To test the functionality of simpcomp you can run the function SCRunTest (15.3.1) from the GAP console:

Example
```
gap> SCRunTest();
+ test simpcomp package, Version 2.1.2
+ GAP4stones: 69988
true
gap>
```

SCRunTest (15.3.1) should return true, otherwise the correct functionality of simpcomp cannot be guaranteed.

1.6 Controlling simpcomp log messages

Note that the verbosity of the output of information to the screen during calls to functions of the package simpcomp can be controlled by setting the info level parameter via the function `SCInfoLevel` (15.1.1).

1.7 How to cite simpcomp

If you would like to cite simpcomp using BibTeX, you can use the following BibTeX entry for the current simpcomp version (remember to include the `url` package in your LaTeX document):

```
@manual{simpcomp,
 author = "Felix Effenberger and Jonathan Spreer",
 title  = "{\tt simpcomp} - a {\tt GAP} toolkit for simplicial complexes,
           {V}ersion 2.1.2",
 year   = "2015",
 url    = "\url{https://github.com/simpcomp-team/simpcomp/}",
}
```

If you are not using BibTeX, you can use the following entry inside the bibliography environment of LaTeX.

```
\bibitem{simpcomp}
F.~Effenberger and J.~Spreer,
\emph{{\tt simpcomp} -- a {\tt GAP} toolkit for simplicial complexes},
Version 2.1.2,
2015,
\url{https://github.com/simpcomp-team/simpcomp/}.
```

Chapter 2

Theoretical foundations

The purpose of this chapter is to recall some basic definitions regarding polytopes, triangulations, polyhedral Morse theory, discrete normal surfaces, slicings, tight triangulations and simplicial blowups. The expert in these fields may well skip to the next chapter.

For a more detailed look the authors recommend the books [Hud69], [RS72] on PL-topology and [Zie95], [Grü03] on the theory of polytopes.

An overview of the more recent developments in the field of combinatorial topology can be found in [Lut05] and [Dat07].

2.1 Polytopes and polytopal complexes

A convex d-*polytope* is the convex hull of n points $p_i \in E^d$ in the d-dimensional euclidean space:

$$P = \operatorname{conv}\{v_1, \dots, v_n\} \subset E^d,$$

where the $v_1, \dots, v_n$ do not lie in a hyperplane of E^d.

From now on when talking about polytopes in this document always convex polytopes are meant unless explicitly stated otherwise.

For any supporting hyperplane $h \subset E^d$, $P \cap h$ is called a k-*face* of P if $\dim(P \cap h) = k$. The 0-faces are called *vertices*, the 1-faces *edges* and the $(d-1)$-faces are called *facets* of P.

A d-polytope P for which all facets are congruent regular $(d-1)$-polytopes and for which all vertex links are congruent regular $(d-1)$-polytopes is called regular, where the regular 2-polytopes are regular polygons.

Figure 1 below shows the only five regular convex 3-polytopes (also known as *platonic solids*).

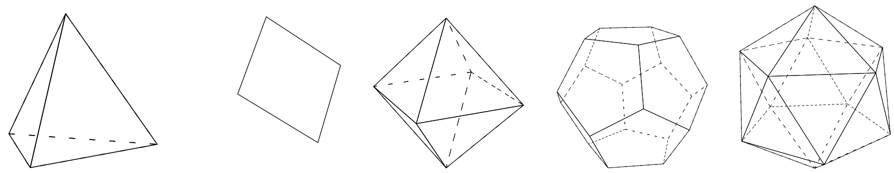


Figure 1. The *platonic solids* as the five regular convex 3-polytopes.

The set of all k-faces of P is called the k-*skeleton* of P, written as $\operatorname{skel}_k(P)$.

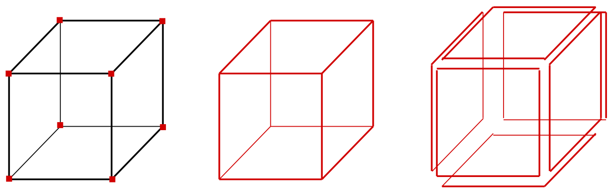


Figure 2. From left to right, drawn in grey: the 0-skeleton, the 1-skeleton and the 2-skeleton of the cube.

A *polytopal complex* C is a finite collection of polytopes P_i, $1 \leq i \leq n$ for which the intersection of any two polytopes $P_i \cap P_j$ is either empty or a common face of P_i and P_j. The polytopes of maximal dimension are called the *facets* of C. The *dimension* of a polytopal complex C is defined as the maximum over all dimensions of its facets.

For every d-dimensional polytopal complex the $(d+1)$-tuple, containing its number of i-faces in the i-th entry is called the f*-vector* of the polytopal complex.

Every polytope P gives rise to a polytopal complex consisting of all the proper faces of P. This polytopal complex is called the *boundary complex* $C(\partial P)$ *of the polytope* P.

Figure 2 below shows the boundary complex of the cube.

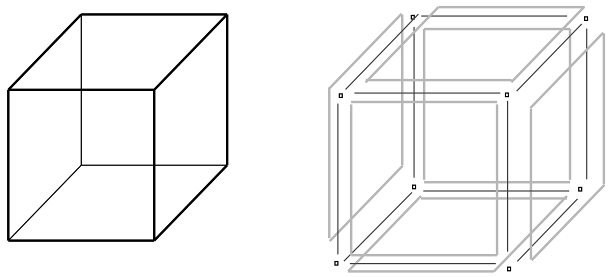


Figure 3. The 3-cube (left) and its boundary complex (right) where the 0-faces shown in black, the 1-faces dark gray and the 2-faces in light gray.

2.2 Simplices and simplicial complexes

A d-dimensional *simplex* or d*-simplex* for short is the convex hull of $d+1$ points in E^d in general position. Thus the d-simplex is the smallest (with respect to the number of vertices) possible d-polytope. Every face of the d-simplex is a m-simplex, $m \leq d$.

A 0-simplex is a point, a 1-simplex is a line segment, a 2-simplex is a triangle, a 3-simplex a tetrahedron, and so on.

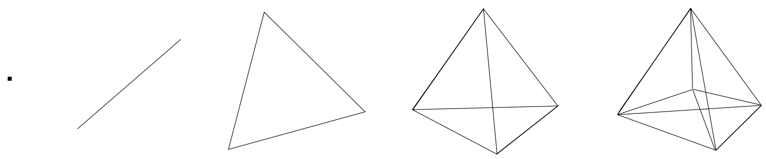


Figure 4. From left to right: a 0-simplex, a 1-simplex, a 2-simplex, a 3-simplex and a Schlegel diagram of a 4-simplex.

A polytopal complex which entirely consists of simplices is called a *simplicial complex* (for this it actually suffices that the facets (i. e., the faces that are not included in any other face of the complex) of a polytopal complex are simplices).

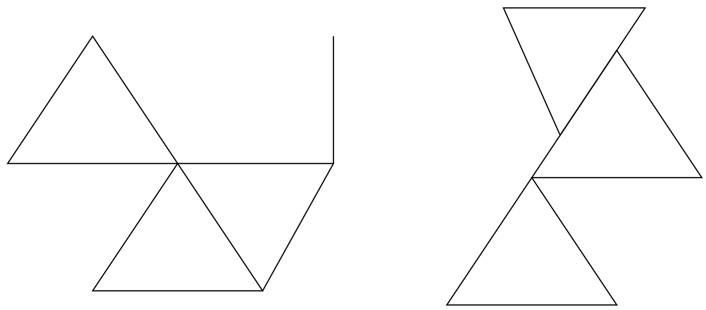


Figure 4. A simplicial complex (left) and a collection of simplices that does not form a simplicial complex (right).

The dimension of a simplicial complex is the maximal dimension of a facet. A simplicial complex is said to be *pure* if all facets are of the same dimension. A pure simplicial complex of dimension d satisfies the *weak pseudomanifold property* if every $(d-1)$-face is part of exactly two facets.

Since simplices are polytopes and, hence, simplicial complexes are polytopal complexes all of the terminology regarding simplicial complexes can be transfered from polytope theory.

2.3 From geometry to combinatorics

Every d-simplex has an *underlying set* in E^d, as the set of all points of that simplex. In the same way one can define the *underlying set* $|C|$ of a simplicial complex C. If the underlying set of a simplicial complex C is a topological manifold, then C is called *triangulated manifold* (or *triangulation of* $|C|$).

One can also go the other way and assign an abstract simplicial complex to a geometrical one by identifying each simplex with its vertex set. This obviously defines a set of sets with a natural partial ordering given by the inclusion (a socalled *poset*).

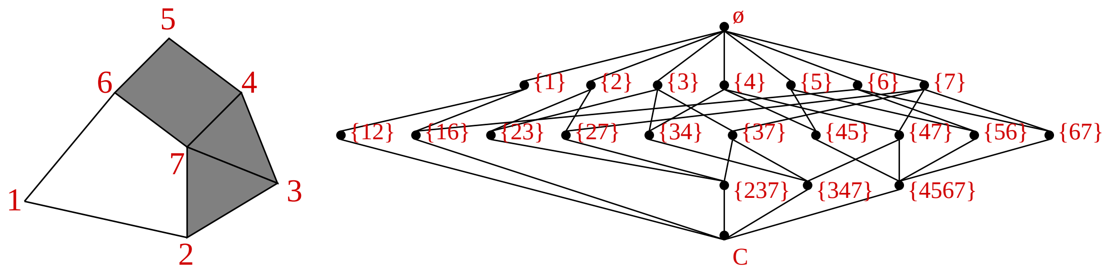


Figure 5. A geometrical polytopal complex (left) and its abstract version in form of a poset (right).

Let v be a vertex of C. The set of all facets that contain v is called *star of v in C* and is denoted by $\mathrm{star}_C(v)$. The subcomplex of $\mathrm{star}_C(v)$ that contains all faces not containing v is called *link of v in C*, written as $\mathrm{lk}_C(v)$.

A *combinatorial d-manifold* is a d-dimensional simplicial complex whose vertex links are all triangulated $(d-1)$-dimensional spheres with standard PL-structure. A *combinatorial pseudomanifold* is a simplicial complex whose vertex links are all combinatorial $(d-1)$-manifolds.

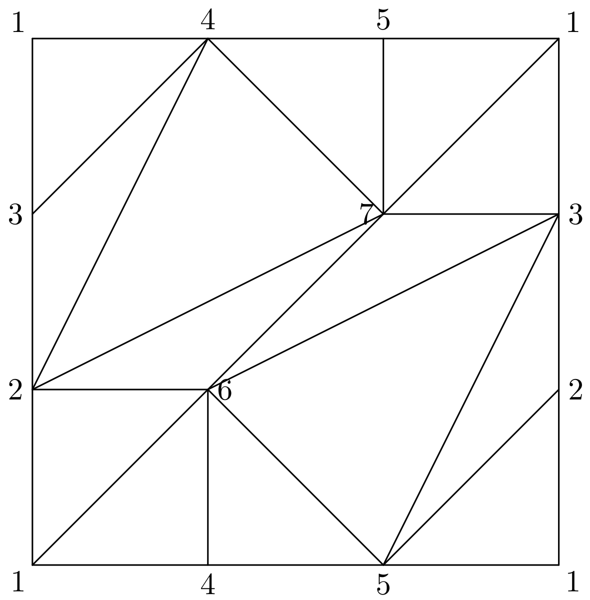


Figure 6. A simplicial complex that is a vertex-minimal combinatorial triangulation of the torus T^2 (so called Möbius' torus) – each vertex link is a hexagon.

Note that every combinatorial manifold is a triangulated manifold. The opposite is wrong: for example, there exists a triangulation of the 5-sphere that is not combinatorial, the so called *Edward's sphere*, see [BL00].

A combinatorial manifold carries an induced PL-structure and can be understood in terms of an abstract simplicial complex. If the complex has d vertices there exists a natural embedding of C into the $(d-1)$ simplex and, thus, into E^{d-1}. In general, there is no canonical embedding into any lower dimensional space. However, combinatorial methods allow to examine a given simplicial complex independently from an embedding and, in particular, independently from vertex coordinates.

Some fundamental properties of an abstract simplicial complex C are the following:

Dimensionality.
: The dimension of C.

f, g and h-vector.
: The f-vector (f_k equals the number of k-faces of a simplicial complex), the g- and h-vector can be obtained from the f-vector via linear transformations.

(Co-)Homology.
: The simplicial (co-)homology groups and Betti numbers.

Euler characteristic
: The Euler characteristic as the alternating sum over the Betti numbers / the f-vector.

Connectedness and closedness.
: Whether C is strongly connected, path connected, has a boundary or not.

Symmetries.
: The automorphism group, i. e. the group of all permutations on the set of vertex labels that do not change the complex as a whole.

All of those properties and many more can be computed on a strictly combinatorial basis.

2.4 Discrete Normal surfaces

The concept of *normal surfaces* is originally due to Kneser [Kne29] and Haken [Hak61]: A surface S, properly embedded into a 3-manifold M, is said to be *normal*, if it respects a given cell decomposition of M in the following sense: It does not intersect any vertex nor touch any 3-cell of the manifold and does not intersect with any 2-cell in a circle or an arc starting and ending in a point of the same edge. Here we will look at normal surfaces in the case that M is given as a combinatorial 3-manifold and we will call the corresponding objects *discrete normal surfaces*. In order to do this let us first define:

DEFINITION
A *polytopal manifold* is a polytopal complex M such that there exists a simplicial subdivision of M which is a combinatorial manifold. If M is a surface we will call it a *polytopal map*. If, in addition M entirely consists of m-gons, we call it a *polytopal m-gon map*.

DEFINITION (Discrete Normal surface, [Spr11b])
Let M be a combinatorial 3-manifold (3-pseudomanifold), $\Delta \in M$ one of its tetrahedra and P the intersection of Δ with a plane that does not include any vertex of Δ. Then P is called a *normal subset* of Δ. Up to an isotopy that respects the face lattice of Δ, P is equal to one of the triangles P_i, $1 \leq i \leq 4$, or quadrilaterals P_i, $5 \leq i \leq 7$, shown in Figure 7.

A polyhedral map $S \subset M$ that entirely consists of facets P_i such that every tetrahedron contains at most one facet is called *discrete normal surface* of M.

The second author has recently investigated on the combinatorial theory of discrete normal surfaces, see [Spr11b].

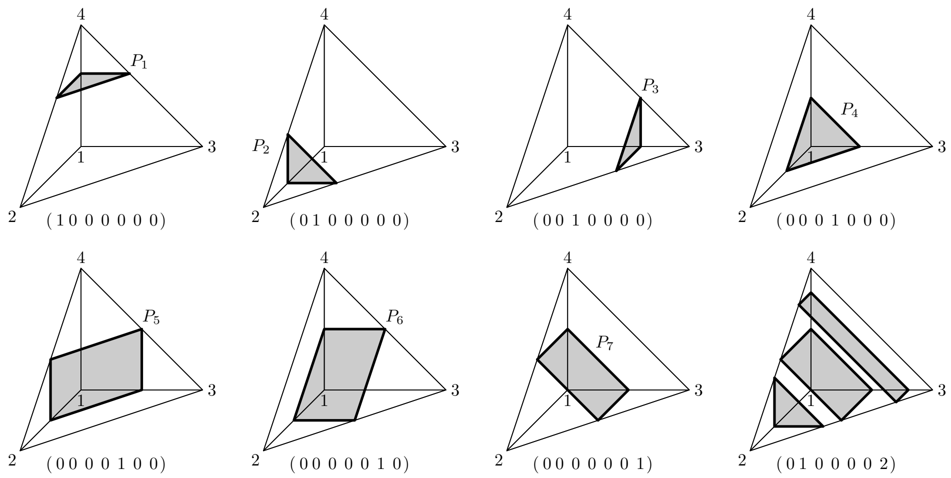


Figure 7. The seven different normal subsets of the tetrahedron. Note that the rightmost picture of the bottom row can not be part of a discrete normal surface.

2.5 Polyhedral Morse theory and slicings

In the field of PL-topology Kühnel developed what one might call a polyhedral Morse theory (compare [Küh95], not to be confused with Forman's discrete Morse theory for cell complexes which is decribed in Section 2.6):

Let M be a combinatorial d-manifold. A function $f : M \to \mathbb{R}$ is called *regular simplexwise linear (rsl)* if $f(v) \neq f(w)$ for any two vertices $w \neq v$ and if f is linear when restricted to an arbitrary simplex of the triangulation.

A vertex $x \in M$ is said to be *critical* for an rsl-function $f : M \to \mathbb{R}$, if $H_\star(M_x, M_x \backslash \{x\}, F) \neq 0$ where $M_x := \{y \in M | f(y) \leq f(x)\}$ and F is a field.

It follows that no point of M can be critical except possibly the vertices. In arbitrary dimensions we define:

DEFINITION (Slicing, [Spr11b])
Let M be a combinatorial pseudomanifold of dimension d and $f : M \to \mathbb{R}$ an rsl-function. Then we call the pre-image $f^{-1}(\alpha)$ a *slicing* of M whenever $\alpha \neq f(v)$ for any vertex $v \in M$.

By construction, a slicing is a polytopal $(d-1)$-manifold and for any ordered pair $\alpha \leq \beta$ we have $f^{-1}(\alpha) \cong f^{-1}(\beta)$ whenever $f^{-1}([\alpha, \beta])$ contains no vertex of M. In particular, a slicing S of a closed combinatorial 3-manifold M is a discrete normal surface: It follows from the simplexwise linearity of f that the intersection of the pre-image with any tetrahedron of M either forms a single triangle or a single quadrilateral. In addition, if two facets of S lie in adjacent tetrahedra they either are disjoint or glued together along the intersection line of the pre-image and the common triangle.

Any partition of the set of vertices $V = V_1 \dot\cup V_2$ of M already determines a slicing: Just define an rsl-function $f : M \to \mathbb{R}$ with $f(v) \leq f(w)$ for all $v \in V_1$ and $w \in V_2$ and look at a suitable pre-image. In the following we will write $S_{(V_1,V_2)}$ for the slicing defined by the vertex partition $V = V_1 \dot\cup V_2$.

Every vertex of a slicing is given as an intersection point of the corresponding pre-image with an edge $\langle u, w \rangle$ of the combinatorial manifold. Since there is at most one such intersection point per edge, we usually label this vertex of the slicing according to the vertices of the corresponding edge, that is $\binom{u}{w}$ with $u \in V_1$ and $w \in V_2$.

Every slicing decomposes the surrounding combinatorial manifold M into at least 2 pieces (an upper part M^+ and a lower part M^-). This is not the case for discrete normal surfaces (see 2.4) in general. However, we will focus on the case where discrete normal surfaces are slicings and we will apply the above notation for both types of objects.

Since every combinatorial pseudomanifold M has a finite number of vertices, there exist only a finite number of slicings of M. Hence, if f is chosen carefully, the induced slicings admit a useful visualization of M, c.f. [SK11].

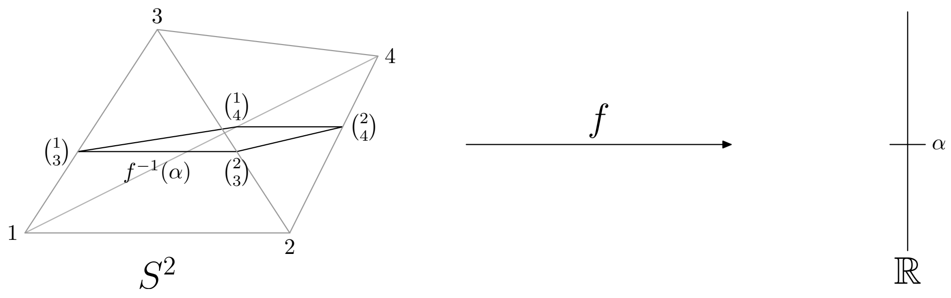


Figure 8. One dimensional slicing of the 2-sphere (represented as the boundary of the 3-simplex) seen as a level set of a regular point of a simplicial Morse function.

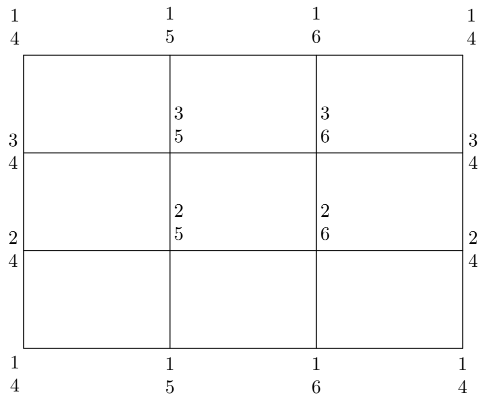


Figure 9. Handlebody decomposition of genus 1 of a 6-vertex 3-sphere - a 3×3-grid torus.

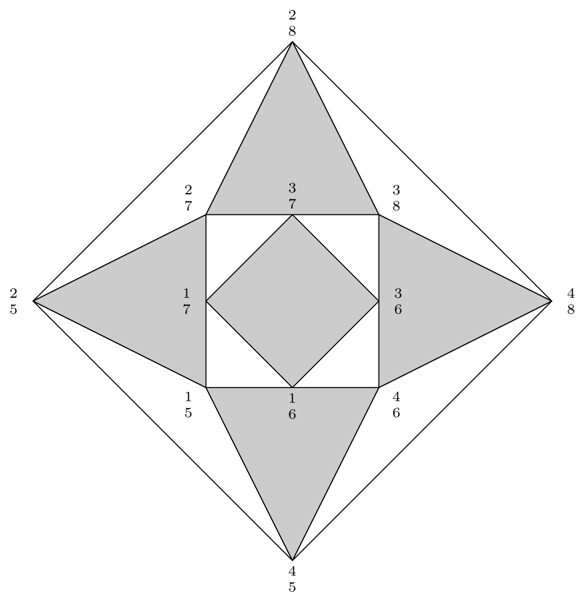


Figure 10. Separating sphere of an 8-vertex cylinder $S^2_4 \times [0,1]$ - A cuboctahedron (drawn as a Schlegel diagram of a quadrilateral face).

2.6 Discrete Morse theory

For an introduction into Forman's discrete Morse theory see [For95], not to be confused with Banchoff and Kühnel's theory of regular simplexwise linear functions which is described in Section 2.5).

2.7 Tightness and tight triangulations

Tightness is a notion developed in the field of differential geometry as the equality of the (normalized) *total absolute curvature* of a submanifold with the lower bound *sum of the Betti numbers* [Kui84], [BK97]. It was first studied by Alexandrov, Milnor, Chern and Lashof and Kuiper and later extended to the polyhedral case by Banchoff [Ban65], Kuiper [Kui84] and Kühnel [Küh95]. From a geometrical point of view, tightness can be understood as a generalization of the concept of convexity that applies to objects other than topological balls and their boundary manifolds since it roughly means that an embedding of a submanifold is "as convex as possible" according to its topology. The usual definition is the following:

DEFINITION (Tightness, [Küh95])
Let $\mathbb{F}$ be a field. An embedding $M \to \mathbb{E}^N$ of a compact manifold is called *k-tight with respect to* $\mathbb{F}$ if for any open or closed halfspace $h \subset E^N$ the induced homomorphism

$$H_i(M \cap h; \mathbb{F}) \longrightarrow H_i(M; \mathbb{F})$$

is injective for all $i \leq k$. M is called $\mathbb{F}$-*tight* if it is k-tight for all k. The standard choice for the field of coefficients is $\mathbb{F}_2$ and an $\mathbb{F}_2$-tight embedding is called *tight*.

With regard to PL embeddings of PL manifolds tightness of *combinatorial manifolds* can also be defined via a purely combinatorial condition as follows. For an introduction to PL topology see [RS72].

DEFINITION (Tight triangulation [Küh95])
Let $\mathbb{F}$ be a field. A combinatorial manifold K on n vertices is called *(k-) tight w.r.t.* $\mathbb{F}$ if its canonical embedding $K \subset \Delta^{n-1} \subset E^{n-1}$ is $(k$-)tight w.r.t. $\mathbb{F}$, where Δ^{n-1} denotes the $(n-1)$-dimensional simplex.

In dimension $d = 2$ the following are equivalent for a triangulated surface S on n vertices: (i) S has a complete edge graph K_n, (ii) S appears as a so called *regular case* in Heawood's Map Color Theorem [Rin74], compare [Küh95] and (iii) the induced piecewise linear embedding of S into Euclidean $(n-1)$-space has the two-piece property [Ban74], and it is tight [Küh95].

Kühnel investigated the tightness of combinatorial triangulations of manifolds also in higher dimensions and codimensions, see [Küh94]. It turned out that the tightness of a combinatorial triangulation is closely related to the concept of *Hamiltonicity* of a polyhedral complexes (see [Küh95]): A subcomplex A of a polyhedral complex K is called k-*Hamiltonian* if A contains the full k-dimensional skeleton of K (not to be confused with the notion of a k-Hamiltonian graph). This generalization of the notion of a Hamiltonian circuit in a graph seems to be due to C.Schulz

[Sch94]. A Hamiltonian circuit then becomes a special case of a 0-Hamiltonian subcomplex of a 1-dimensional graph or of a higher-dimensional complex.

A triangulated $2k$-manifold that is a k-Hamiltonian subcomplex of the boundary complex of some higher dimensional simplex is a tight triangulation as Kühnel [Küh95] showed. Such a triangulation is also called $(k+1)$*-neighborly triangulation* since any $k+1$ vertices in a k-dimensional simplex are common neighbors. Moreover, $(k+1)$-neighborly triangulations of $2k$-manifolds are also referred to as *super-neighborly* triangulations – in analogy with neighborly polytopes the boundary complex of a $(2k+1)$-polytope can be at most k-neighborly unless it is a simplex. Notice here that combinatorial $2k$-manifolds can go beyond k-neighborliness, depending on their topology.

Whereas in the 2-dimensional case all tight triangulations of surfaces were classified by Ringel and Jungerman and Ringel, in dimensions $d \geq 3$ there exist only a finite number of known examples of tight triangulations (see [KL99] for a census) apart from the trivial case of the boundary of a simplex and an infinite series of triangulations of sphere bundles over the circle due to Kühnel [Küh95], [Küh86].

2.8 Simplicial blowups

The *blowing up process* or *Hopf σ-process* can be described as the resolution of nodes or ordinary double points of a complex algebraic variety. This was described by H.~Hopf in [Hop51], compare [Hir53] and [Hau00]. From the topological point of view the process consists of cutting out some subspace and gluing in some other subspace. In complex algebraic geometry one point is replaced by the projective line $\mathbb{C}P^1 \cong S^2$ of all complex lines through that point. This is often called *blowing up* of the point or just *blowup*. In general the process can be applied to non-singular 4-manifolds and yields a transformation of a manifold M to $M\#(+\mathbb{C}P^2)$ or $M\#(-\mathbb{C}P^2)$, depending on the choice of an orientation. The same construction is possible for nodes or ordinary double points (a special type of singularities), and also the ambiguity of the orientation is the same for the blowup process of a node. Similarly it has been used in arbitrary even dimension by Spanier [Spa56] as a so-called *dilatation process*.

A PL version of the blowing up process is the following: We cut out the star of one of the singular vertices which is, in the case of an ordinary double point, nothing but a cone over a triangulated $\mathbb{R}P^3$. The boundary of the resulting space is this triangulated $\mathbb{R}P^3$. Now we glue back in a triangulated version $\mathbf{C}$ of a complex projective plane with a 4-ball removed where antipodal points of the boundary are identified. $\mathbf{C}$ is called a triangulated mapping cylinder and by construction its boundary is PL homeomorphic to $\mathbb{R}P^3$.

For a combinatorial version with concrete triangulations, however, we face the problem that these two triangulations are not isomorphic. This implies that before cutting out and gluing in we have to modify the triangulations by bistellar moves until they coincide:

DEFINITION (Simplicial blowup, [SK11])
Let v be a vertex of a combinatorial 4-pseudomanifold M whose link is isomorphic with the particular 11-vertex triangulation of $\mathbb{R}P^3$ which is given by the boundary complex of the triangulated $\mathbf{C}$ given in [SK11]. Let $\psi : \mathrm{lk}(v) \to \partial\mathbf{C}$ denote such an isomorphism. A simplicial resolution of the singularity

v is given by the following construction $M \mapsto \widetilde{M} := (M \setminus \operatorname{star}(v)^{\circ}) \cup_{\psi} \mathbf{C}$.

The process is described in more detail in [SK11]. In particular it is used to transform a 4-dimensional Kummer variety into a K3 surface.

Chapter 3

The new GAP object types of simpcomp

In order to meet the particular requirements of piecewise linear geometric objects and their invariants, simpcomp defines a number of new GAP object types.

All new object types are derived from the object type `SCPropertyObject` which is a subtype of `Record`. It is a GAP object consisting of permanent and temporary attributes. While simpcomp makes use of GAP's internal attribute caching mechanism for permanent attributes (see below), this is not the case for temporary ones.

The temporary properties of a `SCPropertyObject` can be accessed directly with the functions `SCPropertyTmpByName` and changed with `SCPropertyTmpSet`. But this direct access to property objects is discouraged when working with simpcomp, as the internal consistency of the objects cannot be guaranteed when the properties of the objects are modified in this way.

Important note: The temporary properties of `SCPropertyObject` are not used to hold properties (in the GAP sense) of simplicial complexes or other geometric objects. This is done by the GAP4 type system [BL98]. Instead, the properties handled by simpcomp's own caching mechanism are used to store changing information, e.g. the complex library (see Section 13) of the package or any other data which possibly is subject to changes (and thus not suited to be stored by the GAP type system).

To realize its complex library (see Section 13), simpcomp defines a GAP object type `SCLibRepository` which provides the possibility to store, load, etc. any defined geometric object to and from the build-in complex library as well as customized user libraries. In addition, a searching mechanism is provided.

Geometric objects are represented by the GAP object type `SCPolyhedralComplex`, which as well is a subtype of `SCPropertyObject`. `SCPolyhedralComplex` is designed to represent any kind of piecewise linear geometric object given by a certain cell decomposition. Here, as already mentioned, the GAP4 type system [BL98] is used to cache properties of the object. In this way, a property is not calculated multiple times in case the object is not altered (see `SCPropertiesDropped` (5.1.4) for a way of dropping previously calculated properties).

As of Version 1.4, simpcomp makes use of two different subtypes of `SCPolyhedralComplex`: `SCSimplicialComplex` to handle simplicial complexes and `SCNormalSurface` to deal with discrete normal surfaces (slicings of dimension 2). Whenever possible, only one method per operations is implemented to deal with all subtypes of `SCPolyhedralComplex`, these functions are described in Chapter 4. For all other operations, the different methods for `SCSimplicialComplex` and `SCNormalSurface` are documented separately.

3.1 Accessing properties of a `SCPolyhedralComplex` object

As described above the object type `SCPolyhedralComplex` (and thus also the GAP object types `SCSimplicialComplex` and `SCNormalSurface`) has properties that are handled by the GAP4 type system. Hence, GAP takes care of the internal consistency of objects of type `SCSimplicialComplex`.

There are two ways of accessing properties of a `SCPolyhedralComplex` object. The first is to call a property handler function of the property one wishes to calculate. The first argument of such a property handler function is always the simplicial complex for which the property should be calculated, in some cases followed by further arguments of the property handler function. An example would be:

Example
```
gap> c:=SCBdSimplex(3);; # create a SCSimplicialComplex object
gap> SCFVector(c);
[ 4, 6, 4 ]
gap> SCSkel(c,0);
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ] ]
```

Here the functions `SCFVector` and `SCSkel` are the property handler functions, see Chapter 16 for a list of all property handlers of a `SCPolyhedralComplex`, `SCSimplicialComplex` or `SCNormalSurface` object. Apart from this (standard) method of calling the property handlers directly with a `SCPolyhedralComplex` object, simpcomp provides the user with another more object oriented method which calls property handlers of a `SCPolyhedralComplex` object indirectly and more conveniently:

Example
```
gap> c:=SCBdSimplex(3);; # create a SCSimplicialComplex object
gap> c.F;
[ 4, 6, 4 ]
gap> c.Skel(0);
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ] ]
```

Note that the code in this example calculates the same properties as in the first example above, but the properties of a `SCPolyhedralComplex` object are accessed via the `.` operator (the record access operator).

For each property handler of a `SCPolyhedralComplex` object the object oriented form of this property handler equals the name of the corresponding operation. However, in most cases abbreviations are available: Usually the prefix “SC” can be dropped, in other cases even shorter names are available. See Chapter 16 for a complete list of all abbreviations available.

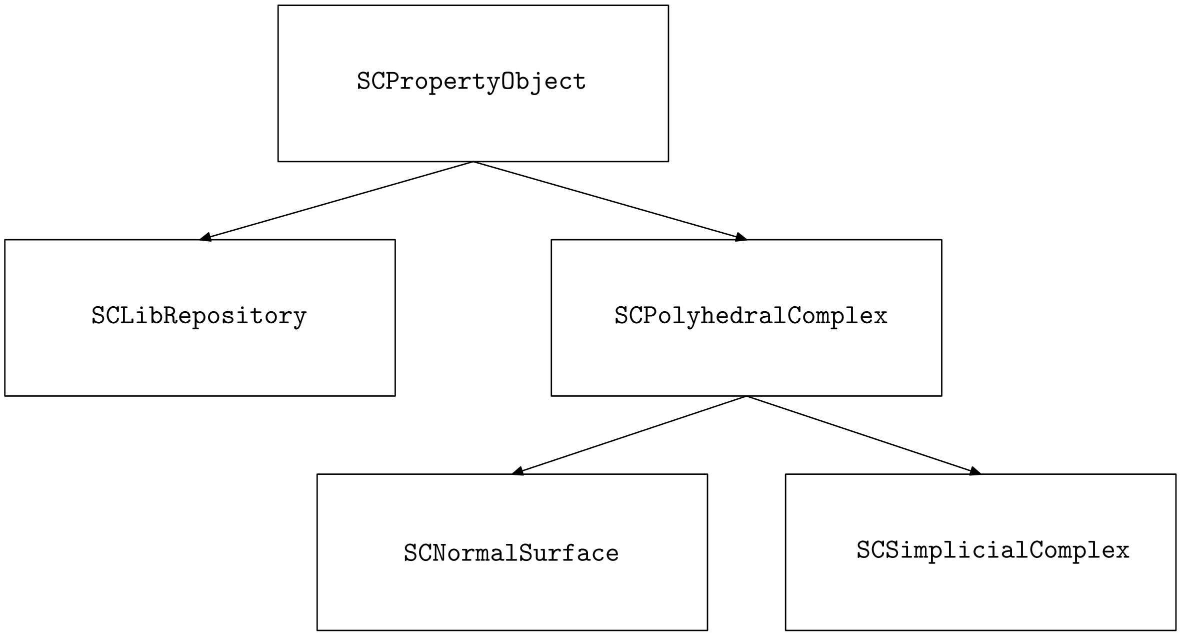


Figure 11. Overview over all GAP object types defined by simpcomp.

Chapter 4

Functions and operations for the GAP object type `SCPolyhedralComplex`

In the following all operations for the GAP object type `SCPolyhedralComplex` are listed. I. e. for the following operations only one method is implemented to deal with all geometric objects derived from this object type.

4.1 Computing properties of objects of type `SCPolyhedralComplex`

The following functions compute basic properties of objects of type `SCPolyhedralComplex` (and thus also of objects of type `SCSimplicialComplex` and `SCNormalSurface`). None of these functions alter the complex. All properties are returned as immutable objects (this ensures data consistency of the cached properties of a simplicial complex). Use `ShallowCopy` or the internal simpcomp function `SCIntFunc.DeepCopy` to get a mutable copy.

Note: every object is internally stored with the standard vertex labeling from 1 to n and a maptable to restore the original vertex labeling. Thus, we have to relabel some of the complex properties (facets, etc...) whenever we want to return them to the user. As a consequence, some of the functions exist twice, one of them with the appendix "Ex". These functions return the standard labeling whereas the other ones relabel the result to the original labeling.

4.1.1 SCFacets

▷ `SCFacets(`*`complex`*`)` (method)

Returns: a facet list upon success, `fail` otherwise.

Returns the facets of a simplicial complex in the original vertex labeling.

Example
```
gap> c:=SC([[2,3],[3,4],[4,2]]);;
gap> SCFacets(c);
[ [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

4.1.2 SCFacetsEx

▷ `SCFacetsEx(`*`complex`*`)` (method)

Returns: a facet list upon success, `fail` otherwise.

Returns the facets of a simplicial complex as they are stored, i. e. with standard vertex labeling from 1 to n.

Example
```
gap> c:=SC([[2,3],[3,4],[4,2]]);;
gap> SCFacetsEx(c);
[ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ]
```

4.1.3 SCVertices

▷ `SCVertices(complex)` (method)

Returns: a list of vertex labels of `complex` upon success, `fail` otherwise.

Returns the vertex labels of a simplicial complex `complex`.

Example
```
gap> sphere:=SC([["x",45,[1,1]],["x",45,["b",3]],["x",[1,1],
  ["b",3]],[45,[1,1],["b",3]]]);;
gap> SCVerticesEx(sphere);
[ 1 .. 4 ]
gap> SCVertices(sphere);
[ 45, [ 1, 1 ], "x", [ "b", 3 ] ]
```

4.1.4 SCVerticesEx

▷ `SCVerticesEx(complex)` (method)

Returns: $[1,\ldots,n]$ upon success, `fail` otherwise.

Returns $[1,\ldots,n]$, where n is the number of vertices of a simplicial complex `complex`.

Example
```
gap> c:=SC([[1,4,5],[4,9,8],[12,13,14,15,16,17]]);;
gap> SCVerticesEx(c);
[ 1 .. 11 ]
```

4.2 Vertex labelings and label operations

This section focuses on functions operating on the labels of a complex such as the name or the vertex labeling.

Internally, simpcomp uses the standard labeling $[1,\ldots,n]$. It is recommended to use simple vertex labels like integers and, whenever possible, the standard labeling, see also `SCRelabelStandard` (4.2.7).

4.2.1 SCLabelMax

▷ `SCLabelMax(complex)` (method)

Returns: vertex label of `complex` (an integer, a short list, a character, a short string) upon success, `fail` otherwise.

The maximum over all vertex labels is determined by the GAP function `MaximumList`.

Example
```
gap> c:=SCBdSimplex(3);;
gap> SCRelabel(c,[10,100,100000,3500]);;
gap> SCLabelMax(c);
100000
```

Example
```
gap> c:=SCBdSimplex(3);;
gap> SCRelabel(c,["a","bbb",5,[1,1]]);;
gap> SCLabelMax(c);
"bbb"
```

4.2.2 SCLabelMin

▷ `SCLabelMin(complex)` (method)

Returns: vertex label of *complex* (an integer, a short list, a character, a short string) upon success, `fail` otherwise.

The minimum over all vertex labels is determined by the GAP function `MinimumList`.

Example
```
gap> c:=SCBdSimplex(3);;
gap> SCRelabel(c,[10,100,100000,3500]);;
gap> SCLabelMin(c);
10
```

Example
```
gap> c:=SCBdSimplex(3);;
gap> SCRelabel(c,["a","bbb",5,[1,1]]);;
gap> SCLabelMin(c);
5
```

4.2.3 SCLabels

▷ `SCLabels(complex)` (method)

Returns: a list of vertex labels of *complex* (a list of integers, short lists, characters, short strings, ...) upon success, `fail` otherwise.

Returns the vertex labels of *complex* as a list. This is a synonym of `SCVertices` (4.1.3).

Example
```
gap> c:=SCFromFacets(Combinations(["a","b","c","d"],3));;
gap> SCLabels(c);
[ "a", "b", "c", "d" ]
```

4.2.4 SCName

▷ `SCName(complex)` (operation)

Returns: a string upon success, `fail` otherwise.

Returns the name of a simplicial complex *complex*.

Example
```
gap> c:=SCBdSimplex(5);;
gap> SCName(c);
"S^4_6"
```

Example
```
gap> c:=SC([[1,2],[2,3],[3,1]]);;
gap> SCName(c);
"unnamed complex 2"
```

4.2.5 SCReference

▷ `SCReference(complex)` (operation)

Returns: a string upon success, `fail` otherwise.

Returns a literature reference of a polyhedral complex *complex*.

Example
```
gap> c:=SCLib.Load(253);;
gap> SCReference(c);
"F.H.Lutz: 'The Manifold Page', http://www.math.tu-berlin.de/diskregeom/stella\
r/"
gap> c:=SC([[1,2],[2,3],[3,1]]);;
gap> SCReference(c);
#I  SCReference: complex lacks reference.
fail
```

4.2.6 SCRelabel

▷ `SCRelabel(complex, maptable)` (method)

Returns: `true` upon success, `fail` otherwise.

maptable has to be a list of length n where n is the number of vertices of *complex*. The function maps the i-th entry of *maptable* to the i-th entry of the current vertex labels. If *complex* has the standard vertex labeling $[1,\ldots,n]$ the vertex label i is mapped to *maptable[i]*.

Note that the elements of *maptable* must admit a total ordering. Hence, following Section 4.11 of the GAP manual, they must be members of one of the following families: rationals `IsRat`, cyclotomics `IsCyclotomic`, finite field elements `IsFFE`, permutations `IsPerm`, booleans `IsBool`, characters `IsChar` and lists (strings) `IsList`.

Internally the property "SCVertices" of *complex* is replaced by *maptable*.

Example
```
gap> list:=SCLib.SearchByAttribute("F[1]=12");;
gap> c:=SCLib.Load(list[1][1]);;
gap> SCVertices(c);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ]
gap> SCRelabel(c,["a","b","c","d","e","f","g","h","i","j","k","l"]);
true
gap> SCLabels(c);
[ "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l" ]
```

4.2.7 SCRelabelStandard

▷ `SCRelabelStandard(complex)` (method)

Returns: `true` upon success, `fail` otherwise.

Maps vertex labels $v_1,\dots,v_n$ of *complex* to $[1,\dots,n]$. Internally the property "SCVertices" is replaced by $[1,\dots,n]$.

Example
```
gap> list:=SCLib.SearchByAttribute("F[1]=12");;
gap> c:=SCLib.Load(list[1][1]);;
gap> SCRelabel(c,[4..15]);
true
gap> SCVertices(c);
[ 4 .. 15 ]
gap> SCRelabelStandard(c);
true
gap> SCLabels(c);
[ 1 .. 12 ]
```

4.2.8 SCRelabelTransposition

▷ `SCRelabelTransposition(complex, pair)` (method)

Returns: `true` upon success, `fail` otherwise.

Permutes vertex labels of a single pair of vertices. *pair* has to be a list of length 2 and a sublist of the property “SCVertices”.

The function is equivalent to `SCRelabel` (4.2.6) with *maptable* = $[SCVertices[1],\dots,SCVertices[j],\dots,SCVertices[i],\dots,SCVertices[n]]$ if *pair* = $[SCVertices[j],SCVertices[i]]$, $j \leq i$, $j \neq i$.

Example
```
gap> c:=SCBdSimplex(3);;
gap> SCVertices(c);
[ 1, 2, 3, 4 ]
gap> SCRelabelTransposition(c,[1,2]);;
gap> SCLabels(c);
[ 2, 1, 3, 4 ]
```

4.2.9 SCRename

▷ `SCRename(complex, name)` (method)

Returns: `true` upon success, `fail` otherwise.

Renames a polyhedral complex. The argument *name* has to be given in form of a string.

Example
```
gap> c:=SCBdSimplex(5);;
gap> SCName(c);
"S^4_6"
gap> SCRename(c,"mySphere");
true
gap> SCName(c);
"mySphere"
```

4.2.10 SCSetReference

▷ `SCSetReference(complex, ref)` (method)

Returns: `true` upon success, `fail` otherwise.

Sets the literature reference of a polyhedral complex. The argument *ref* has to be given in form of a string.

Example
```
gap> c:=SCBdSimplex(5);;
gap> SCReference(c);
#I  SCReference: complex lacks reference.
fail
gap> SCSetReference(c,"my 5-sphere in my cool paper");
true
gap> SCReference(c);
"my 5-sphere in my cool paper"
```

4.2.11 SCUnlabelFace

▷ `SCUnlabelFace(complex, face)` (method)

Returns: a list upon success, `fail` otherwise.

Computes the standard labeling of *face* in *complex*.

Example
```
gap> c:=SCBdSimplex(3);;
gap> SCRelabel(c,["a","bbb",5,[1,1]]);;
gap> SCUnlabelFace(c,["a","bbb",5]);
[ 1, 2, 3 ]
```

4.3 Operations on objects of type `SCPolyhedralComplex`

The following functions perform operations on objects of type `SCPolyhedralComplex` and all of its subtypes. Most of them return simplicial complexes. Thus, this section is closely related to the Sections 6.6 (for objects of type `SCSimplicialComplex`), ”Generate new complexes from old”. However, the data generated here is rather seen as an intrinsic attribute of the original complex and not as an independent complex.

4.3.1 SCAntiStar

▷ `SCAntiStar(complex, face)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise .

Computes the anti star of *face* (a face given as a list of vertices or a scalar interpreted as vertex) in *complex*, i. e. the complement of *face* in *complex*.

Example
```
gap> SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> rp2:=SCLib.Load(last[1][1]);;
gap> SCVertices(rp2);
[ 1, 2, 3, 4, 5, 6 ]
```

```
gap> SCAntiStar(rp2,1);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="ast([ 1 ]) in RP^2 (VT)"
 Dim=2

/SimplicialComplex]
gap> last.Facets;
[ [ 2, 3, 4 ], [ 2, 4, 5 ], [ 2, 5, 6 ], [ 3, 4, 6 ], [ 3, 5, 6 ] ]
```

4.3.2 SCLink

▷ SCLink(*complex, face*) (method)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Computes the link of *face* (a face given as a list of vertices or a scalar interpreted as vertex) in a polyhedral complex *complex*, i. e. all facets containing *face*, reduced by *face*. if *complex* is pure, the resulting complex is of dimension dim(*complex*) - dim(*face*) -1. If *face* is not a face of *complex* the empty complex is returned.

Example
```
gap> SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> rp2:=SCLib.Load(last[1][1]);;
gap> SCVertices(rp2);
[ 1, 2, 3, 4, 5, 6 ]
gap> SCLink(rp2,[1]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 1 ]) in RP^2 (VT)"
 Dim=1

/SimplicialComplex]
gap> last.Facets;
[ [ 2, 3 ], [ 2, 6 ], [ 3, 5 ], [ 4, 5 ], [ 4, 6 ] ]
```

4.3.3 SCLinks

▷ SCLinks(*complex, k*) (method)

Returns: a list of simplicial complexes of type SCSimplicialComplex upon success, fail otherwise.

Computes the link of all *k*-faces of the polyhedral complex *complex* and returns them as a list of simplicial complexes. Internally calls SCLink (4.3.2) for every *k*-face of *complex*.

Example
```
gap> c:=SCBdSimplex(4);;
gap> SCLinks(c,0);
```

```
[ [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="lk([ 1 ]) in S^3_5"
     Dim=2

    /SimplicialComplex], [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="lk([ 2 ]) in S^3_5"
     Dim=2

    /SimplicialComplex], [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="lk([ 3 ]) in S^3_5"
     Dim=2

    /SimplicialComplex], [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="lk([ 4 ]) in S^3_5"
     Dim=2

    /SimplicialComplex], [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="lk([ 5 ]) in S^3_5"
     Dim=2

    /SimplicialComplex] ]
gap> SCLinks(c,1);
[ [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="lk([ 1, 2 ]) in S^3_5"
     Dim=1

    /SimplicialComplex], [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="lk([ 1, 3 ]) in S^3_5"
     Dim=1

    /SimplicialComplex], [SimplicialComplex
```

```
 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 1, 4 ]) in S^3_5"
 Dim=1

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 1, 5 ]) in S^3_5"
 Dim=1

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 2, 3 ]) in S^3_5"
 Dim=1

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 2, 4 ]) in S^3_5"
 Dim=1

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 2, 5 ]) in S^3_5"
 Dim=1

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 3, 4 ]) in S^3_5"
 Dim=1

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 3, 5 ]) in S^3_5"
 Dim=1

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 4, 5 ]) in S^3_5"
 Dim=1
```

```
    /SimplicialComplex] ]
```

4.3.4 SCStar

▷ SCStar(*complex, face*) (method)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise .

Computes the star of *face* (a face given as a list of vertices or a scalar interpreted as vertex) in a polyhedral complex *complex*, i. e. the set of facets of *complex* that contain *face*.

Example
```
gap> SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> rp2:=SCLib.Load(last[1][1]);;
gap> SCVertices(rp2);
[ 1, 2, 3, 4, 5, 6 ]
gap> SCStar(rp2,1);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="star([ 1 ]) in RP^2 (VT)"
 Dim=2

/SimplicialComplex]
gap> last.Facets;
[ [ 1, 2, 3 ], [ 1, 2, 6 ], [ 1, 3, 5 ], [ 1, 4, 5 ], [ 1, 4, 6 ] ]
```

4.3.5 SCStars

▷ SCStars(*complex, k*) (method)

Returns: a list of simplicial complexes of type SCSimplicialComplex upon success, fail otherwise.

Computes the star of all *k*-faces of the polyhedral complex *complex* and returns them as a list of simplicial complexes. Internally calls SCStar (4.3.4) for every *k*-face of *complex*.

Example
```
gap> SCLib.SearchByName("T^2"){[1..6]};
[ [ 4, "T^2 (VT)" ], [ 5, "T^2 (VT)" ], [ 9, "T^2 (VT)" ], [ 10, "T^2 (VT)" ],
  [ 18, "T^2 (VT)" ], [ 20, "(T^2)#2" ] ]
gap> torus:=SCLib.Load(last[1][1]);; # the minimal 7-vertex torus
gap> SCStars(torus,0); # 7 2-discs as vertex stars
[ [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="star([ 1 ]) in T^2 (VT)"
     Dim=2

    /SimplicialComplex], [SimplicialComplex
```

```
 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="star([ 2 ]) in T^2 (VT)"
 Dim=2

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="star([ 3 ]) in T^2 (VT)"
 Dim=2

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="star([ 4 ]) in T^2 (VT)"
 Dim=2

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="star([ 5 ]) in T^2 (VT)"
 Dim=2

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="star([ 6 ]) in T^2 (VT)"
 Dim=2

/SimplicialComplex], [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="star([ 7 ]) in T^2 (VT)"
 Dim=2

/SimplicialComplex] ]
```

Chapter 5

The GAP object types `SCSimplicialComplex` and `SCNormalSurface`

Currently, the GAP package simpcomp supports data structures for two different kinds of geometric objects, namely simplicial complexes (`SCSimplicialComplex`) and discrete normal surfaces (`SCNormalSurface`) which are both subtypes of the GAP object type `SCPolyhedralComplex`

5.1 The object type `SCSimplicialComplex`

A major part of simpcomp deals with the object type `SCSimplicialComplex`. For a complete list of properties that `SCSimplicialComplex` handles, see Chapter 6. For a few fundamental methods and functions (such as checking the object class, copying objects of this type, etc.) for `SCSimplicialComplex` see below.

5.1.1 SCIsSimplicialComplex

▷ `SCIsSimplicialComplex(`*`object`*`)` (filter)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if *`object`* is of type `SCSimplicialComplex`. The object type `SCSimplicialComplex` is derived from the object type `SCPropertyObject`.

Example
```
gap> c:=SCEmpty();;
gap> SCIsSimplicialComplex(c);
true
```

5.1.2 SCCopy

▷ `SCCopy(`*`complex`*`)` (method)

Returns: a copy of *`complex`* upon success, `fail` otherwise.

Makes a “deep copy” of *`complex`* – this is a copy such that all properties of the copy can be altered without changing the original complex.

Example
```
gap> c:=SCBdSimplex(4);;
gap> d:=SCCopy(c)-1;;
gap> c.Facets=d.Facets;
false
```

Example
```
gap> c:=SCBdSimplex(4);;
gap> d:=SCCopy(c);;
gap> IsIdenticalObj(c,d);
false
```

5.1.3 ShallowCopy (SCSimplicialComplex)

▷ `ShallowCopy (SCSimplicialComplex)(`*`complex`*`)` (method)

Returns: a copy of *`complex`* upon success, `fail` otherwise.

Makes a copy of *`complex`*. This is actually a "deep copy" such that all properties of the copy can be altered without changing the original complex. Internally calls `SCCopy` (7.2.1).

Example
```
gap> c:=SCBdCrossPolytope(7);;
gap> d:=ShallowCopy(c)+10;;
gap> c.Facets=d.Facets;
false
```

5.1.4 SCPropertiesDropped

▷ `SCPropertiesDropped(`*`complex`*`)` (function)

Returns: a object of type `SCSimplicialComplex` upon success, `fail` otherwise.

An object of the type `SCSimplicialComplex` caches its previously calculated properties such that each property only has to be calculated once. This function returns a copy of *`complex`* with all properties (apart from Facets, Dim and Name) dropped, clearing all previously computed properties. See also `SCPropertyDrop` (18.1.8) and `SCPropertyTmpDrop` (18.1.13).

Example
```
gap> c:=SC(SCFacets(SCBdCyclicPolytope(10,12)));
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 27"
 Dim=9

/SimplicialComplex]
gap> c.F; time;
[ 12, 66, 220, 495, 792, 922, 780, 465, 180, 36 ]
144
gap> c.F; time;
[ 12, 66, 220, 495, 792, 922, 780, 465, 180, 36 ]
0
```

```
gap> c:=SCPropertiesDropped(c);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 27"
 Dim=9

/SimplicialComplex]
gap> c.F; time;
[ 12, 66, 220, 495, 792, 922, 780, 465, 180, 36 ]
140
```

5.2 Overloaded operators of `SCSimplicialComplex`

simpcomp overloads some standard operations for the object type `SCSimplicialComplex` if this definition is intuitive and mathematically sound. See a list of overloaded operators below.

5.2.1 Operation + (SCSimplicialComplex, Integer)

▷ `Operation + (SCSimplicialComplex, Integer)(`*`complex, value`*`)` (method)

Returns: the simplicial complex passed as argument upon success, `fail` otherwise.

Positively shifts the vertex labels of *`complex`* (provided that all labels satisfy the property `IsAdditiveElement`) by the amount specified in *`value`*.

Example
```
gap> c:=SCBdSimplex(3)+10;;
gap> c.Facets;
[ [ 11, 12, 13 ], [ 11, 12, 14 ], [ 11, 13, 14 ], [ 12, 13, 14 ] ]
```

5.2.2 Operation - (SCSimplicialComplex, Integer)

▷ `Operation - (SCSimplicialComplex, Integer)(`*`complex, value`*`)` (method)

Returns: the simplicial complex passed as argument upon success, `fail` otherwise.

Negatively shifts the vertex labels of *`complex`* (provided that all labels satisfy the property `IsAdditiveElement`) by the amount specified in *`value`*.

Example
```
gap> c:=SCBdSimplex(3)-1;;
gap> c.Facets;
[ [ 0, 1, 2 ], [ 0, 1, 3 ], [ 0, 2, 3 ], [ 1, 2, 3 ] ]
```

5.2.3 Operation mod (SCSimplicialComplex, Integer)

▷ `Operation mod (SCSimplicialComplex, Integer)(`*`complex, value`*`)` (method)

Returns: the simplicial complex passed as argument upon success, `fail` otherwise.

Takes all vertex labels of *complex* modulo the value specified in *value* (provided that all labels satisfy the property `IsAdditiveElement`). Warning: this might result in different vertices being assigned the same label or even in invalid facet lists, so be careful.

Example
```
gap> c:=(SCBdSimplex(3)*10) mod 7;;
gap> c.Facets;
[ [ 2, 3, 5 ], [ 2, 3, 6 ], [ 2, 5, 6 ], [ 3, 5, 6 ] ]
```

5.2.4 Operation ^ (SCSimplicialComplex, Integer)

▷ `Operation ^ (SCSimplicialComplex, Integer)(`*complex, value*`)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Forms the *value*-th simplicial cartesian power of *complex*, i.e. the *value*-fold cartesian product of copies of *complex*. The complex passed as argument is not altered. Internally calls `SCCartesianPower` (6.6.1).

Example
```
gap> c:=SCBdSimplex(2)^2; #a torus
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, TopologicalType, Vertices.

 Name="(S^1_3)^2"
 Dim=2
 TopologicalType="(S^1)^2"

/SimplicialComplex]
```

5.2.5 Operation + (SCSimplicialComplex, SCSimplicialComplex)

▷ `Operation + (SCSimplicialComplex, SCSimplicialComplex)(`*complex1, complex2*`)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Forms the connected sum of *complex1* and *complex2*. Uses the lexicographically first facets of both complexes to do the gluing. The complexes passed as arguments are not altered. Internally calls `SCConnectedSum` (6.6.5).

Example
```
gap> SCLib.SearchByName("RP^3");
[ [ 45, "RP^3" ], [ 113, "RP^3=L(2,1) (VT)" ], [ 589, "(S^2~S^1)#RP^3" ],
  [ 590, "(S^2xS^1)#RP^3" ], [ 632, "(S^2~S^1)#2#RP^3" ],
  [ 633, "(S^2xS^1)#2#RP^3" ], [ 2414, "RP^3#RP^3" ],
  [ 2426, "RP^3=L(2,1) (VT)" ], [ 2488, "(S^2~S^1)#3#RP^3" ],
  [ 2489, "(S^2xS^1)#3#RP^3" ], [ 2502, "RP^3=L(2,1) (VT)" ],
  [ 7473, "(S^2~S^1)#4#RP^3" ], [ 7474, "(S^2xS^1)#4#RP^3" ],
  [ 7504, "(S^2~S^1)#5#RP^3" ], [ 7505, "(S^2xS^1)#5#RP^3" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCLib.SearchByName("S^2~S^1"){[1..3]};
[ [ 12, "S^2~S^1 (VT)" ], [ 27, "S^2~S^1 (VT)" ], [ 28, "S^2~S^1 (VT)" ] ]
gap> d:=SCLib.Load(last[1][1]);;
```

```
gap> c:=c+d; #form RP^3#(S^2~S^1)
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="RP^3#+-S^2~S^1 (VT)"
 Dim=3

/SimplicialComplex]
```

5.2.6 Operation - (SCSimplicialComplex, SCSimplicialComplex)

▷ `Operation - (SCSimplicialComplex, SCSimplicialComplex)(`*`complex1, complex2`*`)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Calls `SCDifference` (6.10.5)(*`complex1`*, *`complex2`*)

5.2.7 Operation * (SCSimplicialComplex, SCSimplicialComplex)

▷ `Operation * (SCSimplicialComplex, SCSimplicialComplex)(`*`complex1, complex2`*`)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Forms the simplicial cartesian product of *`complex1`* and *`complex2`*. Internally calls `SCCartesianProduct` (6.6.2).

Example
```
gap> SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> c:=SCLib.Load(last[1][1])*SCBdSimplex(3); #form RP^2 x S^2
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="RP^2 (VT)xS^2_4"
 Dim=4

/SimplicialComplex]
```

5.2.8 Operation = (SCSimplicialComplex, SCSimplicialComplex)

▷ `Operation = (SCSimplicialComplex, SCSimplicialComplex)(`*`complex1, complex2`*`)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Calculates whether two simplicial complexes are isomorphic, i.e. are equal up to a relabeling of the vertices.

Example
```
gap> c:=SCBdSimplex(3);;
gap> c=c+10;
true
```

```
gap> c=SCBdCrossPolytope(4);
false
```

5.3 SCSimplicialComplex as a subtype of Set

Apart from being a subtype of SCPropertyObject, an object of type SCSimplicialComplex also behaves like a GAP Set type. The elements of the set are given by the facets of the simplical complex, grouped by their dimensionality, i.e. if complex is an object of type SCSimplicialComplex, c[1] refers to the 0-faces of complex, c[2] to the 1-faces, etc.

5.3.1 Operation Union (SCSimplicialComplex, SCSimplicialComplex)

▷ Operation Union (SCSimplicialComplex, SCSimplicialComplex)(*complex1*, *complex2*) (method)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Computes the union of two simplicial complexes by calling SCUnion (7.3.16).

Example
```
gap> c:=Union(SCBdSimplex(3),SCBdSimplex(3)+3); #a wedge of two 2-spheres
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="S^2_4 cup S^2_4"
 Dim=2

/SimplicialComplex]
```

5.3.2 Operation Difference (SCSimplicialComplex, SCSimplicialComplex)

▷ Operation Difference (SCSimplicialComplex, SCSimplicialComplex)(*complex1*, *complex2*) (method)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Computes the “difference” of two simplicial complexes by calling SCDifference (6.10.5).

Example
```
gap> c:=SCBdSimplex(3);;
gap> d:=SC([[1,2,3]]);;
gap> disc:=Difference(c,d);;
gap> disc.Facets;
[ [ 1, 2, 4 ], [ 1, 3, 4 ], [ 2, 3, 4 ] ]
gap> empty:=Difference(d,c);;
gap> empty.Dim;
-1
```

5.3.3 Operation Intersection (SCSimplicialComplex, SCSimplicialComplex)

▷ `Operation Intersection (SCSimplicialComplex, SCSimplicialComplex)(`*complex1*`,` *complex2*`)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Computes the "intersection" of two simplicial complexes by calling `SCIntersection` (6.10.8).

Example
```
gap> c:=SCBdSimplex(3);;
gap> d:=SCBdSimplex(3);;
gap> d:=SCMove(d,[[1,2,3],[]]);;
gap> d:=d+1;;
gap> s1:=SCIntersection(c,d);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="S^2_4 cap unnamed complex 20"
 Dim=1

/SimplicialComplex]
gap> s1.Facets;
[ [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

5.3.4 Size (SCSimplicialComplex)

▷ `Size (SCSimplicialComplex)(`*complex*`)` (method)

Returns: an integer upon success, `fail` otherwise.

Returns the "size" of a simplicial complex. This is $d+1$, where d is the dimension of the complex. $d+1$ is returned instead of d, as all lists in GAP are indexed beginning with 1 – thus this also holds for all the face lattice related properties of the complex.

Example
```
gap> SCLib.SearchByAttribute("F=[12,66,108,54]");
[ [ 139, "L_3_1" ], [ 140, "S^2~S^1 (VT)" ],
  [ 141, "(S^2xS^1)#(S^2xS^1) (VT)" ], [ 142, "S^2xS^1 (VT)" ],
  [ 143, "S^2xS^1 (VT)" ], [ 144, "S^2xS^1 (VT)" ], [ 145, "S^2xS^1 (VT)" ],
  [ 146, "S^2~S^1 (VT)" ], [ 147, "S^2~S^1 (VT)" ], [ 148, "S^2~S^1 (VT)" ],
  [ 149, "S^2~S^1 (VT)" ], [ 150, "S^2~S^1 (VT)" ],
  [ 151, "(S^2xS^1)#(S^2xS^1) (VT)" ], [ 152, "S^2xS^1 (VT)" ],
  [ 153, "(S^2xS^1)#(S^2xS^1) (VT)" ], [ 154, "S^2xS^1 (VT)" ],
  [ 155, "S^2xS^1 (VT)" ], [ 156, "S^2~S^1 (VT)" ], [ 157, "S^2~S^1 (VT)" ],
  [ 158, "(S^2xS^1)#(S^2xS^1) (VT)" ], [ 159, "S^2xS^1 (VT)" ],
  [ 160, "S^2xS^1 (VT)" ], [ 161, "(S^2xS^1)#(S^2xS^1) (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> for i in [1..Size(c)] do Print(c.F[i],"\n"); od;
12
66
108
54
```

5.3.5 Length (SCSimplicialComplex)

▷ `Length (SCSimplicialComplex)(complex)` (method)

Returns: an integer upon success, `fail` otherwise.

Returns the "size" of a simplicial complex by calling `Size(complex)`.

Example
```
gap> SCLib.SearchByAttribute("F=[12,66,108,54]");
[ [ 139, "L_3_1" ], [ 140, "S^2~S^1 (VT)" ],
  [ 141, "(S^2xS^1)#(S^2xS^1) (VT)" ], [ 142, "S^2xS^1 (VT)" ],
  [ 143, "S^2xS^1 (VT)" ], [ 144, "S^2xS^1 (VT)" ], [ 145, "S^2xS^1 (VT)" ],
  [ 146, "S^2~S^1 (VT)" ], [ 147, "S^2~S^1 (VT)" ], [ 148, "S^2~S^1 (VT)" ],
  [ 149, "S^2~S^1 (VT)" ], [ 150, "S^2~S^1 (VT)" ],
  [ 151, "(S^2xS^1)#(S^2xS^1) (VT)" ], [ 152, "S^2xS^1 (VT)" ],
  [ 153, "(S^2xS^1)#(S^2xS^1) (VT)" ], [ 154, "S^2xS^1 (VT)" ],
  [ 155, "S^2xS^1 (VT)" ], [ 156, "S^2~S^1 (VT)" ], [ 157, "S^2~S^1 (VT)" ],
  [ 158, "(S^2xS^1)#(S^2xS^1) (VT)" ], [ 159, "S^2xS^1 (VT)" ],
  [ 160, "S^2xS^1 (VT)" ], [ 161, "(S^2xS^1)#(S^2xS^1) (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> for i in [1..Length(c)] do Print(c.F[i],"\n"); od;
12
66
108
54
```

5.3.6 Operation [] (SCSimplicialComplex)

▷ `Operation [] (SCSimplicialComplex)(complex, pos)` (method)

Returns: a list of faces upon success, `fail` otherwise.

Returns the $(pos-1)$-dimensional faces of `complex` as a list. If $pos \geq d+2$, where d is the dimension of `complex`, the empty set is returned. Note that `pos` must be ≥ 1.

Example
```
gap> SCLib.SearchByName("K^2");
[ [ 17, "K^2 (VT)" ], [ 571, "K^2 (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> c[2];
[ [ 1, 2 ], [ 1, 3 ], [ 1, 5 ], [ 1, 7 ], [ 1, 9 ], [ 1, 10 ], [ 2, 3 ],
  [ 2, 4 ], [ 2, 6 ], [ 2, 8 ], [ 2, 10 ], [ 3, 4 ], [ 3, 5 ], [ 3, 7 ],
  [ 3, 9 ], [ 4, 5 ], [ 4, 6 ], [ 4, 8 ], [ 4, 10 ], [ 5, 6 ], [ 5, 7 ],
  [ 5, 9 ], [ 6, 7 ], [ 6, 8 ], [ 6, 10 ], [ 7, 8 ], [ 7, 9 ], [ 8, 9 ],
  [ 8, 10 ], [ 9, 10 ] ]
gap> c[4];
[  ]
```

5.3.7 Iterator (SCSimplicialComplex)

▷ `Iterator (SCSimplicialComplex)(complex)` (method)

Returns: an iterator on the face lattice of `complex` upon success, `fail` otherwise.

Provides an iterator object for the face lattice of a simplicial complex.

Example
```
gap> c:=SCBdCrossPolytope(4);;
gap> for faces in c do Print(Length(faces),"\n"); od;
8
24
32
16
```

5.4 The object type SCNormalSurface

The GAP object type `SCNormalSurface` is designed to describe slicings (level sets of discrete Morse functions) of combinatorial 3-manifolds, i. e. discrete normal surfaces. Internally `SCNormalSurface` is a subtype of `SCPolyhedralComplex` and, thus, mostly behaves like a `SCSimplicialComplex` object (see Section 5.1). For a very short introduction to normal surfaces see 2.4, for a more thorough introduction to the field see [Spr11b]. For some fundamental methods and functions for `SCNormalSurface` see below. For more functions related to the `SCNormalSurface` object type see Chapter 7.

5.5 Overloaded operators of SCNormalSurface

As with the object type `SCSimplicialComplex`, simpcomp overloads some standard operations for the object type `SCNormalSurface`. See a list of overloaded operators below.

5.5.1 Operation + (SCNormalSurface, Integer)

▷ `Operation + (SCNormalSurface, Integer)(`*complex, value*`)` (method)

Returns: the discrete normal surface passed as argument upon success, `fail` otherwise.

Positively shifts the vertex labels of *complex* (provided that all labels satisfy the property `IsAdditiveElement`) by the amount specified in *value*.

Example
```
gap> sl:=SCNSSlicing(SCBdSimplex(4),[[1],[2..5]]);;
gap> sl.Facets;
[ [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ] ], [ [ 1, 2 ], [ 1, 3 ], [ 1, 5 ] ],
  [ [ 1, 2 ], [ 1, 4 ], [ 1, 5 ] ], [ [ 1, 3 ], [ 1, 4 ], [ 1, 5 ] ] ]
gap> sl:=sl + 2;;
gap> sl.Facets;
[ [ [ 3, 4 ], [ 3, 5 ], [ 3, 6 ] ], [ [ 3, 4 ], [ 3, 5 ], [ 3, 7 ] ],
  [ [ 3, 4 ], [ 3, 6 ], [ 3, 7 ] ], [ [ 3, 5 ], [ 3, 6 ], [ 3, 7 ] ] ]
```

5.5.2 Operation - (SCNormalSurface, Integer)

▷ `Operation - (SCNormalSurface, Integer)(`*complex, value*`)` (method)

Returns: the discrete normal surface passed as argument upon success, `fail` otherwise.

Negatively shifts the vertex labels of *complex* (provided that all labels satisfy the property `IsAdditiveElement`) by the amount specified in *value*.

Example
```
gap> sl:=SCNSSlicing(SCBdSimplex(4),[[1],[2..5]]);;
gap> sl.Facets;
[ [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ] ], [ [ 1, 2 ], [ 1, 3 ], [ 1, 5 ] ],
  [ [ 1, 2 ], [ 1, 4 ], [ 1, 5 ] ], [ [ 1, 3 ], [ 1, 4 ], [ 1, 5 ] ] ]
gap> sl:=sl - 2;;
gap> sl.Facets;
[ [ [ -1, 0 ], [ -1, 1 ], [ -1, 2 ] ], [ [ -1, 0 ], [ -1, 1 ], [ -1, 3 ] ],
  [ [ -1, 0 ], [ -1, 2 ], [ -1, 3 ] ], [ [ -1, 1 ], [ -1, 2 ], [ -1, 3 ] ] ]
```

5.5.3 Operation mod (SCNormalSurface, Integer)

▷ `Operation mod (SCNormalSurface, Integer)(`*`complex, value`*`)` (method)

Returns: the discrete normal surface passed as argument upon success, `fail` otherwise.

Takes all vertex labels of *`complex`* modulo the value specified in *`value`* (provided that all labels satisfy the property `IsAdditiveElement`). Warning: this might result in different vertices being assigned the same label or even invalid facet lists, so be careful.

Example
```
gap> sl:=SCNSSlicing(SCBdSimplex(4),[[1],[2..5]]);;
gap> sl.Facets;
[ [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ] ], [ [ 1, 2 ], [ 1, 3 ], [ 1, 5 ] ],
  [ [ 1, 2 ], [ 1, 4 ], [ 1, 5 ] ], [ [ 1, 3 ], [ 1, 4 ], [ 1, 5 ] ] ]
gap> sl:=sl mod 2;;
gap> sl.Facets;
[ [ [ 1, 0 ], [ 1, 0 ], [ 1, 1 ] ], [ [ 1, 0 ], [ 1, 0 ], [ 1, 1 ] ],
  [ [ 1, 0 ], [ 1, 1 ], [ 1, 1 ] ], [ [ 1, 0 ], [ 1, 1 ], [ 1, 1 ] ] ]
```

5.6 `SCNormalSurface` as a subtype of `Set`

Like objects of type `SCSimplicialComplex`, an object of type `SCNormalSurface` behaves like a GAP `Set` type. The elements of the set are given by the facets of the normal surface, grouped by their dimensionality and type, i.e. if `complex` is an object of type `SCNormalSurface`, `c[1]` refers to the 0-faces of `complex`, `c[2]` to the 1-faces, `c[3]` to the triangles and `c[4]` to the quadrilaterals. See below for some examples and Section 5.3 for details.

5.6.1 Operation Union (SCNormalSurface, SCNormalSurface)

▷ `Operation Union (SCNormalSurface, SCNormalSurface)(`*`complex1, complex2`*`)` (method)

Returns: discrete normal surface of type `SCNormalSurface` upon success, `fail` otherwise.

Computes the union of two discrete normal surfaces by calling `SCUnion` (7.3.16).

Example
```
gap> SCLib.SearchByAttribute("F = [ 10, 35, 50, 25 ]");
[ [ 19, "S^3 (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> sl1:=SCNSSlicing(c,[[1,3,5,7,9],[2,4,6,8,10]]);;
gap> sl2:=sl1+10;;
```

```
gap> SCTopologicalType(sl1);
"T^2"
gap> sl3:=Union(sl1,sl2);;
gap> SCTopologicalType(sl3);
"T^2 U T^2"
```

Chapter 6

Functions and operations for `SCSimplicialComplex`

6.1 Creating an `SCSimplicialComplex` object from a facet list

This section contains functions to generate or to construct new simplicial complexes. Some of them obtain new complexes from existing ones, some generate new complexes from scratch.

6.1.1 SCFromFacets

▷ `SCFromFacets(`*`facets`*`)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Constructs a simplicial complex object from the given facet list. The facet list *`facets`* has to be a duplicate free list (or set) which consists of duplicate free entries, which are in turn lists or sets. For the vertex labels (i. e. the entries of the list items of *`facets`*) an ordering via the less-operator has to be defined. Following Section 4.11 of the GAP manual this is the case for objects of the following families: rationals `IsRat`, cyclotomics `IsCyclotomic`, finite field elements `IsFFE`, permutations `IsPerm`, booleans `IsBool`, characters `IsChar` and lists (strings) `IsList`.

Internally the vertices are mapped to the standard labeling $1..n$, where n is the number of vertices of the complex and the vertex labels of the original complex are stored in the property "VertexLabels", see `SCLabels` (4.2.3) and the `SCRelabel..` functions like `SCRelabel` (4.2.6) or `SCRelabelStandard` (4.2.7).

Example
```
gap> c:=SCFromFacets([[1,2,5], [1,4,5], [1,4,6], [2,3,5], [3,4,6], [3,5,6]]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 9"
 Dim=2

/SimplicialComplex]
gap> c:=SCFromFacets([["a","b","c"], ["a","b",1], ["a","c",1], ["b","c",1]]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.
```

```
 Name="unnamed complex 10"
 Dim=2

/SimplicialComplex]
```

6.1.2 SC

▷ SC(*facets*) (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

A shorter function to create a simplicial complex from a facet list, just calls `SCFromFacets` (6.1.1)(*facets*).

Example
```
gap> c:=SC(Combinations([1..6],5));
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 11"
 Dim=4

/SimplicialComplex]
```

6.1.3 SCFromDifferenceCycles

▷ SCFromDifferenceCycles(*diffcycles*) (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Creates a simplicial complex object from the list of difference cycles provided. If *diffcycles* is of length 1 the computation is equivalent to the one in `SCDifferenceCycleExpand` (6.6.8). Otherwise the induced modulus (the sum of all entries of a difference cycle) of all cycles has to be equal and the union of all expanded difference cycles is returned.

A n-dimensional difference cycle $D = (d_1 : \ldots : d_{n+1})$ induces a simplex $\Delta = (v_1, \ldots, v_{n+1})$ by $v_1 = d_1$, $v_i = v_{i-1} + d_i$ and a cyclic group action by $\mathbb{Z}_\sigma$ where $\sigma = \sum d_i$ is the modulus of D. The function returns the $\mathbb{Z}_\sigma$-orbit of Δ.

Note that modulo operations in GAP are often a little bit cumbersome, since all integer ranges usually start from 1.

Example
```
gap> c:=SCFromDifferenceCycles([[1,1,6],[2,3,3]]);;
gap> c.F;
[ 8, 24, 16 ]
gap> c.Homology;
[ [ 0, [  ] ], [ 2, [  ] ], [ 1, [  ] ] ]
gap> c.Chi;
0
gap> c.HasBoundary;
false
gap> SCIsPseudoManifold(c);
true
```

```
gap> SCIsManifold(c);
true
```

6.1.4 SCFromGenerators

▷ `SCFromGenerators(group, generators)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Constructs a simplicial complex object from the set of *generators* on which the group *group* acts, i.e. a complex which has *group* as a subgroup of the automorphism group and a facet list that consists of the *group*-orbits specified by the list of representatives passed in *generators*. Note that *group* is not stored as an attribute of the resulting complex as it might just be a subgroup of the actual automorphism group. Internally calls `Orbits` and `SCFromFacets` (6.1.1).

Example
```
gap> #group: AGL(1,7) of order 42
gap> G:=Group([(2,6,5,7,3,4),(1,3,5,7,2,4,6)]);;
gap> c:=SCFromGenerators(G,[[ 1, 2, 4 ]]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="complex from generators under unknown group"
 Dim=2

/SimplicialComplex]
gap> SCLib.DetermineTopologicalType(c);
[SimplicialComplex

 Properties known: BoundaryEx, Dim, FacetsEx, HasBoundary,
                   IsPseudoManifold, IsPure, Name, SkelExs[],
                   Vertices.

 Name="complex from generators under unknown group"
 Dim=2
 HasBoundary=false
 IsPseudoManifold=true
 IsPure=true

/SimplicialComplex]
```

6.2 Isomorphism signatures

This section contains functions to construct simplicial complexes from isomorphism signatures and to compress closed and strongly connected weak pseudomanifolds to strings.

The isomorphism signature of a closed and strongly connected weak pseudomanifold is a representation which is invariant under relabelings of the underlying complex and thus unique for a combinatorial type, i.e. two complexes are isomorphic iff they have the same isomorphism signature.

To compute the isomorphism signature of a closed and strongly connected weak pseudomanifold P we have to compute all canonical labelings of P and chose the one that is lexicographically minimal.

A canonical labeling of P is determined by chosing a facet $\Delta \in P$ and a numbering $1, 2, \ldots, d+1$ of the vertices of Δ (which in turn determines a numbering of the co-dimension one faces of Δ by identifying each face with its opposite vertex). This numbering can then be uniquely extended to a numbering (and thus a labeling) on all vertices of P by the weak pseudomanifold property: start at face 1 of Δ and label the opposite vertex of the unique other facet δ meeting face 1 by $d+2$, go on with face 2 of Δ and so on. After finishing with the first facet we now have a numbering on δ, repeat the procedure for δ, etc. Whenever the opposite vertex of a face is already labeled (and also, if the vertex occurs for the first time) we note this label. Whenever a facet is already visited we skip this step and keep track of the number of skippings between any two newly discovered facets. This results in a sequence of $m-1$ vertex labels together with $m-1$ skipping numbers (where m denotes the number of facets in P) which then can by encoded by characters via a lookup table.

Note that there are precisely $(d+1)!m$ canonical labelings we have to check in order to find the lexicographically minimal one. Thus, computing the isomorphism signature of a large or highly dimensional complex can be time consuming. If you are not interested in the isomorphism signature but just in the compressed string representation use `SCExportToString` (6.2.1) which just computes the first canonical labeling of the complex provided as argument and returns the resulting string.

Note: Another way of storing and loading complexes is provided by simpcomp's library functionality, see Section 13.1 for details.

6.2.1 SCExportToString

▷ `SCExportToString(c)` (function)

Returns: string upon success, `fail` otherwise.

Computes one string representation of a closed and strongly connected weak pseudomanifold. Compare `SCExportIsoSig` (6.2.2), which returns the lexicographically minimal string representation.

Example
```
gap> c:=SCSeriesBdHandleBody(3,9);;
gap> s:=SCExportToString(c); time;
"deffg.h.f.fahaiciai.i.hai.fbgeiagihbhceceba.g.gag"
12
gap> s:=SCExportIsoSig(c); time;
"deefgaf.hbi.gbh.eaiaeaicg.g.ibf.heg.iff.hggcfffgg"
48
```

6.2.2 SCExportIsoSig

▷ `SCExportIsoSig(c)` (method)

Returns: string upon success, `fail` otherwise.

Computes the isomorphism signature of a closed, strongly connected weak pseudomanifold. The isomorphism signature is stored as an attribute of the complex.

Example
```
gap> c:=SCSeriesBdHandleBody(3,9);;
gap> s:=SCExportIsoSig(c);
"deefgaf.hbi.gbh.eaiaeaicg.g.ibf.heg.iff.hggcfffgg"
```

6.2.3 SCFromIsoSig

▷ SCFromIsoSig(*str*) (method)

Returns: a SCSimplicialComplex object upon success, `fail` otherwise.

Computes a simplicial complex from its isomorphism signature. If a file with isomorphism signatures is provided a list of all complexes is returned.

Example
```
gap> s:="deeee";;
gap> c:=SCFromIsoSig(s);;
gap> SCIsIsomorphic(c,SCBdSimplex(4));
true
```

Example
```
gap> s:="deeee";;
gap> PrintTo("tmp.txt",s,"\n");;
gap> cc:=SCFromIsoSig("tmp.txt");
[ [SimplicialComplex

     Properties known: Dim, ExportIsoSig, FacetsEx, Name, Vertices.

     Name="unnamed complex 7"
     Dim=3

    /SimplicialComplex] ]
gap> cc[1].F;
[ 5, 10, 10, 5 ]
```

6.3 Generating some standard triangulations

6.3.1 SCBdCyclicPolytope

▷ SCBdCyclicPolytope(*d*, *n*) (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates the boundary complex of the d-dimensional cyclic polytope (a combinatorial $d-1$-sphere) on n vertices, where $n \geq d+2$.

Example
```
gap> SCBdCyclicPolytope(3,8);
[SimplicialComplex

 Properties known: Dim, EulerCharacteristic, FacetsEx, HasBoundary,
                   Homology, IsConnected, IsStronglyConnected, Name,
                   NumFaces[], TopologicalType, Vertices.

 Name="Bd(C_3(8))"
 Dim=2
 EulerCharacteristic=2
 HasBoundary=false
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsConnected=true
 IsStronglyConnected=true
```

```
 TopologicalType="S^2"

/SimplicialComplex]
```

6.3.2 SCBdSimplex

▷ `SCBdSimplex(d)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates the boundary of the d-simplex Δ^d, a combinatorial $d-1$-sphere.

Example
```
gap> SCBdSimplex(5);
[SimplicialComplex

 Properties known: AutomorphismGroup, AutomorphismGroupSize,
                   AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, Dim,
                   EulerCharacteristic, FacetsEx, GeneratorsEx,
                   HasBoundary, Homology, IsConnected,
                   IsStronglyConnected, Name, NumFaces[],
                   TopologicalType, Vertices.

 Name="S^4_6"
 Dim=4
 AutomorphismGroupSize=720
 AutomorphismGroupStructure="S6"
 AutomorphismGroupTransitivity=6
 EulerCharacteristic=2
 HasBoundary=false
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsConnected=true
 IsStronglyConnected=true
 TopologicalType="S^4"

/SimplicialComplex]
```

6.3.3 SCEmpty

▷ `SCEmpty()` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates an empty complex (of dimension -1), i. e. a `SCSimplicialComplex` object with empty facet list.

Example
```
gap> SCEmpty();
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="empty complex"
 Dim=-1
```

```
/SimplicialComplex]
```

6.3.4 SCSimplex

▷ SCSimplex(d) (function)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Generates the d-simplex.

Example
```
gap> SCSimplex(3);
[SimplicialComplex

 Properties known: Dim, EulerCharacteristic, FacetsEx, Name,
                   NumFaces[], TopologicalType, Vertices.

 Name="B^3_4"
 Dim=3
 EulerCharacteristic=1
 TopologicalType="B^3"

/SimplicialComplex]
```

6.3.5 SCSeriesTorus

▷ SCSeriesTorus(d) (function)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Generates the d-torus described in [Küh86].

Example
```
gap> t4:=SCSeriesTorus(4);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="4-torus T^4"
 Dim=4
 TopologicalType="T^4"

/SimplicialComplex]
gap> t4.Homology;
[ [ 0, [  ] ], [ 4, [  ] ], [ 6, [  ] ], [ 4, [  ] ], [ 1, [  ] ] ]
```

6.3.6 SCSurface

▷ SCSurface(g, $orient$) (function)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Generates the surface of genus g where the boolean argument *orient* specifies whether the surface is orientable or not. The surfaces have transitive cyclic group actions and can be described using the minimum amount of $O(\log(g))$ memory. If *orient* is `true` and $g \geq 50$ or if *orient* is `false` and $g \geq 100$ only the difference cycles of the surface are returned

Example
```
gap> c:=SCSurface(23,true);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="S_23^or"
 Dim=2
 TopologicalType="(T^2)^#23"

/SimplicialComplex]
gap> c.Homology;
[ [ 0, [  ] ], [ 46, [  ] ], [ 1, [  ] ] ]
gap> c.TopologicalType;
"(T^2)^#23"
gap> c:=SCSurface(23,false);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="S_23^non"
 Dim=2
 TopologicalType="(RP^2)^#23"

/SimplicialComplex]
gap> c.Homology;
[ [ 0, [  ] ], [ 22, [ 2 ] ], [ 0, [  ] ] ]
gap> c.TopologicalType;
"(RP^2)^#23"
```

Example
```
gap> dc:=SCSurface(345,true);
[ [ 1, 1, 1374 ], [ 2, 343, 1031 ], [ 343, 345, 688 ] ]
gap> c:=SCFromDifferenceCycles(dc);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name, Vertices.

 Name="complex from diffcycles [ [ 1, 1, 1374 ], [ 2, 343, 1031 ], [ 343, 345,\
 688 ] ]"
 Dim=2

/SimplicialComplex]
gap> c.Chi;
-688
gap> dc:=SCSurface(12345678910,true); time;
```

```
[ [ 1, 1, 24691357816 ], [ 2, 4, 24691357812 ], [ 3, 3, 24691357812 ],
  [ 4, 12345678907, 12345678907 ] ]
0
```

6.3.7 SCFVectorBdCrossPolytope

▷ `SCFVectorBdCrossPolytope(d)` (function)

Returns: a list of integers of size `d + 1` upon success, `fail` otherwise.

Computes the f-vector of the d-dimensional cross polytope without generating the underlying complex.

Example
```
gap> SCFVectorBdCrossPolytope(50);
[ 100, 4900, 156800, 3684800, 67800320, 1017004800, 12785203200,
  137440934400, 1282782054400, 10518812846080, 76500457062400,
  497252970905600, 2907017368371200, 15365663232819200, 73755183517532160,
  322678927889203200, 1290715711556812800, 4732624275708313600,
  15941471244491161600, 49418560857922600960, 141195888165493145600,
  372243705163572838400, 906332499528699084800, 2039248123939572940800,
  4241636097794311716864, 8156992495758291763200, 14501319992459185356800,
  23823597130468661657600, 36146147370366245273600, 50604606318512743383040,
  65296266217435797913600, 77539316133205010022400, 84588344872587283660800,
  84588344872587283660800, 77337915312079802204160, 64448262760066501836800,
  48771658304915190579200, 33370081998099867238400, 20535435075753764454400,
  11294489291664570449920, 5509506971543692902400, 2361217273518725529600,
  878592473867432755200, 279552150776001331200, 74547240206933688320,
  16205921784116019200, 2758454771764428800, 344806846470553600,
  28147497671065600, 1125899906842624 ]
```

6.3.8 SCFVectorBdCyclicPolytope

▷ `SCFVectorBdCyclicPolytope(d, n)` (function)

Returns: a list of integers of size `d+1` upon success, `fail` otherwise.

Computes the f-vector of the d-dimensional cyclic polytope on n vertices, $n \geq d+2$, without generating the underlying complex.

Example
```
gap> SCFVectorBdCyclicPolytope(25,198);
[ 198, 19503, 1274196, 62117055, 2410141734, 77526225777, 2126433621312,
  50768602708824, 1071781612741840, 20256672480820776, 346204947854027808,
  5395027104058600008, 48354596155522298656, 262068846498922699590,
  940938105142239825104, 2379003007642628680027, 4396097923113038784642,
  6062663500381642763609, 6294919173643129209180, 4911378208855785427761,
  2840750019404460890298, 1183225500922302444568, 335951678686835900832,
  58265626173398052500, 4661250093871844200 ]
```

6.3.9 SCFVectorBdSimplex

▷ `SCFVectorBdSimplex(d)` (function)

Returns: a list of integers of size `d + 1` upon success, `fail` otherwise.

Computes the f-vector of the d-simplex without generating the underlying complex.

Example

```
gap> SCFVectorBdSimplex(100);
[ 101, 5050, 166650, 4082925, 79208745, 1267339920, 17199613200,
  202095455100, 2088319702700, 19212541264840, 158940114100040,
  1192050855750300, 8160963550905900, 51297485177122800, 297525414027312240,
  1599199100396803290, 7995995501984016450, 37314645675925410100,
  163006083742200475700, 668324943343021950370, 2577824781465941808570,
  9373908296239788394800, 32197337191432316660400, 104641345872155029146300,
  322295345286237489770604, 942094086221309585483304,
  2616928017281415515231400, 6916166902815169575968700,
  17409661513983013070541900, 41783187633559231369300560,
  95696978128474368620010960, 209337139656037681356273975,
  437704928371715151926754675, 875409856743430303853509350,
  1675784582908852295948146470, 3072271735332895875904935195,
  5397234129638871133346507775, 9090078534128625066688855200,
  14683973016669317415420458400, 22760158175837441993901710520,
  33862674359172779551902544920, 48375249084532542217003635600,
  66375341767149302111702662800, 87494768693060443692698964600,
  110826707011209895344085355160, 134919469404951176940625649760,
  157884485473879036845412994400, 177620046158113916451089618700,
  192119641762857909630770403900, 199804427433372226016001220056,
  199804427433372226016001220056, 192119641762857909630770403900,
  177620046158113916451089618700, 157884485473879036845412994400,
  134919469404951176940625649760, 110826707011209895344085355160,
  87494768693060443692698964600, 66375341767149302111702662800,
  48375249084532542217003635600, 33862674359172779551902544920,
  22760158175837441993901710520, 14683973016669317415420458400,
  9090078534128625066688855200, 5397234129638871133346507775,
  3072271735332895875904935195, 1675784582908852295948146470,
  875409856743430303853509350, 437704928371715151926754675,
  209337139656037681356273975, 95696978128474368620010960,
  41783187633559231369300560, 17409661513983013070541900,
  6916166902815169575968700, 2616928017281415515231400,
  942094086221309585483304, 322295345286237489770604,
  104641345872155029146300, 32197337191432316660400, 9373908296239788394800,
  2577824781465941808570, 668324943343021950370, 163006083742200475700,
  37314645675925410100, 7995995501984016450, 1599199100396803290,
  297525414027312240, 51297485177122800, 8160963550905900, 1192050855750300,
  158940114100040, 19212541264840, 2088319702700, 202095455100, 17199613200,
  1267339920, 79208745, 4082925, 166650, 5050, 101 ]
```

6.4 Generating infinite series of transitive triangulations

6.4.1 SCSeriesAGL

▷ `SCSeriesAGL(p)` (function)

Returns: a permutation group and a list of 5-tuples of integers upon success, `fail` otherwise.

For a given prime p the automorphism group (AGL$(1,p)$) and the generators of all members of the series of 2-transitive combinatorial 4-pseudomanifolds with p vertices from [Spr11a], Section 5.2, is computed. The affine linear group AGL$(1,p)$ is returned as the first argument. If no member of the series with p vertices exists only the group is returned.

Example
```
gap> gens:=SCSeriesAGL(17);
[ AGL(1,17), [ [ 1, 2, 4, 8, 16 ] ] ]
gap> c:=SCFromGenerators(gens[1],gens[2]);;
gap> SCIsManifold(SCLink(c,1));
true
```

Example
```
gap> List([19..23],x->SCSeriesAGL(x));
#I  SCSeriesAGL: argument must be a prime > 13.
#I  SCSeriesAGL: argument must be a prime > 13.
#I  SCSeriesAGL: argument must be a prime > 13.
[ [ AGL(1,19), [ [ 1, 2, 10, 12, 17 ] ] ], fail, fail, fail,
  [ AGL(1,23), [ [ 1, 2, 7, 9, 19 ], [ 1, 2, 4, 8, 22 ] ] ] ]
gap> for i in [80000..80100] do if IsPrime(i) then Print(i,"\n"); fi; od;
80021
80039
80051
80071
80077
gap> SCSeriesAGL(80021);
AGL(1,80021)
gap> SCSeriesAGL(80039);
[ AGL(1,80039), [ [ 1, 2, 6496, 73546, 78018 ] ] ]
gap> SCSeriesAGL(80051);
[ AGL(1,80051), [ [ 1, 2, 31498, 37522, 48556 ] ] ]
gap> SCSeriesAGL(80071);
AGL(1,80071)
gap> SCSeriesAGL(80077);
[ AGL(1,80077), [ [ 1, 2, 4126, 39302, 40778 ] ] ]
```

6.4.2 SCSeriesBrehmKuehnelTorus

▷ `SCSeriesBrehmKuehnelTorus(n)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates a neighborly 3-torus with n vertices if n is odd and a centrally symmetric 3-torus if n is even ($n \geq 15$. The triangulations are taken from [BK12]

Example
```
gap> T3:=SCSeriesBrehmKuehnelTorus(15);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="Neighborly 3-Torus NT3(15)"
 Dim=3
 TopologicalType="T^3"

/SimplicialComplex]
```

```
gap> T3.Homology;
[ [ 0, [  ] ], [ 3, [  ] ], [ 3, [  ] ], [ 1, [  ] ] ]
gap> T3.Neighborliness;
2
gap> T3:=SCSeriesBrehmKuehnelTorus(16);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="Centrally symmetric 3-Torus SCT3(16)"
 Dim=3
 TopologicalType="T^3"

/SimplicialComplex]
gap> T3.Homology;
[ [ 0, [  ] ], [ 3, [  ] ], [ 3, [  ] ], [ 1, [  ] ] ]
gap> T3.IsCentrallySymmetric;
true
```

6.4.3 SCSeriesBdHandleBody

▷ `SCSeriesBdHandleBody(d, n)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

`SCSeriesBdHandleBody(d,n)` generates a transitive d-dimensional sphere bundle ($d \geq 2$) with n vertices ($n \geq 2d+3$) which coincides with the boundary of `SCSeriesHandleBody` (6.4.9)`(d,n)`. The sphere bundle is orientable if d is even or if d is odd and n is even, otherwise it is not orientable. Internally calls `SCFromDifferenceCycles` (6.1.3).

Example

```
gap> c:=SCSeriesBdHandleBody(2,7);
[SimplicialComplex

 Properties known: Dim, FacetsEx, IsOrientable, Name, TopologicalType,
                   Vertices.

 Name="Sphere bundle S^1 x S^1"
 Dim=2
 IsOrientable=true
 TopologicalType="S^1 x S^1"

/SimplicialComplex]
gap> SCLib.DetermineTopologicalType(c);
[SimplicialComplex

 Properties known: BoundaryEx, Dim, FacetsEx, HasBoundary,
                   IsOrientable, IsPseudoManifold, IsPure, Name,
                   SkelExs[], TopologicalType, Vertices.

 Name="Sphere bundle S^1 x S^1"
 Dim=2
 HasBoundary=false
```

```
 IsOrientable=true
 IsPseudoManifold=true
 IsPure=true
 TopologicalType="S^1 x S^1"

/SimplicialComplex]
gap> SCIsIsomorphic(c,SCSeriesHandleBody(3,7).Boundary);
true
```

6.4.4 SCSeriesBid

▷ `SCSeriesBid(i, d)` (function)

Returns: a simplicial complex upon success, `fail` otherwise.

Constructs the complex $B(i,d)$ as described in [KN12], cf. [Eff11a], [Spa99]. The complex $B(i,d)$ is a i-Hamiltonian subcomplex of the d-cross polytope and its boundary topologically is a sphere product $S^i \times S^{d-i-2}$ with vertex transitive automorphism group.

Example

```
gap> b26:=SCSeriesBid(2,6);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Reference, Vertices.

 Name="B(2,6)"
 Dim=5

/SimplicialComplex]
gap> s2s2:=SCBoundary(b26);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="Bd(B(2,6))"
 Dim=4

/SimplicialComplex]
gap> SCFVector(s2s2);
[ 12, 60, 160, 180, 72 ]
gap> SCAutomorphismGroup(s2s2);
TransitiveGroup(12,28) = D(4)[x]S(3)
gap> SCIsManifold(s2s2);
true
gap> SCHomology(s2s2);
[ [ 0, [  ] ], [ 0, [  ] ], [ 2, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
```

6.4.5 SCSeriesC2n

▷ `SCSeriesC2n(n)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates the combinatorial 3-manifold C_{2n}, $n \geq 8$, with $2n$ vertices from [Spr11a], Section 4.5.3 and Section 5.2. The complex is homeomorphic to $S^2 \times S^1$ for n odd and homeomorphic to $S^2 \times S^1$ in case n is an even number. In the latter case C_{2n} is isomorphic to D_{2n} from SCSeriesD2n (6.4.8). The complexes are believed to appear as the vertex links of some of the members of the series of 2-transitive 4-pseudomanifolds from SCSeriesAGL (6.4.1). Internally calls SCFromDifferenceCycles (6.1.3).

Example

```
gap> c:=SCSeriesC2n(8);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="C_16 = { (1:1:3:11),(1:1:11:3),(1:3:1:11),(2:3:2:9),(2:5:2:7) }"
 Dim=3
 TopologicalType="S^2 ~ S^1"

/SimplicialComplex]
gap> SCGenerators(c);
[ [ [ 1, 2, 3, 6 ], 32 ], [ [ 1, 2, 5, 6 ], 16 ], [ [ 1, 3, 6, 8 ], 16 ],
  [ [ 1, 3, 8, 10 ], 16 ] ]
```

Example

```
gap> c:=SCSeriesC2n(8);;
gap> d:=SCSeriesD2n(8);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="D_16 = { (1:1:1:13),(1:2:11:2),(3:4:5:4),(2:3:4:7),(2:7:4:3) }"
 Dim=3
 TopologicalType="S^2 ~ S^1"

/SimplicialComplex]
gap> SCIsIsomorphic(c,d);
true
gap> c:=SCSeriesC2n(11);;
gap> d:=SCSeriesD2n(11);;
gap> c.Homology;
[ [ 0, [  ] ], [ 1, [  ] ], [ 1, [  ] ], [ 1, [  ] ] ]
gap> d.Homology;
[ [ 0, [  ] ], [ 1, [  ] ], [ 0, [ 2 ] ], [ 0, [  ] ] ]
```

6.4.6 SCSeriesConnectedSum

▷ SCSeriesConnectedSum(k) (function)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Generates a combinatorial manifold of type $(S^2xS^1)^k$ for k even. The complex is a combinatorial 3-manifold with transitive cyclic symmetry as described in [BS14].

Example
```
gap> c:=SCSeriesConnectedSum(12);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="(S^2xS^1)^#12)"
 Dim=3
 TopologicalType="(S^2xS^1)^#12)"

/SimplicialComplex]
gap> c.Homology;
[ [ 0, [  ] ], [ 12, [  ] ], [ 12, [  ] ], [ 1, [  ] ] ]
gap> g:=SimplifiedFpGroup(SCFundamentalGroup(c));
<fp group of size infinity on the generators
[ [2,3], [2,14], [3,4], [6,7], [9,10], [10,11], [11,12], [12,13], [26,32],
  [26,34], [29,31], [33,35] ]>
gap> RelatorsOfFpGroup(g);
[  ]
```

6.4.7 SCSeriesCSTSurface

▷ `SCSeriesCSTSurface(l[, j], 2k)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

`SCSeriesCSTSurface(l,j,2k)` generates the centrally symmetric transitive (cst) surface $S_{(l,j,2k)}$, `SCSeriesCSTSurface(l,2k)` generates the cst surface $S_{(l,2k)}$ from [Spr12], Section 4.4.

Example
```
gap> SCSeriesCSTSurface(2,4,14);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name, Vertices.

 Name="cst surface S_{(2,4,14)} = { (2:4:8),(2:8:4) }"
 Dim=2

/SimplicialComplex]
gap> last.Homology;
[ [ 1, [  ] ], [ 4, [  ] ], [ 2, [  ] ] ]
gap> SCSeriesCSTSurface(2,10);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name, Vertices.

 Name="cst surface S_{(2,10)} = { (2:2:6),(3:3:4) }"
 Dim=2

/SimplicialComplex]
gap> last.Homology;
[ [ 0, [  ] ], [ 1, [ 2 ] ], [ 0, [  ] ] ]
```

6.4.8 SCSeriesD2n

▷ SCSeriesD2n(n) (function)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Generates the combinatorial 3-manifold D_{2n}, $n \geq 8$, $n \neq 9$, with $2n$ vertices from [Spr11a], Section 4.5.3 and Section 5.2. The complex is homeomorphic to $S^2 \times S^1$. In the case that n is even D_{2n} is isomorphic to C_{2n} from SCSeriesC2n (6.4.5). The complexes are believed to appear as the vertex links of some of the members of the series of 2-transitive 4-pseudomanifolds from SCSeriesAGL (6.4.1). Internally calls SCFromDifferenceCycles (6.1.3).

Example
```
gap> d:=SCSeriesD2n(15);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="D_30 = { (1:1:1:27),(1:2:25:2),(3:11:5:11),(2:3:11:14),(2:14:11:3) }"
 Dim=3
 TopologicalType="S^2 ~ S^1"

/SimplicialComplex]
gap> SCAutomorphismGroup(d);
TransitiveGroup(30,14) = t30n14
gap> StructureDescription(last);
"D60"
```

Example
```
gap> c:=SCSeriesC2n(8);;
gap> d:=SCSeriesD2n(8);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="D_16 = { (1:1:1:13),(1:2:11:2),(3:4:5:4),(2:3:4:7),(2:7:4:3) }"
 Dim=3
 TopologicalType="S^2 ~ S^1"

/SimplicialComplex]
gap> SCIsIsomorphic(c,d);
true
gap> c:=SCSeriesC2n(11);;
gap> d:=SCSeriesD2n(11);;
gap> c.Homology;
[ [ 0, [  ] ], [ 1, [  ] ], [ 1, [  ] ], [ 1, [  ] ] ]
gap> d.Homology;
[ [ 0, [  ] ], [ 1, [  ] ], [ 0, [ 2 ] ], [ 0, [  ] ] ]
```

6.4.9 SCSeriesHandleBody

▷ `SCSeriesHandleBody(d, n)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

`SCSeriesHandleBody(d,n)` generates a transitive d-dimensional handle body ($d \geq 3$) with n vertices ($n \geq 2d+1$). The handle body is orientable if d is odd or if d and n are even, otherwise it is not orientable. The complex equals the difference cycle $(1 : \ldots : 1 : n-d)$ To obtain the boundary complexes of `SCSeriesHandleBody(d,n)` use the function `SCSeriesBdHandleBody` (6.4.3). Internally calls `SCFromDifferenceCycles` (6.1.3).

Example
```
gap> c:=SCSeriesHandleBody(3,7);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, IsOrientable,
                   Name, TopologicalType, Vertices.

 Name="Handle body B^2 x S^1"
 Dim=3
 IsOrientable=true
 TopologicalType="B^2 x S^1"

/SimplicialComplex]
gap> SCAutomorphismGroup(c);
PrimitiveGroup(7,2) = D(2*7)
gap> bd:=SCBoundary(c);;
gap> SCAutomorphismGroup(bd);
PrimitiveGroup(7,4) = AGL(1, 7)
gap> SCIsIsomorphic(bd,SCSeriesBdHandleBody(2,7));
true
```

6.4.10 SCSeriesHomologySphere

▷ `SCSeriesHomologySphere(p, q, r)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates a combinatorial Brieskorn homology sphere of type $\Sigma(p,q,r)$, p, q and r pairwise coprime. The complex is a combinatorial 3-manifold with transitive cyclic symmetry as described in [BS14].

Example
```
gap> c:=SCSeriesHomologySphere(2,3,5);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="Homology sphere Sigma(2,3,5)"
 Dim=3
 TopologicalType="Sigma(2,3,5)"

/SimplicialComplex]
gap> c.Homology;
```

```
[ [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> c:=SCSeriesHomologySphere(3,4,13);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="Homology sphere Sigma(3,4,13)"
 Dim=3
 TopologicalType="Sigma(3,4,13)"

/SimplicialComplex]
gap> c.Homology;
[ [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
```

6.4.11 SCSeriesK

▷ `SCSeriesK(i, k)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates the k-th member ($k \geq 0$) of the series `K^i` ($1 \leq i \leq 396$) from [Spr11a]. The 396 series describe a complete classification of all dense series (i. e. there is a member of the series for every integer, $f_0(K^i(k+1)) = f_0(K^i(k)) + 1$) of cyclic 3-manifolds with a fixed number of difference cycles and at least one member with less than 23 vertices. See `SCSeriesL` (6.4.13) for a list of series of order 2.

Example
```
gap> cc:=List([1..10],x->SCSeriesK(x,0));;
gap> Set(List(cc,x->x.F));
[ [ 9, 36, 54, 27 ], [ 11, 55, 88, 44 ], [ 13, 65, 104, 52 ],
  [ 13, 78, 130, 65 ], [ 15, 90, 150, 75 ], [ 15, 105, 180, 90 ] ]
gap> cc:=List([1..10],x->SCSeriesK(x,10));;
gap> gap> cc:=List([1..10],x->SCSeriesK(x,10));;
gap> Set(List(cc,x->x.Homology));
[ [ [ 0, [  ] ], [ 1, [  ] ], [ 0, [ 2 ] ], [ 0, [  ] ] ] ]
gap> Set(List(cc,x->x.IsManifold));
[ true ]
```

6.4.12 SCSeriesKu

▷ `SCSeriesKu(n)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Computes the symmetric orientable sphere bundle Ku(n) with $4n$ vertices from [Spr11a], Section 4.5.2. The series is defined as a generalization of the slicings from [Spr11a], Section 3.3.

Example
```
gap> c:=SCSeriesKu(4);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.
```

```
 Name="Sl_16 = G{ [1,2,5,9],[1,2,9,10],[1,5,9,16] }"
 Dim=3

/SimplicialComplex]
gap> SCSlicing(c,[[1,2,3,4,9,10,11,12],[5,6,7,8,13,14,15,16]]);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector, Fac\
etsEx, Genus, IsConnected, IsOrientable, NSTriangulation, Name, TopologicalTyp\
e, Vertices.

 Name="slicing [ [ 1, 2, 3, 4, 9, 10, 11, 12 ], [ 5, 6, 7, 8, 13, 14, 15, 16 ]\
 ] of Sl_16 = G{ [1,2,5,9],[1,2,9,10],[1,5,9,16] }"
 Dim=2
 FVector=[ 32, 80, 32, 16 ]
 EulerCharacteristic=0
 IsOrientable=true
 TopologicalType="T^2"

/NormalSurface]
gap> Mminus:=SCSpan(c,[1,2,3,4,9,10,11,12]);;
gap> Mplus:=SCSpan(c,[5,6,7,8,13,14,15,16]);;
gap> SCCollapseGreedy(Mminus).Facets;
[ [ 1, 2 ], [ 1, 12 ], [ 2, 10 ], [ 10, 11 ], [ 11, 12 ] ]
gap> SCCollapseGreedy(Mplus).Facets;
[ [ 6, 7 ], [ 6, 14 ], [ 7, 16 ], [ 13, 14 ], [ 13, 16 ] ]
```

6.4.13 SCSeriesL

▷ `SCSeriesL(i, k)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates the `k`-th member ($k \geq 0$) of the series `L^i`, $1 \leq i \leq 18$ from [Spr11a]. The 18 series describe a complete classification of all series of cyclic 3-manifolds with a fixed number of difference cycles of order 2 (i. e. there is a member of the series for every second integer, $f_0(L^i(k+1)) = f_0(L^i(k)) + 2$) and at least one member with less than 15 vertices where each series does not appear as a sub series of one of the series K^i from `SCSeriesK` (6.4.11).

Example

```
gap> cc:=List([1..18],x->SCSeriesL(x,0));;
gap> Set(List(cc,x->x.F));
[ [ 10, 45, 70, 35 ], [ 12, 60, 96, 48 ], [ 12, 66, 108, 54 ],
  [ 14, 77, 126, 63 ], [ 14, 84, 140, 70 ], [ 14, 91, 154, 77 ] ]
gap> cc:=List([1..18],x->SCSeriesL(x,10));;
gap> Set(List(cc,x->x.IsManifold));
[ true ]
```

6.4.14 SCSeriesLe

▷ `SCSeriesLe(k)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates the k-th member ($k \geq 7$) of the series Le from [Spr11a], Section 4.5.1. The series can be constructed as the generalization of the boundary of a genus 1 handlebody decomposition of the manifold manifold_3_14_1_5 from the classification in [Lut03].

Example
```
gap> c:=SCSeriesLe(7);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name, Vertices.

 Name="Le_14 = { (1:1:1:11),(1:2:4:7),(1:4:2:7),(2:1:4:7),(2:5:2:5),(2:4:2:6) \
}"
 Dim=3

/SimplicialComplex]
gap> d:=SCLib.DetermineTopologicalType(c);;
gap> SCReference(d);
"manifold_3_14_1_5 in F.H.Lutz: 'The Manifold Page', http://www.math.tu-berlin\
.de/diskregeom/stellar/,\r\nF.H.Lutz: 'Triangulated manifolds with few vertice\
s and vertex-transitive group actions', Doctoral Thesis TU Berlin 1999, Shaker\
-Verlag, Aachen 1999"
```

6.4.15 SCSeriesLensSpace

▷ SCSeriesLensSpace(p, q) (function)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Generates the lens space $L(p,q)$ whenever $p = (k+2)^2 - 1$ and $q = k+2$ or $p = 2k+3$ and $q = 1$ for a $k \geq 0$ and fail otherwise. All complexes have a transitive cyclic automorphism group.

Example
```
gap> l154:=SCSeriesLensSpace(15,4);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="Lens space L(15,4)"
 Dim=3
 TopologicalType="L(15,4)"

/SimplicialComplex]
gap> l154.Homology;
[ [ 0, [  ] ], [ 0, [ 15 ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> g:=SimplifiedFpGroup(SCFundamentalGroup(l154));
<fp group on the generators [ [2,5] ]>
gap> StructureDescription(g);
"C15"
```

Example
```
gap> l151:=SCSeriesLensSpace(15,1);
[SimplicialComplex
```

```
 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="Lens space L(15,1)"
 Dim=3
 TopologicalType="L(15,1)"

/SimplicialComplex]
gap> l151.Homology;
[ [ 0, [  ] ], [ 0, [ 15 ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> g:=SimplifiedFpGroup(SCFundamentalGroup(l151));
<fp group on the generators [ [2,3] ]>
gap> StructureDescription(g);
"C15"
```

6.4.16 SCSeriesPrimeTorus

▷ SCSeriesPrimeTorus(l, j, p) (function)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Generates the well known triangulated torus $\{(l:j:p-l-j),(l:p-l-j:j)\}$ with p vertices, $3p$ edges and $2p$ triangles where j has to be greater than l and p must be any prime number greater than 6.

Example
```
gap> l:=List([2..19],x->SCSeriesPrimeTorus(1,x,41));;
gap> Set(List(l,x->SCHomology(x)));
[ [ [ 0, [  ] ], [ 2, [  ] ], [ 1, [  ] ] ] ]
```

6.4.17 SCSeriesSeifertFibredSpace

▷ SCSeriesSeifertFibredSpace(p, q, r) (function)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Generates a combinatorial Seifert fibred space of type

$$SFS[(\mathbb{T}^2)^{(a-1)(b-1)}:(p/a,b_1)^b,(q/b,b_2)^a,(r/ab,b_3)]$$

where p and q are co-prime, $a=\gcd(p,r)$, $b=\gcd(p,r)$, and the b_i are given by the identity

$$\frac{b_1}{p}+\frac{b_2}{q}+\frac{b_3}{r}=\frac{\pm ab}{pqr}.$$

This 3-parameter family of combinatorial 3-manifolds contains the families generated by SCSeriesHomologySphere (6.4.10), SCSeriesConnectedSum (6.4.6) and parts of SCSeriesLensSpace (6.4.15), internally calls SCIntFunc.SeifertFibredSpace(p,q,r). The complexes are combinatorial 3-manifolds with transitive cyclic symmetry as described in [BS14].

Example
```
gap> c:=SCSeriesSeifertFibredSpace(2,3,15);
[SimplicialComplex
```

```
 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="SFS [ S^2 : (2,b1)^3, (5,b3) ]"
 Dim=3
 TopologicalType="SFS [ S^2 : (2,b1)^3, (5,b3) ]"

/SimplicialComplex]
gap> c.Homology;
[ [ 0, [  ] ], [ 0, [ 2, 2 ] ], [ 0, [  ] ], [ 1, [  ] ] ]
```

6.4.18 SCSeriesS2xS2

▷ `SCSeriesS2xS2(k)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates a combinatorial version of $(S^2 \times S^2)^{\#k}$.

Example
```
gap> c:=SCSeriesS2xS2(3);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="(S^2 x S^2)^(# 3)"
 Dim=4
 TopologicalType="(S^2 x S^2)^(# 3)"

/SimplicialComplex]
gap> c.Homology;
[ [ 0, [  ] ], [ 0, [  ] ], [ 6, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
```

6.5 A census of regular and chiral maps

6.5.1 SCChiralMap

▷ `SCChiralMap(m, g)` (function)

Returns: a `SCSimplicialComplex` object upon success, `fail` otherwise.

Returns the (hyperbolic) chiral map of vertex valence m and genus g if existent and `fail` otherwise. The list was generated with the help of the classification of regular maps by Marston Conder [Con09]. Use `SCChiralMaps` (6.5.2) to get a list of all chiral maps available.

Example
```
gap> SCChiralMaps();
[ [ 7, 17 ], [ 8, 10 ], [ 8, 28 ], [ 8, 37 ], [ 8, 46 ], [ 8, 82 ],
  [ 9, 43 ], [ 10, 73 ], [ 12, 22 ], [ 12, 33 ], [ 12, 40 ], [ 12, 51 ],
  [ 12, 58 ], [ 12, 64 ], [ 12, 85 ], [ 12, 94 ], [ 12, 97 ], [ 18, 28 ] ]
gap> c:=SCChiralMap(8,10);
[SimplicialComplex
```

```
 Properties known: Dim, FacetsEx, Name, TopologicalType, Vertices.

 Name="Chiral map {8,10}"
 Dim=2
 TopologicalType="(T^2)^#10"

/SimplicialComplex]
gap> c.Homology;
[ [ 0, [  ] ], [ 20, [  ] ], [ 1, [  ] ] ]
```

6.5.2 SCChiralMaps

▷ SCChiralMaps() (function)

Returns: a list of lists upon success, fail otherwise.

Returns a list of all simplicial (hyperbolic) chiral maps of orientable genus up to 100. The list was generated with the help of the classification of regular maps by Marston Conder [Con09]. Every chiral map is given by a 2-tuple (m,g) where m is the vertex valence and g is the genus of the map. Use the 2-tuples of the list together with SCChiralMap (6.5.1) to get the corresponding triangulations.

Example
```
gap> ll:=SCChiralMaps();
[ [ 7, 17 ], [ 8, 10 ], [ 8, 28 ], [ 8, 37 ], [ 8, 46 ], [ 8, 82 ],
  [ 9, 43 ], [ 10, 73 ], [ 12, 22 ], [ 12, 33 ], [ 12, 40 ], [ 12, 51 ],
  [ 12, 58 ], [ 12, 64 ], [ 12, 85 ], [ 12, 94 ], [ 12, 97 ], [ 18, 28 ] ]
gap> c:=SCChiralMap(ll[18][1],ll[18][2]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, TopologicalType, Vertices.

 Name="Chiral map {18,28}"
 Dim=2
 TopologicalType="(T^2)^#28"

/SimplicialComplex]
gap> SCHomology(c);
[ [ 0, [  ] ], [ 56, [  ] ], [ 1, [  ] ] ]
```

6.5.3 SCChiralTori

▷ SCChiralTori(n) (function)

Returns: a SCSimplicialComplex object upon success, fail otherwise.

Returns a list of chiral triangulations of the torus with n vertices. See [BK08] for details.

Example
```
gap> cc:=SCChiralTori(91);
[ [SimplicialComplex

    Properties known: AutomorphismGroup, Dim, FacetsEx, Name,
                      TopologicalType, Vertices.
```

```
    Name="{3,6}_(9,1)"
    Dim=2
    TopologicalType="T^2"

   /SimplicialComplex], [SimplicialComplex

    Properties known: AutomorphismGroup, Dim, FacetsEx, Name,
                      TopologicalType, Vertices.

    Name="{3,6}_(6,5)"
    Dim=2
    TopologicalType="T^2"

   /SimplicialComplex] ]
gap> SCIsIsomorphic(cc[1],cc[2]);
false
```

6.5.4 SCNrChiralTori

▷ SCNrChiralTori(*n*) (function)

Returns: an integer upon success, fail otherwise.

Returns the number of simplicial chiral maps on the torus with n vertices, cf. [BK08] for details.

Example
```
gap> SCNrChiralTori(7);
1
gap> SCNrChiralTori(343);
2
```

6.5.5 SCNrRegularTorus

▷ SCNrRegularTorus(*n*) (function)

Returns: an integer upon success, fail otherwise.

Returns the number of simplicial regular maps on the torus with n vertices, cf. [BK08] for details.

Example
```
gap> SCNrRegularTorus(9);
1
gap> SCNrRegularTorus(10);
0
```

6.5.6 SCRegularMap

▷ SCRegularMap(*m, g, orient*) (function)

Returns: a SCSimplicialComplex object upon success, fail otherwise.

Returns the (hyperbolic) regular map of vertex valence *m*, genus *g* and orientability *orient* if existent and fail otherwise. The triangulations were generated with the help of the classification of regular maps by Marston Conder [Con09]. Use SCRegularMaps (6.5.7) to get a list of all regular maps available.

Example
```
gap> SCRegularMaps(){[1..10]};
[ [ 7, 3, true ], [ 7, 7, true ], [ 7, 8, false ], [ 7, 14, true ],
  [ 7, 15, false ], [ 7, 147, false ], [ 8, 3, true ], [ 8, 5, true ],
  [ 8, 8, true ], [ 8, 9, false ] ]
gap> c:=SCRegularMap(7,7,true);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, TopologicalType, Vertices.

 Name="Orientable regular map {7,7}"
 Dim=2
 TopologicalType="(T^2)^#7"

/SimplicialComplex]
gap> g:=SCAutomorphismGroup(c);
#I  group not listed
C2 x PSL(2,8)
gap> Size(g);
1008
```

6.5.7 SCRegularMaps

▷ SCRegularMaps() (function)

Returns: a list of lists upon success, fail otherwise.

Returns a list of all simplicial (hyperbolic) regular maps of orientable genus up to 100 or non-orientable genus up to 200. The list was generated with the help of the classification of regular maps by Marston Conder [Con09]. Every regular map is given by a 3-tuple (m,g,or) where m is the vertex valence, g is the genus and or is a boolean stating if the map is orientable or not. Use the 3-tuples of the list together with SCRegularMap (6.5.6) to get the corresponding triangulations. g

Example
```
gap> ll:=SCRegularMaps(){[1..10]};
[ [ 7, 3, true ], [ 7, 7, true ], [ 7, 8, false ], [ 7, 14, true ],
  [ 7, 15, false ], [ 7, 147, false ], [ 8, 3, true ], [ 8, 5, true ],
  [ 8, 8, true ], [ 8, 9, false ] ]
gap> c:=SCRegularMap(ll[5][1],ll[5][2],ll[5][3]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, TopologicalType, Vertices.

 Name="Non-orientable regular map {7,15}"
 Dim=2
 TopologicalType="(RP^2)^#15"

/SimplicialComplex]
gap> SCHomology(c);
[ [ 0, [  ] ], [ 14, [ 2 ] ], [ 0, [  ] ] ]
gap> SCGenerators(c);
```

```
[ [ [ 1, 4, 7 ], 182 ] ]
```

6.5.8 SCRegularTorus

▷ `SCRegularTorus(n)` (function)

Returns: a `SCSimplicialComplex` object upon success, `fail` otherwise.

Returns a list of regular triangulations of the torus with n vertices (the length of the list will be at most 1). See [BK08] for details.

Example
```
gap> cc:=SCRegularTorus(9);
[ [SimplicialComplex

     Properties known: AutomorphismGroup, Dim, FacetsEx, Name,
                       TopologicalType, Vertices.

     Name="{3,6}_(3,0)"
     Dim=2
     TopologicalType="T^2"

    /SimplicialComplex] ]
gap> g:=SCAutomorphismGroup(cc[1]);
Group([ (2,7)(3,4)(5,9), (1,4,2)(3,7,9)(5,8,6), (2,8,7,3,6,4)(5,9) ])
gap> SCNumFaces(cc[1],0)*12 = Size(g);
true
```

6.5.9 SCSeriesSymmetricTorus

▷ `SCSeriesSymmetricTorus(p, q)` (function)

Returns: a `SCSimplicialComplex` object upon success, `fail` otherwise.

Returns the equivarient triangulation of the torus $\{3,6\}_{(p,q)}$ with fundamental domain (p,q) on the 2-dimensional integer lattice. See [BK08] for details.

Example
```
gap> c:=SCSeriesSymmetricTorus(2,1);
[SimplicialComplex

 Properties known: AutomorphismGroup, Dim, FacetsEx, Name,
                   TopologicalType, Vertices.

 Name="{3,6}_(2,1)"
 Dim=2
 TopologicalType="T^2"

/SimplicialComplex]
gap> SCFVector(c);
[ 7, 21, 14 ]
```

See also `SCSurface` (6.3.6) for example triangulations of all compact closed surfaces with transitive cyclic automorphism group.

6.6 Generating new complexes from old

6.6.1 SCCartesianPower

▷ `SCCartesianPower(complex, n)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

The new complex is *PL*-homeomorphic to n times the cartesian product of *complex*, of dimensions $n \cdot d$ and has $f_d^n \cdot n \cdot \frac{2n-1}{2^{n-1}}!$ facets where d denotes the dimension and f_d denotes the number of facets of *complex*. Note that the complex returned by the function is not the n-fold cartesian product *complex*^{n} of *complex* (which, in general, is not simplicial) but a simplicial subdivision of *complex*^{n}.

Example
```
gap> c:=SCBdSimplex(2);;
gap> 4torus:=SCCartesianPower(c,4);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, TopologicalType, Vertices.

 Name="(S^1_3)^4"
 Dim=4
 TopologicalType="(S^1)^4"

/SimplicialComplex]
gap> 4torus.Homology;
[ [ 0, [  ] ], [ 4, [  ] ], [ 6, [  ] ], [ 4, [  ] ], [ 1, [  ] ] ]
gap> 4torus.Chi;
0
gap> 4torus.F;
[ 81, 1215, 4050, 4860, 1944 ]
```

6.6.2 SCCartesianProduct

▷ `SCCartesianProduct(complex1, complex2)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Computes the simplicial cartesian product of *complex1* and *complex2* where *complex1* and *complex2* are pure, simplicial complexes. The original vertex labeling of *complex1* and *complex2* is changed into the standard one. The new complex has vertex labels of type $[v_i, v_j]$ where v_i is a vertex of *complex1* and v_j is a vertex of *complex2*.

If n_i, $i = 1,2$, are the number facets and d_i, $i = 1,2$, are the dimensions of *complexi*, then the new complex has $n_1 \cdot n_2 \cdot \binom{d_1+d_2}{d_1}$ facets. The number of vertices of the new complex equals the product of the numbers of vertices of the arguments.

Example
```
gap> c1:=SCBdSimplex(2);;
gap> c2:=SCBdSimplex(3);;
gap> c3:=SCCartesianProduct(c1,c2);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, TopologicalType, Vertices.

 Name="S^1_3xS^2_4"
 Dim=3
```

```
 TopologicalType="S^1xS^2"

/SimplicialComplex]
gap> c3.Homology;
[ [ 0, [  ] ], [ 1, [  ] ], [ 1, [  ] ], [ 1, [  ] ] ]
gap> c3.F;
[ 12, 48, 72, 36 ]
```

6.6.3 SCConnectedComponents

▷ `SCConnectedComponents(complex)` (method)

Returns: a list of simplicial complexes of type `SCSimplicialComplex` upon success, `fail` otherwise.

Computes all connected components of an arbitrary simplicial complex.

Example
```
gap> c:=SC([[1,2,3],[3,4,5],[4,5,6,7,8]]);;
gap> SCRename(c,"connected complex");;
gap> SCConnectedComponents(c);
[ [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="Connected component #1 of connected complex"
     Dim=4

    /SimplicialComplex] ]
gap> c:=SC([[1,2,3],[4,5],[6,7,8]]);;
gap> SCRename(c,"non-connected complex");;
gap> SCConnectedComponents(c);
[ [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="Connected component #1 of non-connected complex"
     Dim=2

    /SimplicialComplex], [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="Connected component #2 of non-connected complex"
     Dim=1

    /SimplicialComplex], [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="Connected component #3 of non-connected complex"
     Dim=2
```

```
    /SimplicialComplex] ]
```

6.6.4 SCConnectedProduct

▷ SCConnectedProduct(*complex, n*) (method)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

If $n \geq 2$, the function internally calls $1\times$ SCConnectedSum (6.6.5) and $(n-2)\times$ SCConnectedSumMinus (6.6.6).

Example

```
gap> SCLib.SearchByName("T^2"){[1..6]};
[ [ 4, "T^2 (VT)" ], [ 5, "T^2 (VT)" ], [ 9, "T^2 (VT)" ], [ 10, "T^2 (VT)" ],
  [ 18, "T^2 (VT)" ], [ 20, "(T^2)#2" ] ]
gap> torus:=SCLib.Load(last[1][1]);;
gap> genus10:=SCConnectedProduct(torus,10);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="T^2 (VT)#+-T^2 (VT)#+-T^2 (VT)#+-T^2 (VT)#+-T^2 (VT)#+-T^2 (VT)#+-T^2 (\
VT)#+-T^2 (VT)#+-T^2 (VT)#+-T^2 (VT)"
 Dim=2

/SimplicialComplex]
gap> genus10.Chi;
-18
gap> genus10.F;
[ 43, 183, 122 ]
```

6.6.5 SCConnectedSum

▷ SCConnectedSum(*complex1, complex2*) (method)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

In a lexicographic ordering the smallest facet of both *complex1* and *complex2* is removed and the complexes are glued together along the resulting boundaries. The bijection used to identify the vertices of the boundaries differs from the one chosen in SCConnectedSumMinus (6.6.6) by a transposition. Thus, the topological type of SCConnectedSum is different from the one of SCConnectedSumMinus (6.6.6) whenever *complex1* and *complex2* do not allow an orientation reversing homeomorphism.

Example

```
gap> SCLib.SearchByName("T^2"){[1..6]};
[ [ 4, "T^2 (VT)" ], [ 5, "T^2 (VT)" ], [ 9, "T^2 (VT)" ], [ 10, "T^2 (VT)" ],
  [ 18, "T^2 (VT)" ], [ 20, "(T^2)#2" ] ]
gap> torus:=SCLib.Load(last[1][1]);;
gap> genus2:=SCConnectedSum(torus,torus);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="T^2 (VT)#+-T^2 (VT)"
```

```
 Dim=2

/SimplicialComplex]
gap> genus2.Homology;
[ [ 0, [  ] ], [ 4, [  ] ], [ 1, [  ] ] ]
gap> genus2.Chi;
-2
```

Example

```
gap> SCLib.SearchByName("CP^2");
[ [ 16, "CP^2 (VT)" ], [ 99, "CP^2#-CP^2" ], [ 100, "CP^2#CP^2" ],
  [ 400, "CP^2#(S^2xS^2)" ], [ 2486, "Gaifullin CP^2" ],
  [ 4401, "(S^3~S^1)#(CP^2)^{#5} (VT)" ] ]
gap> cp2:=SCLib.Load(last[1][1]);;
gap> c1:=SCConnectedSum(cp2,cp2);;
gap> c2:=SCConnectedSumMinus(cp2,cp2);;
gap> c1.F=c2.F;
true
gap> c1.ASDet=c2.ASDet;
true
gap> SCIsIsomorphic(c1,c2);
false
gap> PrintArray(SCIntersectionForm(c1));
[ [  1,  0 ],
  [  0,  1 ] ]
gap> PrintArray(SCIntersectionForm(c2));
[ [   1,   0 ],
  [   0,  -1 ] ]
```

6.6.6 SCConnectedSumMinus

▷ `SCConnectedSumMinus(complex1, complex2)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

In a lexicographic ordering the smallest facet of both *complex1* and *complex2* is removed and the complexes are glued together along the resulting boundaries. The bijection used to identify the vertices of the boundaries differs from the one chosen in `SCConnectedSum` (6.6.5) by a transposition. Thus, the topological type of `SCConnectedSumMinus` is different from the one of `SCConnectedSum` (6.6.5) whenever *complex1* and *complex2* do not allow an orientation reversing homeomorphism.

Example

```
gap> SCLib.SearchByName("T^2"){[1..6]};
[ [ 4, "T^2 (VT)" ], [ 5, "T^2 (VT)" ], [ 9, "T^2 (VT)" ], [ 10, "T^2 (VT)" ],
  [ 18, "T^2 (VT)" ], [ 20, "(T^2)#2" ] ]
gap> torus:=SCLib.Load(last[1][1]);;
gap> genus2:=SCConnectedSumMinus(torus,torus);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="T^2 (VT)#+-T^2 (VT)"
 Dim=2
```

```
/SimplicialComplex]
gap> genus2.Homology;
[ [ 0, [  ] ], [ 4, [  ] ], [ 1, [  ] ] ]
gap> genus2.Chi;
-2
```

Example

```
gap> SCLib.SearchByName("CP^2");
[ [ 16, "CP^2 (VT)" ], [ 99, "CP^2#-CP^2" ], [ 100, "CP^2#CP^2" ],
  [ 400, "CP^2#(S^2xS^2)" ], [ 2486, "Gaifullin CP^2" ],
  [ 4401, "(S^3~S^1)#(CP^2)^{#5} (VT)" ] ]
gap> cp2:=SCLib.Load(last[1][1]);;
gap> c1:=SCConnectedSum(cp2,cp2);;
gap> c2:=SCConnectedSumMinus(cp2,cp2);;
gap> c1.F=c2.F;
true
gap> c1.ASDet=c2.ASDet;
true
gap> SCIsIsomorphic(c1,c2);
false
gap> PrintArray(SCIntersectionForm(c1));
[ [  1,  0 ],
  [  0,  1 ] ]
gap> PrintArray(SCIntersectionForm(c2));
[ [   1,   0 ],
  [   0,  -1 ] ]
```

6.6.7 SCDifferenceCycleCompress

▷ `SCDifferenceCycleCompress(simplex, modulus)` (function)

Returns: list with possibly duplicate entries upon success, `fail` otherwise.

A difference cycle is returned, i. e. a list of integer values of length $(d+1)$, if d is the dimension of *simplex*, and a sum equal to *modulus*. In some sense this is the inverse operation of `SCDifferenceCycleExpand` (6.6.8).

Example

```
gap> sphere:=SCBdSimplex(4);;
gap> gens:=SCGenerators(sphere);
[ [ [ 1, 2, 3, 4 ], [ 5 ] ] ]
gap> diffcycle:=SCDifferenceCycleCompress(gens[1][1],5);
[ 1, 1, 1, 2 ]
gap> c:=SCDifferenceCycleExpand([1,1,1,2]);;
gap> c.Facets;
[ [ 1, 2, 3, 4 ], [ 1, 2, 3, 5 ], [ 1, 2, 4, 5 ], [ 1, 3, 4, 5 ],
  [ 2, 3, 4, 5 ] ]
```

6.6.8 SCDifferenceCycleExpand

▷ `SCDifferenceCycleExpand(diffcycle)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

diffcycle induces a simplex $\Delta = (v_1, \ldots, v_{n+1})$ by $v_1 =$ *diffcycle[1]*, $v_i = v_{i-1} +$ *diffcycle[i]* and a cyclic group action by $\mathbb{Z}_\sigma$ where $\sigma = \sum$ *diffcycle[i]* is the modulus of `diffcycle`. The function returns the $\mathbb{Z}_\sigma$-orbit of Δ.

Note that modulo operations in GAP are often a little bit cumbersome, since all integer ranges usually start from 1.

Example

```
gap> c:=SCDifferenceCycleExpand([1,1,2]);;
gap> c.Facets;
[ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 4 ], [ 2, 3, 4 ] ]
```

6.6.9 SCStronglyConnectedComponents

▷ `SCStronglyConnectedComponents(complex)` (method)

Returns: a list of simplicial complexes of type `SCSimplicialComplex` upon success, `fail` otherwise.

Computes all strongly connected components of a pure simplicial complex.

Example

```
gap> c:=SC([[1,2,3],[2,3,4],[4,5,6],[5,6,7]]);;
gap> comps:=SCStronglyConnectedComponents(c);
[ [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="Strongly connected component #1 of unnamed complex 82"
     Dim=2

    /SimplicialComplex], [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="Strongly connected component #2 of unnamed complex 82"
     Dim=2

    /SimplicialComplex] ]
gap> comps[1].Facets;
[ [ 1, 2, 3 ], [ 2, 3, 4 ] ]
gap> comps[2].Facets;
[ [ 4, 5, 6 ], [ 5, 6, 7 ] ]
```

6.7 Simplicial complexes from transitive permutation groups

Beginning from Version 1.3.0, simpcomp is able to generate triangulations from a prescribed transitive group action on its set of vertices. Note that the corresponding group is a subgroup of the full automorphism group, but not necessarily the full automorphism group of the triangulations obtained in this way. The methods and algorithms are based on the works of Frank H. Lutz [Lut03], [Lut] and in particular his program MANIFOLD_VT.

6.7.1 SCsFromGroupExt

▷ SCsFromGroupExt(*G, n, d, objectType, cache, removeDoubleEntries, outfile, maxLinkSize, subset*) (function)

Returns: a list of simplicial complexes of type SCSimplicialComplex upon success, fail otherwise.

Computes all combinatorial d-pseudomanifolds, $d = 2$ / all strongly connected combinatorial d-pseudomanifolds, $d \geq 3$, as a union of orbits of the group action of *G* on (d+1)-tuples on the set of *n* vertices, see [Lut03]. The integer argument *objectType* specifies, whether complexes exceeding the maximal size of each vertex link for combinatorial manifolds are sorted out (objectType = 0) or not (objectType = 1, in this case some combinatorial pseudomanifolds won't be found, but no combinatorial manifold will be sorted out). The integer argument *cache* specifies if the orbits are held in memory during the computation, a value of 0 means that the orbits are discarded, trading speed for memory, any other value means that they are kept, trading memory for speed. The boolean argument *removeDoubleEntries* specifies whether the results are checked for combinatorial isomorphism, preventing isomorphic entries. The argument *outfile* specifies an output file containing all complexes found by the algorithm, if *outfile* is anything else than a string, not output file is generated. The argument *maxLinkSize* determines a maximal link size of any output complex. If *maxLinkSize* = 0 or if *maxLinkSize* is anything else than an integer the argument is ignored. The argument *subset* specifies a set of orbits (given by a list of indices of repHigh) which have to be contained in any output complex. If *subset* is anything else than a subset of matrixAllowedRows the argument is ignored.

Example

```
gap> G:=PrimitiveGroup(8,5);
PGL(2, 7)
gap> Size(G);
336
gap> Transitivity(G);
3
gap> list:=SCsFromGroupExt(G,8,3,1,0,true,false,0,[]);
[ "defgh.g.h.fah.e.gaf.h.eag.e.faf.a.haa.g.fah.a.gjhzh" ]
gap> c:=SCFromIsoSig(list[1]);
[SimplicialComplex

 Properties known: Dim, ExportIsoSig, FacetsEx, Name, Vertices.

 Name="unnamed complex 6"
 Dim=3

/SimplicialComplex]
gap> SCNeighborliness(c);
3
gap> c.F;
[ 8, 28, 56, 28 ]
gap> c.IsManifold;
false
```

```
gap> SCLibDetermineTopologicalType(SCLink(c,1));
[SimplicialComplex

 Properties known: BoundaryEx, Dim, FacetsEx, HasBoundary,
                   IsPseudoManifold, IsPure, Name, SkelExs[],
                   Vertices.

 Name="lk([ 1 ]) in unnamed complex 6"
 Dim=2
 HasBoundary=false
 IsPseudoManifold=true
 IsPure=true

/SimplicialComplex]
gap> # there are no 3-neighborly 3-manifolds with 8 vertices
gap> list:=SCsFromGroupExt(PrimitiveGroup(8,5),8,3,0,0,true,false,0,[]);
gap> [  ]
```

6.7.2 SCsFromGroupByTransitivity

▷ SCsFromGroupByTransitivity(*n, d, k, maniflag, computeAutGroup, removeDoubleEntries*) (function)

Returns: a list of simplicial complexes of type SCSimplicialComplex upon success, fail otherwise.

Computes all combinatorial d-pseudomanifolds, $d = 2$ / all strongly connected combinatorial d-pseudomanifolds, $d \geq 3$, as union of orbits of group actions for all k-transitive groups on (d+1)-tuples on the set of n vertices, see [Lut03]. The boolean argument *maniflag* specifies, whether the resulting complexes should be listed separately by combinatorial manifolds, combinatorial pseudomanifolds and complexes where the verification that the object is at least a combinatorial pseudomanifold failed. The boolean argument *computeAutGroup* specifies whether or not the real automorphism group should be computed (note that a priori the generating group is just a subgroup of the automorphism group). The boolean argument *removeDoubleEntries* specifies whether the results are checked for combinatorial isomorphism, preventing isomorphic entries. Internally calls SCsFromGroupExt (6.7.1) for every group.

Example
```
gap> list:=SCsFromGroupByTransitivity(8,3,2,true,true,true);
#I  SCsFromGroupByTransitivity: Building list of groups...
#I  SCsFromGroupByTransitivity: ...2 groups found.
#I  degree 8: [ AGL(1, 8), PSL(2, 7) ]
#I  SCsFromGroupByTransitivity: Processing dimension 3.
#I  SCsFromGroupByTransitivity: Processing degree 8.
#I  SCsFromGroupByTransitivity: 1 / 2 groups calculated, found 0 complexes.
#I  SCsFromGroupByTransitivity: Calculating 0 automorphism and homology groups...
#I  SCsFromGroupByTransitivity: ...all automorphism groups calculated for group 1 / 2.
#I  SCsFromGroupByTransitivity: 2 / 2 groups calculated, found 1 complexes.
#I  SCsFromGroupByTransitivity: Calculating 1 automorphism and homology groups...
#I  group not listed
#I  SCsFromGroupByTransitivity: 1 / 1 automorphism groups calculated.
#I  SCsFromGroupByTransitivity: ...all automorphism groups calculated for group 2 / 2.
```

```
#I  SCsFromGroupByTransitivity: ...done dim = 3, deg =  8, 0 manifolds, 1 pseudomanifolds, 0 ca
#I  SCsFromGroupByTransitivity: ...done dim = 3.
[ [  ], [  ], [  ] ]
```

6.8 The classification of cyclic combinatorial 3-manifolds

This section contains functions to access the classification of combinatorial 3-manifolds with transitive cyclic symmetry and up to 22 vertices as presented in [Spr14].

6.8.1 SCNrCyclic3Mflds

▷ `SCNrCyclic3Mflds(i)` (function)

Returns: integer upon success, `fail` otherwise.

Returns the number of combinatorial 3-manifolds with transitive cyclic symmetry with *i* vertices. See [Spr14] for more about the classification of combinatorial 3-manifolds with transitive cyclic symmetry up to 22 vertices.

Example

```
gap> SCNrCyclic3Mflds(22);
3090
```

6.8.2 SCCyclic3MfldTopTypes

▷ `SCCyclic3MfldTopTypes(i)` (function)

Returns: a list of strings upon success, `fail` otherwise.

Returns a list of all topological types that occur in the classification combinatorial 3-manifolds with transitive cyclic symmetry with *i* vertices. See [Spr14] for more about the classification of combinatorial 3-manifolds with transitive cyclic symmetry up to 22 vertices.

Example

```
gap> SCCyclic3MfldTopTypes(19);
[ "B2", "RP^2xS^1", "SFS[RP^2:(2,1)(3,1)]", "S^2~S^1", "S^3", "Sigma(2,3,7)",
  "T^3" ]
```

6.8.3 SCCyclic3Mfld

▷ `SCCyclic3Mfld(i, j)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Returns the *j*th combinatorial 3-manifold with *i* vertices in the classification of combinatorial 3-manifolds with transitive cyclic symmetry. See [Spr14] for more about the classification of combinatorial 3-manifolds with transitive cyclic symmetry up to 22 vertices.

Example

```
gap> SCCyclic3Mfld(15,34);
[SimplicialComplex

 Properties known: AutomorphismGroupTransitivity, DifferenceCycles,
                   Dim, FacetsEx, IsManifold, Name, TopologicalType,
```

```
                     Vertices.

 Name="Cyclic 3-mfld (15,34): T^3"
 Dim=3
 AutomorphismGroupTransitivity=1
 TopologicalType="T^3"

/SimplicialComplex]
```

6.8.4 SCCyclic3MfldByType

▷ `SCCyclic3MfldByType(type)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Returns the smallest combinatorial 3-manifolds in the classification of combinatorial 3-manifolds with transitive cyclic symmetry of topological type *type*. See [Spr14] for more about the classification of combinatorial 3-manifolds with transitive cyclic symmetry up to 22 vertices.

Example

```
gap> SCCyclic3MfldByType("T^3");
[SimplicialComplex

 Properties known: AutomorphismGroupTransitivity, DifferenceCycles,
                   Dim, FacetsEx, IsManifold, Name, TopologicalType,
                   Vertices.

 Name="Cyclic 3-mfld (15,34): T^3"
 Dim=3
 AutomorphismGroupTransitivity=1
 TopologicalType="T^3"

/SimplicialComplex]
```

6.8.5 SCCyclic3MfldListOfGivenType

▷ `SCCyclic3MfldListOfGivenType(type)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Returns a list of indices $\{(i_1,j_1),(i_1,j_1),\dots(i_n,j_n)\}$ of all combinatorial 3-manifolds in the classification of combinatorial 3-manifolds with transitive cyclic symmetry of topological type *type*. Complexes can be obtained by calling `SCCyclic3Mfld` (6.8.3) using these indices. See [Spr14] for more about the classification of combinatorial 3-manifolds with transitive cyclic symmetry up to 22 vertices.

Example

```
gap> SCCyclic3MfldListOfGivenType("Sigma(2,3,7)");
[ [ 19, 100 ], [ 19, 118 ], [ 19, 120 ], [ 19, 130 ] ]
```

6.9 Computing properties of simplicial complexes

The following functions compute basic properties of simplicial complexes of type `SCSimplicialComplex`. None of these functions alter the complex. All properties are returned as immutable objects (this ensures data consistency of the cached properties of a simplicial complex). Use `ShallowCopy` or the internal simpcomp function `SCIntFunc.DeepCopy` to get a mutable copy.

Note: every simplicial complex is internally stored with the standard vertex labeling from 1 to n and a maptable to restore the original vertex labeling. Thus, we have to relabel some of the complex properties (facets, face lattice, generators, etc...) whenever we want to return them to the user. As a consequence, some of the functions exist twice, one of them with the appendix "Ex". These functions return the standard labeling whereas the other ones relabel the result to the original labeling.

6.9.1 SCAltshulerSteinberg

▷ `SCAltshulerSteinberg(complex)` (method)

Returns: a non-negative integer upon success, `fail` otherwise.

Computes the Altshuler-Steinberg determinant.

Definition: Let v_i, $1 \le i \le n$ be the vertices and let F_j, $1 \le j \le m$ be the facets of a pure simplicial complex C, then the determinant of $AS \in \mathbb{Z}^{n\times m}$, $AS_{ij} = 1$ if $v_i \in F_j$, $AS_{ij} = 0$ otherwise, is called the Altshuler-Steinberg matrix. The Altshuler-Steinberg determinant is the determinant of the quadratic matrix $AS \cdot AS^T$.

The Altshuler-Steinberg determinant is a combinatorial invariant of C and can be checked before searching for an isomorphism between two simplicial complexes.

Example
```
gap> list:=SCLib.SearchByName("T^2");;
gap> torus:=SCLib.Load(last[1][1]);;
gap> SCAltshulerSteinberg(torus);
73728
gap> c:=SCBdSimplex(3);;
gap> SCAltshulerSteinberg(c);
9
gap> c:=SCBdSimplex(4);;
gap> SCAltshulerSteinberg(c);
16
gap> c:=SCBdSimplex(5);;
gap> SCAltshulerSteinberg(c);
25
```

6.9.2 SCAutomorphismGroup

▷ `SCAutomorphismGroup(complex)` (method)

Returns: a GAP permutation group upon success, `fail` otherwise.

Computes the automorphism group of a strongly connected pseudomanifold *complex*, i. e. the group of all automorphisms on the set of vertices of *complex* that do not change the complex as a whole. Necessarily the group is a subgroup of the symmetric group S_n where n is the number of vertices of the simplicial complex.

The function uses an efficient algorithm provided by the package GRAPE (see [Soi12], which is based on the program nauty by Brendan McKay [MP14]). If the package GRAPE is not available, this function call falls back to SCAutomorphismGroupInternal (6.9.3).

The position of the group in the GAP libraries of small groups, transitive groups or primitive groups is given. If the group is not listed, its structure description, provided by the GAP function StructureDescription(), is returned as the name of the group. Note that the latter form is not always unique, since every non trivial semi-direct product is denoted by ":".

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> k3surf:=SCLib.Load(last[1][1]);;
gap> SCAutomorphismGroup(k3surf);
Group([ (1,3,8,4,9,16,15,2,14,12,6,7,13,5,10),
  (1,13)(2,14)(3,15)(4,16)(5,9)(6,10)(7,11)(8,12) ])
```

6.9.3 SCAutomorphismGroupInternal

▷ SCAutomorphismGroupInternal(*complex*) (method)

Returns: a GAP permutation group upon success, fail otherwise.

Computes the automorphism group of a strongly connected pseudomanifold *complex*, i. e. the group of all automorphisms on the set of vertices of *complex* that do not change the complex as a whole. Necessarily the group is a subgroup of the symmetric group S_n where n is the number of vertices of the simplicial complex.

The position of the group in the GAP libraries of small groups, transitive groups or primitive groups is given. If the group is not listed, its structure description, provided by the GAP function StructureDescription(), is returned as the name of the group. Note that the latter form is not always unique, since every non trivial semi-direct product is denoted by ":".

Example
```
gap> c:=SCBdSimplex(5);;
gap> SCAutomorphismGroupInternal(c);
Sym( [ 1 .. 6 ] )
```

Example
```
gap> c:=SC([[1,2],[2,3],[1,3]]);;
gap> g:=SCAutomorphismGroupInternal(c);
PrimitiveGroup(3,2) = S(3)
gap> List(g);
[ (), (1,2,3), (1,3,2), (2,3), (1,2), (1,3) ]
gap> StructureDescription(g);
"S3"
```

6.9.4 SCAutomorphismGroupSize

▷ SCAutomorphismGroupSize(*complex*) (method)

Returns: a positive integer group upon success, fail otherwise.

Computes the size of the automorphism group of a strongly connected pseudomanifold *complex*, see SCAutomorphismGroup (6.9.2).

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> k3surf:=SCLib.Load(last[1][1]);;
gap> SCAutomorphismGroupSize(k3surf);
240
```

6.9.5 SCAutomorphismGroupStructure

▷ SCAutomorphismGroupStructure(*complex*) (method)

Returns: the GAP structure description upon success, fail otherwise.

Computes the GAP structure description of the automorphism group of a strongly connected pseudomanifold *complex*, see SCAutomorphismGroup (6.9.2).

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> k3surf:=SCLib.Load(last[1][1]);;
gap> SCAutomorphismGroupStructure(k3surf);
"((C2 x C2 x C2 x C2) : C5) : C3"
```

6.9.6 SCAutomorphismGroupTransitivity

▷ SCAutomorphismGroupTransitivity(*complex*) (method)

Returns: a positive integer upon success, fail otherwise.

Computes the transitivity of the automorphism group of a strongly connected pseudomanifold *complex*, i. e. the maximal integer t such that for any two ordered t-tuples T_1 and T_2 of vertices of *complex*, there exists an element g in the automorphism group of *complex* for which $gT_1 = T_2$, see [Hup67].

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> k3surf:=SCLib.Load(last[1][1]);;
gap> SCAutomorphismGroupTransitivity(k3surf);
2
```

6.9.7 SCBoundary

▷ SCBoundary(*complex*) (method)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

The function computes the boundary of a simplicial complex *complex* satisfying the weak pseudomanifold property and returns it as a simplicial complex. In addition, it is stored as a property of *complex*.

The boundary of a simplicial complex is defined as the simplicial complex consisting of all $d-1$-faces that are contained in exactly one facet.

If *complex* does not fulfill the weak pseudomanifold property (i. e. if the valence of any $d-1$-face exceeds 2) the function returns fail.

Example

```
gap> c:=SC([[1,2,3,4],[1,2,3,5],[1,2,4,5],[1,3,4,5]]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 52"
 Dim=3

/SimplicialComplex]
gap> SCBoundary(c);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="Bd(unnamed complex 52)"
 Dim=2

/SimplicialComplex]
gap> c;
[SimplicialComplex

 Properties known: BoundaryEx, Dim, FacetsEx, HasBoundary,
                   IsPseudoManifold, IsPure, Name, SkelExs[],
                   Vertices.

 Name="unnamed complex 52"
 Dim=3
 HasBoundary=true
 IsPseudoManifold=true
 IsPure=true

/SimplicialComplex]
```

6.9.8 SCDehnSommervilleCheck

▷ `SCDehnSommervilleCheck(c)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if the simplicial complex `c` fulfills the Dehn Sommerville equations: $h_j - h_{d+1-j} = (-1)^{d+1-j}\binom{d+1}{j}(\chi(M)-2)$ for $0 \leq j \leq \frac{d}{2}$ and d even, and $h_j - h_{d+1-j} = 0$ for $0 \leq j \leq \frac{d-1}{2}$ and d odd. Where h_j is the jth component of the h-vector, see `SCHVector` (6.9.26).

Example

```
gap> c:=SCBdCrossPolytope(6);;
gap> SCDehnSommervilleCheck(c);
true
gap> c:=SC([[1,2,3],[1,4,5]]);;
gap> SCDehnSommervilleCheck(c);
false
```

6.9.9 SCDehnSommervilleMatrix

▷ `SCDehnSommervilleMatrix(d)` (method)

Returns: a `(d+1)`$\times$`Int(d+1/2)` matrix with integer entries upon success, `fail` otherwise.

Computes the coefficients of the Dehn Sommerville equations for dimension `d`: $h_j - h_{d+1-j} = (-1)^{d+1-j}\binom{d+1}{j}(\chi(M)-2)$ for $0 \leq j \leq \frac{d}{2}$ and d even, and $h_j - h_{d+1-j} = 0$ for $0 \leq j \leq \frac{d-1}{2}$ and d odd. Where h_j is the jth component of the h-vector, see `SCHVector` (6.9.26).

Example
```
gap> m:=SCDehnSommervilleMatrix(6);;
gap> PrintArray(m);
[ [    1,   -1,    1,   -1,    1,   -1,    1 ],
  [    0,   -2,    3,   -4,    5,   -6,    7 ],
  [    0,    0,    0,   -4,   10,  -20,   35 ],
  [    0,    0,    0,    0,    0,   -6,   21 ] ]
```

6.9.10 SCDifferenceCycles

▷ `SCDifferenceCycles(complex)` (method)

Returns: a list of lists upon success, `fail` otherwise.

Computes the difference cycles of *complex* in standard labeling if *complex* is invariant under a shift of the vertices of type $v \mapsto v+1 \mod n$. The function returns the difference cycles as lists where the sum of the entries equals the number of vertices n of *complex*.

Example
```
gap> torus:=SCFromDifferenceCycles([[1,2,4],[1,4,2]]);
[SimplicialComplex

 Properties known: DifferenceCycles, Dim, FacetsEx, Name, Vertices.

 Name="complex from diffcycles [ [ 1, 2, 4 ], [ 1, 4, 2 ] ]"
 Dim=2

/SimplicialComplex]
gap> torus.Homology;
[ [ 0, [  ] ], [ 2, [  ] ], [ 1, [  ] ] ]
gap> torus.DifferenceCycles;
[ [ 1, 2, 4 ], [ 1, 4, 2 ] ]
```

6.9.11 SCDim

▷ `SCDim(complex)` (method)

Returns: an integer ≥ -1 upon success, `fail` otherwise.

Computes the dimension of a simplicial complex. If the complex is not pure, the dimension of the highest dimensional simplex is returned.

Example
```
gap> complex:=SC([[1,2,3], [1,2,4], [1,3,4], [2,3,4]]);;
gap> SCDim(complex);
2
gap> c:=SC([[1], [2,4], [3,4], [5,6,7,8]]);;
```

```
gap> SCDim(c);
3
```

6.9.12 SCDualGraph

▷ SCDualGraph(*complex*) (method)

Returns: 1-dimensional simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Computes the dual graph of the pure simplicial complex *complex*.

Example
```
gap> sphere:=SCBdSimplex(5);;
gap> graph:=SCFaces(sphere,1);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 1, 6 ], [ 2, 3 ], [ 2, 4 ],
  [ 2, 5 ], [ 2, 6 ], [ 3, 4 ], [ 3, 5 ], [ 3, 6 ], [ 4, 5 ], [ 4, 6 ],
  [ 5, 6 ] ]
gap> graph:=SC(graph);;
gap> dualGraph:=SCDualGraph(sphere);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="dual graph of S^4_6"
 Dim=1

/SimplicialComplex]
gap> graph.Facets = dualGraph.Facets;
true
```

6.9.13 SCEulerCharacteristic

▷ SCEulerCharacteristic(*complex*) (method)

Returns: integer upon success, `fail` otherwise.

Computes the Euler characteristic $\chi(C) = \sum_{i=0}^{d} (-1)^i f_i$ of a simplicial complex C, where f_i denotes the i-th component of the f-vector.

Example
```
gap> complex:=SCFromFacets([[1,2,3], [1,2,4], [1,3,4], [2,3,4]]);;
gap> SCEulerCharacteristic(complex);
2
gap> s2:=SCBdSimplex(3);;
gap> s2.EulerCharacteristic;
2
```

6.9.14 SCFVector

▷ SCFVector(*complex*) (method)

Returns: a list of non-negative integers upon success, `fail` otherwise.

Computes the f-vector of the simplicial complex *complex*, i. e. the number of i-dimensional faces for $0 \leq i \leq d$, where d is the dimension of *complex*. A memory-saving implicit algorithm is used that avoids calculating the face lattice of the complex. Internally calls `SCNumFaces` (6.9.52).

Example
```
gap> complex:=SC([[1,2,3], [1,2,4], [1,3,4], [2,3,4]]);;
gap> SCFVector(complex);
[ 4, 6, 4 ]
```

6.9.15 SCFaceLattice

▷ `SCFaceLattice(complex)` (method)

Returns: a list of face lists upon success, `fail` otherwise.

Computes the entire face lattice of a d-dimensional simplicial complex, i. e. all of its i-skeletons for $0 \leq i \leq d$. The faces are returned in the original labeling.

Example
```
gap> c:=SC([["a","b","c"],["a","b","d"], ["a","c","d"], ["b","c","d"]]);;
gap> SCFaceLattice(c);
[ [ [ "a" ], [ "b" ], [ "c" ], [ "d" ] ],
  [ [ "a", "b" ], [ "a", "c" ], [ "a", "d" ], [ "b", "c" ], [ "b", "d" ],
      [ "c", "d" ] ],
  [ [ "a", "b", "c" ], [ "a", "b", "d" ], [ "a", "c", "d" ],
      [ "b", "c", "d" ] ] ]
```

6.9.16 SCFaceLatticeEx

▷ `SCFaceLatticeEx(complex)` (method)

Returns: a list of face lists upon success, `fail` otherwise.

Computes the entire face lattice of a d-dimensional simplicial complex, i. e. all of its i-skeletons for $0 \leq i \leq d$. The faces are returned in the standard labeling.

Example
```
gap> c:=SC([["a","b","c"],["a","b","d"], ["a","c","d"], ["b","c","d"]]);;
gap> SCFaceLatticeEx(c);
[ [ [ 1 ], [ 2 ], [ 3 ], [ 4 ] ],
  [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ],
  [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 4 ], [ 2, 3, 4 ] ] ]
```

6.9.17 SCFaces

▷ `SCFaces(complex, k)` (method)

Returns: a face list upon success, `fail` otherwise.

This is a synonym of the function `SCSkel` (7.3.13).

6.9.18 SCFacesEx

▷ `SCFacesEx(complex, k)` (method)

Returns: a face list upon success, `fail` otherwise.

This is a synonym of the function `SCSkelEx` (7.3.14).

6.9.19 SCFacets

▷ `SCFacets(complex)` (method)

Returns: a facet list upon success, `fail` otherwise.

Returns the facets of a simplicial complex in the original vertex labeling.

Example
```
gap> c:=SC([[2,3],[3,4],[4,2]]);;
gap> SCFacets(c);
[ [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

6.9.20 SCFacetsEx

▷ `SCFacetsEx(complex)` (method)

Returns: a facet list upon success, `fail` otherwise.

Returns the facets of a simplicial complex as they are stored, i. e. with standard vertex labeling from 1 to n.

Example
```
gap> c:=SC([[2,3],[3,4],[4,2]]);;
gap> SCFacetsEx(c);
[ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ]
```

6.9.21 SCFpBettiNumbers

▷ `SCFpBettiNumbers(complex, p)` (method)

Returns: a list of non-negative integers upon success, `fail` otherwise.

Computes the Betti numbers of a simplicial complex with respect to the field $\mathbb{F}_p$ for any prime number p.

Example
```
gap> SCLib.SearchByName("K^2");
[ [ 17, "K^2 (VT)" ], [ 571, "K^2 (VT)" ] ]
gap> kleinBottle:=SCLib.Load(last[1][1]);;
gap> SCHomology(kleinBottle);
[ [ 0, [  ] ], [ 1, [ 2 ] ], [ 0, [  ] ] ]
gap> SCFpBettiNumbers(kleinBottle,2);
[ 1, 2, 1 ]
gap> SCFpBettiNumbers(kleinBottle,3);
[ 1, 1, 0 ]
```

6.9.22 SCFundamentalGroup

▷ `SCFundamentalGroup(complex)` (method)

Returns: a GAP fp group upon success, `fail` otherwise.

Computes the first fundamental group of `complex`, which must be a connected simplicial complex, and returns it in form of a finitely presented group. The generators of the group are given as 2-tuples that correspond to the edges of `complex` in standard labeling. You can use GAP's `SimplifiedFpGroup` to simplify the group presenation.

Example
```
gap> list:=SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> c:=SCLib.Load(list[1][1]);
[SimplicialComplex

 Properties known: AltshulerSteinberg, AutomorphismGroup,
                   AutomorphismGroupSize, AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, ConnectedComponents,
                   Dim, DualGraph, EulerCharacteristic, FVector,
                   FacetsEx, GVector, GeneratorsEx, HVector,
                   HasBoundary, HasInterior, Homology, Interior,
                   IsCentrallySymmetric, IsConnected,
                   IsEulerianManifold, IsManifold, IsOrientable,
                   IsPseudoManifold, IsPure, IsStronglyConnected,
                   MinimalNonFacesEx, Name, Neighborliness,
                   NumFaces[], Orientation, Reference, SkelExs[],
                   Vertices.

 Name="RP^2 (VT)"
 Dim=2
 AltshulerSteinberg=3645
 AutomorphismGroupSize=60
 AutomorphismGroupStructure="A5"
 AutomorphismGroupTransitivity=2
 EulerCharacteristic=1
 FVector=[ 6, 15, 10 ]
 GVector=[ 2, 3 ]
 HVector=[ 3, 6, 0 ]
 HasBoundary=false
 HasInterior=true
 Homology=[ [ 0, [ ] ], [ 0, [ 2 ] ], [ 0, [ ] ] ]
 IsCentrallySymmetric=false
 IsConnected=true
 IsEulerianManifold=true
 IsOrientable=false
 IsPseudoManifold=true
 IsPure=true
 IsStronglyConnected=true
 Neighborliness=2

/SimplicialComplex]
gap> g:=SCFundamentalGroup(c);;
gap> StructureDescription(g);
"C2"
```

6.9.23 SCGVector

▷ `SCGVector(`*`complex`*`)` (method)

Returns: a list of integers upon success, `fail` otherwise.

Computes the g-vector of a simplicial complex. The *g*-vector is defined as follows:

Let h be the h-vector of a d-dimensional simplicial complex C, then $g_i := h_{i+1} - h_i$; $\frac{d}{2} \geq i \geq 0$ is called the g-vector of C. For the definition of the h-vector see SCHVector (6.9.26). The information contained in g suffices to determine the f-vector of C.

Example
```
gap> SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> rp2_6:=SCLib.Load(last[1][1]);;
gap> SCFVector(rp2_6);
[ 6, 15, 10 ]
gap> SCHVector(rp2_6);
[ 3, 6, 0 ]
gap> SCGVector(rp2_6);
[ 2, 3 ]
```

6.9.24 SCGenerators

▷ SCGenerators(*complex*) (method)

Returns: a list of pairs of the form `[ list, integer ]` upon success, `fail` otherwise.

Computes the generators of a simplicial complex in the original vertex labeling.

The generating set of a simplicial complex is a list of simplices that will generate the complex by uniting their G-orbits if G is the automorphism group of *complex*.

The function returns the simplices together with the length of their orbits.

Example
```
gap> list:=SCLib.SearchByName("T^2");;
gap> torus:=SCLib.Load(list[1][1]);;
gap> SCGenerators(torus);
[ [ [ 1, 2, 4 ], 14 ] ]
```

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> SCLib.Load(last[1][1]);
[SimplicialComplex

 Properties known: AltshulerSteinberg, AutomorphismGroup,
                   AutomorphismGroupSize, AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, ConnectedComponents,
                   Dim, DualGraph, EulerCharacteristic, FVector,
                   FacetsEx, GVector, GeneratorsEx, HVector,
                   HasBoundary, HasInterior, Homology, Interior,
                   IsCentrallySymmetric, IsConnected,
                   IsEulerianManifold, IsManifold, IsOrientable,
                   IsPseudoManifold, IsPure, IsStronglyConnected,
                   MinimalNonFacesEx, Name, Neighborliness,
                   NumFaces[], Orientation, SkelExs[], Vertices.

 Name="K3_16"
 Dim=4
 AltshulerSteinberg=883835714748069945165599539200
 AutomorphismGroupSize=240
```

```
 AutomorphismGroupStructure="((C2 x C2 x C2 x C2) : C5) : C3"
 AutomorphismGroupTransitivity=2
 EulerCharacteristic=24
 FVector=[ 16, 120, 560, 720, 288 ]
 GVector=[ 10, 55, 220 ]
 HVector=[ 11, 66, 286, -99, 23 ]
 HasBoundary=false
 HasInterior=true
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 22, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsCentrallySymmetric=false
 IsConnected=true
 IsEulerianManifold=true
 IsOrientable=true
 IsPseudoManifold=true
 IsPure=true
 IsStronglyConnected=true
 Neighborliness=3

/SimplicialComplex]
gap> SCGenerators(last);
[ [ [ 1, 2, 3, 8, 12 ], 240 ], [ [ 1, 2, 5, 8, 14 ], 48 ] ]
```

6.9.25 SCGeneratorsEx

▷ SCGeneratorsEx(*complex*) (method)

Returns: a list of pairs of the form `[ list, integer ]` upon success, `fail` otherwise.

Computes the generators of a simplicial complex in the standard vertex labeling.

The generating set of a simplicial complex is a list of simplices that will generate the complex by uniting their G-orbits if G is the automorphism group of *complex*.

The function returns the simplices together with the length of their orbits.

Example
```
gap> list:=SCLib.SearchByName("T^2");;
gap> torus:=SCLib.Load(list[1][1]);;
gap> SCGeneratorsEx(torus);
[ [ [ 1, 2, 4 ], 14 ] ]
```

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> SCLib.Load(last[1][1]);
[SimplicialComplex

 Properties known: AltshulerSteinberg, AutomorphismGroup,
                   AutomorphismGroupSize, AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, ConnectedComponents,
                   Dim, DualGraph, EulerCharacteristic, FVector,
                   FacetsEx, GVector, GeneratorsEx, HVector,
                   HasBoundary, HasInterior, Homology, Interior,
                   IsCentrallySymmetric, IsConnected,
                   IsEulerianManifold, IsManifold, IsOrientable,
```

```
                     IsPseudoManifold, IsPure, IsStronglyConnected,
                     MinimalNonFacesEx, Name, Neighborliness,
                     NumFaces[], Orientation, SkelExs[], Vertices.

 Name="K3_16"
 Dim=4
 AltshulerSteinberg=883835714748069945165599539200
 AutomorphismGroupSize=240
 AutomorphismGroupStructure="((C2 x C2 x C2 x C2) : C5) : C3"
 AutomorphismGroupTransitivity=2
 EulerCharacteristic=24
 FVector=[ 16, 120, 560, 720, 288 ]
 GVector=[ 10, 55, 220 ]
 HVector=[ 11, 66, 286, -99, 23 ]
 HasBoundary=false
 HasInterior=true
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 22, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsCentrallySymmetric=false
 IsConnected=true
 IsEulerianManifold=true
 IsOrientable=true
 IsPseudoManifold=true
 IsPure=true
 IsStronglyConnected=true
 Neighborliness=3

/SimplicialComplex]
gap> SCGeneratorsEx(last);
[ [ [ 1, 2, 3, 8, 12 ], 240 ], [ [ 1, 2, 5, 8, 14 ], 48 ] ]
```

6.9.26 SCHVector

▷ `SCHVector(complex)` (method)

Returns: a list of integers upon success, `fail` otherwise.

Computes the h-vector of a simplicial complex. The h-vector is defined as $h_k := \sum_{i=-1}^{k-1} (-1)^{k-i-1} \binom{d-i-1}{k-i-1} f_i$ for $0 \le k \le d$, where $f_{-1} := 1$. For all simplicial complexes we have $h_0 = 1$, hence the returned list starts with the second entry of the h-vector.

Example

```
gap> SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> rp2_6:=SCLib.Load(last[1][1]);;
gap> SCFVector(rp2_6);
[ 6, 15, 10 ]
gap> SCHVector(rp2_6);
[ 3, 6, 0 ]
```

6.9.27 SCHasBoundary

▷ `SCHasBoundary(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if a simplicial complex *complex* that fulfills the weak pseudo manifold property has a boundary, i. e. $d-1$-faces of valence 1. If *complex* is closed `false` is returned, if *complex* does not fulfill the weak pseudomanifold property, `fail` is returned, otherwise `true` is returned.

Example
```
gap> SCLib.SearchByName("K^2");
[ [ 17, "K^2 (VT)" ], [ 571, "K^2 (VT)" ] ]
gap> kleinBottle:=SCLib.Load(last[1][1]);;
gap> SCHasBoundary(kleinBottle);
false
```

Example
```
gap> c:=SC([[1,2,3,4],[1,2,3,5],[1,2,4,5],[1,3,4,5]]);;
gap> SCHasBoundary(c);
true
```

6.9.28 SCHasInterior

▷ `SCHasInterior(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Returns `true` if a simplicial complex *complex* that fulfills the weak pseudomanifold property has at least one $d-1$-face of valence 2, i. e. if there exist at least one $d-1$-face that is not in the boundary of *complex*, if no such face can be found `false` is returned. It *complex* does not fulfill the weak pseudomanifold property `fail` is returned.

Example
```
gap> c:=SC([[1,2,3,4],[1,2,3,5],[1,2,4,5],[1,3,4,5]]);;
gap> SCHasInterior(c)
true
gap> c:=SC([[1,2,3,4]]);;
gap> SCHasInterior(c);
false
```

6.9.29 SCHeegaardSplittingSmallGenus

▷ `SCHeegaardSplittingSmallGenus(M)` (method)

Returns: a list of an integer, a list of two sublists and a string upon success, `fail` otherwise.

Computes a Heegaard splitting of the combinatorial 3-manifold *M* of small genus. The function returns the genus of the Heegaard splitting, the vertex partition of the Heegaard splitting and information whether the splitting is minimal or just small (i. e. the Heegaard genus could not be determined). See also `SCHeegaardSplitting` (6.9.30) for a faster computation of a Heegaard splitting of arbitrary genus and `SCIsHeegaardSplitting` (6.9.40) for a test whether or not a given splitting defines a Heegaard splitting.

Example
```
gap> c:=SCSeriesBdHandleBody(3,10);;
gap> M:=SCConnectedProduct(c,3);;
gap> list:=SCHeegaardSplittingSmallGenus(M);
This creates an error
```

6.9.30 SCHeegaardSplitting

▷ `SCHeegaardSplitting(M)` (method)

Returns: a list of an integer, a list of two sublists and a string upon success, `fail` otherwise.

Computes a Heegaard splitting of the combinatorial 3-manifold `M`. The function returns the genus of the Heegaard splitting, the vertex partition of the Heegaard splitting and a note, that splitting is arbitrary and in particular possibly not minimal. See also `SCHeegaardSplittingSmallGenus` (6.9.29) for the calculation of a Heegaard splitting of small genus and `SCIsHeegaardSplitting` (6.9.40) for a test whether or not a given splitting defines a Heegaard splitting.

Example
```
gap> M:=SCSeriesBdHandleBody(3,12);;
gap> list:=SCHeegaardSplitting(M);
[ 1, [ [ 1, 2, 3, 5, 9 ], [ 4, 6, 7, 8, 10, 11, 12 ] ], "arbitrary" ]
gap> sl:=SCSlicing(M,list[2]);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector, Fac\
etsEx, Genus, IsConnected, IsOrientable, NSTriangulation, Name, TopologicalTyp\
e, Vertices.

 Name="slicing [ [ 1, 2, 3, 5, 9 ], [ 4, 6, 7, 8, 10, 11, 12 ] ] of Sphere bun\
dle S^2 x S^1"
 Dim=2
 FVector=[ 24, 55, 14, 17 ]
 EulerCharacteristic=0
 IsOrientable=true
 TopologicalType="T^2"

/NormalSurface]
```

6.9.31 SCHomologyClassic

▷ `SCHomologyClassic(complex)` (function)

Returns: a list of pairs of the form `[ integer, list ]`.

Computes the integral simplicial homology groups of a simplicial complex `complex` (internally calls the function `SimplicialHomology(complex.FacetsEx)` from the homology package, see [DHSW11]).

If the homology package is not available, this function call falls back to `SCHomologyInternal` (8.1.5). The output is a list of homology groups of the form $[H_0,....,H_d]$, where d is the dimension of `complex`. The format of the homology groups H_i is given in terms of their maximal cyclic subgroups, i.e. a homology group $H_i \cong \mathbb{Z}^f + \mathbb{Z}/t_1\mathbb{Z} \times \dots \times \mathbb{Z}/t_n\mathbb{Z}$ is returned in form of a list $[f,[t_1,...,t_n]]$, where f is the (integer) free part of H_i and t_i denotes the torsion parts of H_i ordered in weakly increasing size.

Example

```
gap> SCLib.SearchByName("K^2");
[ [ 17, "K^2 (VT)" ], [ 571, "K^2 (VT)" ] ]
gap> kleinBottle:=SCLib.Load(last[1][1]);;
gap> kleinBottle.Homology;
[ [ 0, [  ] ], [ 1, [ 2 ] ], [ 0, [  ] ] ]
gap> SCLib.SearchByName("L_"){[1..10]};
[ [ 139, "L_3_1" ], [ 634, "L_4_1" ], [ 754, "L_5_2" ],
  [ 2416, "(S^2~S^1)#L_3_1" ], [ 2417, "(S^2xS^1)#L_3_1" ], [ 2490, "L_5_1" ],
  [ 2492, "(S^2~S^1)#2#L_3_1" ], [ 2494, "(S^2xS^1)#2#L_3_1" ],
  [ 7467, "L_7_2" ], [ 7468, "L_8_3" ] ]
gap> c:=SCConnectedSum(SCLib.Load(last[9][1]),
                       SCConnectedProduct(SCLib.Load(last[10][1]),2));
> [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="L_7_2#+-L_8_3#+-L_8_3"
 Dim=3

/SimplicialComplex]
gap> SCHomology(c);
[ [ 0, [  ] ], [ 0, [ 8, 56 ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> SCFpBettiNumbers(c,2);
[ 1, 2, 2, 1 ]
gap> SCFpBettiNumbers(c,3);
[ 1, 0, 0, 1 ]
```

6.9.32 SCIncidences

▷ SCIncidences(*complex, k*) (method)

Returns: a list of face lists upon success, `fail` otherwise.

Returns a list of all `k`-faces of the simplicial complex `complex`. The list is sorted by the valence of the faces in the `k`+1-skeleton of the complex, i. e. the *i*-th entry of the list contains all `k`-faces of valence *i*. The faces are returned in the original labeling.

Example

```
gap> c:=SC([[1,2,3],[2,3,4],[3,4,5],[4,5,6],[1,5,6],[1,4,6],[2,3,6]]);;
gap> SCIncidences(c,1);
[ [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 4 ], [ 2, 6 ], [ 3, 5 ],
      [ 3, 6 ] ], [ [ 1, 6 ], [ 3, 4 ], [ 4, 5 ], [ 4, 6 ], [ 5, 6 ] ],
  [ [ 2, 3 ] ] ]
```

6.9.33 SCIncidencesEx

▷ SCIncidencesEx(*complex, k*) (method)

Returns: a list of face lists upon success, `fail` otherwise.

Returns a list of all `k`-faces of the simplicial complex `complex`. The list is sorted by the valence of the faces in the `k`+1-skeleton of the complex, i. e. the *i*-th entry of the list contains all `k`-faces of valence *i*. The faces are returned in the standard labeling.

Example
```
gap> c:=SC([[1,2,3],[2,3,4],[3,4,5],[4,5,6],[1,5,6],[1,4,6],[2,3,6]]);;
gap> SCIncidences(c,1);
[ [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 2, 4 ], [ 2, 6 ], [ 3, 5 ],
      [ 3, 6 ] ], [ [ 1, 6 ], [ 3, 4 ], [ 4, 5 ], [ 4, 6 ], [ 5, 6 ] ],
  [ [ 2, 3 ] ] ]
```

6.9.34 SCInterior

▷ `SCInterior(complex)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Computes all $d-1$-faces of valence 2 of a simplicial complex *complex* that fulfills the weak pseudomanifold property, i. e. the function returns the part of the $d-1$-skeleton of C that is not part of the boundary.

Example
```
gap> c:=SC([[1,2,3,4],[1,2,3,5],[1,2,4,5],[1,3,4,5]]);;
gap> SCInterior(c).Facets;
[ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 2, 5 ], [ 1, 3, 4 ], [ 1, 3, 5 ],
  [ 1, 4, 5 ] ]
gap> c:=SC([[1,2,3,4]]);;
gap> SCInterior(c).Facets;
[  ]
```

6.9.35 SCIsCentrallySymmetric

▷ `SCIsCentrallySymmetric(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if a simplicial complex *complex* is centrally symmetric, i. e. if its automorphism group contains a fixed point free involution.

Example
```
gap> c:=SCBdCrossPolytope(4);;
gap> SCIsCentrallySymmetric(c);
true
```

Example
```
gap> c:=SCBdSimplex(4);;
gap> SCIsCentrallySymmetric(c);
false
```

6.9.36 SCIsConnected

▷ `SCIsConnected(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if a simplicial complex *complex* is connected.

Example

```
gap> c:=SCBdSimplex(1);;
gap> SCIsConnected(c);
false
gap> c:=SCBdSimplex(2);;
gap> SCIsConnected(c);
true
```

6.9.37 SCIsEmpty

▷ `SCIsEmpty(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if a simplicial complex *complex* is the empty complex, i. e. a `SCSimplicialComplex` object with empty facet list.

Example

```
gap> c:=SC([[1]]);;
gap> SCIsEmpty(c);
false
gap> c:=SC([]);;
gap> SCIsEmpty(c);
true
gap> c:=SC([[]]);;
gap> SCIsEmpty(c);
true
```

6.9.38 SCIsEulerianManifold

▷ `SCIsEulerianManifold(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks whether a given simplicial complex *complex* is a Eulerian manifold or not, i. e. checks if all vertex links of *complex* have the Euler characteristic of a sphere. In particular the function returns `false` in case *complex* has a non-empty boundary.

Example

```
gap> c:=SCBdSimplex(4);;
gap> SCIsEulerianManifold(c);
true
gap> SCLib.SearchByName("Moebius");
[ [ 1, "Moebius Strip" ] ]
gap> moebius:=SCLib.Load(last[1][1]); # a moebius strip
[SimplicialComplex

 Properties known: Dim, EulerCharacteristic, FVector, FacetsEx,
                   GVector, HVector, HasBoundary, Homology,
                   IsConnected, IsManifold, IsPseudoManifold,
                   MinimalNonFacesEx, Name, NumFaces[], SkelExs[],
                   Vertices.

 Name="Moebius Strip"
 Dim=2
```

```
 EulerCharacteristic=0
 FVector=[ 5, 10, 5 ]
 GVector=[ 1, 1 ]
 HVector=[ 2, 3, -1 ]
 HasBoundary=true
 Homology=[ [ 0 ], [ 1 ], [ 0 ] ]
 IsConnected=true
 IsPseudoManifold=true

/SimplicialComplex]
gap> SCIsEulerianManifold(moebius);
false
```

6.9.39 SCIsFlag

▷ `SCIsFlag(complex, k)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if *complex* is flag. A simplicial complex is a flag complex if all edges of a potential face of the complex are in the complex, or equivalently if all of its minimal non-faces are edges (cf. [Fro08]).

Example
```
gap> SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> rp2_6:=SCLib.Load(last[1][1]);;
gap> SCIsFlag(rp2_6);
true
```

6.9.40 SCIsHeegaardSplitting

▷ `SCIsHeegaardSplitting(c, list)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks whether *list* defines a Heegaard splitting of *c* or not. See also `SCHeegaardSplitting` (6.9.30) and `SCHeegaardSplittingSmallGenus` (6.9.29) for functions to compute Heegaard splittings.

Example
```
gap> c:=SCSeriesBdHandleBody(3,9);;
gap> list:=[[1..3],[4..9]];
[ [ 1 .. 3 ], [ 4 .. 9 ] ]
gap> SCIsHeegaardSplitting(c,list);
false
gap> splitting:=SCHeegaardSplitting(c);
[ 1, [ [ 1, 2, 3, 6 ], [ 4, 5, 7, 8, 9 ] ], "arbitrary" ]
gap> SCIsHeegaardSplitting(c,splitting[2]);
true
```

6.9.41 SCIsHomologySphere

▷ `SCIsHomologySphere(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks whether a simplicial complex *complex* is a homology sphere, i. e. has the homology of a sphere, or not.

Example
```
gap> c:=SC([[2,3],[3,4],[4,2]]);;
gap> SCIsHomologySphere(c);
true
```

6.9.42 SCIsInKd

▷ `SCIsInKd(complex, k)` (method)

Returns: `true` / `false` upon success, `fail` otherwise.

Checks whether the simplicial complex *complex* that must be a combinatorial d-manifold is in the class $\mathcal{K}^k(d)$, $1 \leq k \leq \lfloor \frac{d+1}{2} \rfloor$, of simplicial complexes that only have k-stacked spheres as vertex links, see [Eff11b]. Note that it is not checked whether *complex* is a combinatorial manifold – if not, the algorithm will not succeed. Returns `true` / `false` upon success. If `true` is returned this means that *complex* is at least *k*-stacked and thus that the complex is in the class $\mathcal{K}^k(d)$, i.e. all vertex links are `i`-stacked spheres. If `false` is returnd the complex cannot be *k*-stacked. In some cases the question can not be decided. In this case `fail` is returned.

Internally calls `SCIsKStackedSphere` (9.2.5) for all links. Please note that this is a radomized algorithm that may give an indefinite answer to the membership problem.

Example
```
gap> list:=SCLib.SearchByName("S^2~S^1");;{[1..3]};
gap> c:=SCLib.Load(list[1][1]);;
gap> c.AutomorphismGroup;
Group([ (1,3)(4,9)(5,8)(6,7), (1,9,8,7,6,5,4,3,2) ])
gap> SCIsInKd(c,1);
#I  SCIsKStackedSphere: checking if complex is a 1-stacked sphere...
#I  SCIsKStackedSphere: try 1/50
#I  SCIsKStackedSphere: complex is a 1-stacked sphere.
true
```

6.9.43 SCIsKNeighborly

▷ `SCIsKNeighborly(complex, k)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Example
```
gap> SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> rp2_6:=SCLib.Load(last[1][1]);;
gap> SCFVector(rp2_6);
[ 6, 15, 10 ]
gap> SCIsKNeighborly(rp2_6,2);
true
gap> SCIsKNeighborly(rp2_6,3);
```

```
false
```

6.9.44 SCIsOrientable

▷ `SCIsOrientable(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if a simplicial complex *complex*, satisfying the weak pseudomanifold property, is orientable.

Example

```
gap> c:=SCBdCrossPolytope(4);;
gap> SCIsOrientable(c);
true
```

6.9.45 SCIsPseudoManifold

▷ `SCIsPseudoManifold(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if a simplicial complex *complex* fulfills the weak pseudomanifold property, i. e. if every $d-1$-face of *complex* is contained in at most 2 facets.

Example

```
gap> c:=SC([[1,2,3],[1,2,4],[1,3,4],[2,3,4],[1,5,6],[1,5,7],[1,6,7],[5,6,7]]);;
gap> SCIsPseudoManifold(c);
true
gap> c:=SC([[1,2],[2,3],[3,1],[1,4],[4,5],[5,1]]);;
gap> SCIsPseudoManifold(c);
false
```

6.9.46 SCIsPure

▷ `SCIsPure(complex)` (method)

Returns: a boolean upon success, `fail` otherwise.

Checks if a simplicial complex *complex* is pure.

Example

```
gap> c:=SC([[1,2], [1,4], [2,4], [2,3,4]]);;
gap> SCIsPure(c);
false
gap> c:=SC([[1,2], [1,4], [2,4]]);;
gap> SCIsPure(c);
true
```

6.9.47 SCIsShellable

▷ `SCIsShellable(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

The simplicial complex *complex* must be pure, strongly connected and must fulfill the weak pseudomanifold property with non-empty boundary (cf. SCBoundary (6.9.7)).

The function checks whether *complex* is shellable or not. An ordering $(F_1, F_2, \ldots, F_r)$ on the facet list of a simplicial complex is called a shelling if and only if $F_i \cap (F_1 \cup \ldots \cup F_{i-1})$ is a pure simplicial complex of dimension $d-1$ for all $i = 1, \ldots, r$. A simplicial complex is called shellable, if at least one shelling exists.

See [Zie95], [Pac87] to learn more about shellings.

Example

```
gap> c:=SCBdCrossPolytope(4);;
gap> c:=Difference(c,SC([[1,3,5,7]]));; # bounded version
gap> SCIsShellable(c);
true
```

6.9.48 SCIsStronglyConnected

▷ SCIsStronglyConnected(*complex*) (method)

Returns: true or false upon success, fail otherwise.

Checks if a simplicial complex *complex* is strongly connected, i. e. if for any pair of facets $(\hat{\Delta}, \tilde{\Delta})$ there exists a sequence of facets $(\Delta_1, \ldots, \Delta_k)$ with $\Delta_1 = \hat{\Delta}$ and $\Delta_k = \tilde{\Delta}$ and $\dim(\Delta_i, \Delta_{i+1}) = d-1$ for all $1 \leq i \leq k-1$.

Example

```
gap> c:=SC([[1,2,3],[1,2,4],[1,3,4],[2,3,4], [1,5,6],[1,5,7],[1,6,7],[5,6,7]]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 24"
 Dim=2

/SimplicialComplex]
gap> SCIsConnected(c);
true
gap> SCIsStronglyConnected(c);
false
```

6.9.49 SCMinimalNonFaces

▷ SCMinimalNonFaces(*complex*) (method)

Returns: a list of face lists upon success, fail otherwise.

Computes all missing proper faces of a simplicial complex *complex* by calling SCMinimalNonFacesEx (6.9.50). The simplices are returned in the original labeling of *complex*.

Example

```
gap> c:=SCFromFacets(["abc","abd"]);;
gap> SCMinimalNonFaces(c);
[ [  ], [ "cd" ] ]
```

6.9.50 SCMinimalNonFacesEx

▷ `SCMinimalNonFacesEx(complex)` (method)

Returns: a list of face lists upon success, `fail` otherwise.

Computes all missing proper faces of a simplicial complex *complex*, i.e. the missing $(i+1)$-tuples in the i-dimensional skeleton of a *complex*. A missing $i+1$-tuple is not listed if it only consists of missing i-tuples. Note that whenever *complex* is k-neighborly the first $k+1$ entries are empty. The simplices are returned in the standard labeling $1,\ldots,n$, where n is the number of vertices of *complex*.

Example
```
gap> SCLib.SearchByName("T^2"){[1..10]};
[ [ 4, "T^2 (VT)" ], [ 5, "T^2 (VT)" ], [ 9, "T^2 (VT)" ], [ 10, "T^2 (VT)" ],
  [ 18, "T^2 (VT)" ], [ 20, "(T^2)#2" ], [ 24, "(T^2)#3" ],
  [ 41, "T^2 (VT)" ], [ 44, "(T^2)#4" ], [ 65, "T^2 (VT)" ] ]
gap> torus:=SCLib.Load(last[1][1]);;
gap> SCFVector(torus);
[ 7, 21, 14 ]
gap> SCMinimalNonFacesEx(torus);
[ [  ], [  ] ]
gap> SCMinimalNonFacesEx(SCBdCrossPolytope(4));
[ [  ], [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ], [ 7, 8 ] ], [  ] ]
```

6.9.51 SCNeighborliness

▷ `SCNeighborliness(complex)` (method)

Returns: a positive integer upon success, `fail` otherwise.

Returns k if a simplicial complex *complex* is k-neighborly but not $(k+1)$-neighborly. See also `SCIsKNeighborly` (6.9.43).

Note that every complex is at least 1-neighborly.

Example
```
gap> c:=SCBdSimplex(4);;
gap> SCNeighborliness(c);
4
gap> c:=SCBdCrossPolytope(4);;
gap> SCNeighborliness(c);
1
gap> SCLib.SearchByAttribute("F[3]=Binomial(F[1],3) and Dim=4");
[ [ 16, "CP^2 (VT)" ], [ 7648, "K3_16" ] ]
gap> cp2:=SCLib.Load(last[2][1]);;
gap> SCNeighborliness(cp2);
3
```

6.9.52 SCNumFaces

▷ `SCNumFaces(complex[, i])` (method)

Returns: an integer or a list of integers upon success, `fail` otherwise.

If `i` is not specified the function computes the f-vector of the simplicial complex *complex* (cf. `SCFVector` (7.3.4)). If the optional integer parameter `i` is passed, only the `i`-th position of the f-

vector of *complex* is calculated. In any case a memory-saving implicit algorithm is used that avoids calculating the face lattice of the complex.

Example
```
gap> complex:=SC([[1,2,3], [1,2,4], [1,3,4], [2,3,4]]);;
gap> SCNumFaces(complex,1);
6
```

6.9.53 SCOrientation

▷ `SCOrientation(complex)` (method)

Returns: a list of the type $\{\pm 1\}^{f_d}$ or `[ ]` upon success, `fail` otherwise.

This function tries to compute an orientation of a pure simplicial complex *complex* that fulfills the weak pseudomanifold property. If *complex* is orientable, an orientation in form of a list of orientations for the facets of *complex* is returned, otherwise an empty set.

Example
```
gap> c:=SCBdCrossPolytope(4);;
gap> SCOrientation(c);
[ 1, -1, -1, 1, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1, -1, 1 ]
```

6.9.54 SCSkel

▷ `SCSkel(complex, k)` (method)

Returns: a face list or a list of face lists upon success, `fail` otherwise.

If *k* is an integer, the *k*-skeleton of a simplicial complex *complex*, i. e. all *k*-faces of *complex*, is computed. If *k* is a list, a list of all `k[i]`-faces of *complex* for each entry `k[i]` (which has to be an integer) is returned. The faces are returned in the original labeling.

Example
```
gap> SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> rp2_6:=SCLib.Load(last[1][1]);;
gap> rp2_6:=SC(rp2_6.Facets+10);;
gap> SCSkelEx(rp2_6,1);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 1, 6 ], [ 2, 3 ], [ 2, 4 ],
  [ 2, 5 ], [ 2, 6 ], [ 3, 4 ], [ 3, 5 ], [ 3, 6 ], [ 4, 5 ], [ 4, 6 ],
  [ 5, 6 ] ]
gap> SCSkel(rp2_6,1);
[ [ 11, 12 ], [ 11, 13 ], [ 11, 14 ], [ 11, 15 ], [ 11, 16 ], [ 12, 13 ],
  [ 12, 14 ], [ 12, 15 ], [ 12, 16 ], [ 13, 14 ], [ 13, 15 ], [ 13, 16 ],
  [ 14, 15 ], [ 14, 16 ], [ 15, 16 ] ]
```

6.9.55 SCSkelEx

▷ `SCSkelEx(complex, k)` (method)

Returns: a face list or a list of face lists upon success, `fail` otherwise.

If *k* is an integer, the *k*-skeleton of a simplicial complex *complex*, i. e. all *k*-faces of *complex*, is computed. If *k* is a list, a list of all `k[i]`-faces of *complex* for each entry `k[i]` (which has to be an integer) is returned. The faces are returned in the standard labeling.

Example
```
gap> SCLib.SearchByName("RP^2");
[ [ 3, "RP^2 (VT)" ], [ 635, "RP^2xS^1" ] ]
gap> rp2_6:=SCLib.Load(last[1][1]);;
gap> rp2_6:=SC(rp2_6.Facets+10);;
gap> SCSkelEx(rp2_6,1);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 1, 6 ], [ 2, 3 ], [ 2, 4 ],
  [ 2, 5 ], [ 2, 6 ], [ 3, 4 ], [ 3, 5 ], [ 3, 6 ], [ 4, 5 ], [ 4, 6 ],
  [ 5, 6 ] ]
gap> SCSkel(rp2_6,1);
[ [ 11, 12 ], [ 11, 13 ], [ 11, 14 ], [ 11, 15 ], [ 11, 16 ], [ 12, 13 ],
  [ 12, 14 ], [ 12, 15 ], [ 12, 16 ], [ 13, 14 ], [ 13, 15 ], [ 13, 16 ],
  [ 14, 15 ], [ 14, 16 ], [ 15, 16 ] ]
```

6.9.56 SCSpanningTree

▷ `SCSpanningTree(`*`complex`*`)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Computes a spanning tree of a connected simplicial complex *`complex`* using a greedy algorithm.

Example
```
gap> c:=SC([["a","b","c"],["a","b","d"], ["a","c","d"], ["b","c","d"]]);;
gap> s:=SCSpanningTree(c);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="spanning tree of unnamed complex 1"
 Dim=1

/SimplicialComplex]
gap> s.Facets;
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ] ]
```

6.10 Operations on simplicial complexes

The following functions perform operations on simplicial complexes. Most of them return simplicial complexes. Thus, this section is closely related to the Sections 6.6 ”Generate new complexes from old”. However, the data generated here is rather seen as an intrinsic attribute of the original complex and not as an independent complex.

6.10.1 SCAlexanderDual

▷ `SCAlexanderDual(`*`complex`*`)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

The Alexander dual of a simplicial complex *`complex`* with set of vertices V is the simplicial complex where any subset of V spans a face if and only if its complement in V is a non-face of *`complex`*.

Example
```
gap> c:=SC([[1,2],[2,3],[3,4],[4,1]]);;
gap> dual:=SCAlexanderDual(c);;
gap> dual.F;
[ 4, 2 ]
gap> dual.IsConnected;
false
gap> dual.Facets;
[ [ 1, 3 ], [ 2, 4 ] ]
```

6.10.2 SCClose

▷ `SCClose(complex[, apex])` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Closes a simplicial complex *complex* by building a cone over its boundary. If *apex* is specified it is assigned to the apex of the cone and the original vertex labeling of *complex* is preserved, otherwise an arbitrary vertex label is chosen and *complex* is returned in the standard labeling.

Example
```
gap> s:=SCSimplex(5);;
gap> b:=SCSimplex(5);;
gap> s:=SCClose(b,13);;
gap> SCIsIsomorphic(s,SCBdSimplex(6));
true
```

6.10.3 SCCone

▷ `SCCone(complex, apex)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

If the second argument is passed every facet of the simplicial complex *complex* is united with *apex*. If not, an arbitrary vertex label v is used (which is not a vertex of *complex*). In the first case the vertex labeling remains unchanged. In the second case the function returns the new complex in the standard vertex labeling from 1 to $n+1$ and the apex of the cone is $n+1$.

If called with a facet list instead of a `SCSimplicialComplex` object and *apex* is not specified, internally falls back to the homology package [DHSW11], if available.

Example
```
gap> SCLib.SearchByName("RP^3");
[ [ 45, "RP^3" ], [ 113, "RP^3=L(2,1) (VT)" ], [ 589, "(S^2~S^1)#RP^3" ],
  [ 590, "(S^2xS^1)#RP^3" ], [ 632, "(S^2~S^1)#2#RP^3" ],
  [ 633, "(S^2xS^1)#2#RP^3" ], [ 2414, "RP^3#RP^3" ],
  [ 2426, "RP^3=L(2,1) (VT)" ], [ 2488, "(S^2~S^1)#3#RP^3" ],
  [ 2489, "(S^2xS^1)#3#RP^3" ], [ 2502, "RP^3=L(2,1) (VT)" ],
  [ 7473, "(S^2~S^1)#4#RP^3" ], [ 7474, "(S^2xS^1)#4#RP^3" ],
  [ 7504, "(S^2~S^1)#5#RP^3" ], [ 7505, "(S^2xS^1)#5#RP^3" ] ]
gap> rp3:=SCLib.Load(last[1][1]);;
gap> rp3.F;
[ 11, 51, 80, 40 ]
gap> cone:=SCCone(rp3);;
gap> cone.F;
```

```
[ 12, 62, 131, 120, 40 ]
```

Example

```
gap> s:=SCBdSimplex(4)+12;;
gap> s.Facets;
[ [ 13, 14, 15, 16 ], [ 13, 14, 15, 17 ], [ 13, 14, 16, 17 ],
  [ 13, 15, 16, 17 ], [ 14, 15, 16, 17 ] ]
gap> cc:=SCCone(s,13);;
gap> cc:=SCCone(s,12);;
gap> cc.Facets;
[ [ 12, 13, 14, 15, 16 ], [ 12, 13, 14, 15, 17 ], [ 12, 13, 14, 16, 17 ],
  [ 12, 13, 15, 16, 17 ], [ 12, 14, 15, 16, 17 ] ]
```

6.10.4 SCDeletedJoin

▷ `SCDeletedJoin(complex1, complex2)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Calculates the simplicial deleted join of the simplicial complexes *complex1* and *complex2*. If called with a facet list instead of a `SCSimplicialComplex` object, the function internally falls back to the homology package [DHSW11], if available.

Example

```
gap> deljoin:=SCDeletedJoin(SCBdSimplex(3),SCBdSimplex(3));
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="S^2_4 deljoin S^2_4"
 Dim=3

/SimplicialComplex]
gap> bddeljoin:=SCBoundary(deljoin);;
gap> bddeljoin.Homology;
[ [ 1, [  ] ], [ 0, [  ] ], [ 2, [  ] ] ]
gap> deljoin.Facets;
[ [ [ 1, 1 ], [ 2, 1 ], [ 3, 1 ], [ 4, 2 ] ],
  [ [ 1, 1 ], [ 2, 1 ], [ 3, 2 ], [ 4, 1 ] ],
  [ [ 1, 1 ], [ 2, 1 ], [ 3, 2 ], [ 4, 2 ] ],
  [ [ 1, 1 ], [ 2, 2 ], [ 3, 1 ], [ 4, 1 ] ],
  [ [ 1, 1 ], [ 2, 2 ], [ 3, 1 ], [ 4, 2 ] ],
  [ [ 1, 1 ], [ 2, 2 ], [ 3, 2 ], [ 4, 1 ] ],
  [ [ 1, 1 ], [ 2, 2 ], [ 3, 2 ], [ 4, 2 ] ],
  [ [ 1, 2 ], [ 2, 1 ], [ 3, 1 ], [ 4, 1 ] ],
  [ [ 1, 2 ], [ 2, 1 ], [ 3, 1 ], [ 4, 2 ] ],
  [ [ 1, 2 ], [ 2, 1 ], [ 3, 2 ], [ 4, 1 ] ],
  [ [ 1, 2 ], [ 2, 1 ], [ 3, 2 ], [ 4, 2 ] ],
  [ [ 1, 2 ], [ 2, 2 ], [ 3, 1 ], [ 4, 1 ] ],
  [ [ 1, 2 ], [ 2, 2 ], [ 3, 1 ], [ 4, 2 ] ],
  [ [ 1, 2 ], [ 2, 2 ], [ 3, 2 ], [ 4, 1 ] ] ]
```

6.10.5 SCDifference

▷ `SCDifference(complex1, complex2)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Forms the "difference" of two simplicial complexes *complex1* and *complex2* as the simplicial complex formed by the difference of the face lattices of *complex1* minus *complex2*. The two arguments are not altered. Note: for the difference process the vertex labelings of the complexes are taken into account, see also `Operation Difference (SCSimplicialComplex, SCSimplicialComplex)` (5.3.2).

Example

```
gap> c:=SCBdSimplex(3);;
gap> d:=SC([[1,2,3]]);;
gap> disc:=SCDifference(c,d);;
gap> disc.Facets;
[ [ 1, 2, 4 ], [ 1, 3, 4 ], [ 2, 3, 4 ] ]
gap> empty:=SCDifference(d,c);;
gap> empty.Dim;
-1
```

6.10.6 SCFillSphere

▷ `SCFillSphere(complex[, vertex])` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise .

Fills the given simplicial sphere *complex* by forming the suspension of the anti star of *vertex* over *vertex*. This is a triangulated $(d+1)$-ball with the boundary *complex*, see [BD08a]. If the optional argument *vertex* is not supplied, the first vertex of *complex* is chosen.

Note that it is not checked whether *complex* really is a simplicial sphere – this has to be done by the user!

Example

```
gap> SCLib.SearchByName("S^4");
[ [ 36, "S^4 (VT)" ], [ 37, "S^4 (VT)" ], [ 38, "S^4 (VT)" ],
  [ 130, "S^4 (VT)" ], [ 463, "S^4~S^1 (VT)" ], [ 713, "S^4xS^1 (VT)" ],
  [ 1472, "S^4xS^1 (VT)" ], [ 1473, "S^4~S^1 (VT)" ],
  [ 1474, "S^4~S^1 (VT)" ], [ 1475, "S^4xS^1 (VT)" ],
  [ 2477, "S^4~S^1 (VT)" ], [ 2478, "S^4 (VT)" ], [ 3435, "S^4 (VT)" ],
  [ 4395, "S^4~S^1 (VT)" ], [ 4396, "S^4~S^1 (VT)" ],
  [ 4397, "S^4~S^1 (VT)" ], [ 4398, "S^4~S^1 (VT)" ],
  [ 4399, "S^4~S^1 (VT)" ], [ 4402, "S^4~S^1 (VT)" ],
  [ 4403, "S^4~S^1 (VT)" ], [ 4404, "S^4~S^1 (VT)" ], [ 7479, "S^4xS^2" ],
  [ 7539, "S^4xS^3" ], [ 7573, "S^4xS^4" ] ]
gap> s:=SCLib.Load(last[1][1]);;
gap> filled:=SCFillSphere(s);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="FilledSphere(S^4 (VT)) at vertex [ 1 ]"
 Dim=5

/SimplicialComplex]
```

```
gap> SCHomology(filled);
[ [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ],
  [ 0, [  ] ] ]
gap> SCCollapseGreedy(filled);
[SimplicialComplex

 Properties known: Dim, FVector, FacetsEx, IsPure, Name, NumFaces[],
                   SkelExs[], Vertices.

 Name="collapsed version of FilledSphere(S^4 (VT)) at vertex [ 1 ]"
 Dim=0
 FVector=[ 1 ]
 IsPure=true

/SimplicialComplex]
gap> bd:=SCBoundary(filled);;
gap> bd=s;
true
```

6.10.7 SCHandleAddition

▷ `SCHandleAddition(complex, f1, f2)` (method)

Returns: simplicial complex of type `SCSimplicialComplex`, `fail` otherwise.

Returns a simplicial complex obtained by identifying the vertices of facet *f1* with the ones from facet *f2* in *complex*. A combinatorial handle addition is possible, whenever we have $\mathrm{d}(v,w) \geq 3$ for any two vertices $v \in f1$ and $w \in f2$, where $\mathrm{d}(\cdot,\cdot)$ is the length of the shortest path from v to w. This condition is not checked by this algorithm. See [BD08b] for further information.

Example

```
gap> c:=SC([[1,2,4],[2,4,5],[2,3,5],[3,5,6],[1,3,6],[1,4,6]]);;
gap> c:=SCUnion(c,SCUnion(SCCopy(c)+3,SCCopy(c)+6));;
gap> c:=SCUnion(c,SC([[1,2,3],[10,11,12]]));;
gap> c.Facets;
[ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 6 ], [ 1, 4, 6 ], [ 2, 3, 5 ],
  [ 2, 4, 5 ], [ 3, 5, 6 ], [ 4, 5, 7 ], [ 4, 6, 9 ], [ 4, 7, 9 ],
  [ 5, 6, 8 ], [ 5, 7, 8 ], [ 6, 8, 9 ], [ 7, 8, 10 ], [ 7, 9, 12 ],
  [ 7, 10, 12 ], [ 8, 9, 11 ], [ 8, 10, 11 ], [ 9, 11, 12 ], [ 10, 11, 12 ] ]
gap> c.Homology;
[ [ 0, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> torus:=SCHandleAddition(c,[1,2,3],[10,11,12]);;
gap> torus.Homology;
[ [ 0, [  ] ], [ 2, [  ] ], [ 1, [  ] ] ]
gap> ism:=SCIsManifold(torus);;
gap> ism;
true
```

6.10.8 SCIntersection

▷ `SCIntersection(complex1, complex2)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Forms the "intersection" of two simplicial complexes *complex1* and *complex2* as the simplicial complex formed by the intersection of the face lattices of *complex1* and *complex2*. The two arguments are not altered. Note: for the intersection process the vertex labelings of the complexes are taken into account. See also `Operation Intersection (SCSimplicialComplex, SCSimplicialComplex)` (5.3.3).

Example

```
gap> c:=SCBdSimplex(3);;
gap> d:=SCBdSimplex(3)+1;;
gap> d.Facets;
[ [ 2, 3, 4 ], [ 2, 3, 5 ], [ 2, 4, 5 ], [ 3, 4, 5 ] ]
gap> c:=SCBdSimplex(3);;
gap> d:=SCBdSimplex(3);;
gap> d:=SCMove(d,[[1,2,3],[]])+1;;
gap> s1:=SCIntersection(c,d);;
gap> s1.Facets;
[ [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

6.10.9 SCIsIsomorphic

▷ `SCIsIsomorphic(complex1, complex2)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

The function returns `true`, if the simplicial complexes *complex1* and *complex2* are combinatorially isomorphic, `false` if not.

Example

```
gap> c1:=SC([[11,12,13],[11,12,14],[11,13,14],[12,13,14]]);;
gap> c2:=SCBdSimplex(3);;
gap> SCIsIsomorphic(c1,c2);
true
gap> c3:=SCBdCrossPolytope(3);;
gap> SCIsIsomorphic(c1,c3);
false
```

6.10.10 SCIsSubcomplex

▷ `SCIsSubcomplex(sc1, sc2)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Returns `true` if the simplicial complex *sc2* is a sub-complex of simplicial complex *sc1*, `false` otherwise. If dim(*sc2*) ≤ dim(*sc1*) the facets of *sc2* are compared with the dim(*sc2*)-skeleton of *sc1*. Only works for pure simplicial complexes. Note: for the intersection process the vertex labelings of the complexes are taken into account.

Example

```
gap> SCLib.SearchByAttribute("F[1]=10"){[1..10]};
[ [ 17, "K^2 (VT)" ], [ 18, "T^2 (VT)" ], [ 19, "S^3 (VT)" ],
  [ 20, "(T^2)#2" ], [ 21, "S^3 (VT)" ], [ 22, "S^2xS^1 (VT)" ],
  [ 23, "S^3 (VT)" ], [ 24, "(T^2)#3" ], [ 25, "(P^2)#7 (VT)" ],
  [ 26, "S^3 (VT)" ] ]
gap> k:=SCLib.Load(last[1][1]);;
gap> c:=SCBdSimplex(9);;
```

```
gap> k.F;
[ 10, 30, 20 ]
gap> c.F;
[ 10, 45, 120, 210, 252, 210, 120, 45, 10 ]
gap> SCIsSubcomplex(c,k);
true
gap> SCIsSubcomplex(k,c);
false
```

6.10.11 SCIsomorphism

▷ SCIsomorphism(*complex1*, *complex2*) (method)

Returns: a list of pairs of vertex labels or `false` upon success, `fail` otherwise.

Returns an isomorphism of simplicial complex *complex1* to simplicial complex *complex2* in the standard labeling if they are combinatorially isomorphic, `false` otherwise. Internally calls `SCIsomorphismEx` (6.10.12).

Example

```
gap> c1:=SC([[11,12,13],[11,12,14],[11,13,14],[12,13,14]]);;
gap> c2:=SCBdSimplex(3);;
gap> SCIsomorphism(c1,c2);
[ [ 11, 1 ], [ 12, 2 ], [ 13, 3 ], [ 14, 4 ] ]
gap> SCIsomorphismEx(c1,c2);
[ [ [ 1, 1 ], [ 2, 2 ], [ 3, 3 ], [ 4, 4 ] ] ]
```

6.10.12 SCIsomorphismEx

▷ SCIsomorphismEx(*complex1*, *complex2*) (method)

Returns: a list of pairs of vertex labels or `false` upon success, `fail` otherwise.

Returns an isomorphism of simplicial complex *complex1* to simplicial complex *complex2* in the standard labeling if they are combinatorially isomorphic, `false` otherwise. If the f-vector and the Altshuler-Steinberg determinant of *complex1* and *complex2* are equal, the internal function `SCIntFunc.SCComputeIsomorphismsEx(complex1,complex2,true)` is called.

Example

```
gap> c1:=SC([[11,12,13],[11,12,14],[11,13,14],[12,13,14]]);;
gap> c2:=SCBdSimplex(3);;
gap> SCIsomorphism(c1,c2);
[ [ 11, 1 ], [ 12, 2 ], [ 13, 3 ], [ 14, 4 ] ]
gap> SCIsomorphismEx(c1,c2);
[ [ [ 1, 1 ], [ 2, 2 ], [ 3, 3 ], [ 4, 4 ] ] ]
```

6.10.13 SCJoin

▷ SCJoin(*complex1*, *complex2*) (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Calculates the simplicial join of the simplicial complexes *complex1* and *complex2*. If facet lists instead of `SCSimplicialComplex` objects are passed as arguments, the function internally falls back to the homology package [DHSW11], if available. Note that the vertex labelings of the complexes passed as arguments are not propagated to the new complex.

Example
```
gap> sphere:=SCJoin(SCBdSimplex(2),SCBdSimplex(2));
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="S^1_3 join S^1_3"
 Dim=3

/SimplicialComplex]
gap> SCHasBoundary(sphere);
false
gap> sphere.Facets;
[ [ [ 1, 1 ], [ 1, 2 ], [ 2, 1 ], [ 2, 2 ] ],
  [ [ 1, 1 ], [ 1, 2 ], [ 2, 1 ], [ 2, 3 ] ],
  [ [ 1, 1 ], [ 1, 2 ], [ 2, 2 ], [ 2, 3 ] ],
  [ [ 1, 1 ], [ 1, 3 ], [ 2, 1 ], [ 2, 2 ] ],
  [ [ 1, 1 ], [ 1, 3 ], [ 2, 1 ], [ 2, 3 ] ],
  [ [ 1, 1 ], [ 1, 3 ], [ 2, 2 ], [ 2, 3 ] ],
  [ [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 2, 2 ] ],
  [ [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 2, 3 ] ],
  [ [ 1, 2 ], [ 1, 3 ], [ 2, 2 ], [ 2, 3 ] ] ]
gap> sphere.Homology;
[ [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
```

Example
```
gap> ball:=SCJoin(SC([[1]]),SCBdSimplex(2));
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 4 join S^1_3"
 Dim=2

/SimplicialComplex]
gap> ball.Homology;
[ [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ] ]
gap> ball.Facets;
[ [ [ 1, 1 ], [ 2, 1 ], [ 2, 2 ] ], [ [ 1, 1 ], [ 2, 1 ], [ 2, 3 ] ],
  [ [ 1, 1 ], [ 2, 2 ], [ 2, 3 ] ] ]
```

6.10.14 SCNeighbors

▷ `SCNeighbors(complex, face)` (method)

Returns: a list of faces upon success, `fail` otherwise.

In a simplicial complex *complex* all neighbors of the `k`-face *face*, i. e. all `k`-faces distinct from *face* intersecting with *face* in a common $(k-1)$-face, are returned in the original labeling.

Example
```
gap> c:=SCFromFacets(Combinations(["a","b","c"],2));
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 22"
 Dim=1

/SimplicialComplex]
gap> SCNeighbors(c,["a","d"]);
[ [ "a", "b" ], [ "a", "c" ] ]
```

6.10.15 SCNeighborsEx

▷ `SCNeighborsEx(complex, face)` (method)

Returns: a list of faces upon success, `fail` otherwise.

In a simplicial complex *complex* all neighbors of the `k`-face *face*, i. e. all `k`-faces distinct from *face* intersecting with *face* in a common $(k-1)$-face, are returned in the standard labeling.

Example
```
gap> c:=SCFromFacets(Combinations(["a","b","c"],2));
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 21"
 Dim=1

/SimplicialComplex]
gap> SCLabels(c);
[ "a", "b", "c" ]
gap> SCNeighborsEx(c,[1,2]);
[ [ 1, 3 ], [ 2, 3 ] ]
```

6.10.16 SCShelling

▷ `SCShelling(complex)` (method)

Returns: a facet list or `false` upon success, `fail` otherwise.

The simplicial complex *complex* must be pure, strongly connected and must fulfill the weak pseudomanifold property with non-empty boundary (cf. `SCBoundary` (6.9.7)).

An ordering $(F_1, F_2, \dots, F_r)$ on the facet list of a simplicial complex is a shelling if and only if $F_i \cap (F_1 \cup \dots \cup F_{i-1})$ is a pure simplicial complex of dimension $d-1$ for all $i = 1, \dots, r$.

The function checks whether *complex* is shellable or not. In the first case a permuted version of the facet list of *complex* is returned encoding a shelling of *complex*, otherwise `false` is returned.

Internally calls `SCShellingExt` (6.10.17)`(complex,false,[]);`. To learn more about shellings see [Zie95], [Pac87].

Example
```
gap> c:=SC([[1,2,3],[1,2,4],[1,3,4]]);;
gap> SCShelling(c);
[ [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 4 ] ] ]
```

6.10.17 SCShellingExt

▷ `SCShellingExt(complex, all, checkvector)` (method)

Returns: a list of facet lists (if *checkvector* = *[]*) or `true` or `false` (if *checkvector* is not empty), `fail` otherwise.

The simplicial complex *complex* must be pure of dimension d, strongly connected and must fulfill the weak pseudomanifold property with non-empty boundary (cf. `SCBoundary` (6.9.7)).

An ordering $(F_1,F_2,\ldots,F_r)$ on the facet list of a simplicial complex is a shelling if and only if $F_i \cap (F_1 \cup \ldots \cup F_{i-1})$ is a pure simplicial complex of dimension $d-1$ for all $i = 1,\ldots,r$.

If *all* is set to `true` all possible shellings of *complex* are computed. If *all* is set to `false`, at most one shelling is computed.

Every shelling is represented as a permuted version of the facet list of *complex*. The list *checkvector* encodes a shelling in a shorter form. It only contains the indices of the facets. If an order of indices is assigned to *checkvector* the function tests whether it is a valid shelling or not.

See [Zie95], [Pac87] to learn more about shellings.

Example
```
gap> c:=SCBdSimplex(4);;
gap> c:=SCDifference(c,SC([c.Facets[1]]));; # bounded version
gap> all:=SCShellingExt(c,true,[]);;
gap> Size(all);
24
gap> all[1];
[ [ 1, 2, 3, 5 ], [ 1, 2, 4, 5 ], [ 1, 3, 4, 5 ], [ 2, 3, 4, 5 ] ]
gap> all:=SCShellingExt(c,false,[]);
[ [ [ 1, 2, 3, 5 ], [ 1, 2, 4, 5 ], [ 1, 3, 4, 5 ], [ 2, 3, 4, 5 ] ] ]
gap> all:=SCShellingExt(c,true,[1..4]);
true
```

6.10.18 SCShellings

▷ `SCShellings(complex)` (method)

Returns: a list of facet lists upon success, `fail` otherwise.

The simplicial complex *complex* must be pure, strongly connected and must fulfill the weak pseudomanifold property with non-empty boundary (cf. `SCBoundary` (6.9.7)).

An ordering $(F_1,F_2,\ldots,F_r)$ on the facet list of a simplicial complex is a shelling if and only if $F_i \cap (F_1 \cup \ldots \cup F_{i-1})$ is a pure simplicial complex of dimension $d-1$ for all $i = 1,\ldots,r$.

The function checks whether *complex* is shellable or not. In the first case a list of permuted facet lists of *complex* is returned containing all possible shellings of *complex*, otherwise `false` is returned.

Internally calls `SCShellingExt` (6.10.17)`(complex,true,[]);`. To learn more about shellings see [Zie95], [Pac87].

Example
```
gap> c:=SC([[1,2,3],[1,2,4],[1,3,4]]);;
gap> SCShellings(c);
[ [ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 4 ] ],
  [ [ 1, 2, 3 ], [ 1, 3, 4 ], [ 1, 2, 4 ] ],
  [ [ 1, 2, 4 ], [ 1, 2, 3 ], [ 1, 3, 4 ] ],
  [ [ 1, 3, 4 ], [ 1, 2, 3 ], [ 1, 2, 4 ] ],
  [ [ 1, 2, 4 ], [ 1, 3, 4 ], [ 1, 2, 3 ] ],
  [ [ 1, 3, 4 ], [ 1, 2, 4 ], [ 1, 2, 3 ] ] ]
```

6.10.19 SCSpan

▷ `SCSpan(complex, subset)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Computes the reduced face lattice of all faces of a simplicial complex *complex* that are spanned by *subset*, a subset of the set of vertices of *complex*.

Example
```
gap> c:=SCBdCrossPolytope(4);;
gap> SCVertices(c);
[ 1, 2, 3, 4, 5, 6, 7, 8 ]
gap> span:=SCSpan(c,[1,2,3,4]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="span([ 1, 2, 3, 4 ]) in Bd(\beta^4)"
 Dim=1

/SimplicialComplex]
gap> span.Facets;
[ [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ] ]
```

Example
```
gap> c:=SC([[1,2],[1,4,5],[2,3,4]]);;
gap> span:=SCSpan(c,[2,3,5]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="span([ 2, 3, 5 ]) in unnamed complex 121"
 Dim=1

/SimplicialComplex]
gap> SCFacets(span);
[ [ 2, 3 ], [ 5 ] ]
```

6.10.20 SCSuspension

▷ `SCSuspension(complex)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Calculates the simplicial suspension of the simplicial complex *complex*. Internally falls back to the homology package [DHSW11] (if available) if a facet list is passed as argument. Note that the vertex labelings of the complexes passed as arguments are not propagated to the new complex.

Example
```
gap> SCLib.SearchByName("Poincare");
[ [ 7469, "Poincare_sphere" ] ]
gap> phs:=SCLib.Load(last[1][1]);
[SimplicialComplex

 Properties known: AltshulerSteinberg, AutomorphismGroup,
                   AutomorphismGroupSize, AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, ConnectedComponents,
                   Dim, DualGraph, EulerCharacteristic, FVector,
                   FacetsEx, GVector, GeneratorsEx, HVector,
                   HasBoundary, HasInterior, Homology, Interior,
                   IsCentrallySymmetric, IsConnected,
                   IsEulerianManifold, IsManifold, IsOrientable,
                   IsPseudoManifold, IsPure, IsStronglyConnected,
                   MinimalNonFacesEx, Name, Neighborliness,
                   NumFaces[], Orientation, SkelExs[], Vertices.

 Name="Poincare_sphere"
 Dim=3
 AltshulerSteinberg=115400413872363901952
 AutomorphismGroupSize=1
 AutomorphismGroupStructure="1"
 AutomorphismGroupTransitivity=0
 EulerCharacteristic=0
 FVector=[ 16, 106, 180, 90 ]
 GVector=[ 11, 52 ]
 HVector=[ 12, 64, 12, 1 ]
 HasBoundary=false
 HasInterior=true
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsCentrallySymmetric=false
 IsConnected=true
 IsEulerianManifold=true
 IsOrientable=true
 IsPseudoManifold=true
 IsPure=true
 IsStronglyConnected=true
 Neighborliness=1

/SimplicialComplex]
gap> susp:=SCSuspension(phs);;
gap> edwardsSphere:=SCSuspension(susp);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.
```

```
 Name="susp of susp of Poincare_sphere"
 Dim=5

/SimplicialComplex]
```

6.10.21 SCUnion

▷ SCUnion(*complex1, complex2*) (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Forms the union of two simplicial complexes *complex1* and *complex2* as the simplicial complex formed by the union of their facets sets. The two arguments are not altered. Note: for the union process the vertex labelings of the complexes are taken into account, see also `Operation Union (SCSimplicialComplex, SCSimplicialComplex)` (5.3.1). Facets occurring in both arguments are treated as one facet in the new complex.

Example
```
gap> c:=SCUnion(SCBdSimplex(3),SCBdSimplex(3)+3); #a wedge of two 2-spheres
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="S^2_4 cup S^2_4"
 Dim=2

/SimplicialComplex]
```

6.10.22 SCVertexIdentification

▷ SCVertexIdentification(*complex, v1, v2*) (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Identifies vertex *v1* with vertex *v2* in a simplicial complex *complex* and returns the result as a new object. A vertex identification of *v1* and *v2* is possible whenever d(*v1*,*v2*) ≥ 3. This is not checked by this algorithm.

Example
```
gap> c:=SC([[1,2],[2,3],[3,4]]);;
gap> circle:=SCVertexIdentification(c,[1],[4]);;
gap> circle.Facets;
[ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ]
gap> circle.Homology;
[ [ 0, [  ] ], [ 1, [  ] ] ]
```

6.10.23 SCWedge

▷ SCWedge(*complex1, complex2*) (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Calculates the wedge product of the complexes supplied as arguments. Note that the vertex labelings of the complexes passed as arguments are not propagated to the new complex.

Example

```
gap> wedge:=SCWedge(SCBdSimplex(2),SCBdSimplex(2));
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 17"
 Dim=1

/SimplicialComplex]
gap> wedge.Facets;
[ [ 1, [ 1, 2 ] ], [ 1, [ 1, 3 ] ], [ 1, [ 2, 2 ] ], [ 1, [ 2, 3 ] ],
  [ [ 1, 2 ], [ 1, 3 ] ], [ [ 2, 2 ], [ 2, 3 ] ] ]
```

Chapter 7

Functions and operations for SCNormalSurface

7.1 Creating an SCNormalSurface object

This section contains functions to construct discrete normal surfaces that are slicings from a list of 2-dimensional facets (triangles and quadrilaterals) or combinatorial 3-manifolds.

For a very short introduction to the theory of discrete normal surfaces and slicings see Section 2.4 and Section 2.5, for an introduction to the GAP object type SCNormalSurface see 5.4, for more information see the article [Spr11b].

7.1.1 SCNSEmpty

▷ SCNSEmpty() (function)

Returns: discrete normal surface of type SCNormalSurface upon success, fail otherwise.

Generates an empty complex (of dimension -1), i. e. an object of type SCNormalSurface with empty facet list.

Example

```
gap> SCNSEmpty();
[NormalSurface

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="empty normal surface"
 Dim=-1

/NormalSurface]
```

7.1.2 SCNSFromFacets

▷ SCNSFromFacets(*facets*) (method)

Returns: discrete normal surface of type SCNormalSurface upon success, fail otherwise.

Constructor for a discrete normal surface from a facet list, see SCFromFacets (6.1.1) for details.

Example
```
gap> sl:=SCNSFromFacets([[1,2,3],[1,2,4,5],[1,3,4,6],[2,3,5,6],[4,5,6]]);
[NormalSurface

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 86"
 Dim=2

/NormalSurface]
```

7.1.3 SCNS

▷ SCNS(*facets*) (method)

Returns: discrete normal surface of type SCNormalSurface upon success, fail otherwise.

Internally calls SCNSFromFacets (7.1.2).

Example
```
gap> sl:=SCNS([[1,2,3],[1,2,4,5],[1,3,4,6],[2,3,5,6],[4,5,6]]);
[NormalSurface

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 87"
 Dim=2

/NormalSurface]
```

7.1.4 SCNSSlicing

▷ SCNSSlicing(*complex, slicing*) (function)

Returns: discrete normal surface of type SCNormalSurface upon success, fail otherwise.

Computes a slicing defined by a partition *slicing* of the set of vertices of the 3-dimensional combinatorial pseudomanifold *complex*. In particular, *slicing* has to be a pair of lists of vertex labels and has to contain all vertex labels of *complex*.

Example
```
gap> SCLib.SearchByAttribute("F=[ 10, 35, 50, 25 ]");
[ [ 19, "S^3 (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> sl:=SCNSSlicing(c,[[1..5],[6..10]]);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector, Fac\
etsEx, Genus, IsConnected, IsOrientable, NSTriangulation, Name, TopologicalTyp\
e, Vertices.

 Name="slicing [ [ 1, 2, 3, 4, 5 ], [ 6, 7, 8, 9, 10 ] ] of S^3 (VT)"
 Dim=2
 FVector=[ 17, 36, 12, 9 ]
 EulerCharacteristic=2
```

```
 IsOrientable=true
 TopologicalType="S^2"

/NormalSurface]
gap> sl.Facets;
[ [ [ 1, 6 ], [ 1, 8 ], [ 1, 9 ] ], [ [ 1, 6 ], [ 1, 8 ], [ 3, 6 ], [ 3, 8 ] ]
    , [ [ 1, 6 ], [ 1, 9 ], [ 4, 6 ], [ 4, 9 ] ],
  [ [ 1, 6 ], [ 3, 6 ], [ 4, 6 ] ], [ [ 1, 8 ], [ 1, 9 ], [ 1, 10 ] ],
  [ [ 1, 8 ], [ 1, 10 ], [ 3, 8 ], [ 3, 10 ] ],
  [ [ 1, 9 ], [ 1, 10 ], [ 2, 9 ], [ 2, 10 ] ],
  [ [ 1, 9 ], [ 2, 9 ], [ 4, 9 ] ], [ [ 1, 10 ], [ 2, 10 ], [ 3, 10 ] ],
  [ [ 2, 7 ], [ 2, 9 ], [ 2, 10 ] ],
  [ [ 2, 7 ], [ 2, 9 ], [ 4, 7 ], [ 4, 9 ] ],
  [ [ 2, 7 ], [ 2, 10 ], [ 5, 7 ], [ 5, 10 ] ],
  [ [ 2, 7 ], [ 4, 7 ], [ 5, 7 ] ], [ [ 2, 10 ], [ 3, 10 ], [ 5, 10 ] ],
  [ [ 3, 6 ], [ 3, 8 ], [ 5, 6 ], [ 5, 8 ] ], [ [ 3, 6 ], [ 4, 6 ], [ 5, 6 ] ]
    , [ [ 3, 8 ], [ 3, 10 ], [ 5, 8 ], [ 5, 10 ] ],
  [ [ 4, 6 ], [ 4, 7 ], [ 4, 9 ] ], [ [ 4, 6 ], [ 4, 7 ], [ 5, 6 ], [ 5, 7 ] ]
    , [ [ 5, 6 ], [ 5, 7 ], [ 5, 8 ] ], [ [ 5, 7 ], [ 5, 8 ], [ 5, 10 ] ] ]
gap> sl:=SCNSSlicing(c,[[1,3,5,7,9],[2,4,6,8,10]]);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector, Fac\
etsEx, Genus, IsConnected, IsOrientable, NSTriangulation, Name, TopologicalTyp\
e, Vertices.

 Name="slicing [ [ 1, 3, 5, 7, 9 ], [ 2, 4, 6, 8, 10 ] ] of S^3 (VT)"
 Dim=2
 FVector=[ 25, 50, 0, 25 ]
 EulerCharacteristic=0
 IsOrientable=true
 TopologicalType="T^2"

/NormalSurface]
gap> sl.Facets;
[ [ [ 1, 2 ], [ 1, 4 ], [ 3, 2 ], [ 3, 4 ] ],
  [ [ 1, 2 ], [ 1, 4 ], [ 9, 2 ], [ 9, 4 ] ],
  [ [ 1, 2 ], [ 1, 10 ], [ 3, 2 ], [ 3, 10 ] ],
  [ [ 1, 2 ], [ 1, 10 ], [ 9, 2 ], [ 9, 10 ] ],
  [ [ 1, 4 ], [ 1, 6 ], [ 3, 4 ], [ 3, 6 ] ],
  [ [ 1, 4 ], [ 1, 6 ], [ 9, 4 ], [ 9, 6 ] ],
  [ [ 1, 6 ], [ 1, 8 ], [ 3, 6 ], [ 3, 8 ] ],
  [ [ 1, 6 ], [ 1, 8 ], [ 9, 6 ], [ 9, 8 ] ],
  [ [ 1, 8 ], [ 1, 10 ], [ 3, 8 ], [ 3, 10 ] ],
  [ [ 1, 8 ], [ 1, 10 ], [ 9, 8 ], [ 9, 10 ] ],
  [ [ 3, 2 ], [ 3, 4 ], [ 5, 2 ], [ 5, 4 ] ],
  [ [ 3, 2 ], [ 3, 10 ], [ 5, 2 ], [ 5, 10 ] ],
  [ [ 3, 4 ], [ 3, 6 ], [ 5, 4 ], [ 5, 6 ] ],
  [ [ 3, 6 ], [ 3, 8 ], [ 5, 6 ], [ 5, 8 ] ],
  [ [ 3, 8 ], [ 3, 10 ], [ 5, 8 ], [ 5, 10 ] ],
  [ [ 5, 2 ], [ 5, 4 ], [ 7, 2 ], [ 7, 4 ] ],
  [ [ 5, 2 ], [ 5, 10 ], [ 7, 2 ], [ 7, 10 ] ],
  [ [ 5, 4 ], [ 5, 6 ], [ 7, 4 ], [ 7, 6 ] ],
```

```
  [ [ 5, 6 ], [ 5, 8 ], [ 7, 6 ], [ 7, 8 ] ],
  [ [ 5, 8 ], [ 5, 10 ], [ 7, 8 ], [ 7, 10 ] ],
  [ [ 7, 2 ], [ 7, 4 ], [ 9, 2 ], [ 9, 4 ] ],
  [ [ 7, 2 ], [ 7, 10 ], [ 9, 2 ], [ 9, 10 ] ],
  [ [ 7, 4 ], [ 7, 6 ], [ 9, 4 ], [ 9, 6 ] ],
  [ [ 7, 6 ], [ 7, 8 ], [ 9, 6 ], [ 9, 8 ] ],
  [ [ 7, 8 ], [ 7, 10 ], [ 9, 8 ], [ 9, 10 ] ] ]
```

7.2 Generating new objects from discrete normal surfaces

simpcomp provides the possibility to copy and / or triangulate normal surfaces. Note that other constructions like the connected sum or the cartesian product do not make sense for (embedded) normal surfaces in general.

7.2.1 SCCopy

▷ `SCCopy(complex)` (method)

Returns: discrete normal surface of type `SCNormalSurface` upon success, `fail` otherwise.

Copies a GAP object of type `SCNormalSurface` (cf. `SCCopy`).

Example

```
gap> sl:=SCNSSlicing(SCBdSimplex(4),[[1],[2..5]]);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector, Fac\
etsEx, Genus, IsConnected, IsOrientable, NSTriangulation, Name, TopologicalTyp\
e, Vertices.

 Name="slicing [ [ 1 ], [ 2, 3, 4, 5 ] ] of S^3_5"
 Dim=2
 FVector=[ 4, 6, 4 ]
 EulerCharacteristic=2
 IsOrientable=true
 TopologicalType="S^2"

/NormalSurface]
gap> sl_2:=SCCopy(sl);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector, Fac\
etsEx, Genus, IsConnected, IsOrientable, NSTriangulation, Name, TopologicalTyp\
e, Vertices.

 Name="slicing [ [ 1 ], [ 2, 3, 4, 5 ] ] of S^3_5"
 Dim=2
 FVector=[ 4, 6, 4 ]
 EulerCharacteristic=2
 IsOrientable=true
 TopologicalType="S^2"

/NormalSurface]
```

```
gap> IsIdenticalObj(sl,sl_2);
false
```

7.2.2 SCNSTriangulation

▷ SCNSTriangulation(*sl*) (method)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Computes a simplicial subdivision of a slicing *sl* without introducing new vertices. The subdivision is stored as a property of *sl* and thus is returned as an immutable object. Note that symmetry may be lost during the computation.

Example

```
gap> SCLib.SearchByAttribute("F=[ 10, 35, 50, 25 ]");
[ [ 19, "S^3 (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> sl:=SCNSSlicing(c,[[1,3,5,7,9],[2,4,6,8,10]]);;
gap> sl.F;
[ 25, 50, 0, 25 ]
gap> sc:=SCNSTriangulation(sl);;
gap> sc.F;
[ 25, 75, 50 ]
```

7.3 Properties of SCNormalSurface objects

Although some properties of a discrete normal surface can be computed by using the functions for simplicial complexes, there is a variety of properties needing specially designed functions. See below for a list.

7.3.1 SCConnectedComponents

▷ SCConnectedComponents(*complex*) (method)

Returns: a list of simplicial complexes of type SCNormalSurface upon success, fail otherwise.

Computes all connected components of an arbitrary normal surface.

Example

```
gap> sl:=SCNSSlicing(SCBdCrossPolytope(4),[[1,2],[3..8]]);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector, Fac\
etsEx, IsConnected, IsOrientable, NSTriangulation, Name, TopologicalType, Vert\
ices.

 Name="slicing [ [ 1, 2 ], [ 3, 4, 5, 6, 7, 8 ] ] of Bd(\beta^4)"
 Dim=2
 FVector=[ 12, 24, 16 ]
 EulerCharacteristic=4
 IsOrientable=true
 TopologicalType="S^2 U S^2"
```

```
/NormalSurface]
gap> cc:=SCConnectedComponents(sl);
[ [NormalSurface

     Properties known: Dim, EulerCharacteristic, FVector, FacetsEx, Genus, IsC\
onnected, IsOrientable, NSTriangulation, Name, TopologicalType, Vertices.

     Name="unnamed complex 302_cc_#1"
     Dim=2
     FVector=[ 6, 12, 8 ]
     EulerCharacteristic=2
     IsOrientable=true
     TopologicalType="S^2"

    /NormalSurface], [NormalSurface

     Properties known: Dim, EulerCharacteristic, FVector, FacetsEx, Genus, IsC\
onnected, IsOrientable, NSTriangulation, Name, TopologicalType, Vertices.

     Name="unnamed complex 302_cc_#2"
     Dim=2
     FVector=[ 6, 12, 8 ]
     EulerCharacteristic=2
     IsOrientable=true
     TopologicalType="S^2"

    /NormalSurface] ]
```

7.3.2 SCDim

▷ `SCDim(sl)` (method)

Returns: an integer upon success, `fail` otherwise.

Computes the dimension of a discrete normal surface (which is always 2 if the slicing *sl* is not empty).

Example

```
gap> sl:=SCNSEmpty();;
gap> SCDim(sl);
-1
gap> sl:=SCNSFromFacets([[1,2,3],[1,2,4,5],[1,3,4,6],[2,3,5,6],[4,5,6]]);;
gap> SCDim(sl);
2
```

7.3.3 SCEulerCharacteristic

▷ `SCEulerCharacteristic(sl)` (method)

Returns: an integer upon success, `fail` otherwise.

Computes the Euler characteristic of a discrete normal surface *sl*, cf. `SCEulerCharacteristic`.

Example

```
gap> list:=SCLib.SearchByName("S^2xS^1");;
gap> c:=SCLib.Load(list[1][1]);;
gap> sl:=SCNSSlicing(c,[[1..5],[6..10]]);;
gap> SCEulerCharacteristic(sl);
4
```

7.3.4 SCFVector

▷ SCFVector(*sl*) (method)

Returns: a 1, 3 or 4 tuple of (non-negative) integer values upon success, `fail` otherwise.

Computes the f-vector of a discrete normal surface, i. e. the number of vertices, edges, triangles and quadrilaterals of *sl*, cf. `SCFVector`.

Example

```
gap> list:=SCLib.SearchByName("S^2xS^1");;
gap> c:=SCLib.Load(list[1][1]);;
gap> sl:=SCNSSlicing(c,[[1..5],[6..10]]);;
gap> SCFVector(sl);
[ 20, 40, 16, 8 ]
```

7.3.5 SCFaceLattice

▷ SCFaceLattice(*complex*) (method)

Returns: a list of facet lists upon success, `fail` otherwise.

Computes the face lattice of a discrete normal surface *sl* in the original labeling. Triangles and quadrilaterals are stored separately (cf. `SCSkel` (7.3.13)).

Example

```
gap> c:=SCBdSimplex(4);;
gap> sl:=SCNSSlicing(c,[[1,2],[3..5]]);;
gap> SCFaceLattice(sl);
[ [ [ [ 1, 3 ] ], [ [ 1, 4 ] ], [ [ 1, 5 ] ], [ [ 2, 3 ] ], [ [ 2, 4 ] ],
      [ [ 2, 5 ] ] ],
  [ [ [ 1, 3 ], [ 1, 4 ] ], [ [ 1, 3 ], [ 1, 5 ] ], [ [ 1, 3 ], [ 2, 3 ] ],
      [ [ 1, 4 ], [ 1, 5 ] ], [ [ 1, 4 ], [ 2, 4 ] ], [ [ 1, 5 ], [ 2, 5 ] ],
      [ [ 2, 3 ], [ 2, 4 ] ], [ [ 2, 3 ], [ 2, 5 ] ], [ [ 2, 4 ], [ 2, 5 ] ] ]
    , [ [ [ 1, 3 ], [ 1, 4 ], [ 1, 5 ] ], [ [ 2, 3 ], [ 2, 4 ], [ 2, 5 ] ] ],
  [ [ [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ] ],
      [ [ 1, 3 ], [ 1, 5 ], [ 2, 3 ], [ 2, 5 ] ],
      [ [ 1, 4 ], [ 1, 5 ], [ 2, 4 ], [ 2, 5 ] ] ] ]
gap> sl.F;
[ 6, 9, 2, 3 ]
```

7.3.6 SCFaceLatticeEx

▷ SCFaceLatticeEx(*complex*) (method)

Returns: a list of face lists upon success, `fail` otherwise.

Computes the face lattice of a discrete normal surface *sl* in the standard labeling. Triangles and quadrilaterals are stored separately (cf. SCSkelEx (7.3.14)).

Example
```
gap> c:=SCBdSimplex(4);;
gap> sl:=SCNSSlicing(c,[[1,2],[3..5]]);;
gap> SCFaceLatticeEx(sl);
[ [ [ 1 ], [ 2 ], [ 3 ], [ 4 ], [ 5 ], [ 6 ] ],
  [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 5 ], [ 3, 6 ], [ 4, 5 ],
      [ 4, 6 ], [ 5, 6 ] ], [ [ 1, 2, 3 ], [ 4, 5, 6 ] ],
  [ [ 1, 2, 4, 5 ], [ 1, 3, 4, 6 ], [ 2, 3, 5, 6 ] ] ]
gap> sl.F;
[ 6, 9, 2, 3 ]
```

7.3.7 SCFpBettiNumbers

▷ SCFpBettiNumbers(*sl, p*) (method)

Returns: a list of non-negative integers upon success, fail otherwise.

Computes the Betti numbers modulo *p* of a slicing *sl*. Internally, *sl* is triangulated (using SCNSTriangulation (7.2.2)) and the Betti numbers are computed via SCFpBettiNumbers using the triangulation.

Example
```
gap> SCLib.SearchByName("(S^2xS^1)#20");
[ [ 7617, "(S^2xS^1)#20" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> c.F;
[ 27, 298, 542, 271 ]
gap> sl:=SCNSSlicing(c,[[1..13],[14..27]]);;
gap> SCFpBettiNumbers(sl,2);
[ 2, 14, 2 ]
```

7.3.8 SCGenus

▷ SCGenus(*sl*) (method)

Returns: a non-negative integer upon success, fail otherwise.

Computes the genus of a discrete normal surface *sl*.

Example
```
gap> SCLib.SearchByName("(S^2xS^1)#20");
[ [ 7617, "(S^2xS^1)#20" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> c.F;
[ 27, 298, 542, 271 ]
gap> sl:=SCNSSlicing(c,[[1..12],[13..27]]);;
gap> SCIsConnected(sl);
true
gap> SCGenus(sl);
7
```

7.3.9 SCHomology

▷ SCHomology(*sl*) (method)

Returns: a list of homology groups upon success, `fail` otherwise.

Computes the homology of a slicing *sl*. Internally, *sl* is triangulated (cf. `SCNSTriangulation` (7.2.2)) and simplicial homology is computed via `SCHomology` using the triangulation.

Example
```
gap> SCLib.SearchByName("(S^2xS^1)#20");
[ [ 7617, "(S^2xS^1)#20" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> c.F;
[ 27, 298, 542, 271 ]
gap> sl:=SCNSSlicing(c,[[1..12],[13..27]]);;
gap> sl.Homology;
[ [ 0, [  ] ], [ 14, [  ] ], [ 1, [  ] ] ]
gap> sl:=SCNSSlicing(c,[[1..13],[14..27]]);;
gap> sl.Homology;
[ [ 1, [  ] ], [ 14, [  ] ], [ 2, [  ] ] ]
```

7.3.10 SCIsConnected

▷ SCIsConnected(*complex*) (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if a normal surface *complex* is connected.

Example
```
gap> list:=SCLib.SearchByAttribute("Dim=3 and F[1]=10");;
gap> c:=SCLib.Load(list[1][1]);
[SimplicialComplex

 Properties known: AltshulerSteinberg, AutomorphismGroup,
                   AutomorphismGroupSize, AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, ConnectedComponents,
                   Dim, DualGraph, EulerCharacteristic, FVector,
                   FacetsEx, GVector, GeneratorsEx, HVector,
                   HasBoundary, HasInterior, Homology, Interior,
                   IsCentrallySymmetric, IsConnected,
                   IsEulerianManifold, IsManifold, IsOrientable,
                   IsPseudoManifold, IsPure, IsStronglyConnected,
                   MinimalNonFacesEx, Name, Neighborliness,
                   NumFaces[], Orientation, Reference, SkelExs[],
                   Vertices.

 Name="S^3 (VT)"
 Dim=3
 AltshulerSteinberg=0
 AutomorphismGroupSize=200
 AutomorphismGroupStructure="(D10 x D10) : C2"
 AutomorphismGroupTransitivity=1
 EulerCharacteristic=0
 FVector=[ 10, 35, 50, 25 ]
 GVector=[ 5, 5 ]
```

```
 HVector=[ 6, 11, 6, 1 ]
 HasBoundary=false
 HasInterior=true
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsCentrallySymmetric=false
 IsConnected=true
 IsEulerianManifold=true
 IsOrientable=true
 IsPseudoManifold=true
 IsPure=true
 IsStronglyConnected=true
 Neighborliness=1

/SimplicialComplex]
gap> sl:=SCNSSlicing(c,[[1..5],[6..10]]);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector, Fac\
etsEx, Genus, IsConnected, IsOrientable, NSTriangulation, Name, TopologicalTyp\
e, Vertices.

 Name="slicing [ [ 1, 2, 3, 4, 5 ], [ 6, 7, 8, 9, 10 ] ] of S^3 (VT)"
 Dim=2
 FVector=[ 17, 36, 12, 9 ]
 EulerCharacteristic=2
 IsOrientable=true
 TopologicalType="S^2"

/NormalSurface]
gap> SCIsConnected(sl);
true
```

7.3.11 SCIsEmpty

▷ `SCIsEmpty(`*`complex`*`)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if a normal surface *`complex`* is the empty complex, i. e. a `SCNormalSurface` object with empty facet list.

Example

```
gap> sl:=SCNS([]);;
gap> SCIsEmpty(sl);
true
```

7.3.12 SCIsOrientable

▷ `SCIsOrientable(`*`sl`*`)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks if a discrete normal surface *`sl`* is orientable.

Example

```
gap> c:=SCBdSimplex(4);;
gap> sl:=SCNSSlicing(c,[[1,2],[3,4,5]]);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector, Fac\
etsEx, Genus, IsConnected, IsOrientable, NSTriangulation, Name, TopologicalTyp\
e, Vertices.

 Name="slicing [ [ 1, 2 ], [ 3, 4, 5 ] ] of S^3_5"
 Dim=2
 FVector=[ 6, 9, 2, 3 ]
 EulerCharacteristic=2
 IsOrientable=true
 TopologicalType="S^2"

/NormalSurface]
gap> SCIsOrientable(sl);
true
```

7.3.13 SCSkel

▷ SCSkel(*sl*, *k*) (method)

Returns: a face list (of *k+1* tuples) or a list of face lists upon success, `fail` otherwise.

Computes all faces of cardinality *k+1* in the original labeling: $k = 0$ computes the vertices, $k = 1$ computes the edges, $k = 2$ computes the triangles, $k = 3$ computes the quadrilaterals.

If *k* is a list (necessarily a sublist of `[ 0,1,2,3 ]`) all faces of all cardinalities contained in *k* are computed.

Example

```
gap> c:=SCBdSimplex(4);;
gap> sl:=SCNSSlicing(c,[[1],[2..5]]);;
gap> SCSkel(sl,1);
[ [ [ 1, 2 ], [ 1, 3 ] ], [ [ 1, 2 ], [ 1, 4 ] ], [ [ 1, 2 ], [ 1, 5 ] ],
  [ [ 1, 3 ], [ 1, 4 ] ], [ [ 1, 3 ], [ 1, 5 ] ], [ [ 1, 4 ], [ 1, 5 ] ] ]
```

Example

```
gap> c:=SCBdSimplex(4);;
gap> sl:=SCNSSlicing(c,[[1],[2..5]]);;
gap> SCSkel(sl,3);
[  ]
gap> sl:=SCNSSlicing(c,[[1,2],[3..5]]);;
gap> SCSkelEx(sl,3);
[ [ 1, 2, 4, 5 ], [ 1, 3, 4, 6 ], [ 2, 3, 5, 6 ] ]
```

7.3.14 SCSkelEx

▷ SCSkelEx(*sl*, *k*) (method)

Returns: a face list (of *k+1* tuples) or a list of face lists upon success, `fail` otherwise.

Computes all faces of cardinality *k+1* in the standard labeling: $k = 0$ computes the vertices, $k = 1$ computes the edges, $k = 2$ computes the triangles, $k = 3$ computes the quadrilaterals.

If *k* is a list (necessarily a sublist of `[ 0,1,2,3 ]`) all faces of all cardinalities contained in *k* are computed.

Example
```
gap> c:=SCBdSimplex(4);;
gap> sl:=SCNSSlicing(c,[[1],[2..5]]);;
gap> SCSkelEx(sl,1);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

Example
```
gap> c:=SCBdSimplex(4);;
gap> sl:=SCNSSlicing(c,[[1],[2..5]]);;
gap> SCSkelEx(sl,3);
[  ]
gap> sl:=SCNSSlicing(c,[[1,2],[3..5]]);;
gap> SCSkelEx(sl,3);
[ [ 1, 2, 4, 5 ], [ 1, 3, 4, 6 ], [ 2, 3, 5, 6 ] ]
```

7.3.15 SCTopologicalType

▷ `SCTopologicalType(sl)` (method)

Returns: a string upon success, `fail` otherwise.

Determines the topological type of *sl* via the classification theorem for closed compact surfaces. If *sl* is not connected, the topological type of each connected component is computed.

Example
```
gap> SCLib.SearchByName("(S^2xS^1)#20");
[ [ 7617, "(S^2xS^1)#20" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> c.F;
[ 27, 298, 542, 271 ]
gap> for i in [1..26] do sl:=SCNSSlicing(c,[[1..i],[i+1..27]]); Print(sl.TopologicalType,"\n");
S^2
S^2
S^2
S^2
S^2 U S^2
S^2 U S^2
S^2
(T^2)#3
(T^2)#5
(T^2)#4
(T^2)#3
(T^2)#7
(T^2)#7 U S^2
(T^2)#7 U S^2
(T^2)#7 U S^2
(T^2)#8 U S^2
(T^2)#7 U S^2
(T^2)#8
```

```
(T^2)#6
(T^2)#6
(T^2)#5
(T^2)#3
(T^2)#2
T^2
S^2
S^2
```

7.3.16 SCUnion

▷ `SCUnion(complex1, complex2)` (method)

Returns: normal surface of type `SCNormalSurface` upon success, `fail` otherwise.

Forms the union of two normal surfaces *complex1* and *complex2* as the normal surface formed by the union of their facet sets. The two arguments are not altered. Note: for the union process the vertex labelings of the complexes are taken into account, see also `Operation Union (SCNormalSurface, SCNormalSurface)` (5.6.1). Facets occurring in both arguments are treated as one facet in the new complex.

Example

```
gap> list:=SCLib.SearchByAttribute("Dim=3 and F[1]=10");;
gap> c:=SCLib.Load(list[1][1]);
[SimplicialComplex

 Properties known: AltshulerSteinberg, AutomorphismGroup,
                   AutomorphismGroupSize, AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, ConnectedComponents,
                   Dim, DualGraph, EulerCharacteristic, FVector,
                   FacetsEx, GVector, GeneratorsEx, HVector,
                   HasBoundary, HasInterior, Homology, Interior,
                   IsCentrallySymmetric, IsConnected,
                   IsEulerianManifold, IsManifold, IsOrientable,
                   IsPseudoManifold, IsPure, IsStronglyConnected,
                   MinimalNonFacesEx, Name, Neighborliness,
                   NumFaces[], Orientation, Reference, SkelExs[],
                   Vertices.

 Name="S^3 (VT)"
 Dim=3
 AltshulerSteinberg=0
 AutomorphismGroupSize=200
 AutomorphismGroupStructure="(D10 x D10) : C2"
 AutomorphismGroupTransitivity=1
 EulerCharacteristic=0
 FVector=[ 10, 35, 50, 25 ]
 GVector=[ 5, 5 ]
 HVector=[ 6, 11, 6, 1 ]
 HasBoundary=false
 HasInterior=true
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsCentrallySymmetric=false
```

```
 IsConnected=true
 IsEulerianManifold=true
 IsOrientable=true
 IsPseudoManifold=true
 IsPure=true
 IsStronglyConnected=true
 Neighborliness=1

/SimplicialComplex]
gap> sl1:=SCNSSlicing(c,[[1..5],[6..10]]);;
gap> sl2:=sl1+10;;
gap> sl3:=SCUnion(sl1,sl2);;
gap> SCTopologicalType(sl3);
"S^2 U S^2"
```

Chapter 8

(Co-)Homology of simplicial complexes

By default, simpcomp uses an algorithm based on discrete Morse theory (see Chapter 12, `SCHomology` (12.1.12)) for its homology computations. However, some additional (co-)homology related functionality cannot be realised using this algorithm. For this, simpcomp contains an additional (co-)homology algorithm (cf. `SCHomologyInternal` (8.1.5)), which will be presented in this chapter.

Furthermore, whenever possible simpcomp makes use of the GAP package "homology" [DHSW11], for an alternative method to calculate homology groups (cf. `SCHomologyClassic` (6.9.31)) which sometimes is much faster than the built-in discrete Morse theory algorithm.

8.1 Homology computation

Apart from calculating boundaries of simplices, boundary matrices or the simplicial homology of a given complex, simpcomp is also able to compute a basis of the homology groups.

8.1.1 SCBoundaryOperatorMatrix

▷ `SCBoundaryOperatorMatrix(complex, k)` (method)

Returns: a rectangular matrix upon success, `fail` otherwise.

Calculates the matrix of the boundary operator ∂_{k+1} of a simplicial complex *complex*. Note that each column contains the boundaries of a $k+1$-simplex as a list of oriented k-simplices and that the matrix is stored as a list of row vectors (as usual in GAP).

Example

```
gap> c:=SCFromFacets([[1,2,3],[1,2,6],[1,3,5],[1,4,5],[1,4,6],\
                       [2,3,4],[2,4,5],[2,5,6],[3,4,6],[3,5,6]]);;
gap> mat:=SCBoundaryOperatorMatrix(c,1);
[ [ 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ -1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0 ],
  [ 0, -1, 0, 0, 0, -1, 0, 0, 0, 1, 1, 1, 0, 0, 0 ],
  [ 0, 0, -1, 0, 0, 0, -1, 0, 0, -1, 0, 0, 1, 1, 0 ],
  [ 0, 0, 0, -1, 0, 0, 0, -1, 0, 0, -1, 0, -1, 0, 1 ],
  [ 0, 0, 0, 0, -1, 0, 0, 0, -1, 0, 0, -1, 0, -1, -1 ] ]
```

8.1.2 SCBoundarySimplex

▷ `SCBoundarySimplex(simplex, orientation)` (function)

Returns: a list upon success, `fail` otherwise.

Calculates the boundary of a given *simplex*. If the flag *orientation* is set to `true`, the function returns the boundary as a list of oriented simplices of the form [ORIENTATION, SIMPLEX], where ORIENTATION is either +1 or -1 and a value of +1 means that SIMPLEX is positively oriented and a value of -1 that SIMPLEX is negatively oriented. If *orientation* is set to `false`, an unoriented list of simplices is returned.

Example
```
gap> SCBoundarySimplex([1..5],true);
[ [ -1, [ 2, 3, 4, 5 ] ], [ 1, [ 1, 3, 4, 5 ] ], [ -1, [ 1, 2, 4, 5 ] ],
  [ 1, [ 1, 2, 3, 5 ] ], [ -1, [ 1, 2, 3, 4 ] ] ]
gap> SCBoundarySimplex([1..5],false);
[ [ 2, 3, 4, 5 ], [ 1, 3, 4, 5 ], [ 1, 2, 4, 5 ], [ 1, 2, 3, 5 ],
  [ 1, 2, 3, 4 ] ]
```

8.1.3 SCHomologyBasis

▷ `SCHomologyBasis(complex, k)` (method)

Returns: a list of pairs of the form `[ integer, list of linear combinations of simplices ]` upon success, `fail` otherwise.

Calculates a set of basis elements for the *k*-dimensional homology group (with integer coefficients) of a simplicial complex *complex*. The entries of the returned list are of the form [MODULUS, [BASEELM1, BASEELM2, ...]], where the value MODULUS is 1 for the basis elements of the free part of the *k*-th homology group and $q \geq 2$ for the basis elements of the q-torsion part. In contrast to the function `SCHomologyBasisAsSimplices` (8.1.4) the basis elements are stored as lists of coefficient-index pairs referring to the simplices of the complex, i.e. a basis element of the form $[[\lambda_1, i], [\lambda_2, j], \dots]\dots$ encodes the linear combination of simplices of the form $\lambda_1 * \Delta_1 + \lambda_2 * \Delta_2$ with Δ_1=`SCSkel(complex,k)[i]`, Δ_2=`SCSkel(complex,k)[j]` and so on.

Example
```
gap> SCLib.SearchByName("(S^2xS^1)#RP^3");
[ [ 590, "(S^2xS^1)#RP^3" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCHomologyBasis(c,1);
[ [ 1, [ [ 1, 12 ], [ -1, 7 ], [ 1, 1 ] ] ],
  [ 2, [ [ 1, 68 ], [ -1, 69 ], [ -1, 71 ], [ 2, 72 ], [ -2, 73 ] ] ] ]
```

8.1.4 SCHomologyBasisAsSimplices

▷ `SCHomologyBasisAsSimplices(complex, k)` (method)

Returns: a list of pairs of the form `[ integer, list of linear combinations of simplices ]` upon success, `fail` otherwise.

Calculates a set of basis elements for the *k*-dimensional homology group (with integer coefficients) of a simplicial complex *complex*. The entries of the returned list are of the form [MODULUS, [BASEELM1, BASEELM2, ...]], where the value MODULUS is 1 for the basis elements of the free part of the *k*-th homology group and $q \geq 2$ for the basis elements of the q-torsion part. In contrast to the function SCHomologyBasis (8.1.3) the basis elements are stored as lists of coefficient-simplex pairs, i.e. a basis element of the form $[[\lambda_1,\Delta_1],[\lambda_2,\Delta_2],\dots]$ encodes the linear combination of simplices of the form $\lambda_1 * \Delta_1 + \lambda_2 * \Delta_2 + \dots$.

Example
```
gap> SCLib.SearchByName("(S^2xS^1)#RP^3");
[ [ 590, "(S^2xS^1)#RP^3" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCHomologyBasisAsSimplices(c,1);
[ [ 1, [ [ 1, [ 2, 8 ] ], [ -1, [ 1, 8 ] ], [ 1, [ 1, 2 ] ] ] ],
  [ 2,
      [ [ 1, [ 11, 12 ] ], [ -1, [ 11, 13 ] ], [ -1, [ 12, 13 ] ],
          [ 2, [ 12, 14 ] ], [ -2, [ 13, 14 ] ] ] ] ]
```

8.1.5 SCHomologyInternal

▷ SCHomologyInternal(*complex*) (function)

Returns: a list of pairs of the form `[ integer, list ]` upon success, `fail` otherwise.

This function computes the reduced simplicial homology with integer coefficients of a given simplicial complex *complex* with integer coefficients. It uses the algorithm described in [DKT08].

The output is a list of homology groups of the form $[H_0,....,H_d]$, where d is the dimension of *complex*. The format of the homology groups H_i is given in terms of their maximal cyclic subgroups, i.e. a homology group $H_i \cong \mathbb{Z}^f + \mathbb{Z}/t_1\mathbb{Z} \times \dots \times \mathbb{Z}/t_n\mathbb{Z}$ is returned in form of a list $[f,[t_1,...,t_n]]$, where f is the (integer) free part of H_i and t_i denotes the torsion parts of H_i ordered in weakly incresing size. See also SCHomology (12.1.12) and SCHomologyClassic (6.9.31).

Example
```
gap> c:=SCSurface(1,false);;
gap> SCHomologyInternal(c);
[ [ 0, [  ] ], [ 0, [ 2 ] ], [ 0, [  ] ] ]
```

8.2 Cohomology computation

simpcomp can also compute the cohomology groups of simplicial complexes, bases of these cohomology groups, the cup product of two cocycles and the intersection form of (orientable) 4-manifolds.

8.2.1 SCCoboundaryOperatorMatrix

▷ SCCoboundaryOperatorMatrix(*complex, k*) (method)

Returns: a rectangular matrix upon success, `fail` otherwise.

Calculates the matrix of the coboundary operator d^{k+1} as a list of row vectors.

Example
```
gap> c:=SCFromFacets([[1,2,3],[1,2,6],[1,3,5],[1,4,5],[1,4,6],\
                      [2,3,4],[2,4,5],[2,5,6],[3,4,6],[3,5,6]]);
> [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 2"
```

```
 Dim=2

/SimplicialComplex]
gap> mat:=SCCoboundaryOperatorMatrix(c,1);
[ [ -1, 1, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ -1, 0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0 ],
  [ 0, -1, 0, 1, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0 ],
  [ 0, 0, -1, 1, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0 ],
  [ 0, 0, -1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0 ],
  [ 0, 0, 0, 0, 0, -1, 1, 0, 0, -1, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 0, 0, -1, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, -1, 1, 0, 0, 0, 0, 0, -1 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 1, 0, -1, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 1, 0, 0, -1 ] ]
```

8.2.2 SCCohomology

▷ `SCCohomology(complex)` (method)

Returns: a list of pairs of the form `[ integer, list ]` upon success, `fail` otherwise.

This function computes the simplicial cohomology groups of a given simplicial complex *complex* with integer coefficients. It uses the algorithm described in [DKT08].

The output is a list of cohomology groups of the form $[H^0,\ldots.,H^d]$, where d is the dimension of *complex*. The format of the cohomology groups H^i is given in terms of their maximal cyclic subgroups, i.e. a cohomology group $H^i \cong \mathbb{Z}^f + \mathbb{Z}/t_1\mathbb{Z} \times \cdots \times \mathbb{Z}/t_n\mathbb{Z}$ is returned in form of a list $[f,[t_1,\ldots,t_n]]$, where f is the (integer) free part of H^i and t_i denotes the torsion parts of H^i ordered in weakly increasing size.

Example

```
gap> c:=SCFromFacets([[1,2,3],[1,2,6],[1,3,5],[1,4,5],[1,4,6],
                      [2,3,4],[2,4,5],[2,5,6],[3,4,6],[3,5,6]]);
> [SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 4"
 Dim=2

/SimplicialComplex]
gap> SCCohomology(c);
[ [ 1, [  ] ], [ 0, [  ] ], [ 0, [ 2 ] ] ]
```

8.2.3 SCCohomologyBasis

▷ `SCCohomologyBasis(complex, k)` (method)

Returns: a list of pairs of the form `[ integer, list of linear combinations of simplices ]` upon success, `fail` otherwise.

Calculates a set of basis elements for the *k*-dimensional cohomology group (with integer coefficients) of a simplicial complex *complex*. The entries of the returned list are of the form [MODULUS, [BASEELM1, BASEELM2, ...]], where the value MODULUS is 1 for the basis elements of the free part of the k-th homology group and $q \geq 2$ for the basis elements of the q-torsion part. In contrast to the function `SCCohomologyBasisAsSimplices` (8.2.4) the basis elements are stored as lists of coefficient-index pairs referring to the linear forms dual to the simplices in the k-th cochain complex of *complex*, i.e. a basis element of the form $[[\lambda_1,i],[\lambda_2,j],\ldots]\ldots$ encodes the linear combination of simplices (or their dual linear forms in the corresponding cochain complex) of the form $\lambda_1 * \Delta_1 + \lambda_2 * \Delta_2$ with Δ_1=`SCSkel(complex,k)[i]`, Δ_2=`SCSkel(complex,k)[j]` and so on.

Example

```
gap> SCLib.SearchByName("SU(3)/SO(3)");
[ [ 563, "SU(3)/SO(3) (VT)" ], [ 7276, "SU(3)/SO(3) (VT)" ],
  [ 7418, "SU(3)/SO(3) (VT)" ], [ 7419, "SU(3)/SO(3) (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCCohomologyBasis(c,3);
[ [ 2, [ [ -9, 259 ], [ 9, 262 ], [ 9, 263 ], [ -9, 270 ], [ 9, 271 ],
          [ -9, 273 ], [ -9, 274 ], [ -18, 275 ], [ -9, 276 ], [ 9, 278 ],
          [ -9, 279 ], [ -9, 280 ], [ 3, 283 ], [ -3, 285 ], [ 3, 289 ],
          [ -3, 294 ], [ 3, 310 ], [ -3, 313 ], [ 3, 316 ], [ -1, 317 ],
          [ -6, 318 ], [ 3, 319 ], [ -6, 320 ], [ 6, 321 ], [ 1, 322 ],
          [ 3, 325 ], [ -1, 328 ], [ 6, 330 ], [ -2, 331 ], [ 12, 332 ],
          [ 7, 333 ], [ -5, 334 ], [ 1, 345 ], [ 3, 355 ], [ -9, 357 ],
          [ 9, 358 ], [ 1, 363 ], [ 12, 365 ], [ -9, 366 ], [ -3, 370 ],
          [ -1, 371 ], [ -3, 372 ], [ 8, 373 ], [ -1, 374 ], [ 6, 375 ],
          [ 9, 376 ], [ 3, 377 ], [ 1, 380 ], [ 3, 383 ], [ -8, 385 ],
          [ -9, 386 ], [ -9, 388 ], [ -18, 404 ], [ 9, 410 ], [ -9, 425 ],
          [ -18, 426 ], [ -9, 427 ], [ 9, 428 ], [ -9, 429 ], [ 3, 433 ],
          [ -3, 435 ], [ -9, 437 ], [ 10, 442 ], [ 12, 445 ], [ 1, 447 ],
          [ -19, 448 ], [ 2, 449 ], [ -1, 450 ], [ -9, 451 ], [ 3, 453 ],
          [ 1, 455 ], [ 1, 457 ], [ -11, 458 ], [ -9, 459 ], [ 9, 461 ],
          [ 9, 462 ], [ -9, 468 ], [ 9, 469 ], [ -18, 471 ], [ -9, 472 ],
          [ 9, 474 ], [ -9, 475 ], [ 9, 488 ], [ 9, 495 ], [ -9, 500 ],
          [ -3, 504 ], [ 9, 505 ], [ 9, 512 ], [ 9, 515 ], [ 6, 519 ],
          [ 18, 521 ], [ -15, 523 ], [ 9, 524 ], [ -3, 525 ], [ 18, 527 ],
          [ -18, 528 ], [ 6, 529 ], [ 6, 531 ], [ 12, 532 ] ] ] ]
```

8.2.4 SCCohomologyBasisAsSimplices

▷ `SCCohomologyBasisAsSimplices(`*complex, k*`)` (method)

Returns: a list of pars of the form `[ integer, linear combination of simplices ]` upon success, `fail` otherwise.

Calculates a set of basis elements for the k-dimensional cohomology group (with integer coefficients) of a simplicial complex *complex*. The entries of the returned list are of the form [MODULUS, [BASEELM1, BASEELM2, ...]], where the value MODULUS is 1 for the basis elements of the free part of the k-th homology group and $q \geq 2$ for the basis elements of the q-torsion part. In contrast to the function `SCCohomologyBasis` (8.2.3) the basis elements are stored as lists of coefficient-simplex pairs referring to the linear forms dual to the simplices in the k-th cochain complex of *complex*, i.e. a basis element of the form $[[\lambda_1,\Delta_i],[\lambda_2,\Delta_j],\ldots]\ldots$ encodes the linear combination of simplices (or their dual linear forms in the corresponding cochain complex) of the form $\lambda_1 * \Delta_1 + \lambda_2 * \Delta_2 + \ldots$.

Example

```
gap> SCLib.SearchByName("SU(3)/SO(3)");
[ [ 563, "SU(3)/SO(3) (VT)" ], [ 7276, "SU(3)/SO(3) (VT)" ],
```

```
  [ 7418, "SU(3)/SO(3) (VT)" ], [ 7419, "SU(3)/SO(3) (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCCohomologyBasisAsSimplices(c,3);
[ [ 2,
      [ [ -9, [ 2, 7, 8, 9 ] ], [ 9, [ 2, 7, 8, 12 ] ],
          [ 9, [ 2, 7, 8, 13 ] ], [ -9, [ 2, 7, 11, 12 ] ],
          [ 9, [ 2, 7, 11, 13 ] ], [ -9, [ 2, 8, 9, 10 ] ],
          [ -9, [ 2, 8, 9, 11 ] ], [ -18, [ 2, 8, 9, 12 ] ],
          [ -9, [ 2, 8, 9, 13 ] ], [ 9, [ 2, 8, 10, 12 ] ],
          [ -9, [ 2, 8, 10, 13 ] ], [ -9, [ 2, 8, 11, 12 ] ],
          [ 3, [ 2, 9, 10, 12 ] ], [ -3, [ 2, 9, 11, 12 ] ],
          [ 3, [ 3, 4, 5, 7 ] ], [ -3, [ 3, 4, 5, 12 ] ],
          [ 3, [ 3, 4, 10, 12 ] ], [ -3, [ 3, 5, 6, 7 ] ],
          [ 3, [ 3, 5, 6, 11 ] ], [ -1, [ 3, 5, 6, 13 ] ],
          [ -6, [ 3, 5, 7, 8 ] ], [ 3, [ 3, 5, 7, 10 ] ],
          [ -6, [ 3, 5, 7, 11 ] ], [ 6, [ 3, 5, 7, 12 ] ],
          [ 1, [ 3, 5, 7, 13 ] ], [ 3, [ 3, 5, 8, 12 ] ],
          [ -1, [ 3, 5, 9, 13 ] ], [ 6, [ 3, 5, 10, 12 ] ],
          [ -2, [ 3, 5, 10, 13 ] ], [ 12, [ 3, 5, 11, 12 ] ],
          [ 7, [ 3, 5, 11, 13 ] ], [ -5, [ 3, 5, 12, 13 ] ],
          [ 1, [ 3, 6, 9, 13 ] ], [ 3, [ 3, 7, 10, 12 ] ],
          [ -9, [ 3, 7, 11, 12 ] ], [ 9, [ 3, 7, 11, 13 ] ],
          [ 1, [ 3, 8, 9, 13 ] ], [ 12, [ 3, 8, 10, 12 ] ],
          [ -9, [ 3, 8, 10, 13 ] ], [ -3, [ 3, 9, 10, 12 ] ],
          [ -1, [ 3, 9, 10, 13 ] ], [ -3, [ 3, 9, 11, 12 ] ],
          [ 8, [ 3, 9, 11, 13 ] ], [ -1, [ 3, 9, 12, 13 ] ],
          [ 6, [ 3, 10, 11, 12 ] ], [ 9, [ 3, 10, 11, 13 ] ],
          [ 3, [ 3, 10, 12, 13 ] ], [ 1, [ 4, 5, 6, 8 ] ],
          [ 3, [ 4, 5, 6, 11 ] ], [ -8, [ 4, 5, 6, 13 ] ],
          [ -9, [ 4, 5, 7, 8 ] ], [ -9, [ 4, 5, 7, 11 ] ],
          [ -18, [ 4, 6, 8, 9 ] ], [ 9, [ 4, 6, 9, 13 ] ],
          [ -9, [ 4, 8, 9, 10 ] ], [ -18, [ 4, 8, 9, 12 ] ],
          [ -9, [ 4, 8, 9, 13 ] ], [ 9, [ 4, 8, 10, 12 ] ],
          [ -9, [ 4, 8, 10, 13 ] ], [ 3, [ 4, 9, 10, 12 ] ],
          [ -3, [ 4, 9, 11, 12 ] ], [ -9, [ 4, 9, 12, 13 ] ],
          [ 10, [ 5, 6, 7, 8 ] ], [ 12, [ 5, 6, 7, 11 ] ],
          [ 1, [ 5, 6, 7, 13 ] ], [ -19, [ 5, 6, 8, 9 ] ],
          [ 2, [ 5, 6, 8, 11 ] ], [ -1, [ 5, 6, 8, 12 ] ],
          [ -9, [ 5, 6, 8, 13 ] ], [ 3, [ 5, 6, 9, 11 ] ],
          [ 1, [ 5, 6, 9, 13 ] ], [ 1, [ 5, 6, 10, 13 ] ],
          [ -11, [ 5, 6, 11, 13 ] ], [ -9, [ 5, 7, 8, 9 ] ],
          [ 9, [ 5, 7, 8, 12 ] ], [ 9, [ 5, 7, 8, 13 ] ],
          [ -9, [ 5, 7, 11, 12 ] ], [ 9, [ 5, 7, 11, 13 ] ],
          [ -18, [ 5, 8, 9, 12 ] ], [ -9, [ 5, 8, 9, 13 ] ],
          [ 9, [ 5, 8, 10, 12 ] ], [ -9, [ 5, 8, 11, 12 ] ],
          [ 9, [ 6, 7, 8, 13 ] ], [ 9, [ 6, 7, 11, 13 ] ],
          [ -9, [ 6, 8, 10, 13 ] ], [ -3, [ 6, 9, 11, 12 ] ],
          [ 9, [ 6, 9, 11, 13 ] ], [ 9, [ 7, 8, 9, 13 ] ],
          [ 9, [ 7, 8, 11, 12 ] ], [ 6, [ 7, 9, 11, 12 ] ],
          [ 18, [ 7, 11, 12, 13 ] ], [ -15, [ 8, 9, 10, 12 ] ],
          [ 9, [ 8, 9, 10, 13 ] ], [ -3, [ 8, 9, 11, 12 ] ],
          [ 18, [ 8, 10, 11, 12 ] ], [ -18, [ 8, 10, 12, 13 ] ],
          [ 6, [ 9, 10, 11, 12 ] ], [ 6, [ 9, 10, 12, 13 ] ],
```

```
            [ 12, [ 9, 11, 12, 13 ] ] ] ] ]
```

8.2.5 SCCupProduct

▷ `SCCupProduct(complex, cocycle1, cocycle2)` (function)

Returns: a list of pairs of the form `[ ORIENTATION, SIMPLEX ]` upon success, `fail` otherwise.

The cup product is a method of adjoining two cocycles of degree p and q to form a composite cocycle of degree $p+q$. It endows the cohomology groups of a simplicial complex with the structure of a ring.

The construction of the cup product starts with a product of cochains: if *cocycle1* is a p-cochain and *cocylce2* is a q-cochain of a simplicial complex *complex* (given as list of oriented p- (q-)simplices), then

$\mathit{cocycle1} \smile \mathit{cocycle2}(\sigma) = \mathit{cocycle1}(\sigma \circ \iota_{0,1,\dots p}) \cdot \mathit{cocycle2}(\sigma \circ \iota_{p,p+1,\dots,p+q})$

where σ is a $p+q$-simplex and ι_S, $S \subset \{0,1,\dots,p+q\}$ is the canonical embedding of the simplex spanned by S into the $(p+q)$-standard simplex.

$\sigma \circ \iota_{0,1,\dots,p}$ is called the p-th front face and $\sigma \circ \iota_{p,p+1,\dots,p+q}$ is the q-th back face of σ, respectively.

Note that this function only computes the cup product in the case that *complex* is an orientable weak pseudomanifold of dimension $2k$ and $p = q = k$. Furthermore, *complex* must be given in standard labeling, with sorted facet list and *cocylce1* and *cocylce2* must be given in simplex notation and labeled accordingly. Note that the latter condition is usually fulfilled in case the cocycles were computed using `SCCohomologyBasisAsSimplices` (8.2.4).

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> basis:=SCCohomologyBasisAsSimplices(c,2);;
gap> SCCupProduct(c,basis[1][2],basis[1][2]);
[ [ 1, [ 1, 2, 4, 7, 11 ] ], [ 1, [ 2, 3, 4, 5, 9 ] ] ]
gap> SCCupProduct(c,basis[1][2],basis[2][2]);
[ [ -1, [ 1, 2, 4, 7, 11 ] ], [ -1, [ 1, 2, 4, 7, 15 ] ],
  [ -1, [ 2, 3, 4, 5, 9 ] ] ]
```

8.2.6 SCIntersectionForm

▷ `SCIntersectionForm(complex)` (method)

Returns: a square matrix of integer values upon success, `fail` otherwise.

For $2k$-dimensional orientable manifolds M the cup product (see `SCCupProduct` (8.2.5)) defines a bilinear form

$\mathrm{H}^k(M) \times \mathrm{H}^k(M) \to \mathrm{H}^{2k}(M), (a,b) \mapsto a \cup b$

called the intersection form of M. This function returns the intersection form of an orientable combinatorial $2k$-manifold *complex* in form of a matrix `mat` with respect to the basis of $\mathrm{H}^k($*complex*M$)$ computed by `SCCohomologyBasisAsSimplices` (8.2.4). The matrix entry `mat[i][j]` equals the intersection number of the `i`-th base element with the `j`-th base element of $\mathrm{H}^k($*complex*M$)$.

Example
```
gap> SCLib.SearchByName("CP^2");
[ [ 16, "CP^2 (VT)" ], [ 99, "CP^2#-CP^2" ], [ 100, "CP^2#CP^2" ],
```

```
  [ 400, "CP^2#(S^2xS^2)" ], [ 2486, "Gaifullin CP^2" ],
  [ 4401, "(S^3~S^1)#(CP^2)^{#5} (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> c1:=SCConnectedSum(c,c);;
gap> c2:=SCConnectedSumMinus(c,c);;
gap> q1:=SCIntersectionForm(c1);;
gap> q2:=SCIntersectionForm(c2);;
gap> PrintArray(q1);
[ [  1,  0 ],
  [  0,  1 ] ]
gap> PrintArray(q2);
[ [   1,   0 ],
  [   0,  -1 ] ]
```

8.2.7 SCIntersectionFormParity

▷ SCIntersectionFormParity(*complex*) (method)

Returns: 0 or 1 upon success, fail otherwise.

Computes the parity of the intersection form of a combinatorial manifold *complex* (see SCIntersectionForm (8.2.6)). If the intersection for is even (i. e. all diagonal entries are even numbers) 0 is returned, otherwise 1 is returned.

Example

```
gap> SCLib.SearchByName("S^2xS^2");
[ [ 59, "S^2xS^2" ], [ 134, "S^2xS^2 (VT)" ], [ 135, "S^2xS^2 (VT)" ],
  [ 136, "S^2xS^2 (VT)" ], [ 137, "(S^2xS^2)#(S^2xS^2)" ],
  [ 360, "(S^2xS^2)#(S^2xS^2) (VT)" ], [ 361, "(S^2xS^2)#(S^2xS^2) (VT)" ],
  [ 400, "CP^2#(S^2xS^2)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCIntersectionFormParity(c);
0
gap> SCLib.SearchByName("CP^2");
[ [ 16, "CP^2 (VT)" ], [ 99, "CP^2#-CP^2" ], [ 100, "CP^2#CP^2" ],
  [ 400, "CP^2#(S^2xS^2)" ], [ 2486, "Gaifullin CP^2" ],
  [ 4401, "(S^3~S^1)#(CP^2)^{#5} (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCIntersectionFormParity(c);
1
```

8.2.8 SCIntersectionFormDimensionality

▷ SCIntersectionFormDimensionality(*complex*) (method)

Returns: an integer upon success, fail otherwise.

Returns the dimensionality of the intersection form of a combinatorial manifold *complex*, i. e. the length of a minimal generating set of $H^k(M)$ (where $2k$ is the dimension of *complex*). See SCIntersectionForm (8.2.6) for further details.

Example

```
gap> SCLib.SearchByName("CP^2");
[ [ 16, "CP^2 (VT)" ], [ 99, "CP^2#-CP^2" ], [ 100, "CP^2#CP^2" ],
```

```
  [ 400, "CP^2#(S^2xS^2)" ], [ 2486, "Gaifullin CP^2" ],
  [ 4401, "(S^3~S^1)#(CP^2)^{#5} (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCIntersectionFormParity(c);
1
gap> SCCohomology(c);
[ [ 1, [  ] ], [ 0, [  ] ], [ 1, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> SCIntersectionFormDimensionality(c);
1
gap> d:=SCConnectedProduct(c,10);;
gap> SCIntersectionFormDimensionality(d);
10
```

8.2.9 SCIntersectionFormSignature

▷ `SCIntersectionFormSignature(complex)` (method)

Returns: a triple of integers upon success, `fail` otherwise.

Computes the dimensionality (see `SCIntersectionFormDimensionality` (8.2.8)) and the signature of the intersection form of a combinatorial manifold *complex* as a 3-tuple that contains the dimensionality in the first entry and the number of positive / negative eigenvalues in the second and third entry. See `SCIntersectionForm` (8.2.6) for further details.

Internally calls the GAP-functions `Matrix_CharacteristicPolynomialSameField` and `CoefficientsOfLaurentPolynomial` to compute the number of positive / negative eigenvalues of the intersection form.

Example

```
gap> SCLib.SearchByName("CP^2");
[ [ 16, "CP^2 (VT)" ], [ 99, "CP^2#-CP^2" ], [ 100, "CP^2#CP^2" ],
  [ 400, "CP^2#(S^2xS^2)" ], [ 2486, "Gaifullin CP^2" ],
  [ 4401, "(S^3~S^1)#(CP^2)^{#5} (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCIntersectionFormParity(c);
1
gap> SCCohomology(c);
[ [ 1, [  ] ], [ 0, [  ] ], [ 1, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> SCIntersectionFormSignature(c);
[ 1, 0, 1 ]
gap> d:=SCConnectedSum(c,c);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="CP^2 (VT)#+-CP^2 (VT)"
 Dim=4

/SimplicialComplex]
gap> SCIntersectionFormSignature(d);
[ 2, 2, 0 ]
gap> d:=SCConnectedSumMinus(c,c);;
gap> SCIntersectionFormSignature(d);
[ 2, 1, 1 ]
```

Chapter 9

Bistellar flips

9.1 Theory

Since two combinatorial manifolds are already considered distinct to each other as soon as they are not combinatorially isomorphic, a topological PL-manifold is represented by a whole class of combinatorial manifolds. Thus, a frequent question when working with combinatorial manifolds is whether two such objects are PL-homeomorphic or not. One possibility to approach this problem, i. e. to find combinatorially distinct members of the class of a PL-manifold, is a heuristic algorithm using the concept of bistellar moves.

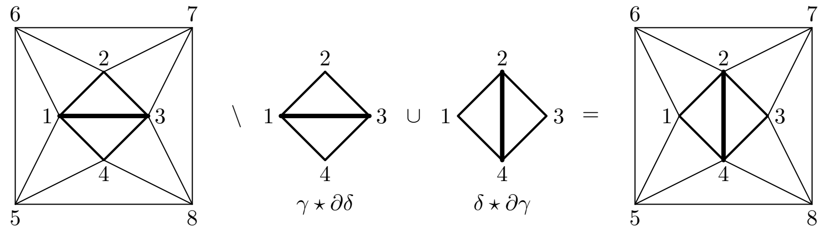
DEFINITION (Bistellar moves [Pac87])
Let M be a combinatorial d-manifold (d-pseudomanifold), $\gamma = \langle v_0, \dots, v_k \rangle$ a k-face and $\delta = \langle w_0, \dots, w_{d-k} \rangle$ a $(d-k+1)$-tuple of vertices of M that does not span a $(d-k)$-face in M, $0 \leq k \leq d$, such that $\{v_0, \dots, v_k\} \cap \{w_0, \dots, w_{d-k}\} = \varnothing$ and $\{v_0, \dots, v_k, w_0, \dots w_{k-d}\}$ spans exactly $d-k+1$ facets. Then the operation

$$\kappa_{(\gamma,\delta)}(M) = M \smallsetminus (\gamma \star \partial\delta) \cup (\partial\gamma \star \delta)$$

is called a *bistellar* $(d-k)$*-move*.

In other words: If there exists a bouquet $D \subset M$ of $d-k+1$ facets on a subset of vertices $W \subset V$ of order $d+2$ with a common k-face γ and the complement δ of the vertices of γ in W does not span a $(d-k)$-face in M we can remove D and replace it by a bouquet of $k+1$ facets $E \subset M$ with vertex set W with a common face spanned by δ. By construction $\partial D = \partial E$ and the altered complex is again a combinatorial d-manifold (d-pseudomanifold). See Fig. 11 for a bistellar 1-move of a 2-dimensional complex, see Fig. 12 for all bistellar moves in dimension 3.

$$M := \langle\langle 1,2,3\rangle, \langle 1,2,5\rangle, \langle 1,3,4\rangle, \langle 1,4,8\rangle, \langle 1,5,8\rangle, \langle 2,3,6\rangle, \langle 2,5,6\rangle, \langle 3,4,7\rangle, \langle 3,6,7\rangle, \langle 4,7,8\rangle\rangle;$$
$$\gamma := \langle\langle 1,3\rangle\rangle; \quad \delta := \langle\langle 2,4\rangle\rangle;$$



$$\kappa_{(\gamma,\delta)}(M) = \langle\langle 1,2,4\rangle, \langle 1,2,5\rangle, \langle 2,3,4\rangle, \langle 1,4,8\rangle, \langle 1,5,8\rangle, \langle 2,3,6\rangle, \langle 2,5,6\rangle, \langle 3,4,7\rangle, \langle 3,6,7\rangle, \langle 4,7,8\rangle\rangle;$$

Figure 11. Bistellar 1-move in dimension 2 with $W = \{1,2,3,4\}$.

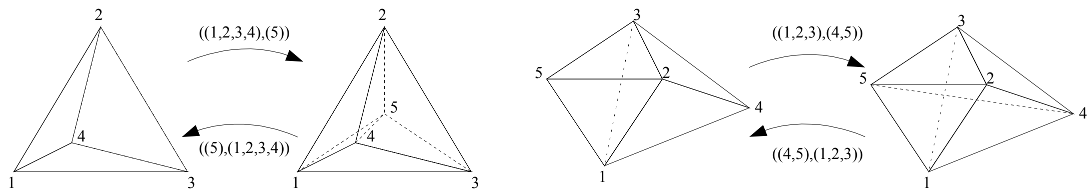


Figure 12. Bistellar moves in dimension $d = 3$ with $W = \{1,2,3,4,5\}$. On the left side a bistellar 0- and a bistellar 3-move, on the right side a bistellar 1- and a bistellar 2-move.

A bistellar 0-move is a *stellar subdivision*, i. e. the subdivision of a facet δ into $d+1$ new facets by introducing a new vertex at the center of δ (cf. Fig. 12 on the left). In particular, the vertex set of a combinatorial manifold (pseudomanifold) is not invariant under bistellar moves. For any bistellar $(d-k)$-move $\kappa_{(\gamma,\delta)}$ we have an inverse bistellar k-move $\kappa^{-1}_{(\gamma,\delta)} = \kappa_{(\delta,\gamma)}$ such that $\kappa_{(\delta,\gamma)}(\kappa_{(\gamma,\delta)}(M)) = M$. If for two combinatorial manifolds M and N there exist a sequence of bistellar moves that transforms one into the other, M and N are called *bistellarly equivalent*. So far bistellar moves are local operations on combinatorial manifolds that change its combinatorial type. However, the strength of the concept in combinatorial topology is a consequence of the following

THEOREM (Bistellar moves [Pac87])
Two combinatorial manifolds (pseudomanifolds) M and N are PL homeomorphic if and only if they are bistellarly equivalent.

Unfortunately Pachners theorem does not guarantee that the search for a connecting sequence of bistellar moves between M and N terminates. Hence, using bistellar moves, we can not prove that M and N are not PL-homeomorphic. However, there is a very effective simulated annealing approach that is able to give a positive answer in a lot of cases. The heuristic was first implemented by Bjoerner and Lutz in [BL00]. The functions presented in this chapter are based on this code which can be used for several tasks:

- Decide, whether two combinatorial manifolds are PL-homeomorphic,

- for a given triangulation of a PL-manifold, try to find a smaller one with less vertices,
- check, if an abstract simplicial complex is a combinatorial manifold by reducing all vertex links to the boundary of the d-simplex (this can also be done using discrete Morse theory, see Chapter <Ref Chap="chap:DMT" />, <Ref Meth="SCBistellarIsManifold" />).

In many cases the heuristic reduces a given triangulation but does not reach a minimal triangulation after a reasonable amount of flips. Thus, we usually can not expect the algorithm to terminate. However, in some cases the program normally stops after a small number of flips:

- Whenever $d = 1$ (in this case the approach is deterministic),
- whenever a complex is PL-homeomorphic to the boundary of the d-simplex,
- in the case of some 3-manifolds, namely $S^2 \underline{\times} S^1$, $S^2 \times S^1$ or $\mathbb{RP}^3$.

Technical note: Since bistellar flips do not respect the combinatorial properties of a complex, no attention to the original vertex labels is payed, i. e. the flipped complex will be relabeled whenever its vertex labels become different from the standard labeling (for example after every reverse 0-move).

9.2 Functions for bistellar flips

9.2.1 SCBistellarOptions

▷ `SCBistellarOptions` (global variable)

Record of global variables to adjust output an behavior of bistellar moves in `SCIntFunc.SCChooseMove` (9.2.4) and `SCReduceComplexEx` (9.2.14) respectively.

1. `BaseRelaxation`: determines the length of the relaxation period. Default: 3
2. `BaseHeating`: determines the length of the heating period. Default: 4
3. `Relaxation`: value of the current relaxation period. Default: 0
4. `Heating`: value of the current heating period. Default: 0
5. `MaxRounds`: maximal over all number of bistellar flips that will be performed. Default: 500000
6. `MaxInterval`: maximal number of bistellar flips that will be performed without a change of the f-vector of the moved complex. Default: 100000
7. `Mode`: flip mode, 0=reducing, 1=comparing, 2=reduce as sub-complex, 3=randomize. Default: 0
8. `WriteLevel`: 0=no output, 1=storing of every vertex minimal complex to user library, 2=e-mail notification. Default: 1
9. `MailNotifyIntervall`: (minimum) number of seconds between two e-mail notifications. Default: $24 \cdot 60 \cdot 60$ (one day)

10. `MaxIntervalIsManifold`: maximal number of bistellar flips that will be performed without a change of the f-vector of a vertex link while trying to prove that the complex is a combinatorial manifold. Default: 5000

11. `MaxIntervalRandomize := 50`: number of flips performed to create a randomized sphere. Default: 50

Example
```
gap> SCBistellarOptions.BaseRelaxation;
3
gap> SCBistellarOptions.BaseHeating;
4
gap> SCBistellarOptions.Relaxation;
0
gap> SCBistellarOptions.Heating;
0
gap> SCBistellarOptions.MaxRounds;
500000
gap> SCBistellarOptions.MaxInterval;
100000
gap> SCBistellarOptions.Mode;
0
gap> SCBistellarOptions.WriteLevel;
0
gap> SCBistellarOptions.MailNotifyInterval;
86400
gap> SCBistellarOptions.MaxIntervalIsManifold;
5000
gap> SCBistellarOptions.MaxIntervalRandomize;
50
```

9.2.2 SCEquivalent

▷ `SCEquivalent(`*`complex1, complex2`*`)` (method)

Returns: `true` or `false` upon success, `fail` or a list of type `[ fail, SCSimplicialComplex, Integer, facet list]` otherwise.

Checks if the simplicial complex *`complex1`* (which has to fulfill the weak pseudomanifold property with empty boundary) can be reduced to the simplicial complex *`complex2`* via bistellar moves, i. e. if *`complex1`* and *`complex2`* are *PL*-homeomorphic. Note that in general the problem is undecidable. In this case `fail` is returned.

It is recommended to use a minimal triangulation *`complex2`* for the check if possible.

Internally calls `SCReduceComplexEx` (9.2.14) `(complex1,complex2,1,SCIntFunc.SCChooseMove);`

Example
```
gap> SCBistellarOptions.WriteLevel:=0;; # do not save complexes to disk
gap> obj:=SC([[1,2],[2,3],[3,4],[4,5],[5,6],[6,1]]);; # hexagon
gap> refObj:=SCBdSimplex(2);; # triangle as a (minimal) reference object
gap> SCEquivalent(obj,refObj);
#I  SCReduceComplexEx: complexes are bistellarly equivalent.
true
```

9.2.3 SCExamineComplexBistellar

▷ `SCExamineComplexBistellar(complex)` (method)

Returns: simplicial complex passed as argument with additional properties upon success, `fail` otherwise.

Computes the face lattice, the f-vector, the AS-determinant, the dimension and the maximal vertex label of *complex*.

Example
```
gap> obj:=SC([[1,2],[2,3],[3,4],[4,5],[5,6],[6,1]]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 7"
 Dim=1

/SimplicialComplex]
gap> SCExamineComplexBistellar(obj);
[SimplicialComplex

 Properties known: AltshulerSteinberg, BoundaryEx, Dim, FVector,
                   FacetsEx, HasBoundary, IsPseudoManifold, IsPure,
                   Name, NumFaces[], SkelExs[], Vertices.

 Name="unnamed complex 7"
 Dim=1
 AltshulerSteinberg=0
 FVector=[ 6, 6 ]
 HasBoundary=false
 IsPseudoManifold=true
 IsPure=true

/SimplicialComplex]
```

9.2.4 SCIntFunc.SCChooseMove

▷ `SCIntFunc.SCChooseMove(dim, moves)` (function)

Returns: a bistellar move, i. e. a pair of lists upon success, `fail` otherwise.

Since the problem of finding a bistellar flip sequence that reduces a simplicial complex is undecidable, we have to use an heuristic approach to choose the next move.

The implemented strategy `SCIntFunc.SCChooseMove` first tries to directly remove vertices, edges, i-faces in increasing dimension etc. If this is not possible it inserts high dimensional faces in decreasing co-dimension. To do this in an efficient way a number of parameters have to be adjusted, namely `SCBistellarOptions.BaseHeating` and `SCBistellarOptions.BaseRelaxation`. See `SCBistellarOptions` (9.2.1) for further options.

If this strategy does not work for you, just implement a customized strategy and pass it to `SCReduceComplexEx` (9.2.14).

See `SCRMoves` (9.2.10) for further information.

9.2.5 SCIsKStackedSphere

▷ `SCIsKStackedSphere(complex, k)` (method)

Returns: a list upon success, `fail` otherwise.

Checks, whether the given simplicial complex *complex* that must be a PL d-sphere is a k-stacked sphere with $1 \leq k \leq \lfloor \frac{d+2}{2} \rfloor$ using a randomized algorithm based on bistellar moves (see [Eff11b], [Eff11a]). Note that it is not checked whether *complex* is a PL sphere – if not, the algorithm will not succeed. Returns a list upon success: the first entry is a boolean, where `true` means that the complex is `k`-stacked and `false` means that the complex cannot be k-stacked. A value of -1 means that the question could not be decided. The second argument contains a simplicial complex that, in case of success, contains the trigangulated $(d+1)$-ball B with $\partial B = S$ and $\mathrm{skel}_{d-k}(B) = \mathrm{skel}_{d-k}(S)$, where S denotes the simplicial complex passed in *complex*.

Internally calls `SCReduceComplexEx` (9.2.14).

Example
```
gap> SCLib.SearchByName("S^4~S^1");
[ [ 463, "S^4~S^1 (VT)" ], [ 1473, "S^4~S^1 (VT)" ], [ 1474, "S^4~S^1 (VT)" ],
  [ 2477, "S^4~S^1 (VT)" ], [ 4395, "S^4~S^1 (VT)" ],
  [ 4396, "S^4~S^1 (VT)" ], [ 4397, "S^4~S^1 (VT)" ],
  [ 4398, "S^4~S^1 (VT)" ], [ 4399, "S^4~S^1 (VT)" ],
  [ 4402, "S^4~S^1 (VT)" ], [ 4403, "S^4~S^1 (VT)" ],
  [ 4404, "S^4~S^1 (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> l:=SCLink(c,1);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 1 ]) in S^4~S^1 (VT)"
 Dim=4

/SimplicialComplex]
gap> SCIsKStackedSphere(l,1);
#I  SCIsKStackedSphere: checking if complex is a 1-stacked sphere...
#I  SCIsKStackedSphere: try 1/50
#I  SCIsKStackedSphere: complex is a 1-stacked sphere.
[ true, [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="Filled 1-stacked sphere (lk([ 1 ]) in S^4~S^1 (VT))"
     Dim=5

    /SimplicialComplex] ]
```

9.2.6 SCBistellarIsManifold

▷ `SCBistellarIsManifold(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Tries to prove that a closed simplicial d-pseudomanifold is a combinatorial manifold by reducing its vertex links to the boundary of the d-simplex.

false is returned if it can be proven that there exists a vertex link which is not PL-homeomorphic to the standard PL-sphere, true is returned if all vertex links are bistellarly equivalent to the boundary of the simplex, fail is returned if the algorithm does not terminate after the number of rounds indicated by SCBistellarOptions.MaxIntervalIsManifold.

Internally calls SCReduceComplexEx (9.2.14) (link,SCEmpty(),0,SCIntFunc.SCChooseMove); for every link of *complex*. Note that false is returned in case of a bounded manifold.

See SCIsManifoldEx (12.1.18) and SCIsManifold (12.1.17) for alternative methods for manifold verification.

Example
```
gap> c:=SCBdCrossPolytope(3);;
gap> SCBistellarIsManifold(c);
true
```

9.2.7 SCIsMovableComplex

▷ SCIsMovableComplex(*complex*) (method)

Returns: true or false upon success, fail otherwise.

Checks if a simplicial complex *complex* can be modified by bistellar moves, i. e. if it is a pure simplicial complex which fulfills the weak pseudomanifold property with empty boundary.

Example
```
gap> c:=SCBdCrossPolytope(3);;
gap> SCIsMovableComplex(c);
true
```

Complex with non-empty boundary:

Example
```
gap> c:=SC([[1,2],[2,3],[3,4],[3,1]]);;
gap> SCIsMovableComplex(c);
false
```

9.2.8 SCMove

▷ SCMove(*c*, *move*) (method)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Applies the bistellar move *move* to a simplicial complex *c*. *move* is given as a $(r+1)$-tuple together with a $(d+1-r)$-tuple if d is the dimension of *c* and if *move* is a r-move. See SCRMoves (9.2.10) for detailed information about bistellar r-moves.

Note: *move* and *c* should be given in standard labeling to ensure a correct result.

Example
```
gap> obj:=SC([[1,2],[2,3],[3,4],[4,1]]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 5"
 Dim=1
```

```
/SimplicialComplex]
gap> moves:=SCMoves(obj);
[ [ [ [ 1, 2 ], [  ] ], [ [ 1, 4 ], [  ] ], [ [ 2, 3 ], [  ] ],
      [ [ 3, 4 ], [  ] ] ],
  [ [ [ 1 ], [ 2, 4 ] ], [ [ 2 ], [ 1, 3 ] ], [ [ 3 ], [ 2, 4 ] ],
      [ [ 4 ], [ 1, 3 ] ] ] ]
gap> obj:=SCMove(obj,last[2][1]);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, NumFaces[], SkelExs[],
                   Vertices.

 Name="unnamed complex 6"
 Dim=1

/SimplicialComplex]
```

9.2.9 SCMoves

▷ SCMoves(*complex*) (method)

Returns: a list of list of pairs of lists upon success, `fail` otherwise.

See SCRMoves (9.2.10) for further information.

Example

```
gap> c:=SCBdCrossPolytope(3);;
gap> moves:=SCMoves(c);
[ [ [ [ 1, 3, 5 ], [  ] ], [ [ 1, 3, 6 ], [  ] ], [ [ 1, 4, 5 ], [  ] ],
      [ [ 1, 4, 6 ], [  ] ], [ [ 2, 3, 5 ], [  ] ], [ [ 2, 3, 6 ], [  ] ],
      [ [ 2, 4, 5 ], [  ] ], [ [ 2, 4, 6 ], [  ] ] ],
  [ [ [ 1, 3 ], [ 5, 6 ] ], [ [ 1, 4 ], [ 5, 6 ] ], [ [ 1, 5 ], [ 3, 4 ] ],
      [ [ 1, 6 ], [ 3, 4 ] ], [ [ 2, 3 ], [ 5, 6 ] ], [ [ 2, 4 ], [ 5, 6 ] ],
      [ [ 2, 5 ], [ 3, 4 ] ], [ [ 2, 6 ], [ 3, 4 ] ], [ [ 3, 5 ], [ 1, 2 ] ],
      [ [ 3, 6 ], [ 1, 2 ] ], [ [ 4, 5 ], [ 1, 2 ] ], [ [ 4, 6 ], [ 1, 2 ] ] ]
    , [  ] ]
```

9.2.10 SCRMoves

▷ SCRMoves(*complex, r*) (method)

Returns: a list of pairs of the form `[ list, list ]`, `fail` otherwise.

A bistellar r-move of a d-dimensional combinatorial manifold *complex* is a r-face m_1 together with a $d-r$-tuple m_2 where m_1 is a common face of exactly $(d+1-r)$ facets and m_2 is not a face of *complex*.

The r-move removes all facets containing m_1 and replaces them by the $(r+1)$ faces obtained by uniting m_2 with any subset of m_1 of order r.

The resulting complex is PL-homeomorphic to *complex*.

Example

```
gap> c:=SCBdCrossPolytope(3);;
gap> moves:=SCRMoves(c,1);
```

```
[ [ [ 1, 3 ], [ 5, 6 ] ], [ [ 1, 4 ], [ 5, 6 ] ], [ [ 1, 5 ], [ 3, 4 ] ],
  [ [ 1, 6 ], [ 3, 4 ] ], [ [ 2, 3 ], [ 5, 6 ] ], [ [ 2, 4 ], [ 5, 6 ] ],
  [ [ 2, 5 ], [ 3, 4 ] ], [ [ 2, 6 ], [ 3, 4 ] ], [ [ 3, 5 ], [ 1, 2 ] ],
  [ [ 3, 6 ], [ 1, 2 ] ], [ [ 4, 5 ], [ 1, 2 ] ], [ [ 4, 6 ], [ 1, 2 ] ] ]
```

9.2.11 SCRandomize

▷ `SCRandomize(`*`complex[[, rounds][, allowedmoves]]`*`)` (function)

Returns: a simplicial complex upon success, `fail` otherwise.

Randomizes the given simplicial complex *`complex`* via bistellar moves chosen at random. By passing the optional array *`allowedmoves`*, which has to be a dense array of integer values of length `SCDim(complex)`, certain moves can be allowed or forbidden in the proccess. An entry `allowedmoves[i]=1` allows $(i-1)$-moves and an entry `allowedmoves[i]=0` forbids $(i-1)$-moves in the randomization process.

With optional positive integer argument *`rounds`*, the amount of randomization can be controlled. The higher the value of *`rounds`*, the more bistellar moves will be randomly performed on *`complex`*. Note that the argument *`rounds`* overrides the global setting `SCBistellarOptions.MaxIntervalRandomize` (this value is used, if *`rounds`* is not specified). Internally calls `SCReduceComplexEx` (9.2.14).

Example
```
gap> c:=SCRandomize(SCBdSimplex(4));
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="Randomized S^3_5"
 Dim=3

/SimplicialComplex]
gap> c.F;
[ 17, 74, 114, 57 ]
```

9.2.12 SCReduceAsSubcomplex

▷ `SCReduceAsSubcomplex(`*`complex1, complex2`*`)` (method)

Returns: `SCBistellarOptions.WriteLevel=0`: a triple of the form `[ boolean, simplicial complex, rounds performed ]` upon termination of the algorithm.

`SCBistellarOptions.WriteLevel=1`: A library of simplicial complexes with a number of complexes from the reducing process and (upon termination) a triple of the form `[ boolean, simplicial complex, rounds performed ]`.

`SCBistellarOptions.WriteLevel=2`: A mail in case a smaller version of *`complex1`* was found, a library of simplicial complexes with a number of complexes from the reducing process and (upon termination) a triple of the form `[ boolean, simplicial complex, rounds performed ]`.

Returns `fail` upon an error.

Reduces a simplicial complex *complex1* (satisfying the weak pseudomanifold property with empty boundary) as a sub-complex of the simplicial complex *complex2*.

Main application: Reduce a sub-complex of the cross polytope without introducing diagonals.

Internally calls `SCReduceComplexEx` (9.2.14) `(complex1,complex2,2,SCIntFunc.SCChooseMove);`

Example
```
gap> c:=SCFromFacets([[1,3],[3,5],[4,5],[4,1]]);;
gap> SCBistellarOptions.WriteLevel:=0;; # do not save complexes
gap> SCReduceAsSubcomplex(c,SCBdCrossPolytope(3));
[ true, [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="unnamed complex 36"
     Dim=1

    /SimplicialComplex], 1 ]
```

9.2.13 SCReduceComplex

▷ `SCReduceComplex(complex)` (method)

Returns: `SCBistellarOptions.WriteLevel=0`: a triple of the form `[ boolean, simplicial complex, rounds performed ]` upon termination of the algorithm.

`SCBistellarOptions.WriteLevel=1`: A library of simplicial complexes with a number of complexes from the reducing process and (upon termination) a triple of the form `[ boolean, simplicial complex, rounds performed ]`.

`SCBistellarOptions.WriteLevel=2`: A mail in case a smaller version of *complex1* was found, a library of simplicial complexes with a number of complexes from the reducing process and (upon termination) a triple of the form `[ boolean, simplicial complex, rounds performed ]`.

Returns `fail` upon an error..

Reduces a pure simplicial complex *complex* satisfying the weak pseudomanifold property via bistellar moves. Internally calls `SCReduceComplexEx` (9.2.14) `(complex,SCEmpty(),0,SCIntFunc.SCChooseMove);`

Example
```
gap> obj:=SC([[1,2],[2,3],[3,4],[4,5],[5,6],[6,1]]);; # hexagon
gap> SCBistellarOptions.WriteLevel:=0;; # do not save complexes
gap> tmp := SCReduceComplex(obj);
[ true, [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="unnamed complex 27"
     Dim=1

    /SimplicialComplex], 3 ]
```

9.2.14 SCReduceComplexEx

▷ `SCReduceComplexEx(complex, refComplex, mode, choosemove)` (function)

Returns: `SCBistellarOptions.WriteLevel=0`: a triple of the form `[ boolean, simplicial complex, rounds ]` upon termination of the algorithm.

`SCBistellarOptions.WriteLevel=1`: A library of simplicial complexes with a number of complexes from the reducing process and (upon termination) a triple of the form `[ boolean, simplicial complex, rounds ]`.

`SCBistellarOptions.WriteLevel=2`: A mail in case a smaller version of *complex1* was found, a library of simplicial complexes with a number of complexes from the reducing process and (upon termination) a triple of the form `[ boolean, simplicial complex, rounds ]`.

Returns `fail` upon an error.

Reduces a pure simplicial complex *complex* satisfying the weak pseudomanifold property via bistellar moves *mode = 0*, compares it to the simplicial complex *refComplex* (*mode = 1*) or reduces it as a sub-complex of *refComplex* (*mode = 2*).

choosemove is a function containing a flip strategy, see also `SCIntFunc.SCChooseMove` (9.2.4).

The currently smallest complex is stored to the variable `minComplex`, the currently smallest f-vector to `minF`. Note that in general the algorithm will not stop until the maximum number of rounds is reached. You can adjust the maximum number of rounds via the property `SCBistellarOptions` (9.2.1). The number of rounds performed is returned in the third entry of the triple returned by this function.

This function is called by

1. `SCReduceComplex` (9.2.13),
2. `SCEquivalent` (9.2.2),
3. `SCReduceAsSubcomplex` (9.2.12),
4. `SCBistellarIsManifold` (9.2.6).
5. `SCRandomize` (9.2.11).

Please see `SCMailIsPending` (15.2.3) for further information about the email notification system in case `SCBistellarOptions.WriteLevel` is set to 2.

Example
```
gap> c:=SCBdCrossPolytope(4);;
gap> SCBistellarOptions.WriteLevel:=0;; # do not save complexes
gap> SCReduceComplexEx(c,SCEmpty(),0,SCIntFunc.SCChooseMove);
[ true, [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="unnamed complex 13"
     Dim=3

    /SimplicialComplex], 7 ]
gap> SCReduceComplexEx(c,SCEmpty(),0,SCIntFunc.SCChooseMove);
[ true, [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.
```

```
     Name="unnamed complex 18"
     Dim=3

    /SimplicialComplex], 7 ]
gap> SCMailSetAddress("johndoe@somehost");
true
gap> SCMailIsEnabled();
true
gap> SCReduceComplexEx(c,SCEmpty(),0,SCIntFunc.SCChooseMove);
[ true, [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="unnamed complex 23"
     Dim=3

    /SimplicialComplex], 7 ]
```

Content of sent mail:

Example

```
Greetings master,

this is simpcomp %%%VERSION%%% running on igt215.mathematik.uni-stuttgart.de
(Linux igt215 2.6.26-2-amd64 #1 SMP Thu Nov 5 02:23:12 UTC 2009 x86_64
GNU/Linux), GAP 4.4.12.

I have been working hard for 0 seconds and have a message for you, see below.

#### START MESSAGE ####

SCReduceComplex:

Computed locally minimal complex after 7 rounds:

[SimplicialComplex

 Properties known: Boundary, Chi, Date, Dim, F, Faces, Facets, G, H,
 HasBoundary, Homology, IsConnected, IsManifold, IsPM, Name, SCVertices,
 Vertices.

 Name="ReducedComplex_5_vertices_7"
 Dim=3
 Chi=0
 F=[ 5, 10, 10, 5 ]
 G=[ 0, 0 ]
 H=[ 1, 1, 1, 1 ]
 HasBoundary=false
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsConnected=true
 IsPM=true
```

```
/SimplicialComplex]

##### END MESSAGE #####

That's all, I hope this is good news! Have a nice day.
```

9.2.15 SCReduceComplexFast

▷ `SCReduceComplexFast(complex)` (function)

Returns: a simplicial complex upon success, `fail` otherwise.

Same as `SCReduceComplex` (9.2.13), but calls an external binary provided with the simpcomp package.

Chapter 10

Simplicial blowups

10.1 Theory

In this chapter functions are provided to perform simplicial blowups as well as the resolution of isolated singularities of certain types of combinatorial 4-manifolds. As of today singularities where the link is homeomorphic to $\mathbb{R}P^3$, $S^2 \times S^1$, $S^2 \underline{\times} S^1$ and the lens spaces $L(k,1)$ are supported. In addition, the program provides the possibility to hand over additional types of mapping cylinders to cover other types of singularities.

Please note that the program is based on a heuristic algorithm using bistellar moves. Hence, the search for a suitable sequence of bistellar moves to perform the blowup does not always terminate. However, especially in the case of ordinary double points (singularities of type $\mathbb{R}P^3$), a lot of blowups have already been successful. For a very short introduction to simplicial blowups see 2.8, for further information see [SK11].

10.2 Functions related to simplicial blowups

10.2.1 SCBlowup

▷ `SCBlowup(`*`pseudomanifold, singularity[, mappingCyl]`*`)` (property)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

If `singularity` is an ordinary double point of a combinatorial 4-pseudomanifold *`pseudomanifold`* (lk(`singularity`) = $\mathbb{R}P^3$) the blowup of `pseudomanifold` at `singularity` is computed. If it is a singularity of type $S^2 \times S^1$, $S^2 \underline{\times} S^1$ or $L(k,1)$, $k \leq 4$, the canonical resolution of `singularity` is computed using the bounded complexes provided in the source code below.

If the optional argument `mappingCyl` of type `SCIsSimplicialComplex` is given, this complex will be used to to resolve the singularity `singularity`.

Note that bistellar moves do not necessarily preserve any orientation. Thus, the orientation of the blowup has to be checked in order to verify which type of blowup was performed. Normally, repeated computation results in both versions.

Example

```
gap> SCLib.SearchByName("Kummer variety");
[ [ 7493, "4-dimensional Kummer variety (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> d:= SCBlowup(c,1);
#I  SCBlowup: checking if singularity is a combinatorial manifold...
```

```
#I  SCBlowup: ...true
#I  SCBlowup: checking type of singularity...
#I  SCReduceComplexEx: complexes are bistellarly equivalent.
#I  SCBlowup: ...ordinary double point (supported type).
#I  SCBlowup: starting blowup...
#I  SCBlowup: map boundaries...
#I  SCBlowup: boundaries not isomorphic, initializing bistellar moves...
#I  SCBlowup: found complex with smaller boundary: f = [ 15, 74, 118, 59 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 14, 70, 112, 56 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 14, 67, 106, 53 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 13, 65, 104, 52 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 13, 64, 102, 51 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 13, 63, 100, 50 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 13, 62, 98, 49 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 12, 58, 92, 46 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 12, 57, 90, 45 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 12, 56, 88, 44 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 12, 55, 86, 43 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 11, 51, 80, 40 ].
#I  SCBlowup: found complex with isomorphic boundaries.
#I  SCBlowup: ...boundaries mapped succesfully.
#I  SCBlowup: build complex...
#I  SCBlowup: ...done.
#I  SCBlowup: ...blowup completed.
#I  SCBlowup: You may now want to reduce the complex via 'SCReduceComplex'.
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 4084 \ star([ 1 ]) in unnamed complex 4084 cup unnamed \
complex 4088 cup unnamed complex 4086"
 Dim=4

/SimplicialComplex]
```

Example

```
gap> # resolving the singularities of a 4 dimensional Kummer variety
gap> SCLib.SearchByName("Kummer variety");
[ [ 7488, "4-dimensional Kummer variety (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> for i in [1..16] do
       for j in SCLabels(c) do
         lk:=SCLink(c,j);
         if lk.Homology = [[0],[0],[0],[1]] then continue; fi;
         singularity := j; break;
       od;
       c:=SCBlowup(c,singularity);
     od;
gap> d.IsManifold;
true
gap> d.Homology;
[ [ 0, [ ] ], [ 0, [ ] ], [ 22, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
```

10.2.2 SCMappingCylinder

▷ SCMappingCylinder(*k*) (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Generates a bounded version of $\mathbb{C}P^2$ (a so-called mapping cylinder for a simplicial blowup, compare [SK11]) with boundary $L(\mathtt{k},1)$.

Example
```
gap> mapCyl:=SCMappingCylinder(3);;
gap> mapCyl.Homology;
[ [ 0, [  ] ], [ 0, [  ] ], [ 1, [  ] ], [ 0, [  ] ], [ 0, [  ] ] ]
gap> l31:=SCBoundary(mapCyl);;
gap> l31.Homology;
[ [ 0, [  ] ], [ 0, [ 3 ] ], [ 0, [  ] ], [ 1, [  ] ] ]
```

Chapter 11

Polyhedral Morse theory

In this chapter we present some useful functions dealing with polyhedral Morse theory. See Section 2.5 for a very short introduction to the field, see [Küh95] for more information. Note: this is not to be confused with Robin Forman's discrete Morse theory for cell complexes which is described in Chapter 12.

If M is a combinatorial d-manifold with n-vertices a rsl-function will be represented as an ordering on the set of vertices, i. e. a list of length n containing all vertex labels of the corresponding simplicial complex.

11.1 Polyhedral Morse theory related functions

11.1.1 SCIsTight

▷ `SCIsTight(complex)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks whether a simplicial complex `complex` is a tight triangulation or not. A simplicial complex with n vertices is said to be a tight triangulation if it can be tightly embedded into the $(n-1)$-simplex. See Section 2.7 for a short introduction to the field of tightness.

First, if `complex` is a $(k+1)$-neighborly $2k$-manifold (cf. [Küh95], Corollary 4.7) or if `complex` is of dimension $d \geq 4$, 2-neighborly and all its vertex links are stacked spheres (i.e. the complex is in Walkup's class $K(d)$, see [Eff11b]), `true` is returned as the complex is a tight triangulation in these cases. Note that it is not computed whether or not `complex` is a combinatorial manifold as this computation might take a long time. Hence, only if the manifold flag of the complex is set (this can be achieved by calling `SCIsManifold` (12.1.17) and the complex indeed is a combinatorial manifold) these checks are performed. In a second step, the algorithm first checks certain rsl-functions allowing slicings between minimal non faces and the rest of the complex. In most cases where `complex` is not tight at least one of these rsl-functions is not perfect and thus `false` is returned as the complex is not a tight triangulation.

If the complex passed all checks so far, the remaining rsl-functions are checked for being perfect functions. As there are "only" 2^n different multiplicity vectors, but $n!$ different rsl-functions, a lookup table containing all possible multiplicity vectors is computed first. Note that nonetheless the complexity of this algorithm is $O(n!)$.

In order to reduce the number of rsl-functions that need to be checked, the automorphism group of `complex` is computed first using `SCAutomorphismGroup` (6.9.2). In case it is k-transitive, the complexity is reduced by the factor of $n\cdot(n-1)\cdot\cdots\cdot(n-k+1)$.

Example

```
gap> list:=SCLib.SearchByName("S^2~S^1 (VT)");
[ [ 12, "S^2~S^1 (VT)" ], [ 27, "S^2~S^1 (VT)" ], [ 28, "S^2~S^1 (VT)" ],
  [ 43, "S^2~S^1 (VT)" ], [ 47, "S^2~S^1 (VT)" ], [ 49, "S^2~S^1 (VT)" ],
  [ 89, "S^2~S^1 (VT)" ], [ 90, "S^2~S^1 (VT)" ], [ 111, "S^2~S^1 (VT)" ],
  [ 140, "S^2~S^1 (VT)" ], [ 146, "S^2~S^1 (VT)" ], [ 147, "S^2~S^1 (VT)" ],
  [ 148, "S^2~S^1 (VT)" ], [ 149, "S^2~S^1 (VT)" ], [ 150, "S^2~S^1 (VT)" ],
  [ 156, "S^2~S^1 (VT)" ], [ 157, "S^2~S^1 (VT)" ], [ 391, "S^2~S^1 (VT)" ],
  [ 393, "S^2~S^1 (VT)" ], [ 394, "S^2~S^1 (VT)" ], [ 396, "S^2~S^1 (VT)" ],
  [ 407, "S^2~S^1 (VT)" ], [ 408, "S^2~S^1 (VT)" ], [ 410, "S^2~S^1 (VT)" ],
  [ 412, "S^2~S^1 (VT)" ], [ 413, "S^2~S^1 (VT)" ], [ 578, "S^2~S^1 (VT)" ],
  [ 579, "S^2~S^1 (VT)" ], [ 582, "S^2~S^1 (VT)" ], [ 596, "S^2~S^1 (VT)" ],
  [ 597, "S^2~S^1 (VT)" ], [ 598, "S^2~S^1 (VT)" ], [ 640, "S^2~S^1 (VT)" ],
  [ 642, "S^2~S^1 (VT)" ], [ 644, "S^2~S^1 (VT)" ], [ 645, "S^2~S^1 (VT)" ],
  [ 769, "S^2~S^1 (VT)" ], [ 770, "S^2~S^1 (VT)" ], [ 774, "S^2~S^1 (VT)" ],
  [ 775, "S^2~S^1 (VT)" ], [ 776, "S^2~S^1 (VT)" ], [ 2401, "S^2~S^1 (VT)" ],
  [ 2409, "S^2~S^1 (VT)" ], [ 2410, "S^2~S^1 (VT)" ],
  [ 2411, "S^2~S^1 (VT)" ], [ 2428, "S^2~S^1 (VT)" ],
  [ 2429, "S^2~S^1 (VT)" ], [ 2430, "S^2~S^1 (VT)" ],
  [ 2431, "S^2~S^1 (VT)" ], [ 2432, "S^2~S^1 (VT)" ],
  [ 2433, "S^2~S^1 (VT)" ], [ 2434, "S^2~S^1 (VT)" ],
  [ 2435, "S^2~S^1 (VT)" ], [ 2500, "S^2~S^1 (VT)" ],
  [ 2501, "S^2~S^1 (VT)" ], [ 2504, "S^2~S^1 (VT)" ],
  [ 2505, "S^2~S^1 (VT)" ], [ 2506, "S^2~S^1 (VT)" ],
  [ 2510, "S^2~S^1 (VT)" ], [ 2511, "S^2~S^1 (VT)" ],
  [ 2512, "S^2~S^1 (VT)" ], [ 2513, "S^2~S^1 (VT)" ],
  [ 2514, "S^2~S^1 (VT)" ], [ 2515, "S^2~S^1 (VT)" ] ]
gap> s2s1:=SCLib.Load(list[1][1]);
[SimplicialComplex

 Properties known: AltshulerSteinberg, AutomorphismGroup,
                   AutomorphismGroupSize, AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, ConnectedComponents,
                   Dim, DualGraph, EulerCharacteristic, FVector,
                   FacetsEx, GVector, GeneratorsEx, HVector,
                   HasBoundary, HasInterior, Homology, Interior,
                   IsCentrallySymmetric, IsConnected,
                   IsEulerianManifold, IsManifold, IsOrientable,
                   IsPseudoManifold, IsPure, IsStronglyConnected,
                   MinimalNonFacesEx, Name, Neighborliness,
                   NumFaces[], Orientation, Reference, SkelExs[],
                   Vertices.

 Name="S^2~S^1 (VT)"
 Dim=3
 AltshulerSteinberg=250838208
 AutomorphismGroupSize=18
 AutomorphismGroupStructure="D18"
 AutomorphismGroupTransitivity=1
 EulerCharacteristic=0
 FVector=[ 9, 36, 54, 27 ]
 GVector=[ 4, 10 ]
```

```
 HVector=[ 5, 15, 5, 1 ]
 HasBoundary=false
 HasInterior=true
 Homology=[ [ 0, [ ] ], [ 1, [ ] ], [ 0, [ 2 ] ], [ 0, [ ] ] ]
 IsCentrallySymmetric=false
 IsConnected=true
 IsEulerianManifold=true
 IsOrientable=false
 IsPseudoManifold=true
 IsPure=true
 IsStronglyConnected=true
 Neighborliness=2

/SimplicialComplex]
gap> SCInfoLevel(2); # print information while running
true
gap> SCIsTight(s2s1); time;
#I  SCIsTight: checking non faces...
#I  SCIsTight: found no contradiction so far.
#I  SCIsTight: generating lookup table...
#I  SCIsTight: lookup table done, size = 2304.
#I  SCIsTight: computing automorphism group...
#I  SCIsTight: automorphism group done, transitivity = 1.
#I  SCIsTight: checking rsl-functions...
#I  SCIsTight: processed 10000 of 40320 rsl-functions so far, all perfect.
#I  SCIsTight: processed 20000 of 40320 rsl-functions so far, all perfect.
#I  SCIsTight: processed 30000 of 40320 rsl-functions so far, all perfect.
#I  SCIsTight: processed 40000 of 40320 rsl-functions so far, all perfect.
true
24592
```

Example

```
gap> SCLib.SearchByAttribute("F[1] = 120");
[ [ 7647, "Bd(600-cell)" ] ]
gap> id:=last[1][1];;
gap> c:=SCLib.Load(id);;
gap> SCIsTight(c); time;
#I  SCIsTight: checking non faces...
#I  SCIsTight: found non perfect rsl-function: [ 1, 3, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
  22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
  41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
  60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78,
  79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
  98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112,
  113, 114, 115, 116, 117, 118, 119, 120 ], complex not tight.
false
52
```

Example

```
gap> SCInfoLevel(0);
true
gap> SCLib.SearchByName("K3");
```

```
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCIsManifold(c);
true
gap> SCInfoLevel(2);
true
gap> c.IsTight;
#I  SCIsTight: complex is (k+1)-neighborly 2k-manifold and thus tight.
true
```

Example

```
gap> list:=SCLib.SearchByName("S^3xS^1");
[ [ 55, "S^3xS^1 (VT)" ], [ 128, "S^3xS^1 (VT)" ], [ 399, "S^3xS^1 (VT)" ],
  [ 459, "S^3xS^1 (VT)" ], [ 460, "S^3xS^1 (VT)" ], [ 461, "S^3xS^1 (VT)" ],
  [ 462, "S^3xS^1 (VT)" ], [ 588, "S^3xS^1 (VT)" ], [ 612, "S^3xS^1 (VT)" ],
  [ 699, "S^3xS^1 (VT)" ], [ 700, "S^3xS^1 (VT)" ], [ 701, "S^3xS^1 (VT)" ],
  [ 703, "S^3xS^1 (VT)" ], [ 1078, "S^3xS^1 (VT)" ], [ 1080, "S^3xS^1 (VT)" ],
  [ 1081, "S^3xS^1 (VT)" ], [ 1082, "S^3xS^1 (VT)" ],
  [ 1083, "S^3xS^1 (VT)" ], [ 1084, "S^3xS^1 (VT)" ],
  [ 1085, "S^3xS^1 (VT)" ], [ 1086, "S^3xS^1 (VT)" ],
  [ 1087, "S^3xS^1 (VT)" ], [ 1088, "S^3xS^1 (VT)" ],
  [ 1089, "S^3xS^1 (VT)" ], [ 1091, "S^3xS^1 (VT)" ],
  [ 2413, "S^3xS^1 (VT)" ], [ 2470, "S^3xS^1 (VT)" ],
  [ 2471, "S^3xS^1 (VT)" ], [ 2472, "S^3xS^1 (VT)" ],
  [ 2473, "S^3xS^1 (VT)" ], [ 2474, "S^3xS^1 (VT)" ],
  [ 2475, "S^3xS^1 (VT)" ], [ 2476, "S^3xS^1 (VT)" ],
  [ 3413, "S^3xS^1 (VT)" ], [ 3414, "S^3xS^1 (VT)" ],
  [ 3415, "S^3xS^1 (VT)" ], [ 3416, "S^3xS^1 (VT)" ],
  [ 3417, "S^3xS^1 (VT)" ], [ 3418, "S^3xS^1 (VT)" ],
  [ 3419, "S^3xS^1 (VT)" ], [ 3420, "S^3xS^1 (VT)" ],
  [ 3421, "S^3xS^1 (VT)" ], [ 3422, "S^3xS^1 (VT)" ],
  [ 3423, "S^3xS^1 (VT)" ], [ 3424, "S^3xS^1 (VT)" ],
  [ 3425, "S^3xS^1 (VT)" ], [ 3426, "S^3xS^1 (VT)" ],
  [ 3427, "S^3xS^1 (VT)" ], [ 3428, "S^3xS^1 (VT)" ],
  [ 3429, "S^3xS^1 (VT)" ], [ 3430, "S^3xS^1 (VT)" ],
  [ 3431, "S^3xS^1 (VT)" ], [ 3432, "S^3xS^1 (VT)" ],
  [ 3433, "S^3xS^1 (VT)" ], [ 3434, "S^3xS^1 (VT)" ] ]
gap> c:=SCLib.Load(list[1][1]);
[SimplicialComplex

 Properties known: AltshulerSteinberg, AutomorphismGroup,
                   AutomorphismGroupSize, AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, ConnectedComponents,
                   Dim, DualGraph, EulerCharacteristic, FVector,
                   FacetsEx, GVector, GeneratorsEx, HVector,
                   HasBoundary, HasInterior, Homology, Interior,
                   IsCentrallySymmetric, IsConnected,
                   IsEulerianManifold, IsManifold, IsOrientable,
                   IsPseudoManifold, IsPure, IsStronglyConnected,
                   MinimalNonFacesEx, Name, Neighborliness,
                   NumFaces[], Orientation, Reference, SkelExs[],
                   Vertices.
```

```
 Name="S^3xS^1 (VT)"
 Dim=4
 AltshulerSteinberg=737125273600
 AutomorphismGroupSize=22
 AutomorphismGroupStructure="D22"
 AutomorphismGroupTransitivity=1
 EulerCharacteristic=0
 FVector=[ 11, 55, 110, 110, 44 ]
 GVector=[ 5, 15, -20 ]
 HVector=[ 6, 21, 1, 16, -1 ]
 HasBoundary=false
 HasInterior=true
 Homology=[ [ 0, [ ] ], [ 1, [ ] ], [ 0, [ ] ], [ 1, [ ] ], [ 1, [ ] ] ]
 IsCentrallySymmetric=false
 IsConnected=true
 IsEulerianManifold=true
 IsOrientable=true
 IsPseudoManifold=true
 IsPure=true
 IsStronglyConnected=true
 Neighborliness=2

/SimplicialComplex]
gap> SCInfoLevel(0);
true
gap> SCIsManifold(c);
true
gap> SCInfoLevel(2);
true
gap> c.IsTight;
#I  SCIsInKd: complex has transitive automorphism group, only checking one link.
#I  SCIsInKd: checking link 1/1
#I  SCIsKStackedSphere: checking if complex is a 1-stacked sphere...
#I  SCIsKStackedSphere: try 1/50
#I  round 0
Reduced complex, F: [ 9, 26, 34, 17 ]
#I  round 1
Reduced complex, F: [ 8, 22, 28, 14 ]
#I  round 2
Reduced complex, F: [ 7, 18, 22, 11 ]
#I  round 3
Reduced complex, F: [ 6, 14, 16, 8 ]
#I  round 4
Reduced complex, F: [ 5, 10, 10, 5 ]
#I  SCReduceComplexEx: computed locally minimal complex after 5 rounds.
#I  SCIsKStackedSphere: complex is a 1-stacked sphere.
#I  SCIsInKd: complex has transitive automorphism group, all links are 1-stacked.
#I  SCIsTight: complex is in class K(1) and 2-neighborly, thus tight.
true
```

11.1.2 SCMorseIsPerfect

▷ `SCMorseIsPerfect(c, f)` (method)

Returns: `true` or `false` upon success, `fail` otherwise.

Checks whether the rsl-function `f` is perfect on the simplicial complex `c` or not. A rsl-function is said to be perfect, if it has the minimum number of critical points, i. e. if the sum of its critical points equals the sum of the Betti numbers of `c`.

Example
```
gap> c:=SCBdCyclicPolytope(4,6);;
gap> SCMinimalNonFaces(c);
[ [  ], [  ], [ [ 1, 3, 5 ], [ 2, 4, 6 ] ] ]
gap> SCMorseIsPerfect(c,[1..6]);
true
gap> SCMorseIsPerfect(c,[1,3,5,2,4,6]);
false
```

11.1.3 SCSlicing

▷ `SCSlicing(complex, slicing)` (method)

Returns: a facet list of a polyhedral complex or a `SCNormalSurface` object upon success, `fail` otherwise.

Returns the pre-image $f^{-1}(\alpha)$ of a rsl-function f on the simplicial complex *complex* where f is given in the second argument *slicing* by a partition of the set of vertices *slicing*$=[V_1,V_2]$ such that $f(v_1)$ ($f(v_2)$) is smaller (greater) than α for all $v_1 \in V_1$ ($v_2 \in V_2$).

If *complex* is of dimension 3, a GAP object of type `SCNormalSurface` is returned. Otherwise only the facet list is returned. See also `SCNSSlicing` (7.1.4).

The vertex labels of the returned slicing are of the form (v_1,v_2) where $v_1 \in V_1$ and $v_2 \in V_2$. They represent the center points of the edges $\rangle v_1,v_2 \langle$ defined by the intersection of *slicing* with *complex*.

Example
```
gap> c:=SCBdCyclicPolytope(4,6);;
gap> v:=SCVertices(c);
[ 1, 2, 3, 4, 5, 6 ]
gap> SCMinimalNonFaces(c);
[ [  ], [  ], [ [ 1, 3, 5 ], [ 2, 4, 6 ] ] ]
gap> ns:=SCSlicing(c,[v{[1,3,5]},v{[2,4,6]}]);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector, Fac\
etsEx, Genus, IsConnected, IsOrientable, NSTriangulation, Name, TopologicalTyp\
e, Vertices.

 Name="slicing [ [ 1, 3, 5 ], [ 2, 4, 6 ] ] of Bd(C_4(6))"
 Dim=2
 FVector=[ 9, 18, 0, 9 ]
 EulerCharacteristic=0
 IsOrientable=true
 TopologicalType="T^2"

/NormalSurface]
```

Example

```
gap> c:=SCBdSimplex(5);;
gap> v:=SCVertices(c);
[ 1, 2, 3, 4, 5, 6 ]
gap> slicing:=SCSlicing(c,[v{[1,3,5]},v{[2,4,6]}]);
[ [ [ 1, 2 ], [ 1, 4 ], [ 3, 2 ], [ 3, 4 ], [ 5, 2 ], [ 5, 4 ] ],
  [ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 3, 2 ], [ 3, 4 ], [ 3, 6 ] ],
  [ [ 1, 2 ], [ 1, 6 ], [ 3, 2 ], [ 3, 6 ], [ 5, 2 ], [ 5, 6 ] ],
  [ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 5, 2 ], [ 5, 4 ], [ 5, 6 ] ],
  [ [ 1, 4 ], [ 1, 6 ], [ 3, 4 ], [ 3, 6 ], [ 5, 4 ], [ 5, 6 ] ],
  [ [ 3, 2 ], [ 3, 4 ], [ 3, 6 ], [ 5, 2 ], [ 5, 4 ], [ 5, 6 ] ] ]
```

11.1.4 SCMorseMultiplicityVector

▷ `SCMorseMultiplicityVector(c, f)` (method)

Returns: a list of $(d+1)$-tuples if `c` is a d-dimensional simplicial complex upon success, `fail` otherwise.

Computes all multiplicity vectors of a rsl-function `f` on a simplicial complex `c`. `f` is given as an ordered list $(v_1,\dots v_n)$ of all vertices of `c` where `f` is defined by $\mathtt{f}(v_i) = \frac{i-1}{n-1}$. The i-th entry of the returned list denotes the multiplicity vector of vertex v_i.

Example

```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> f:=SCVertices(c);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 ]
gap> SCMorseMultiplicityVector(c,f);
[ [ 1, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0 ], [ 0, 0, 1, 0, 0 ],
  [ 0, 0, 2, 0, 0 ], [ 0, 0, 1, 0, 0 ], [ 0, 0, 4, 0, 0 ], [ 0, 0, 3, 0, 0 ],
  [ 0, 0, 3, 0, 0 ], [ 0, 0, 4, 0, 0 ], [ 0, 0, 1, 0, 0 ], [ 0, 0, 2, 0, 0 ],
  [ 0, 0, 1, 0, 0 ], [ 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 1 ] ]
```

11.1.5 SCMorseNumberOfCriticalPoints

▷ `SCMorseNumberOfCriticalPoints(c, f)` (method)

Returns: an integer and a list upon success, `fail` otherwise.

Computes the number of critical points of each index of a rsl-function `f` on a simplicial complex `c` as well as the total number of critical points.

Example

```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> f:=SCVertices(c);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 ]
gap> SCMorseNumberOfCriticalPoints(c,f);
[ 24, [ 1, 0, 22, 0, 1 ] ]
```

Chapter 12

Forman's discrete Morse theory

In this chapter a framework is provided to use Forman's discrete Morse theory [For95] within simpcomp. See Section 2.6 for a brief introduction.

Note: this is not to be confused with Banchoff and Kühnel's theory of regular simplexwise linear functions which is described in Chapter 11.

12.1 Functions using discrete Morse theory

12.1.1 SCCollapseGreedy

▷ `SCCollapseGreedy(`*`complex`*`)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Employs a greedy collapsing algorithm to collapse the simplicial complex *`complex`*. See also `SCCollapseLex` (12.1.2) and `SCCollapseRevLex` (12.1.3).

Example
```
gap> SCLib.SearchByName("T^2"){[1..6]};
[ [ 4, "T^2 (VT)" ], [ 5, "T^2 (VT)" ], [ 9, "T^2 (VT)" ], [ 10, "T^2 (VT)" ],
  [ 18, "T^2 (VT)" ], [ 20, "(T^2)#2" ] ]
gap> torus:=SCLib.Load(last[1][1]);;
gap> bdtorus:=SCDifference(torus,SC([torus.Facets[1]]));;
gap> coll:=SCCollapseGreedy(bdtorus);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="collapsed version of T^2 (VT) \ unnamed complex 8"
 Dim=1

/SimplicialComplex]
gap> coll.Facets;
[ [ 1, 3 ], [ 1, 5 ], [ 2, 3 ], [ 2, 5 ], [ 3, 4 ], [ 3, 7 ], [ 4, 7 ] ]
gap> sphere:=SCBdSimplex(4);;
gap> bdsphere:=SCDifference(sphere,SC([sphere.Facets[1]]));;
gap> coll:=SCCollapseGreedy(bdsphere);
[SimplicialComplex

 Properties known: Dim, FVector, FacetsEx, IsPure, Name, NumFaces[],
```

```
                  SkelExs[], Vertices.

 Name="collapsed version of S^3_5 \ unnamed complex 12"
 Dim=0
 FVector=[ 1 ]
 IsPure=true

/SimplicialComplex]
gap> coll.Facets;
[ [ 5 ] ]
```

12.1.2 SCCollapseLex

▷ `SCCollapseLex(complex)` (method)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Employs a greedy collapsing algorithm in lexicographical order to collapse the simplicial complex *complex*. See also `SCCollapseGreedy` (12.1.1) and `SCCollapseRevLex` (12.1.3).

Example

```
gap> s:=SCSurface(1,true);;
gap> s:=SCDifference(s,SC([SCFacets(s)[1]]));;
gap> coll:=SCCollapseGreedy(s);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="collapsed version of T^2 \ unnamed complex 18"
 Dim=1

/SimplicialComplex]
gap> coll.Facets;
[ [ 1, 3 ], [ 1, 5 ], [ 3, 6 ], [ 3, 7 ], [ 5, 6 ], [ 6, 7 ] ]
gap> sphere:=SCBdSimplex(4);;
gap> ball:=SCDifference(sphere,SC([sphere.Facets[1]]));;
gap> coll:=SCCollapseLex(ball);
[SimplicialComplex

 Properties known: Dim, FVector, FacetsEx, IsPure, Name, NumFaces[],
                  SkelExs[], Vertices.

 Name="collapsed version of S^3_5 \ unnamed complex 22"
 Dim=0
 FVector=[ 1 ]
 IsPure=true

/SimplicialComplex]
gap> coll.Facets;
[ [ 5 ] ]
```

12.1.3 SCCollapseRevLex

▷ SCCollapseRevLex(*complex*) (method)

Returns: simplicial complex of type SCSimplicialComplex upon success, fail otherwise.

Employs a greedy collapsing algorithm in reverse lexicographical order to collapse the simplicial complex *complex*. See also SCCollapseGreedy (12.1.1) and SCCollapseLex (12.1.2).

Example
```
gap> s:=SCSurface(1,true);;
gap> s:=SCDifference(s,SC([SCFacets(s)[1]]));;
gap> coll:=SCCollapseGreedy(s);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="collapsed version of T^2 \ unnamed complex 28"
 Dim=1

/SimplicialComplex]
gap> coll.Facets;
[ [ 1, 3 ], [ 1, 6 ], [ 3, 6 ], [ 3, 7 ], [ 6, 7 ] ]
gap> sphere:=SCBdSimplex(4);;
gap> ball:=SCDifference(sphere,SC([sphere.Facets[1]]));;
gap> coll:=SCCollapseRevLex(ball);
[SimplicialComplex

 Properties known: Dim, FVector, FacetsEx, IsPure, Name, NumFaces[],
                   SkelExs[], Vertices.

 Name="collapsed version of S^3_5 \ unnamed complex 32"
 Dim=0
 FVector=[ 1 ]
 IsPure=true

/SimplicialComplex]
gap> coll.Facets;
[ [ 1 ] ]
```

12.1.4 SCHasseDiagram

▷ SCHasseDiagram(*c*) (function)

Returns: two lists of lists upon success, fail otherweise.

Computes the Hasse diagram of SCSimplicialComplex object *c*. The Hasse diagram is returned as two sets of lists. The first set of lists contains the upward part of the Hasse diagram, the second set of lists contains the downward part of the Hasse diagram.

The i-th list of each set of lists represents the incidences between the $(i-1)$-faces and the i-faces. The faces are given by their indices of the face lattice.

Example
```
gap> c:=SCBdSimplex(3);;
gap> HD:=SCHasseDiagram(c);
[ [ [ [ 1, 2, 3 ], [ 1, 4, 5 ], [ 2, 4, 6 ], [ 3, 5, 6 ] ],
```

```
        [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ], [ 1, 4 ], [ 2, 4 ], [ 3, 4 ] ] ],
    [ [ [ 2, 1 ], [ 3, 1 ], [ 4, 1 ], [ 3, 2 ], [ 4, 2 ], [ 4, 3 ] ],
        [ [ 4, 2, 1 ], [ 5, 3, 1 ], [ 6, 3, 2 ], [ 6, 5, 4 ] ] ] ]
```

12.1.5 SCMorseEngstroem

▷ SCMorseEngstroem(*complex*) (function)

Returns: two lists of small integer lists upon success, `fail` otherweise.

Builds a discrete Morse function following the Engstroem method by reducing the input complex to smaller complexes defined by minimal link and deletion operations. See [Eng09] for details.

Example
```
gap> c:=SCBdSimplex(3);;
gap> f:=SCMorseEngstroem(c);
[ [ [ 2 ], [ 2, 3 ], [ 2, 4 ], [ 2 .. 4 ], [  ], [ 3 ], [ 4 ], [ 3, 4 ],
      [ 1, 3 ], [ 1, 3, 4 ], [ 1 ], [ 1, 4 ], [ 1, 2, 4 ], [ 1, 2 ],
      [ 1 .. 3 ] ], [ [ 2 ], [ 1 .. 3 ] ] ]
```

12.1.6 SCMorseRandom

▷ SCMorseRandom(*complex*) (function)

Returns: two lists of small integer lists upon success, `fail` otherweise.

Builds a discrete Morse function following Lutz and Benedetti's random discrete Morse theory approach: Faces are paired with free co-dimension one faces until now free faces remain. Then a critical face is removed at random. See [BL14] for details.

Example
```
gap> c:=SCBdSimplex(3);;
gap> f:=SCMorseRandom(c);;
gap> Size(f[2]);
2
```

12.1.7 SCMorseRandomLex

▷ SCMorseRandomLex(*complex*) (function)

Returns: two lists of small integer lists upon success, `fail` otherweise.

Builds a discrete Morse function following Adiprasito, Benedetti and Lutz' lexicographic random discrete Morse theory approach. See [BL14], [KAL14] for details.

Example
```
gap> c := SCSurface(3,true);;
gap> f:=SCMorseRandomLex(c);;
gap> Size(f[2]);
8
```

12.1.8 SCMorseRandomRevLex

▷ SCMorseRandomRevLex(*complex*) (function)

Returns: two lists of small integer lists upon success, `fail` otherweise.

Builds a discrete Morse function following Adiprasito, Benedetti and Lutz' reverse lexicographic random discrete Morse theory approach. See [BL14], [KAL14] for details.

Example
```
gap> c := SCSurface(5,false);;
gap> f:=SCMorseRandomRevLex(c);;
gap> Size(f[2]);
7
```

12.1.9 SCMorseSpec

▷ SCMorseSpec(*complex, iter[, morsefunc]*) (function)

Returns: a list upon success, `fail` otherweise.

Computes *iter* versions of a discrete Morse function of *complex* using a randomised method specified by *morsefunc* (default choice is SCMorseRandom (12.1.6), other randomised methods available are SCMorseRandomLex (12.1.7) SCMorseRandomRevLex (12.1.8), and SCMorseUST (12.1.10)). The result is referred to by the Morse spectrum of *complex* and is returned in form of a list containing all Morse vectors sorted by number of critical points together with the actual vector of critical points and how often they ocurred (see [BL14] for details).

Example
```
gap> c:=SCSeriesTorus(2);;
gap> f:=SCMorseSpec(c,30);
[ [ 4, [ 1, 2, 1 ], 30 ] ]
```

Example
```
gap> c:=SCSeriesHomologySphere(2,3,5);;
gap> f:=SCMorseSpec(c,30,SCMorseRandom);
[ [ 6, [ 1, 2, 2, 1 ], 27 ], [ 8, [ 1, 3, 3, 1 ], 3 ] ]
gap> f:=SCMorseSpec(c,30,SCMorseRandomLex);
[ [ 6, [ 1, 2, 2, 1 ], 29 ], [ 8, [ 1, 3, 3, 1 ], 1 ] ]
gap> f:=SCMorseSpec(c,30,SCMorseRandomRevLex);
[ [ 6, [ 1, 2, 2, 1 ], 7 ], [ 8, [ 1, 3, 3, 1 ], 9 ],
  [ 8, [ 2, 3, 2, 1 ], 1 ], [ 10, [ 1, 4, 4, 1 ], 11 ],
  [ 12, [ 1, 5, 5, 1 ], 1 ], [ 14, [ 1, 6, 6, 1 ], 1 ] ]
gap> f:=SCMorseSpec(c,30,SCMorseUST);
[ [ 6, [ 1, 2, 2, 1 ], 18 ], [ 8, [ 1, 3, 3, 1 ], 8 ],
  [ 10, [ 1, 4, 4, 1 ], 3 ], [ 12, [ 1, 5, 5, 1 ], 1 ] ]
```

12.1.10 SCMorseUST

▷ SCMorseUST(*complex*) (function)

Returns: a random Morse function of a simplicial complex and a list of critical faces.

Builds a random Morse function by removing a uniformly sampled spanning tree from the dual 1-skeleton followed by a collapsing approach. *complex* needs to be a closed weak pseudomanifold for this to work. For details of the algorithm, see [PS14].

Example

```
gap> c:=SCBdSimplex(3);;
gap> f:=SCMorseUST(c);;
gap> Size(f[2]);
2
```

12.1.11 SCSpanningTreeRandom

▷ `SCSpanningTreeRandom(HD, top)` (function)

Returns: a list of edges upon success, `fail` otherweise.

Computes a uniformly sampled spanning tree of the complex belonging to the Hasse diagram *HD* using Wilson's algorithm (see [Wil96]). If `top = true` the output is a spanning tree of the dual graph of the underlying complex. If `top = false` the output is a spanning tree of the primal graph (i.e., the 1-skeleton.

Example

```
gap> c:=SCSurface(1,false);;
gap> HD:=SCHasseDiagram(c);;
gap> stTop:=SCSpanningTreeRandom(HD,true);
[ 3, 10, 13, 8, 1, 15, 12, 6, 14 ]
gap> stBot:=SCSpanningTreeRandom(HD,false);
[ 10, 6, 9, 5, 13 ]
```

12.1.12 SCHomology

▷ `SCHomology(complex)` (method)

Returns: a list of pairs of the form `[ integer, list ]` upon success

Computes the homology groups of a given simplicial complex *complex* using `SCMorseRandom` (12.1.6) to obtain a Morse function and `SmithNormalFormIntegerMat`. Use `SCHomologyEx` (12.1.13) to use alternative methods to compute discrete Morse functions (such as `SCMorseEngstroem` (12.1.5), or `SCMorseUST` (12.1.10)) or the Smith normal form.

The output is a list of homology groups of the form $[H_0,....,H_d]$, where d is the dimension of *complex*. The format of the homology groups H_i is given in terms of their maximal cyclic subgroups, i.e. a homology group $H_i \cong \mathbb{Z}^f + \mathbb{Z}/t_1\mathbb{Z} \times \dots \times \mathbb{Z}/t_n\mathbb{Z}$ is returned in form of a list $[f,[t_1,...,t_n]]$, where f is the (integer) free part of H_i and t_i denotes the torsion parts of H_i ordered in weakly increasing size.

Example

```
gap> c:=SCSeriesTorus(2);;
gap> f:=SCHomology(c);
[ [ 0, [  ] ], [ 2, [  ] ], [ 1, [  ] ] ]
```

12.1.13 SCHomologyEx

▷ `SCHomologyEx(c, morsechoice, smithchoice)` (method)

Returns: a list of pairs of the form `[ integer, list ]` upon success, fail otherwise.

Computes the homology groups of a given simplicial complex *c* using the function *morsechoice* for discrete Morse function computations and *smithchoice* for Smith normal form computations.

The output is a list of homology groups of the form $[H_0,....,H_d]$, where d is the dimension of *complex*. The format of the homology groups H_i is given in terms of their maximal cyclic subgroups, i.e. a homology group $H_i \cong \mathbb{Z}^f + \mathbb{Z}/t_1\mathbb{Z} \times \cdots \times \mathbb{Z}/t_n\mathbb{Z}$ is returned in form of a list $[f,[t_1,...,t_n]]$, where f is the (integer) free part of H_i and t_i denotes the torsion parts of H_i ordered in weakly increasing size.

Example
```
gap> c:=SCSeriesTorus(2);;
gap> f:=SCHomology(c);
[ [ 0, [  ] ], [ 2, [  ] ], [ 1, [  ] ] ]
```

Example
```
gap> c := SCSeriesHomologySphere(2,3,5);;
gap> SCHomologyEx(c,SCMorseRandom,SmithNormalFormIntegerMat); time;
[ [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
188
gap> c := SCSeriesHomologySphere(2,3,5);;
gap> SCHomologyEx(c,SCMorseRandomLex,SmithNormalFormIntegerMat); time;
[ [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
188
gap> c := SCSeriesHomologySphere(2,3,5);;
gap> SCHomologyEx(c,SCMorseRandomRevLex,SmithNormalFormIntegerMat); time;
[ [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
204
gap> c := SCSeriesHomologySphere(2,3,5);;
gap> SCHomologyEx(c,SCMorseEngstroem,SmithNormalFormIntegerMat); time;
[ [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
376
gap> c := SCSeriesHomologySphere(2,3,5);;
gap> SCHomologyEx(c,SCMorseUST,SmithNormalFormIntegerMat); time;
[ [ 0, [  ] ], [ 0, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
996
```

12.1.14 SCIsSimplyConnected

▷ `SCIsSimplyConnected(c)` (function)

Returns: a boolean value upon success, `fail` otherweise.

Computes if the `SCSimplicialComplex` object *c* is simply connected. The algorithm is a heuristic method and is described in [PS14]. Internally calls `SCIsSimplyConnectedEx` (12.1.15).

Example
```
gap> rp2:=SCSurface(1,false);;
gap> SCIsSimplyConnected(rp2);
false
gap> c:=SCBdCyclicPolytope(8,18);;
gap> SCIsSimplyConnected(c);
true
```

12.1.15 SCIsSimplyConnectedEx

▷ `SCIsSimplyConnectedEx(c[, top, tries])` (function)

Returns: a boolean value upon success, `fail` otherweise.

Computes if the `SCSimplicialComplex` object *c* is simply connected. The optional boolean argument *top* determines whether a spanning graph in the dual or the primal graph of *c* will be used for a collapsing sequence. The optional positive integer argument *tries* determines the number of times the algorithm will try to find a collapsing sequence. The algorithm is a heuristic method and is described in [PS14].

Example
```
gap> rp2:=SCSurface(1,false);;
gap> SCIsSimplyConnectedEx(rp2);
false
gap> c:=SCBdCyclicPolytope(8,18);;
gap> SCIsSimplyConnectedEx(c);
true
```

12.1.16 SCIsSphere

▷ `SCIsSphere(c)` (function)

Returns: a boolean value upon success, `fail` otherweise.

Determines whether the `SCSimplicialComplex` object *c* is a topological sphere. In dimension $\neq 4$ the algorithm determines whether *c* is PL-homeomorphic to the standard sphere. In dimension 4 the PL type is not specified. The algorithm uses a result due to [KS77] stating that, in dimension $\neq 4$, any simply connected homology sphere with PL structure is a standard PL sphere. The function calls `SCIsSimplyConnected` (12.1.14) which uses a heuristic method described in [PS14].

Example
```
gap> c:=SCBdCyclicPolytope(4,20);;
gap> SCIsSphere(c);
true
gap> c:=SCSurface(1,true);;
gap> SCIsSphere(c);
false
```

12.1.17 SCIsManifold

▷ `SCIsManifold(c)` (function)

Returns: a boolean value upon success, `fail` otherweise.

The algorithm is a heuristic method and is described in [PS14] in more detail. Internally calls `SCIsManifoldEx` (12.1.18).

Example
```
gap> c:=SCBdCyclicPolytope(4,20);;
gap> SCIsManifold(c);
true
```

12.1.18 SCIsManifoldEx

▷ `SCIsManifoldEx(c[, aut, quasi])` (function)

Returns: a boolean value upon success, `fail` otherweise.

If the boolean argument *aut* is `true` the automorphism group is computed and only one link per orbit is checked to be a sphere. If *aut* is not provided symmetry information is only used if the automorphism group is already known. If the boolean argument *quasi* is `false` the algorithm returns whether or not *c* is a combinatorial manifold. If *quasi* is `true` the 4-dimensional links are not verified to be standard PL 4-spheres and *c* is a combinatorial manifold modulo the smooth Poincare conjecture. By default *quasi* is set to `false`. The algorithm is a heuristic method and is described in [PS14] in more detail.

See `SCBistellarIsManifold` (9.2.6) for an alternative method for manifold verification.

Example

```
gap> c:=SCBdCyclicPolytope(4,20);;
gap> SCIsManifold(c);
true
```

Chapter 13

Library and I/O

13.1 Simplicial complex library

simpcomp contains a library of simplicial complexes on few vertices, most of them (combinatorial) triangulations of manifolds and pseudomanifolds. The user can load these known triangulations from the library in order to study their properties or to construct new triangulations out of the known ones. For example, a user could determine the topological type of a given triangulation – which can be quite tedious if done by hand – by establishing a PL equivalence to a complex in the library.

Among other known triangulations, the library contains all of the vertex transitive triangulations of combinatorial manifolds with up to 15 vertices (for $d \in \{2,3,9,10,11,12\}$) and up to 13 vertices (for $d \in \{4,5,6,7,8\}$) and all of the vertex transitive combinatorial pseudomanifolds with up to 15 vertices (for $d = 3$) and up to 13 vertices (for $d \in \{4,5,6,7\}$) classified by Frank Lutz that can be found on his "Manifold Page" `http://www.math.tu-berlin.de/diskregeom/stellar/`, along with some triangulations of sphere bundles and some bounded triangulated PL-manifolds.

See `SCLib` (13.1.2) for a naming convention used for the global library of simpcomp. Note: Another way of storing and loading complexes is provided by the functions `SCExportIsoSig` (6.2.2), `SCExportToString` (6.2.1) and `SCFromIsoSig` (6.2.3), see Section 6.2 for details.

13.1.1 SCIsLibRepository

▷ `SCIsLibRepository(object)` (filter)

Returns: `true` or `false` upon success, `fail` otherwise.

Filter for the category of a library repository `SCIsLibRepository` used by the simpcomp library. The category `SCLibRepository` is derived from the category `SCPropertyObject`.

Example
```
gap> SCIsLibRepository(SCLib); #the global library is stored in SCLib
true
```

13.1.2 SCLib

▷ `SCLib` (global variable)

The global variable `SCLib` contains the library object of the global library of simpcomp through which the user can access the library. The path to the global library is `GAPROOT/pkg/simpcomp/complexes`.

The naming convention in the global library is the following: complexes are usually named by their topological type. As usual, ' S^d' denotes a d-sphere, 'T' a torus, 'x' the cartesian product, '~' the twisted product and '#' the connected sum. The Klein Bottle is denoted by 'K' or 'K^2'.

Example
```
gap> SCLib;
[Simplicial complex library. Properties:
CalculateIndexAttributes=true
Number of complexes in library=7649
IndexAttributes=[ "Name", "Dim", "F", "G", "H", "Chi", "Homology", "IsPM",
  "IsManifold" ]
Loaded=true
Path="/home/jonathan/apps/gap4r5/pkg/simpcomp/complexes/"
]
gap> SCLib.Size;
7649
gap> SCLib.SearchByName("S^4~");
[ [ 463, "S^4~S^1 (VT)" ], [ 1473, "S^4~S^1 (VT)" ], [ 1474, "S^4~S^1 (VT)" ],
  [ 2477, "S^4~S^1 (VT)" ], [ 4395, "S^4~S^1 (VT)" ],
  [ 4396, "S^4~S^1 (VT)" ], [ 4397, "S^4~S^1 (VT)" ],
  [ 4398, "S^4~S^1 (VT)" ], [ 4399, "S^4~S^1 (VT)" ],
  [ 4402, "S^4~S^1 (VT)" ], [ 4403, "S^4~S^1 (VT)" ],
  [ 4404, "S^4~S^1 (VT)" ] ]
gap> SCLib.Load(last[1][1]);
[SimplicialComplex

 Properties known: AltshulerSteinberg, ConnectedComponents, Dim,
                   DualGraph, EulerCharacteristic, FVector, FacetsEx,
                   GVector, HVector, HasBoundary, HasInterior,
                   Homology, Interior, IsConnected,
                   IsEulerianManifold, IsManifold, IsOrientable,
                   IsPseudoManifold, IsPure, IsStronglyConnected,
                   MinimalNonFacesEx, Name, Neighborliness,
                   NumFaces[], Orientation, Reference, SkelExs[],
                   Vertices.

 Name="S^4~S^1 (VT)"
 Dim=5
 AltshulerSteinberg=2417917928025780
 EulerCharacteristic=0
 FVector=[ 13, 78, 195, 260, 195, 65 ]
 GVector=[ 6, 21, -35 ]
 HVector=[ 7, 28, -7, 28, 7, 1 ]
 HasBoundary=false
 HasInterior=true
 Homology=[ [ 0, [ ] ], [ 1, [ ] ], [ 0, [ ] ], [ 0, [ ] ], [ 0, [ 2 ] ], [ 0,\
 [ ] ] ]
 IsConnected=true
 IsEulerianManifold=true
 IsOrientable=false
```

```
 IsPseudoManifold=true
 IsPure=true
 IsStronglyConnected=true
 Neighborliness=2

/SimplicialComplex]
```

13.1.3 SCLibAdd

▷ `SCLibAdd(`*`repository, complex[, name]`*`)` (function)

Returns: `true` upon success, `fail` otherwise.

Adds a given simplicial complex *`complex`* to a given repository *`repository`* of type `SCIsLibRepository`. *`complex`* is saved to a file with suffix `.sc` in the repositories base path, where the file name is either formed from the optional argument *`name`* and the current time or taken from the name of the complex, if it is named.

Example

```
gap> info:=InfoLevel(InfoSimpcomp);;
gap> SCInfoLevel(0);;
gap> myRepository:=SCLibInit("/tmp/repository");
[Simplicial complex library. Properties:
CalculateIndexAttributes=true
Number of complexes in library=0
IndexAttributes=[ "Name", "Dim", "F", "G", "H", "Chi", "Homology", "IsPM",
  "IsManifold" ]
Loaded=true
Path="/tmp/repository/"
]
gap> complex1:=SCBdCrossPolytope(4);;
gap> SCLibAdd(myRepository,complex1);
true
gap> complex2:=SCBdCrossPolytope(4);;
gap> myRepository.Add(complex2);; # alternative syntax
gap> SCInfoLevel(info);;
```

13.1.4 SCLibAllComplexes

▷ `SCLibAllComplexes(`*`repository`*`)` (function)

Returns: list of entries of the form `[ integer, string ]` upon success, `fail` otherwise.

Returns a list with entries of the form `[ ID, NAME ]` of all the complexes in the given repository *`repository`* of type `SCIsLibRepository`.

Example

```
gap> all:=SCLibAllComplexes(SCLib);;
gap> all[1];
[ 1, "Moebius Strip" ]
gap> Length(all);
7649
```

13.1.5 SCLibDelete

▷ SCLibDelete(*repository, id*) (function)

Returns: `true` upon success, `fail` otherwise.

Deletes the simplicial complex with the given id *id* from the given repository *repository*. Apart from deleting the complexes' index entry, the associated `.sc` file is also deleted.

Example
```
gap> myRepository:=SCLibInit("/tmp/repository");
[Simplicial complex library. Properties:
CalculateIndexAttributes=true
Number of complexes in library=2
IndexAttributes=[ "Name", "Dim", "F", "G", "H", "Chi", "Homology", "IsPM",
  "IsManifold" ]
Loaded=true
Path="/tmp/repository/"
]
gap> SCLibAdd(myRepository,SCSimplex(2));;
gap> SCLibDelete(myRepository,1);
true
```

13.1.6 SCLibDetermineTopologicalType

▷ SCLibDetermineTopologicalType(*[repository,]complex*) (function)

Returns: simplicial complex of type `SCSimplicialComplex` or a list of integers upon success, `fail` otherwise.

Tries to determine the topological type of a given complex *complex* by first looking for complexes with matching homology in the library repository *repository* (if no repository is passed, the global repository `SCLib` is used) and either returns a simplicial complex object (that is combinatorially isomorphic to the complex given) or a list of library ids of complexes in the library with the same homology as the complex provided.

The ids obtained in this way can then be used to compare the corresponding complexes with *complex* via the function `SCEquivalent` (9.2.2).

If *complex* is a combinatorial manifold of dimension 1 or 2 its topological type is computed, stored to the property `TopologicalType` and *complex* is returned.

If no complexes with matching homology can be found, the empty set is returned.

Example
```
gap> c:=SCFromFacets([[1,2,3],[1,2,6],[1,3,5],[1,4,5],[1,4,6],
                      [2,3,4],[2,4,5],[2,5,6],[3,4,6],[3,5,6]]);;
gap> SCLibDetermineTopologicalType(c);
[SimplicialComplex

 Properties known: BoundaryEx, Dim, FacetsEx, HasBoundary,
                   IsPseudoManifold, IsPure, Name, SkelExs[],
                   Vertices.

 Name="unnamed complex 167"
 Dim=2
 HasBoundary=false
 IsPseudoManifold=true
 IsPure=true
```

```
/SimplicialComplex]
```

13.1.7 SCLibFlush

▷ SCLibFlush(*repository, confirm*) (function)

Returns: true upon success, fail otherwise.

Completely empties a given repository *repository*. The index and all simplicial complexes in this repository are deleted. The second argument, *confirm*, must be the string "yes" in order to confirm the deletion.

Example
```
gap> myRepository:=SCLibInit("/tmp/repository");;
gap> SCLibFlush(myRepository,"yes");
#I  SCLibInit: invalid parameters.
true
```

13.1.8 SCLibInit

▷ SCLibInit(*dir*) (function)

Returns: library repository of type SCLibRepository upon success, fail otherwise.

This function initializes a library repository object for the given directory *dir* (which has to be provided in form of a GAP object of type String or Directory) and returns that library repository object in case of success. The returned object then provides a mean to access the library repository via the SCLib-functions of simpcomp.

The global library repository of simpcomp is loaded automatically at startup and is stored in the variable SCLib. User repositories can be created by calling SCLibInit with a desired destination directory. Note that each repository must reside in a different path since otherwise data may get lost.

The function first tries to load the repository index for the given directory to rebuild it (by calling SCLibUpdate) if loading the index fails. The library index of a library repository is stored in its base path in the XML file complexes.idx, the complexes are stored in files with suffix .sc, also in XML format.

Example
```
gap> myRepository:=SCLibInit("/tmp/repository");
#I  SCLibInit: made directory "/tmp/repository/" for user library.
#I  SCIntFunc.SCLibInit: index not found -- trying to reconstruct it.
#I  SCLibUpdate: rebuilding index for /tmp/repository/.
#I  SCLibUpdate: rebuilding index done.
[Simplicial complex library. Properties:
CalculateIndexAttributes=true
Number of complexes in library=0
IndexAttributes=[ "Name", "Dim", "F", "G", "H", "Chi", "Homology", "IsPM",
  "IsManifold" ]
Loaded=true
Path="/tmp/repository/"
]
```

13.1.9 SCLibIsLoaded

▷ `SCLibIsLoaded(repository)` (function)

Returns: `true` or `false` upon succes, `fail` otherwise.

Returns `true` when a given library repository *repository* is in loaded state. This means that the directory of this repository is accessible and a repository index file for this repository exists in the repositories' path. If this is not the case `false` is returned.

Example
```
gap> SCLibIsLoaded(SCLib);
true
gap> SCLib.IsLoaded;
true
```

13.1.10 SCLibSearchByAttribute

▷ `SCLibSearchByAttribute(repository, expr)` (function)

Returns: A list of items of the form `[ integer, string ]` upon success, `fail` otherwise.

Searches a given repository *repository* for complexes for which the boolean expression *expr*, passed as string, evaluates to `true` and returns a list of complexes with entries of the form `[ID, NAME]` or `fail` upon error. The expression may use all GAP functions and can access all the indexed attributes of the complexes in the given repository for the query. The standard attributes are: Dim (Dimension), F (f-vector), G (g-vector), H (h-vector), Chi (Euler characteristic), Homology, Name, IsPM, IsManifold. See `SCLib` for the set of indexed attributes of the global library of simpcomp.

Example
```
gap> SCLibSearchByAttribute(SCLib,"Dim=4 and F[3]=Binomial(F[1],3)");
[ [ 16, "CP^2 (VT)" ], [ 7648, "K3_16" ] ]
gap> SCLib.SearchByAttribute("Dim=4 and F[3]=Binomial(F[1],3)");
[ [ 16, "CP^2 (VT)" ], [ 7648, "K3_16" ] ]
```

13.1.11 SCLibSearchByName

▷ `SCLibSearchByName(repository, name)` (function)

Returns: A list of items of the form `[ integer, string ]` upon success, `fail` otherwise.

Searches a given repository *repository* for complexes that contain the string *name* as a substring of their name attribute and returns a list of the complexes found with entries of the form `[ID, NAME]`. See `SCLib` (13.1.2) for a naming convention used for the global library of simpcomp.

Example
```
gap> SCLibSearchByName(SCLib,"K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> SCLib.SearchByName("K3"); #alternative syntax
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> SCLib.SearchByName("S^4x"); #search for products with S^4
[ [ 713, "S^4xS^1 (VT)" ], [ 1472, "S^4xS^1 (VT)" ], [ 1475, "S^4xS^1 (VT)" ],
  [ 7479, "S^4xS^2" ], [ 7539, "S^4xS^3" ], [ 7573, "S^4xS^4" ] ]
```

13.1.12 SCLibSize

▷ `SCLibSize(repository)` (function)

Returns: integer upon success, `fail` otherwise.

Returns the number of complexes contained in the given repository *repository*. Fails if the library repository was not previously loaded with `SCLibInit`.

Example
```
gap> SCLibSize(SCLib); #SCLib is the repository of the global library
7649
```

13.1.13 SCLibUpdate

▷ `SCLibUpdate(repository[, recalc])` (function)

Returns: library repository of type `SCLibRepository` upon success, `fail` otherwise.

Recreates the index of a given repository (either via a repository object or a base path of a repository *repository*) by scanning the base path for all `.sc` files containing simplicial complexes of the repository. Returns a repository object with the newly created index on success or `fail` in case of an error. The optional boolean argument *recalc* forces simpcomp to recompute all the indexed properties (such as f-vector, homology, etc.) of the simplicial complexes in the repository if set to `true`.

Example
```
gap> myRepository:=SCLibInit("/tmp/repository");;
gap> SCLibUpdate(myRepository);
#I  SCLibUpdate: rebuilding index for /tmp/repository/.
#I  SCLibUpdate: rebuilding index done.
[Simplicial complex library. Properties:
CalculateIndexAttributes=true
Number of complexes in library=0
IndexAttributes=[ "Name", "Dim", "F", "G", "H", "Chi", "Homology", "IsPM",
  "IsManifold" ]
Loaded=true
Path="/tmp/repository/"
]
```

13.1.14 SCLibStatus

▷ `SCLibStatus(repository)` (function)

Returns: library repository of type `SCLibRepository` upon success, `fail` otherwise.

Lets GAP print the status of a given library repository *repository*. `IndexAttributes` is the list of attributes indexed for this repository. If `CalculateIndexAttributes` is true, the index attributes for a complex added to the library are calculated automatically upon addition of the complex, otherwise this is left to the user and only pre-calculated attributes are indexed.

Example
```
gap> SCLibStatus(SCLib);
[Simplicial complex library. Properties:
CalculateIndexAttributes=true
Number of complexes in library=7649
IndexAttributes=[ "Name", "Dim", "F", "G", "H", "Chi", "Homology", "IsPM",
```

```
  "IsManifold" ]
Loaded=true
Path="/home/jonathan/apps/gap4r5/pkg/simpcomp/complexes/"
]
```

13.2 simpcomp input / output functions

This section contains a description of the input/output-functionality provided by simpcomp. The package provides the functionality to save and load simplicial complexes (and their known properties) to, respectively from files in XML format. Furthermore, it provides the user with functions to export simplicial complexes into polymake format (for this format there also exists rudimentary import functionality), as JavaView geometry or in form of a LaTeX table. For importing more complex polymake data the package polymaking [RÏ3] can be used.

13.2.1 SCLoad

▷ `SCLoad(filename)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Loads a simplicial complex stored in a binary format (using `IO_Pickle`) from a file specified in *filename* (as string). If *filename* does not end in `.scb`, this suffix is appended to the file name.

Example

```
gap> c:=SCBdSimplex(3);;
gap> SCSave(c,"/tmp/bddelta3");
true
gap> d:=SCLoad("/tmp/bddelta3");
[SimplicialComplex

 Properties known: AutomorphismGroup, AutomorphismGroupSize,
                   AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, Dim,
                   EulerCharacteristic, FacetsEx, GeneratorsEx,
                   HasBoundary, Homology, IsConnected,
                   IsStronglyConnected, Name, NumFaces[],
                   TopologicalType, Vertices.

 Name="S^2_4"
 Dim=2
 AutomorphismGroupSize=24
 AutomorphismGroupStructure="S4"
 AutomorphismGroupTransitivity=4
 EulerCharacteristic=2
 HasBoundary=false
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsConnected=true
 IsStronglyConnected=true
 TopologicalType="S^2"

/SimplicialComplex]
gap> c=d;
```

```
true
```

13.2.2 SCLoadXML

▷ SCLoadXML(*filename*) (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Loads a simplicial complex stored in XML format from a file specified in *filename* (as string). If *filename* does not end in `.sc`, this suffix is appended to the file name.

Example
```
gap> c:=SCBdSimplex(3);;
gap> SCSaveXML(c,"/tmp/bddelta3");
true
gap> d:=SCLoadXML("/tmp/bddelta3");
[SimplicialComplex

 Properties known: AutomorphismGroup, AutomorphismGroupSize,
                   AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity, Dim,
                   EulerCharacteristic, FacetsEx, GeneratorsEx,
                   HasBoundary, Homology, IsConnected,
                   IsStronglyConnected, Name, NumFaces[],
                   TopologicalType, Vertices.

 Name="S^2_4"
 Dim=2
 AutomorphismGroupSize=24
 AutomorphismGroupStructure="S4"
 AutomorphismGroupTransitivity=4
 EulerCharacteristic=2
 HasBoundary=false
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsConnected=true
 IsStronglyConnected=true
 TopologicalType="S^2"

/SimplicialComplex]
gap> c=d;
true
```

13.2.3 SCSave

▷ SCSave(*complex, filename*) (function)

Returns: `true` upon success, `fail` otherwise.

Saves a simplicial complex in a binary format (using `IO_Pickle`) to a file specified in *filename* (as string). If *filename* does not end in `.scb`, this suffix is appended to the file name.

Example
```
gap> c:=SCBdSimplex(3);;
gap> SCSave(c,"/tmp/bddelta3");
```

```
true
```

13.2.4 SCSaveXML

▷ `SCSaveXML(complex, filename)` (function)

Returns: `true` upon success, `fail` otherwise.

Saves a simplicial complex *complex* to a file specified by *filename* (as string) in XML format. If *filename* does not end in `.sc`, this suffix is appended to the file name.

Example
```
gap> c:=SCBdSimplex(3);;
gap> SCSaveXML(c,"/tmp/bddelta3");
true
```

13.2.5 SCExportMacaulay2

▷ `SCExportMacaulay2(complex, ring, filename[, alphalabels])` (function)

Returns: `true` upon success, `fail` otherwise.

Exports the facet list of a given simplicial complex *complex* in `Macaulay2` format to a file specified by *filename*. The argument *ring* can either be the ring of integers (specified by `Integers`) or the ring of rationals (sepcified by `Rationals`). The optional boolean argument *alphalabels* labels the complex with characters from $a,\dots,z$ in the exported file if a value of `true` is supplied, while the standard labeling of the vertices is $v_1,\dots,v_n$ where n is the number of vertices of *complex*. If *complex* has more than 26 vertices, the argument *alphalabels* is ignored.

Example
```
gap> c:=SCBdCrossPolytope(4);;
gap> SCExportMacaulay2(c,Integers,"/tmp/bdbeta4.m2");
true
```

13.2.6 SCExportPolymake

▷ `SCExportPolymake(complex, filename)` (function)

Returns: `true` upon success, `fail` otherwise.

Exports the facet list with vertex labels of a given simplicial complex *complex* in `polymake` format to a file specified by *filename*. Currently, only the export in the format of `polymake` version 2.3 is supported.

Example
```
gap> c:=SCBdCrossPolytope(4);;
gap> SCExportPolymake(c,"/tmp/bdbeta4.poly");
true
```

13.2.7 SCImportPolymake

▷ `SCImportPolymake(filename)` (function)

Returns: simplicial complex of type `SCSimplicialComplex` upon success, `fail` otherwise.

Imports the facet list of a `topaz` `polymake` file specified by *filename* (discarding any vertex labels) and creates a simplicial complex object from these facets.

Example
```
gap> c:=SCBdCrossPolytope(4);;
gap> SCExportPolymake(c,"/tmp/bdbeta4.poly");
true
gap> d:=SCImportPolymake("/tmp/bdbeta4.poly");
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="polymake import '/tmp/bdbeta4.poly'"
 Dim=3

/SimplicialComplex]
gap> c=d;
true
```

13.2.8 SCExportLatexTable

▷ `SCExportLatexTable(complex, filename, itemsperline)` (function)

Returns: `true` on success, `fail` otherwise.

Exports the facet list of a given simplicial complex *complex* (or any list given as first argument) in form of a LaTeX table to a file specified by *filename*. The argument *itemsperline* specifies how many columns the exported table should have. The faces are exported in the format $\langle v_1, \ldots, v_k \rangle$.

Example
```
gap> c:=SCBdSimplex(5);;
gap> SCExportLatexTable(c,"/tmp/bd5simplex.tex",5);
true
```

13.2.9 SCExportJavaView

▷ `SCExportJavaView(complex, file, coords)` (function)

Returns: `true` on success, `fail` otherwise.

Exports the 2-skeleton of the given simplicial complex *complex* (or the facets if the complex is of dimension 2 or less) in `JavaView` format (file name suffix `.jvx`) to a file specified by *filename* (as string). The list *coords* must contain a 3-tuple of real coordinates for each vertex of *complex*, either as tuple of length three containing the coordinates (Warning: as GAP only has rudimentary support for floating point values, currently only integer numbers can be used as coordinates when providing *coords* as list of 3-tuples) or as string of the form `"x.x y.y z.z"` with decimal numbers `x.x`, `y.y`, `z.z` for the three coordinates (i.e. `"1.0 0.0 0.0"`).

Example
```
gap> coords:=[[1,0,0],[0,1,0],[0,0,1]];;
gap> SCExportJavaView(SCBdSimplex(2),"/tmp/triangle.jvx",coords);
true
```

13.2.10 SCExportPolymake

▷ `SCExportPolymake(complex, filename)` (function)

Returns: `true` upon success, `fail` otherwise.

Exports the gluings of the tetrahedra of a given combinatorial 3-manifold *complex* in a format compatible with Matveev's 3-manifold software `Recognizer`.

Example
```
gap> c:=SCBdCrossPolytope(4);;
gap> SCExportRecognizer(c,"/tmp/bdbeta4.mv");
true
```

13.2.11 SCExportSnapPy

▷ `SCExportSnapPy(complex, filename)` (function)

Returns: `true` upon success, `fail` otherwise.

Exports the facet list and orientability of a given combinatorial 3-pseudomanifold *complex* in `SnapPy` format to a file specified by *filename*.

Example
```
gap> SCLib.SearchByAttribute("Dim=3 and F=[8,28,56,28]");
[ [ 8, "PM^3 - TransitiveGroup(8,43), No. 1" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCExportSnapPy(c,"/tmp/M38.tri");
true
```

Chapter 14

Interfaces to other software packages

simpcomp contains various interfaces to other software packages (see Chapter 13 for file-related export and import formats). In this chapter, some more sophisticated interfaces to other software packages are described.

Note that this chapter is subject to change and extension as it is planned to expand simpcomp's functionality in this area in the course of the next versions.

14.1 Interface to the GAP-package homalg

As of Version 1.5, simpcomp is equipped with an interface to the GAP-package homalg [BR08] by Mohamed Barakat. This allows to use homalg's powerful capabilities in the field of homological algebra to compute topological properties of simplicial complexes.

For the time being, the only functions provided are ones allowing to compute the homology and cohomology groups of simplicial complexes with arbitrary coefficients. It is planned to extend the functionality in future releases of simpcomp. See below for a list of functions that provide an interface to homalg.

14.1.1 SCHomalgBoundaryMatrices

▷ `SCHomalgBoundaryMatrices(`*`complex, modulus`*`)` (method)

Returns: a list of homalg objects upon success, `fail` otherwise.

This function computes the boundary operator matrices for the simplicial complex *`complex`* with a ring of coefficients as specified by *`modulus`*: a value of `0` yields $\mathbb{Q}$-matrices, a value of `1` yields $\mathbb{Z}$-matrices and a value of q, q a prime or a prime power, computes the $\mathbb{F}_q$-matrices.

Example
```
gap> SCLib.SearchByName("CP^2 (VT)");
[ [ 16, "CP^2 (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCHomalgBoundaryMatrices(c,0);
[ <A 36 x 9 mutable matrix over an internal ring>,
  <A 84 x 36 mutable matrix over an internal ring>,
  <A 90 x 84 mutable matrix over an internal ring>,
  <A 36 x 90 mutable matrix over an internal ring>,
  <An unevaluated 0 x 36 zero matrix over an internal ring> ]
```

14.1.2 SCHomalgCoboundaryMatrices

▷ SCHomalgCoboundaryMatrices(*complex, modulus*) (method)

Returns: a list of homalg objects upon success, fail otherwise.

This function computes the coboundary operator matrices for the simplicial complex *complex* with a ring of coefficients as specified by *modulus*: a value of 0 yields $\mathbb{Q}$-matrices, a value of 1 yields $\mathbb{Z}$-matrices and a value of q, q a prime or a prime power, computes the $\mathbb{F}_q$-matrices.

Example
```
gap> SCLib.SearchByName("CP^2 (VT)");
[ [ 16, "CP^2 (VT)" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCHomalgCoboundaryMatrices(c,0);
[ <A 9 x 36 mutable matrix over an internal ring>,
  <A 36 x 84 mutable matrix over an internal ring>,
  <A 84 x 90 mutable matrix over an internal ring>,
  <A 90 x 36 mutable matrix over an internal ring>,
  <An unevaluated 36 x 0 zero matrix over an internal ring> ]
```

14.1.3 SCHomalgHomology

▷ SCHomalgHomology(*complex, modulus*) (method)

Returns: a list of integers upon success, fail otherwise.

This function computes the ranks of the homology groups of *complex* with a ring of coefficients as specified by *modulus*: a value of 0 computes the $\mathbb{Q}$-homology, a value of 1 computes the $\mathbb{Z}$-homology and a value of q, q a prime or a prime power, computes the $\mathbb{F}_q$-homology ranks.

Note that if you are interested not only in the ranks of the homology groups, but rather their full structure, have a look at the function SCHomalgHomologyBasis (14.1.4).

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCHomalgHomology(c,0);
#I  SCHomalgHomologyOp: Q-homology ranks: [ 1, 0, 22, 0, 1 ]
[ 1, 0, 22, 0, 1 ]
```

14.1.4 SCHomalgHomologyBasis

▷ SCHomalgHomologyBasis(*complex, modulus*) (method)

Returns: a homalg object upon success, fail otherwise.

This function computes the homology groups (including explicit bases of the modules involved) of *complex* with a ring of coefficients as specified by *modulus*: a value of 0 computes the $\mathbb{Q}$-homology, a value of 1 computes the $\mathbb{Z}$-homology and a value of q, q a prime or a prime power, computes the $\mathbb{F}_q$-homology groups.

The k-th homology group hk can be obtained by calling hk:=CertainObject(homology,k);, where homology is the homalg object returned by this function. The generators of hk can then be obtained via GeneratorsOfModule(hk);.

Note that if you are only interested in the ranks of the homology groups, then it is better to use the funtion SCHomalgHomology (14.1.3) which is way faster.

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCHomalgHomologyBasis(c,0);
#I  SCHomalgHomologyBasisOp: constructed Q-homology groups.
<A graded homology object consisting of 5 left modules at degrees [ 0 .. 4 ]>
```

14.1.5 SCHomalgCohomology

▷ SCHomalgCohomology(*complex, modulus*) (method)

Returns: a list of integers upon success, fail otherwise.

This function computes the ranks of the cohomology groups of *complex* with a ring of coefficients as specified by *modulus*: a value of 0 computes the $\mathbb{Q}$-cohomology, a value of 1 computes the $\mathbb{Z}$-cohomology and a value of q, q a prime or a prime power, computes the $\mathbb{F}_q$-cohomology ranks.

Note that if you are interested not only in the ranks of the cohomology groups, but rather their full structure, have a look at the function SCHomalgCohomologyBasis (14.1.6).

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCHomalgCohomology(c,0);
#I  SCHomalgCohomologyOp: Q-cohomology ranks: [ 1, 0, 22, 0, 1 ]
[ 1, 0, 22, 0, 1 ]
```

14.1.6 SCHomalgCohomologyBasis

▷ SCHomalgCohomologyBasis(*complex, modulus*) (method)

Returns: a homalg object upon success, fail otherwise.

This function computes the cohomology groups (including explicit bases of the modules involved) of *complex* with a ring of coefficients as specified by *modulus*: a value of 0 computes the $\mathbb{Q}$-cohomology, a value of 1 computes the $\mathbb{Z}$-cohomology and a value of q, q a prime or a prime power, computes the $\mathbb{F}_q$-homology groups.

The k-th cohomology group ck can be obtained by calling ck:=CertainObject(cohomology,k);, where cohomology is the homalg object returned by this function. The generators of ck can then be obtained via GeneratorsOfModule(ck);.

Note that if you are only interested in the ranks of the cohomology groups, then it is better to use the funtion SCHomalgCohomology (14.1.5) which is way faster.

Example
```
gap> SCLib.SearchByName("K3");
[ [ 7648, "K3_16" ], [ 7649, "K3_17" ] ]
gap> c:=SCLib.Load(last[1][1]);;
gap> SCHomalgCohomologyBasis(c,0);
#I  SCHomalgCohomologyBasisOp: constructed Q-cohomology groups.
<A graded cohomology object consisting of 5 left modules at degrees
```

```
[ 1 .. 5 ]>
```

Chapter 15

Miscellaneous functions

The behaviour of simpcomp can be changed by setting cetain global options. This can be achieved by the functions described in the following.

15.1 simpcomp logging

The verbosity of the output of information to the screen during calls to functions of the package simpcomp can be controlled by setting the info level parameter via the function SCInfoLevel (15.1.1).

15.1.1 SCInfoLevel

▷ SCInfoLevel(*level*) (function)

Returns: true

Sets the logging verbosity of simpcomp. A level of 0 suppresses all output, a level of 1 lets simpcomp output normal running information, whereas levels of 2 and higher display verbose running information. Examples of functions using more verbose logging are bistellar flip-related functions.

Example
```
gap> SCInfoLevel(3);
true
gap> c:=SCBdCrossPolytope(3);;
gap> SCReduceComplex(c);
#I  round 0, move: [ [ 4, 6 ], [ 1, 2 ] ]
F: [ 6, 12, 8 ]
#I  round 1, move: [ [ 6 ], [ 1, 2, 3 ] ]
F: [ 5, 9, 6 ]
#I  round 1
Reduced complex, F: [ 5, 9, 6 ]
#I  round 2, move: [ [ 4 ], [ 1, 2, 5 ] ]
F: [ 4, 6, 4 ]
#I  round 2
Reduced complex, F: [ 4, 6, 4 ]
#I  SCReduceComplexEx: computed locally minimal complex after 3 rounds.
[ true, [SimplicialComplex

     Properties known: Dim, FacetsEx, Name, Vertices.

     Name="unnamed complex 3"
```

```
 Dim=2

/SimplicialComplex], 3 ]
```

15.2 Email notification system

simpcomp comes with an email notification system that can be used for being notified of the progress of lengthy computations (such as reducing a complex via bistellar flips). See below for a description of the mail notification related functions. Note that this might not work on non-Unix systems.

See `SCReduceComplexEx` (9.2.14) for an example computation using the email notification system.

15.2.1 SCMailClearPending

▷ `SCMailClearPending()` (function)

Returns: nothing.

Clears a pending mail message.

Example
```
gap> SCMailClearPending();
```

15.2.2 SCMailIsEnabled

▷ `SCMailIsEnabled()` (function)

Returns: `true` or `false` upon success, `fail` otherwise.

Returns `true` when the mail notification system of simpcomp is enabled, `false` otherwise. Default setting is `false`.

Example
```
gap> SCMailSetAddress("johndoe@somehost"); #enables mail notification
true
gap> SCMailIsEnabled();
true
```

15.2.3 SCMailIsPending

▷ `SCMailIsPending()` (function)

Returns: `true` or `false` upon success, `fail` otherwise.

Returns `true` when an email of the simpcomp email notification system is pending, `false` otherwise.

Example
```
gap> SCMailIsPending();
true
```

15.2.4 SCMailSend

▷ `SCMailSend(message[, starttime][, forcesend])` (function)

Returns: `true` when the message was sent, `false` if it was not send, `fail` upon an error.

Tries to send an email to the address specified by `SCMailSetAddress` (15.2.6) using the Unix program `mail`. The optional parameter *starttime* specifies the starting time (as the integer Unix timestamp) a calculation was started (then the duration of the calculation is included in the email), the optional boolean parameter *forcesend* can be used to force the sending of an email, even if this violates the minimal email sending interval, see `SCMailSetMinInterval` (15.2.8).

Example
```
gap> SCMailSetAddress("johndoe@somehost"); #enables mail notification
true
gap> SCMailIsEnabled();
true
gap> SCMailSend("Hello, this is simpcomp.");
true
```

15.2.5 SCMailSendPending

▷ `SCMailSendPending()` (function)

Returns: `true` upon success, `fail` otherwise.

Tries to send a pending email of the simpcomp email notification system. Returns `true` on success or if there was no mail pending.

Example
```
gap> SCMailSendPending();
true
```

15.2.6 SCMailSetAddress

▷ `SCMailSetAddress(address)` (function)

Returns: `true` upon success, `fail` otherwise.

Sets the email address that should be used to send notification messages and enables the mail notification system by calling `SCMailSetEnabled` (15.2.7)(`true`).

Example
```
gap> SCMailSetAddress("johndoe@somehost");
true
```

15.2.7 SCMailSetEnabled

▷ `SCMailSetEnabled(flag)` (function)

Returns: `true` upon success, `fail` otherwise.

Enables or disables the mail notification system of simpcomp. By default it is disabled. Returns `fail` if no email message was previously set with `SCMailSetAddress` (15.2.6).

Example
```
gap> SCMailSetAddress("johndoe@somehost"); #enables mail notification
true
```

```
gap> SCMailSetEnabled(false);
true
```

15.2.8 SCMailSetMinInterval

▷ SCMailSetMinInterval(*interval*) (function)

Returns: true upon success, fail otherwise.

Sets the minimal time interval in seconds that mail messages can be sent by simpcomp. This prevents a flooding of the specified email address with messages sent by simpcomp. Default is 3600, i.e. one hour.

Example
```
gap> SCMailSetMinInterval(7200);
true
```

15.3 Testing the functionality of simpcomp

simpcomp makes use of the GAP internal testing mechanisms and provides the user with a function to test the functionality of the package.

15.3.1 SCRunTest

▷ SCRunTest() (function)

Returns: true upon success, fail otherwise.

Test whether the package simpcomp is functional by calling ReadTest("GAPROOT/pkg/simpcomp/tst/simpcomp.tst");. The returned value of GAP4stones is a measure of your system performance and differs from system to system.

Example
```
gap> SCRunTest();
Line 1220 :
+ simpcomp package test
Line 1221 :
+ GAP4stones: 18144
true
```

On a modern computer, the function SCRunTest should take about a minute to complete when the packages GRAPE [Soi12] and homology [DHSW11] are available. If these packages are missing, the testing will take slightly longer.

Chapter 16

Property handlers

As explained in Chapter 5, objects of the types `SCSimplicialComplex`, `SCNormalSurface` and `SCLibRepository` provide a set of property handlers for ease of access to simpcomp functions using these objects. Accessing these property handlers is possible via the .-operator.

For example, the f-vector of a simplicial complex `c` that is stored as a `SCSimplicialComplex` object can be accessed via the statement `c.F;` instead of writing the longer `SCFVector(c);`. See below for a list of all properties supported by objects of the types `SCPolyhedralComplex`, `SCSimplicialComplex`, `SCNormalSurface` and `SCLibRepository` (Note that the property handlers of `SCPolyhedralComplex` can be used by both `SCSimplicialComplex` and `SCNormalSurface`).

16.1 Property handlers of `SCPolyhedralComplex`

This section contains a table of all property handlers of a `SCPolyhedralComplex` object.

PROPERTY HANDLER	FUNCTION CALLED
AntiStar	`SCAntiStar` (4.3.1)
Ast	`SCAntiStar` (4.3.1)
Facets	`SCFacets` (6.9.19)
FacetsEx	`SCFacetsEx` (6.9.20)
LabelMax	`SCLabelMax` (4.2.1)
LabelMin	`SCLabelMin` (4.2.2)
Labels	`SCLabels` (4.2.3)
Lk	`SCLink` (4.3.2)
Link	`SCLink` (4.3.2)
Links	`SCLinks` (4.3.3)
Lks	`SCLinks` (4.3.3)
Name	`SCName` (4.2.4)
Reference	`SCReference` (4.2.5)
Relabel	`SCRelabel` (4.2.6)
RelabelStandard	`SCRelabelStandard` (4.2.7)
RelabelTransposition	`SCRelabelTransposition` (4.2.8)
Rename	`SCRename` (4.2.9)
SetReference	`SCSetReference` (4.2.10)

Star	SCStar (4.3.4)
Str	SCStar (4.3.4)
Stars	SCStars (4.3.5)
Strs	SCStars (4.3.5)
UnlabelFace	SCUnlabelFace (4.2.11)
Vertices	SCVertices (4.1.3)
VerticesEx	SCVerticesEx (4.1.4)

16.2 Property handlers of SCSimplicialComplex

This section contains a table of all property handlers of a SCSimplicialComplex object.

PROPERTY HANDLER	FUNCTION CALLED
ASDet	SCAltshulerSteinberg (6.9.1)
AlexanderDual	SCAlexanderDual (6.10.1)
AutomorphismGroup	SCAutomorphismGroup (6.9.2)
AutomorphismGroupInternal	SCAutomorphismGroupInternal (6.9.3)
AutomorphismGroupSize	SCAutomorphismGroupSize (6.9.4)
AutomorphismGroupStructure	SCAutomorphismGroupStructure (6.9.5)
AutomorphismGroupTransitivity	SCAutomorphismGroupTransitivity (6.9.6)
Bd	SCBoundary (6.9.7)
Boundary	SCBoundary (6.9.7)
BoundaryOperatorMatrix	SCBoundaryOperatorMatrix (8.1.1)
Chi	SCEulerCharacteristic (7.3.3)
CoboundaryOperatorMatrix	SCCoboundaryOperatorMatrix (8.2.1)
Cohomology	SCCohomology (8.2.2)
CohomologyBasis	SCCohomologyBasis (8.2.3)
CohomologyBasisAsSimplices	SCCohomologyBasisAsSimplices (8.2.4)
CollapseGreedy	SCCollapseGreedy (12.1.1)
Cone	SCCone (6.10.3)
ConnectedComponents	SCConnectedComponents (7.3.1)
Copy	SCCopy (7.2.1)
CupProduct	SCCupProduct (8.2.5)
DehnSommervilleCheck	SCDehnSommervilleCheck (6.9.8)
DeletedJoin	SCDeletedJoin (6.10.4)
DetermineTopologicalType	SCLibDetermineTopologicalType (13.1.6)
Difference	SCDifference (6.10.5)
DifferenceCycles	SCDifferenceCycles (6.9.10)
Dim	SCDim (7.3.2)
DualGraph	SCDualGraph (6.9.12)
Equivalent	SCEquivalent (9.2.2)
EulerCharacteristic	SCEulerCharacteristic (7.3.3)
ExportJavaView	SCExportJavaView (13.2.9)
ExportLatexTable	SCExportLatexTable (13.2.8)
ExportPolymake	SCExportPolymake (13.2.10)

IsManifold	`SCIsManifold` (12.1.17)
IsMovable	`SCIsMovableComplex` (9.2.7)
Isomorphism	`SCIsomorphism` (6.10.11)
IsomorphismEx	`SCIsomorphismEx` (6.10.12)
IsOrientable	`SCIsOrientable` (7.3.12)
IsPM	`SCIsPseudoManifold` (6.9.45)
IsPure	`SCIsPure` (6.9.46)
IsSC	`SCIsSimplyConnected` (12.1.14)
IsSimplyConnected	`SCIsSimplyConnected` (12.1.14)
IsShellable	`SCIsShellable` (6.9.47)
IsSphere	`SCIsSphere` (12.1.16)
IsStronglyConnected	`SCIsStronglyConnected` (6.9.48)
IsSubcomplex	`SCIsSubcomplex` (6.10.10)
IsTight	`SCIsTight` (11.1.1)
Join	`SCJoin` (6.10.13)
Load	`SCLoad` (13.2.1)
MinimalNonFaces	`SCMinimalNonFaces` (6.9.49)
MinimalNonFacesEx	`SCMinimalNonFacesEx` (6.9.50)
MorseIsPerfect	`SCMorseIsPerfect` (11.1.2)
MorseMultiplicityVector	`SCMorseMultiplicityVector` (11.1.4)
MorseNumberOfCriticalPoints	`SCMorseNumberOfCriticalPoints` (11.1.5)
Move	`SCMove` (9.2.8)
Moves	`SCMoves` (9.2.9)
Neighborliness	`SCNeighborliness` (6.9.51)
Neighbors	`SCNeighbors` (6.10.14)
NeighborsEx	`SCNeighborsEx` (6.10.15)
NumFaces	`SCNumFaces` (6.9.52)
Orientation	`SCOrientation` (6.9.53)
PropertiesDropped	`SCPropertiesDropped` (5.1.4)
Randomize	`SCRandomize` (9.2.11)
RMoves	`SCRMoves` (9.2.10)
Reduce	`SCReduceComplex` (9.2.13)
ReduceAsSubcomplex	`SCReduceAsSubcomplex` (9.2.12)
ReduceEx	`SCReduceComplexEx` (9.2.14)
Save	`SCSave` (13.2.3)
Shelling	`SCShelling` (6.10.16)
ShellingExt	`SCShellingExt` (6.10.17)
Shellings	`SCShellings` (6.10.18)
Skel	`SCSkel` (7.3.13)
SkelEx	`SCSkelEx` (7.3.14)
Slicing	`SCSlicing` (11.1.3), `SCNSSlicing` (7.1.4)
Span	`SCSpan` (6.10.19)
SpanningTree	`SCSpanningTree` (6.9.56)
StronglyConnectedComponents	`SCStronglyConnectedComponents` (6.6.9)
Suspension	`SCSuspension` (6.10.20)
Transitivity	`SCAutomorphismGroupTransitivity` (6.9.6)
Union	`SCUnion` (7.3.16)
VertexIdentification	`SCVertexIdentification` (6.10.22)
Wedge	`SCWedge` (6.10.23)

16.3 Property handlers of SCNormalSurface

This section contains a table of all property handlers of a SCNormalSurface object.

PROPERTY HANDLER	FUNCTION CALLED
Betti	SCFpBettiNumbers (7.3.7)
ConnectedComponents	SCConnectedComponents (7.3.1)
FpBettiNumbers	SCFpBettiNumbers (7.3.7)
Chi	SCEulerCharacteristic (7.3.3)
EulerCharacteristic	SCEulerCharacteristic (7.3.3)
Connected	SCIsConnected (7.3.10)
IsConnected	SCIsConnected (7.3.10)
Copy	SCCopy (7.2.1)
D	SCDim (7.3.2)
Dim	SCDim (7.3.2)
F	SCFVector (7.3.4)
FVector	SCFVector (7.3.4)
FaceLattice	SCFaceLattice (7.3.5)
Faces	SCSkel (7.3.13)
Genus	SCGenus (7.3.8)
Homology	SCHomology (12.1.12)
IsEmpty	SCIsEmpty (7.3.11)
Name	SCName (4.2.4)
Triangulation	SCNSTriangulation (7.2.2)
TopologicalType	SCTopologicalType (7.3.15)

16.4 Property handlers of SCLibRepository

This section contains a table of all property handlers of a SCLibRepository object.

PROPERTY HANDLER	FUNCTION CALLED
Update	SCLibUpdate (13.1.13)
IsLoaded	SCLibIsLoaded (13.1.9)
Size	SCLibSize (13.1.12)
Status	SCLibStatus (13.1.14)
Flush	SCLibFlush (13.1.7)
Add	SCLibAdd (13.1.3)
Delete	SCLibDelete (13.1.5)
All	SCLibAllComplexes (13.1.4)
SearchByName	SCLibSearchByName (13.1.11)
SearchByAttribute	SCLibSearchByAttribute (13.1.10)
DetermineTopologicalType	SCLibDetermineTopologicalType (13.1.6)

Chapter 17

A demo session with simpcomp

This chapter contains the transcript of a demo session with simpcomp that is intended to give an insight into what things can be done with this package.

Of course this only scratches the surface of the functions provided by simpcomp. See Chapters 4 through 15 for further functions provided by simpcomp.

17.1 Creating a SCSimplicialComplex object

Simplicial complex objects can either be created from a facet list (complex c1 below), orbit representatives together with a permutation group (complex c2) or difference cycles (complex c3, see Section 6.1), from a function generating triangulations of standard complexes (complex c4, see Section 6.3) or from a function constructing infinite series for combinatorial (pseudo)manifolds (complexes c5, c6, c7, see Section 6.4 and the function prefix SCSeries...). There are also functions creating new simplicial complexes from old, see Section 6.6, which will be described in the next sections.

Example
```
gap> #first run functionality test on simpcomp
gap> SCRunTest();
+ test simpcomp package, version 2.1.2
+ GAP4stones: 69988
true
gap> #all ok
gap> c1:=SCFromFacets([[1,2],[2,3],[3,1]]);
[SimplicialComplex

 Properties known: Dim, Facets, Name, VertexLabels.

 Name="unnamed complex 1"
 Dim=1

/SimplicialComplex]
gap> G:=Group([(2,12,11,6,8,3)(4,7,10)(5,9),(1,11,6,4,5,3,10,8,9,7,2,12)]);
Group([ (2,12,11,6,8,3)(4,7,10)(5,9), (1,11,6,4,5,3,10,8,9,7,2,12) ])
gap> StructureDescription(G);
"S4 x S3"
gap> Size(G);
144
gap> c2:=SCFromGenerators(G,[[1,2,3]]);;
```

```
gap> c2.IsManifold;
true
gap> SCLibDetermineTopologicalType(c2);
[SimplicialComplex

 Properties known: AutomorphismGroup, AutomorphismGroupSize,
                   AutomorphismGroupStructure, AutomorphismGroupTransitivity,\

                   Boundary, Dim, Faces, Facets, Generators, HasBoundary,
                   IsManifold, IsPM, Name, TopologicalType, VertexLabels,
                   Vertices.

 Name="complex from generators under group S4 x S3"
 Dim=2
 AutomorphismGroupSize=144
 AutomorphismGroupStructure="S4 x S3"
 AutomorphismGroupTransitivity=1
 HasBoundary=false
 IsPM=true
 TopologicalType="T^2"

/SimplicialComplex]
gap> c3:=SCFromDifferenceCycles([[1,1,6],[3,3,2]]);
[SimplicialComplex

 Properties known: Dim, Facets, Name, VertexLabels.

 Name="complex from diffcycles [ [ 1, 1, 6 ], [ 3, 3, 2 ] ]"
 Dim=2

/SimplicialComplex]
gap> c4:=SCBdSimplex(2);
[SimplicialComplex

 Properties known: AutomorphismGroup, AutomorphismGroupOrder,
                   AutomorphismGroupStructure, AutomorphismGroupTransitivity,
                   Chi, Dim, F, Facets, Generators, HasBounday, Homology,
                   IsConnected, IsStronglyConnected, Name, TopologicalType,
                   VertexLabels.

 Name="S^1_3"
 Dim=1
 AutomorphismGroupStructure="S3"
 AutomorphismGroupTransitivity=3
 Chi=0
 F=[ 3, 3 ]
 Homology=[ [ 0, [ ] ], [ 1, [ ] ] ]
 IsConnected=true
 IsStronglyConnected=true
 TopologicalType="S^1"

/SimplicialComplex]
gap> c5:=SCSeriesCSTSurface(2,16);;
```

```
gap> SCLibDetermineTopologicalType(c5);
[SimplicialComplex

 Properties known: Boundary, Dim, Faces, Facets, HasBoundary, IsPM, Name,
                   TopologicalType, VertexLabels.

 Name="cst surface S_{(2,16)} = { (2:2:12),(6:6:4) }"
 Dim=2
 HasBoundary=false
 IsPM=true
 TopologicalType="T^2 U T^2"

/SimplicialComplex]
gap> c6:=SCSeriesD2n(22);;
gap> c6.Homology;
[ [ 0, [  ] ], [ 1, [  ] ], [ 0, [ 2 ] ], [ 0, [  ] ] ]
gap> c6.F;
[ 44, 264, 440, 220 ]
gap> SCSeriesAGL(17);
[ AGL(1,17), [ [ 1, 2, 4, 8, 16 ] ] ]
gap> c7:=SCFromGenerators(last[1],last[2]);;
gap> c7.AutomorphismGroupTransitivity;
2
```

17.2 Working with a SCSimplicialComplex object

As described in Section 3.1 there are two several ways of accessing an object of type SCSimplicialComplex. An example for the two equivalent ways is given below. The preference will be given to the object oriented notation in this demo session. The code listed below

Example
```
gap> c:=SCBdSimplex(3);; # create a simplicial complex object
gap> SCFVector(c);
[ 4, 6, 4 ]
gap> SCSkel(c,0);
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ] ]
```

is equivalent to

Example
```
gap> c:=SCBdSimplex(3);; # create a simplicial complex object
gap> c.F;
[ 4, 6, 4 ]
gap> c.Skel(0);
[ [ 1 ], [ 2 ], [ 3 ], [ 4 ] ]
```

17.3 Calculating properties of a SCSimplicialComplex object

simpcomp provides a variety of functions for calculating properties of simplicial complexes, see Section 6.9. All these properties are only calculated once and stored in the SCSimplicialComplex object.

Example

```
gap> c1.F;
[ 3, 3 ]
gap> c1.FaceLattice;
[ [ [ 1 ], [ 2 ], [ 3 ] ], [ [ 1, 2 ], [ 1, 3 ], [ 2, 3 ] ] ]
gap> c1.AutomorphismGroup;
S3
gap> c1.Generators;
[ [ [ 1, 2 ], 3 ] ]
gap> c3.Facets;
[ [ 1, 2, 3 ], [ 1, 2, 8 ], [ 1, 3, 6 ], [ 1, 4, 6 ], [ 1, 4, 7 ],
  [ 1, 7, 8 ], [ 2, 3, 4 ], [ 2, 4, 7 ], [ 2, 5, 7 ], [ 2, 5, 8 ],
  [ 3, 4, 5 ], [ 3, 5, 8 ], [ 3, 6, 8 ], [ 4, 5, 6 ], [ 5, 6, 7 ],
  [ 6, 7, 8 ] ]
gap> c3.F;
[ 8, 24, 16 ]
gap> c3.G;
[ 4 ]
gap> c3.H;
[ 5, 11, -1 ]
gap> c3.ASDet;
186624
gap> c3.Chi;
0
gap> c3.Generators;
[ [ [ 1, 2, 3 ], 16 ] ]
gap> c3.HasBoundary;
false
gap> c3.IsConnected;
true
gap> c3.IsCentrallySymmetric;
true
gap> c3.Vertices;
[ 1, 2, 3, 4, 5, 6, 7, 8 ]
gap> c3.ConnectedComponents;
[ [SimplicialComplex

     Properties known: Dim, Facets, Name, VertexLabels.

     Name="Connected component #1 of complex from diffcycles [ [ 1, 1, 6 ], [ \
3, 3, 2 ] ]"
     Dim=2

    /SimplicialComplex] ]
gap> c3.UnknownProperty;
#I  SCPropertyObject: unhandled property 'UnknownProperty'. Handled properties\
 are [ "Equivalent", "IsKStackedSphere", "IsManifold", "IsMovable", "Move",
  "Moves", "RMoves", "ReduceAsSubcomplex", "Reduce", "ReduceEx", "Copy",
  "Recalc", "ASDet", "AutomorphismGroup", "AutomorphismGroupInternal",
  "Boundary", "ConnectedComponents", "Dim", "DualGraph", "Chi", "F",
  "FaceLattice", "FaceLatticeEx", "Faces", "FacesEx", "Facets", "FacetsEx",
  "FpBetti", "FundamentalGroup", "G", "Generators", "GeneratorsEx", "H",
  "HasBoundary", "HasInterior", "Homology", "Incidences", "IncidencesEx",
```

```
  "Interior", "IsCentrallySymmetric", "IsConnected", "IsEmpty",
  "IsEulerianManifold", "IsHomologySphere", "IsInKd", "IsKNeighborly",
  "IsOrientable", "IsPM", "IsPure", "IsShellable", "IsStronglyConnected",
  "MinimalNonFaces", "MinimalNonFacesEx", "Name", "Neighborliness",
  "Orientation", "Skel", "SkelEx", "SpanningTree",
  "StronglyConnectedComponents", "Vertices", "VerticesEx",
  "BoundaryOperatorMatrix", "HomologyBasis", "HomologyBasisAsSimplices",
  "HomologyInternal", "CoboundaryOperatorMatrix", "Cohomology",
  "CohomologyBasis", "CohomologyBasisAsSimplices", "CupProduct",
  "IntersectionForm", "IntersectionFormParity",
  "IntersectionFormDimensionality", "Load", "Save", "ExportPolymake",
  "ExportLatexTable", "ExportJavaView", "LabelMax", "LabelMin", "Labels",
  "Relabel", "RelabelStandard", "RelabelTransposition", "Rename",
  "SortComplex", "UnlabelFace", "AlexanderDual", "CollapseGreedy", "Cone",
  "DeletedJoin", "Difference", "HandleAddition", "Intersection",
  "IsIsomorphic", "IsSubcomplex", "Isomorphism", "IsomorphismEx", "Join",
  "Link", "Links", "Neighbors", "NeighborsEx", "Shelling", "ShellingExt",
  "Shellings", "Span", "Star", "Stars", "Suspension", "Union",
  "VertexIdentification", "Wedge", "DetermineTopologicalType", "Dim",
  "Facets", "VertexLabels", "Name", "Vertices", "IsConnected",
  "ConnectedComponents" ].

fail
```

17.4 Creating new complexes from a `SCSimplicialComplex` object

As already mentioned, there is the possibility to generate new objects of type `SCSimplicialComplex` from existing ones using standard constructions. The functions used in this section are described in more detail in Section 6.6.

Example

```
gap> d:=c3+c3;
[SimplicialComplex

 Properties known: Dim, Facets, Name, VertexLabels, Vertices.

 Name="complex from diffcycles [ [ 1, 1, 6 ], [ 3, 3, 2 ] ]#+-complex from dif\
fcycles [ [ 1, 1, 6 ], [ 3, 3, 2 ] ]"
 Dim=2

/SimplicialComplex]
gap> SCRename(d,"T^2#T^2");
true
gap> SCLink(d,1);
[SimplicialComplex

 Properties known: Dim, Facets, Name, VertexLabels.

 Name="lk(1) in T^2#T^2"
 Dim=1

/SimplicialComplex]
```

```
gap> SCStar(d,[1,2]);
[SimplicialComplex

 Properties known: Dim, Facets, Name, VertexLabels.

 Name="star([ 1, 2 ]) in T^2#T^2"
 Dim=2

/SimplicialComplex]
gap> SCRename(c3,"T^2");
true
gap> SCConnectedProduct(c3,4);
[SimplicialComplex

 Properties known: Dim, Facets, Name, VertexLabels, Vertices.

 Name="T^2#+-T^2#+-T^2#+-T^2"
 Dim=2

/SimplicialComplex]
gap> SCCartesianProduct(c4,c4);
[SimplicialComplex

 Properties known: Dim, Facets, Name, TopologicalType, VertexLabels.

 Name="S^1_3xS^1_3"
 Dim=2
 TopologicalType="S^1xS^1"

/SimplicialComplex]
gap> SCCartesianPower(c4,3);
[SimplicialComplex

 Properties known: Dim, Facets, Name, TopologicalType, VertexLabels.

 Name="(S^1_3)^3"
 Dim=3
 TopologicalType="(S^1)^3"

/SimplicialComplex]
```

17.5 Homology related calculations

simpcomp relies on the GAP package homology [DHSW11] for its homology computations but provides further (co-)homology related functions, see Chapter 8.

Example

```
gap> s2s2:=SCCartesianProduct(SCBdSimplex(3),SCBdSimplex(3));
[SimplicialComplex

 Properties known: Dim, Facets, Name, TopologicalType, VertexLabels.

```

```
 Name="S^2_4xS^2_4"
 Dim=4
 TopologicalType="S^2xS^2"

/SimplicialComplex]
gap> SCHomology(s2s2);
[ [ 0, [  ] ], [ 0, [  ] ], [ 2, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> SCHomologyInternal(s2s2);
[ [ 0, [  ] ], [ 0, [  ] ], [ 2, [  ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> SCHomologyBasis(s2s2,2);
[ [ 1, [ [ 1, 70 ], [ -1, 12 ], [ 1, 2 ], [ -1, 1 ] ] ],
  [ 1, [ [ 1, 143 ], [ -1, 51 ], [ 1, 29 ], [ -1, 25 ] ] ] ]
gap> SCHomologyBasisAsSimplices(s2s2,2);
[ [ 1,
      [ [ 1, [ 2, 3, 4 ] ], [ -1, [ 1, 3, 4 ] ], [ 1, [ 1, 2, 4 ] ], [ -1, [ 1
                 , 2, 3 ] ] ] ],
  [ 1, [ [ 1, [ 5, 9, 13 ] ], [ -1, [ 1, 9, 13 ] ], [ 1, [ 1, 5, 13 ] ],
          [ -1, [ 1, 5, 9 ] ] ] ] ]
gap> SCCohomologyBasis(s2s2,2);
[ [ 1,
      [ [ 1, 122 ], [ 1, 115 ], [ 1, 112 ], [ 1, 111 ], [ 1, 93 ], [ 1, 90 ],
          [ 1, 89 ], [ 1, 84 ], [ 1, 83 ], [ 1, 82 ], [ 1, 46 ], [ 1, 43 ],
          [ 1, 42 ], [ 1, 37 ], [ 1, 36 ], [ 1, 35 ], [ 1, 28 ], [ 1, 27 ],
          [ 1, 26 ], [ 1, 25 ] ] ],
  [ 1, [ [ 1, 213 ], [ 1, 201 ], [ 1, 192 ], [ 1, 189 ], [ 1, 159 ],
          [ 1, 150 ], [ 1, 147 ], [ 1, 131 ], [ 1, 128 ], [ 1, 125 ],
          [ 1, 67 ], [ 1, 58 ], [ 1, 55 ], [ 1, 39 ], [ 1, 36 ], [ 1, 33 ],
          [ 1, 10 ], [ 1, 7 ], [ 1, 4 ], [ 1, 1 ] ] ] ]
gap> SCCohomologyBasisAsSimplices(s2s2,2);
[ [ 1, [ [ 1, [ 4, 8, 12 ] ], [ 1, [ 3, 8, 12 ] ], [ 1, [ 3, 7, 12 ] ],
          [ 1, [ 3, 7, 11 ] ], [ 1, [ 2, 8, 12 ] ], [ 1, [ 2, 7, 12 ] ],
          [ 1, [ 2, 7, 11 ] ], [ 1, [ 2, 6, 12 ] ], [ 1, [ 2, 6, 11 ] ],
          [ 1, [ 2, 6, 10 ] ], [ 1, [ 1, 8, 12 ] ], [ 1, [ 1, 7, 12 ] ],
          [ 1, [ 1, 7, 11 ] ], [ 1, [ 1, 6, 12 ] ], [ 1, [ 1, 6, 11 ] ],
          [ 1, [ 1, 6, 10 ] ], [ 1, [ 1, 5, 12 ] ], [ 1, [ 1, 5, 11 ] ],
          [ 1, [ 1, 5, 10 ] ], [ 1, [ 1, 5, 9 ] ] ] ],
  [ 1, [ [ 1, [ 13, 14, 15 ] ], [ 1, [ 9, 14, 15 ] ], [ 1, [ 9, 10, 15 ] ],
          [ 1, [ 9, 10, 11 ] ], [ 1, [ 5, 14, 15 ] ], [ 1, [ 5, 10, 15 ] ],
          [ 1, [ 5, 10, 11 ] ], [ 1, [ 5, 6, 15 ] ], [ 1, [ 5, 6, 11 ] ],
          [ 1, [ 5, 6, 7 ] ], [ 1, [ 1, 14, 15 ] ], [ 1, [ 1, 10, 15 ] ],
          [ 1, [ 1, 10, 11 ] ], [ 1, [ 1, 6, 15 ] ], [ 1, [ 1, 6, 11 ] ],
          [ 1, [ 1, 6, 7 ] ], [ 1, [ 1, 2, 15 ] ], [ 1, [ 1, 2, 11 ] ],
          [ 1, [ 1, 2, 7 ] ], [ 1, [ 1, 2, 3 ] ] ] ] ]
gap> PrintArray(SCIntersectionForm(s2s2));
[ [  0,  1 ],
  [  1,  0 ] ]
gap> c:=s2s2+s2s2;
[SimplicialComplex

 Properties known: Dim, Facets, Name, VertexLabels, Vertices.

 Name="S^2_4xS^2_4#+-S^2_4xS^2_4"
 Dim=4
```

```
/SimplicialComplex]
gap> PrintArray(SCIntersectionForm(c));
[ [   0,  -1,   0,   0 ],
  [  -1,   0,   0,   0 ],
  [   0,   0,   0,  -1 ],
  [   0,   0,  -1,   0 ] ]
```

17.6 Bistellar flips

For a more detailed description of functions related to bistellar flips as well as a very short introduction into the topic, see Chapter 9.

Example

```
gap> beta4:=SCBdCrossPolytope(4);;
gap> s3:=SCBdSimplex(4);;
gap> SCEquivalent(beta4,s3);
#I  round 0, move: [ [ 2, 6, 7 ], [ 3, 4 ] ]
[ 8, 25, 34, 17 ]
#I  round 1, move: [ [ 2, 7 ], [ 3, 4, 5 ] ]
[ 8, 24, 32, 16 ]
#I  round 2, move: [ [ 2, 5 ], [ 3, 4, 8 ] ]
[ 8, 23, 30, 15 ]
#I  round 3, move: [ [ 2 ], [ 3, 4, 6, 8 ] ]
[ 7, 19, 24, 12 ]
#I  round 4, move: [ [ 6, 8 ], [ 1, 3, 4 ] ]
[ 7, 18, 22, 11 ]
#I  round 5, move: [ [ 8 ], [ 1, 3, 4, 5 ] ]
[ 6, 14, 16, 8 ]
#I  round 6, move: [ [ 5 ], [ 1, 3, 4, 7 ] ]
[ 5, 10, 10, 5 ]
#I  SCReduceComplexEx: complexes are bistellarly equivalent.
true
gap> SCBistellarOptions.WriteLevel;
0
gap> SCBistellarOptions.WriteLevel:=1;
1
gap> SCEquivalent(beta4,s3);
#I  SCLibInit: made directory "~/PATH" for user library.
#I  SCIntFunc.SCLibInit: index not found -- trying to reconstruct it.
#I  SCLibUpdate: rebuilding index for ~/PATH.
#I  SCLibUpdate: rebuilding index done.

#I  round 0, move: [ [ 2, 4, 6 ], [ 7, 8 ] ]
[ 8, 25, 34, 17 ]
#I  round 1, move: [ [ 2, 4 ], [ 5, 7, 8 ] ]
[ 8, 24, 32, 16 ]
#I  round 2, move: [ [ 4, 5 ], [ 1, 7, 8 ] ]
[ 8, 23, 30, 15 ]
#I  round 3, move: [ [ 4 ], [ 1, 6, 7, 8 ] ]
[ 7, 19, 24, 12 ]
#I  SCLibAdd: saving complex to file "complex_ReducedComplex_7_vertices_3_2009\
```

```
-10-27_11-40-00.sc".
#I  round 4, move: [ [ 2, 6 ], [ 3, 7, 8 ] ]
[ 7, 18, 22, 11 ]
#I  round 5, move: [ [ 2 ], [ 3, 5, 7, 8 ] ]
[ 6, 14, 16, 8 ]
#I  SCLibAdd: saving complex to file "complex_ReducedComplex_6_vertices_5_2009\
-10-27_11-40-00.sc".
#I  round 6, move: [ [ 5 ], [ 1, 3, 7, 8 ] ]
[ 5, 10, 10, 5 ]
#I  SCLibAdd: saving complex to file "complex_ReducedComplex_5_vertices_6_2009\
-10-27_11-40-00.sc".
#I  SCLibAdd: saving complex to file "complex_ReducedComplex_5_vertices_7_2009\
-10-27_11-40-00.sc".
#I  SCReduceComplexEx: complexes are bistellarly equivalent.
true
gap> myLib:=SCLibInit("~/PATH"); # copy path from above
[Simplicial complex library. Properties:
CalculateIndexAttributes=true
Number of complexes in library=4
IndexAttributes=[ "Name", "Date", "Dim", "F", "G", "H", "Chi", "Homology" ]
Loaded=true
Path="/home/spreerjn/reducedComplexes/2009-10-27_11-40-00/"
]
gap> s3:=myLib.Load(3);
[SimplicialComplex

 Properties known: Chi, Date, Dim, F, Faces, Facets, G, H, Homology,
                   IsConnected, Name, VertexLabels.

 Name="ReducedComplex_5_vertices_6"
 Dim=3
 Chi=0
 F=[ 5, 10, 10, 5 ]
 G=[ 0, 0 ]
 H=[ 1, 1, 1, 1 ]
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsConnected=true

/SimplicialComplex]
gap> s3:=myLib.Load(2);
[SimplicialComplex

 Properties known: Chi, Date, Dim, F, Faces, Facets, G, H, Homology,
                   IsConnected, Name, VertexLabels.

 Name="ReducedComplex_6_vertices_5"
 Dim=3
 Chi=0
 F=[ 6, 14, 16, 8 ]
 G=[ 1, 0 ]
 H=[ 2, 2, 2, 1 ]
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsConnected=true
```

```
/SimplicialComplex]
gap> t2:=SCCartesianProduct(SCBdSimplex(2),SCBdSimplex(2));;
gap> t2.F;
[ 9, 27, 18 ]
gap> SCBistellarOptions.WriteLevel:=0;
0
gap> SCBistellarOptions.LogLevel:=0;
0
gap> mint2:=SCReduceComplex(t2);
[ true, [SimplicialComplex

     Properties known: Dim, Facets, Name, VertexLabels.

     Name="unnamed complex 85"
     Dim=2

    /SimplicialComplex], 32 ]
```

17.7 Simplicial blowups

For a more detailed description of functions related to simplicial blowups see Chapter 10.

Example

```
gap> list:=SCLib.SearchByName("Kummer");
[ [ 7493, "4-dimensional Kummer variety (VT)" ] ]
gap> c:=SCLib.Load(7493);
[SimplicialComplex

 Properties known: AltshulerSteinberg, AutomorphismGroup,
                   AutomorphismGroupSize, AutomorphismGroupStructure,
                   AutomorphismGroupTransitivity,
                   ConnectedComponents, Date, Dim, DualGraph,
                   EulerCharacteristic, FacetsEx, GVector,
                   GeneratorsEx, HVector, HasBoundary, HasInterior,
                   Homology, Interior, IsCentrallySymmetric,
                   IsConnected, IsEulerianManifold, IsManifold,
                   IsOrientable, IsPseudoManifold, IsPure,
                   IsStronglyConnected, MinimalNonFacesEx, Name,
                   Neighborliness, NumFaces[], Orientation,
                   SkelExs[], Vertices.

 Name="4-dimensional Kummer variety (VT)"
 Dim=4
 AltshulerSteinberg=45137758519296000000000000
 AutomorphismGroupSize=1920
 AutomorphismGroupStructure="((C2 x C2 x C2 x C2) : A5) : C2"
 AutomorphismGroupTransitivity=1
 EulerCharacteristic=8
 GVector=[ 10, 55, 60 ]
 HVector=[ 11, 66, 126, -19, 7 ]
```

```
 HasBoundary=false
 HasInterior=true
 Homology=[ [0, [ ] ], [0, [ ] ], [6, [2,2,2,2,2] ], [0, [ ] ], [1, [ ] ] ]
 IsCentrallySymmetric=false
 IsConnected=true
 IsEulerianManifold=true
 IsOrientable=true
 IsPseudoManifold=true
 IsPure=true
 IsStronglyConnected=true
 Neighborliness=2

/SimplicialComplex]
gap> lk:=SCLink(c,1);
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="lk([ 1 ]) in 4-dimensional Kummer variety (VT)"
 Dim=3

/SimplicialComplex]
gap> SCHomology(lk);
[ [ 0, [  ] ], [ 0, [ 2 ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> SCLibDetermineTopologicalType(lk);
[ 45, 113, 2426, 2502, 7470 ]
gap> d:=SCLib.Load(45);;
gap> d.Name;
"RP^3"
gap> SCEquivalent(lk,d);
#I  SCReduceComplexEx: complexes are bistellarly equivalent.
true
gap> e:=SCBlowup(c,1);
#I  SCBlowup: checking if singularity is a combinatorial manifold...
#I  SCBlowup: ...true
#I  SCBlowup: checking type of singularity...
#I  SCReduceComplexEx: complexes are bistellarly equivalent.
#I  SCBlowup: ...ordinary double point (supported type).
#I  SCBlowup: starting blowup...
#I  SCBlowup: map boundaries...
#I  SCBlowup: boundaries not isomorphic, initializing bistellar moves...
#I  SCBlowup: found complex with smaller boundary: f = [ 15, 74, 118, 59 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 14, 70, 112, 56 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 14, 69, 110, 55 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 14, 68, 108, 54 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 13, 64, 102, 51 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 13, 63, 100, 50 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 13, 62, 98, 49 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 12, 58, 92, 46 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 12, 57, 90, 45 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 12, 56, 88, 44 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 11, 52, 82, 41 ].
#I  SCBlowup: found complex with smaller boundary: f = [ 11, 51, 80, 40 ].
```

```
#I  SCBlowup: found complex with isomorphic boundaries.
#I  SCBlowup: ...boundaries mapped succesfully.
#I  SCBlowup: build complex...
#I  SCBlowup: ...done.
#I  SCBlowup: ...blowup completed.
#I  SCBlowup: You may now want to reduce the complex via 'SCReduceComplex'.
[SimplicialComplex

 Properties known: Dim, FacetsEx, Name, Vertices.

 Name="unnamed complex 6315 \ star([ 1 ]) in unnamed complex 6315 cup unnamed\
 complex 6319 cup unnamed complex 6317"
 Dim=4

/SimplicialComplex]
gap> SCHomology(c);
[ [ 0, [  ] ], [ 0, [  ] ], [ 6, [ 2, 2, 2, 2, 2 ] ], [ 0, [  ] ], [ 1, [  ] ] ]
gap> SCHomology(e);
[ [ 0, [  ] ], [ 0, [  ] ], [ 7, [ 2, 2, 2, 2 ] ], [ 0, [  ] ], [ 1, [  ] ] ]
```

17.8 Discrete normal surfaces and slicings

For a more detailed description of functions related to discrete normal surfaces and slicings see the Sections 2.4 and 2.5.

Example

```
gap> # the boundary of the cyclic 4-polytope with 6 vertices
gap> c:=SCBdCyclicPolytope(4,6);
[SimplicialComplex

 Properties known: Dim, EulerCharacteristic, FacetsEx, HasBoundary, Homology,\
 IsConnected, IsStronglyConnected, Name, NumFaces[], TopologicalType, Vertices.

 Name="Bd(C_4(6))"
 Dim=3
 EulerCharacteristic=0
 HasBoundary=false
 Homology=[ [ 0, [ ] ], [ 0, [ ] ], [ 0, [ ] ], [ 1, [ ] ] ]
 IsConnected=true
 IsStronglyConnected=true
 TopologicalType="S^3"

/SimplicialComplex]
gap> # slicing in between the odd and the even vertex labels, a polyhedral torus
gap> sl:=SCSlicing(c,[[2,4,6],[1,3,5]]);
[NormalSurface

 Properties known: ConnectedComponents, Dim, EulerCharacteristic, FVector,\
 FacetsEx, Genus, IsConnected, IsOrientable, NSTriangulation, Name,\
 TopologicalType, Vertices.
```

```
 Name="slicing [ [ 2, 4, 6 ], [ 1, 3, 5 ] ] of Bd(C_4(6))"
 Dim=2
 FVector=[ 9, 18, 0, 9 ]
 EulerCharacteristic=0
 IsOrientable=true
 TopologicalType="T^2"

/NormalSurface]
gap> sl.Homology;
[ [ 0, [  ] ], [ 2, [  ] ], [ 1, [  ] ] ]
gap> sl.Genus;
1
gap> sl.F; # the slicing constists of 9 quadrilaterals and 0 triangles
[ 9, 18, 0, 9 ]
gap> PrintArray(sl.Facets);
[ [  [ 2, 1 ],  [ 2, 3 ],  [ 4, 1 ],  [ 4, 3 ] ],
  [  [ 2, 1 ],  [ 2, 3 ],  [ 6, 1 ],  [ 6, 3 ] ],
  [  [ 2, 1 ],  [ 2, 5 ],  [ 4, 1 ],  [ 4, 5 ] ],
  [  [ 2, 1 ],  [ 2, 5 ],  [ 6, 1 ],  [ 6, 5 ] ],
  [  [ 2, 3 ],  [ 2, 5 ],  [ 4, 3 ],  [ 4, 5 ] ],
  [  [ 2, 3 ],  [ 2, 5 ],  [ 6, 3 ],  [ 6, 5 ] ],
  [  [ 4, 1 ],  [ 4, 3 ],  [ 6, 1 ],  [ 6, 3 ] ],
  [  [ 4, 1 ],  [ 4, 5 ],  [ 6, 1 ],  [ 6, 5 ] ],
  [  [ 4, 3 ],  [ 4, 5 ],  [ 6, 3 ],  [ 6, 5 ] ] ]
```

Chapter 18

simpcomp internals

The package simpcomp works with geometric objects for which the GAP object types `SCSimplicialComplex` and `SCNormalSurface` are defined and calculates properties of these objects via so called property handlers. This chapter describes how to extend simpcomp by writing own property handlers.

If you extended simpcomp and want to share your extension with other users please send your extension to one of the authors and we will consider including it (of course with giving credit) in a future release of simpcomp.

18.1 The GAP object type `SCPropertyObject`

In the following, we present a number of functions to manage a GAP object of type `SCPropertyObject`. Since most properties of `SCPolyhedralComplex`, `SCSimplicialComplex` and `SCNormalSurface` are managed by the GAP4 type system (cf. [BL98]), the functions described below are mainly used by the object type `SCLibRepository` and to store temporary properties.

18.1.1 SCProperties

▷ `SCProperties(po)` (method)

Returns: a record upon success.

Returns the record of all stored properties of the `SCPropertyObject` *po*.

18.1.2 SCPropertiesFlush

▷ `SCPropertiesFlush(po)` (method)

Returns: `true` upon success.

Drops all properties and temporary properties of the `SCPropertyObject` *po*.

18.1.3 SCPropertiesManaged

▷ `SCPropertiesManaged(po)` (method)

Returns: a list of managed properties upon success, `fail` otherwise.

Returns a list of all properties that are managed for the `SCPropertyObject` *po* via property handler functions. See `SCPropertyHandlersSet` (18.1.9).

18.1.4 SCPropertiesNames

▷ `SCPropertiesNames(po)` (method)

Returns: a list upon success.

Returns a list of all the names of the stored properties of the `SCPropertyObject` *po*. These can be accessed via `SCPropertySet` (18.1.10) and `SCPropertyDrop` (18.1.8).

18.1.5 SCPropertiesTmp

▷ `SCPropertiesTmp(po)` (method)

Returns: a record upon success.

Returns the record of all stored temporary properties (these are mutable in contrast to regular properties and not serialized when the object is serialized to XML) of the `SCPropertyObject` *po*.

18.1.6 SCPropertiesTmpNames

▷ `SCPropertiesTmpNames(po)` (method)

Returns: a list upon success.

Returns a list of all the names of the stored temporary properties of the `SCPropertyObject` *po*. These can be accessed via `SCPropertyTmpSet` (18.1.14) and `SCPropertyTmpDrop` (18.1.13).

18.1.7 SCPropertyByName

▷ `SCPropertyByName(po, name)` (method)

Returns: any value upon success, `fail` otherwise.

Returns the value of the property with name *name* of the `SCPropertyObject` *po* if this property is known for *po* and `fail` otherwise. The names of known properties can be accessed via the function `SCPropertiesNames` (18.1.4)

18.1.8 SCPropertyDrop

▷ `SCPropertyDrop(po, name)` (method)

Returns: `true` upopn success, `fail` otherwise

Drops the property with name *name* of the `SCPropertyObject` *po*. Returns `true` if the property is successfully dropped and `fail` if a property with that name did not exist.

18.1.9 SCPropertyHandlersSet

▷ `SCPropertyHandlersSet(po, handlers)` (method)

Returns: `true`

Sets the property handling functions for a SCPropertyObject *po* to the functions described in the record *handlers*. The record *handlers* has to contain entries of the following structure: `[Property Name]:=[Function name computing and returning the property]`. For `SCSimplicialComplex` for example simpcomp defines (among many others): `F:=SCFVector`. See the file `lib/prophandler.gd`.

18.1.10 SCPropertySet

▷ `SCPropertySet(po, name, data)` (method)

Returns: `true` upon success.

Sets the value of the property with name *name* of the `SCPropertyObject` *po* to *data*. Note that the argument becomes immutable. If this behaviour is not desired, use `SCPropertySetMutable` (18.1.11) instead.

18.1.11 SCPropertySetMutable

▷ `SCPropertySetMutable(po, name, data)` (method)

Returns: `true` upon success.

Sets the value of the property with name *name* of the `SCPropertyObject` *po* to *data*. Note that the argument does not become immutable. If this behaviour is not desired, use `SCPropertySet` (18.1.10) instead.

18.1.12 SCPropertyTmpByName

▷ `SCPropertyTmpByName(po, name)` (method)

Returns: any value upon success, `fail` otherwise.

Returns the value of the temporary property with the name *name* of the `SCPropertyObject` *po* if this temporary property is known for *po* and `fail` otherwise. The names of known temporary properties can be accessed via the function `SCPropertiesTmpNames` (18.1.6)

18.1.13 SCPropertyTmpDrop

▷ `SCPropertyTmpDrop(po, name)` (method)

Returns: `true` upon success, `fail` otherwise

Drops the temporary property with name *name* of the `SCPropertyObject` *po*. Returns `true` if the property is successfully dropped and `fail` if a temporary property with that name did not exist.

18.1.14 SCPropertyTmpSet

▷ `SCPropertyTmpSet(po, name, data)` (method)

Returns: `true` upon success.

Sets the value of the temporary property with name *name* of the `SCPropertyObject` *po* to *data*. Note that the argument does not become immutable. This is the standard behaviour for temporary properties.

18.2 Example of a common attribute

In this section we will have a look at the property handler `SCEulerCharacteristic` (7.3.3) in order to explain the inner workings of property handlers. This is the code of the property handler for calculating the Euler characteristic of a complex in simpcomp:

Example
```
DeclareAttribute("SCEulerCharacteristic",SCIsPolyhedralComplex);

InstallMethod(SCEulerCharacteristic,
```

```
"for SCSimplicialComplex",
[SCIsSimplicialComplex],
function(complex)

        local  f, chi, i;

        f:=SCFVector(complex);
        if f=fail then
                return fail;
        fi;
        chi:=0;

        for i in [1..Size(f)] do
                        chi:=chi + ((-1)^(i+1))*f[i];
        od;

        return chi;
end);

InstallMethod(SCEulerCharacteristic,
"for SCNormalSurface",
[SCIsNormalSurface],
function(sl)

         local facets, f, chi;


         f:=SCFVector(sl);
         if(f=fail) then
                 return fail;
         fi;

        if Length(f) = 1 then
                         return f[1];
        elif Length(f) =3 then
                          return f[1]-f[2]+f[3];
         elif Length(f) =4 then
                          return f[1]-f[2]+f[3]+f[4];
         else
                 Info(InfoSimpcomp,1,"SCEulerCharacteristic: illegal f-vector found: ",f,". ");
                 return fail;
         fi;

end);
```

When looking at the code one already sees the structure that such a handler needs to have:

1. Each property handler (a GAP operation) needs to be defined. This is done by the first line of code. Once an operation is defined, multiple methods can by implemented for various types of GAP objects (here two methods are implemented for the GAP object types `SCSimplicialComplex` and `SCNormalSurface`).

2. First note that the validity of the arguments is checked by GAP. For example, the first method only accepts an argument of type `SCSimplicialComplex`.

3. If the property was already computed, the GAP4 type system automatically returns the cached property avoiding unnecessary double calculations.

4. If the property is not already known. it is computed and returned (and automatically cached by the GAP4 type system).

18.3 Writing a method for an attribute

This section provides the skeleton of a method that can be used when writing own methods:

Example

```
DeclareAttribute("SCMyPropertyHandler",SCPolyhedralComplex);

InstallMethod(SCMyPropertyHandler,
"for SCSimplicialComplex[ and further arguments]",
[SCIsSimplicialComplex[, further arguments]],
function(complex[, further arguments])

        local myprop, ...;

        # compute the property
 [ do property computation here]

        return myprop;
end);
```

References

[Ban65] Thomas F. Banchoff. Tightly embedded 2-dimensional polyhedral manifolds. *Amer. J. Math.*, 87:462–472, 1965. 18

[Ban74] Thomas F. Banchoff. Tight polyhedral Klein bottles, projective planes, and Möbius bands. *Math. Ann.*, 207:233–243, 1974. 18

[BBP+13] B. A. Burton, Ryan Budney, William Pettersson, et al. Regina: normal surface and 3-manifold topology software, version 4.95. `http://regina.sourceforge.net/`, 1999–2013. 8

[BD08a] Bhaskar Bagchi and Basudeb Datta. Lower bound theorem for normal pseudomanifolds. *Expo. Math.*, 26(4):327–351, 2008. 108

[BD08b] Bhaskar Bagchi and Basudeb Datta. On Walkup's class $\mathcal{K}(d)$ and a minimal triangulation of a 4-manifold. `arXiv:0804.2153v1 [math.GT]`, Preprint, 8 pages, 2008. 109

[BK97] Thomas F. Banchoff and Wolfgang Kühnel. Tight submanifolds, smooth and polyhedral. In *Tight and taut submanifolds (Berkeley, CA, 1994)*, volume 32 of *Math. Sci. Res. Inst. Publ.*, pages 51–118. Cambridge Univ. Press, Cambridge, 1997. 18

[BK08] Ulrich Brehm and Wolfgang Kühnel. Equivelar maps on the torus. *European J. Combin.*, 29(8):1843–1861, 2008. 68, 69, 71

[BK12] Ulrich Brehm and Wolfgang Kühnel. Lattice triangulations of E^3 and of the 3-torus. *Israel J. Math.*, 189:97–133, 2012. 56

[BL98] Thomas Breuer and Steve Linton. The gap 4 type system: organising algebraic algorithms. In *Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, ISSAC '98, pages 38–45, New York, NY, USA, 1998. ACM. 8, 21, 213

[BL00] Anders Björner and Frank H. Lutz. Simplicial manifolds, bistellar flips and a 16-vertex triangulation of the Poincaré homology 3-sphere. *Experiment. Math.*, 9(2):275–289, 2000. 14, 144

[BL14] Bruno Benedetti and Frank H. Lutz. Random discrete Morse theory and a new library of triangulations. *Exp. Math.*, 23(1):66–94, 2014. 169, 170

[BR08] Mohamed Barakat and Daniel Robertz. `homalg`: a meta-package for homological algebra. *J. Algebra Appl.*, 7(3):299–317, 2008. 187

[BS14] B. A. Burton and J. Spreer. Combinatorial seifert fibred spaces with transitive cyclic automorphism group. `arXiv:1404.3005 [math.GT]`, 2014. 26 pages, 10 figures. 59, 62, 66

[CK01] Mario Casella and Wolfgang Kühnel. A triangulated $K3$ surface with the minimum number of vertices. *Topology*, 40(4):753–772, 2001. 7

[Con09] Marston D. E. Conder. Regular maps and hypermaps of Euler characteristic -1 to -200. *J. Combin. Theory Ser. B*, 99(2):455–459, 2009. 67, 68, 70

[Dat07] Basudeb Datta. Minimal triangulations of manifolds. *J. Indian Inst. Sci.*, 87(4):429–449, 2007. 11

[DHSW11] J.-G. Dumas, F. Heckenbach, B. D. Saunders, and V. Welker. Simplicial Homology, v. 1.4.5. `http://www.cis.udel.edu/~dumas/Homology/`, 2001–2011. 2, 7, 95, 106, 107, 112, 116, 133, 194, 205

[DKT08] Mathieu Desbrun, Eva Kanso, and Yiying Tong. Discrete differential forms for computational modeling. In *Discrete differential geometry*, volume 38 of *Oberwolfach Semin.*, pages 287–324. Birkhäuser, Basel, 2008. 135, 136

[Eff11a] Felix Effenberger. *Hamiltonian submanifolds of regular polytopes*. Logos Verlag, Berlin, 2011. Dissertation, University of Stuttgart, 2010. 7, 58, 148

[Eff11b] Felix Effenberger. Stacked polytopes and tight triangulations of manifolds. *Journal of Combinatorial Theory, Series A*, 118(6):1843 – 1862, 2011. 100, 148, 159

[Eng09] Alexander Engström. Discrete Morse functions from Fourier transforms. *Experiment. Math.*, 18(1):45–53, 2009. 169

[For95] Robin Forman. A discrete Morse theory for cell complexes. In *Geometry, topology, & physics*, Conf. Proc. Lecture Notes Geom. Topology, IV, pages 112–125. Int. Press, Cambridge, MA, 1995. 18, 166

[Fro08] Andrew Frohmader. Face vectors of flag complexes. *Israel J. Math.*, 164:153–164, 2008. 99

[GJ00] Ewgenij Gawrilow and Michael Joswig. polymake: a framework for analyzing convex polytopes. In *Polytopes—combinatorics and computation (Oberwolfach, 1997)*, volume 29 of *DMV Sem.*, pages 43–73. Birkhäuser, Basel, 2000. 8

[Grü03] Branko Grünbaum. *Convex polytopes*, volume 221 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 2003. Prepared and with a preface by Volker Kaibel, Victor Klee and Günter M. Ziegler. 11

[GS] Daniel R. Grayson and Michael E. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at http://www.math.uiuc.edu/Macaulay2/. 8

[Hak61] Wolfgang Haken. Theorie der Normalflächen. *Acta Math.*, 105:245–375, 1961. 15

[Hau00] Herwig Hauser. Resolution of singularities 1860–1999. In *Resolution of singularities (Obergurgl, 1997)*, volume 181 of *Progr. Math.*, pages 5–36. Birkhäuser, Basel, 2000. 19

[Hir53] Friedrich E. P. Hirzebruch. über vierdimensionale riemannsche flächen mehrdeutiger analytischer funktionen von zwei komplexen veränderlichen. *Math. Ann.*, 126:1 – 22, 1953. 19

[Hop51] Heinz Hopf. Über komplex-analytische Mannigfaltigkeiten. *Univ. Roma. Ist. Naz. Alta Mat. Rend. Mat. e Appl. (5)*, 10:169–182, 1951. 19

[Hud69] John F. P. Hudson. *Piecewise linear topology*. University of Chicago Lecture Notes prepared with the assistance of J. L. Shaneson and J. Lees. W. A. Benjamin, Inc., New York-Amsterdam, 1969. 11

[Hup67] Bertram Huppert. *Endliche Gruppen. I*. Die Grundlehren der Mathematischen Wissenschaften, Band 134. Springer-Verlag, Berlin, 1967. 84

[KAL14] B. Benedetti K. Adiprasito and F. H. Lutz. Random discrete morse theory ii and a collapsible 5-manifold different from the 5-ball. `arXiv:1404.4239 [math.CO]`, 20 pages, 6 figures, 2 tables, 2014. 169, 170

[KL99] Wolfgang Kühnel and Frank H. Lutz. A census of tight triangulations. *Period. Math. Hungar.*, 39(1-3):161–183, 1999. Discrete geometry and rigidity (Budapest, 1999). 19

[KN12] Steven Klee and Isabella Novik. Centrally symmetric manifolds with few vertices. *Adv. Math.*, 229(1):487–500, 2012. 58

[Kne29] Hellmuth Kneser. Geschlossene Flächen in dreidimensionalen Mannigfaltigkeiten. *Jahresbericht der deutschen Mathematiker-Vereinigung*, 38:248–260, 1929. 15

[KS77] Robion C. Kirby and Laurence C. Siebenmann. *Foundational essays on topological manifolds, smoothings, and triangulations*. Princeton University Press, Princeton, N.J.; University of Tokyo Press, Tokyo, 1977. With notes by John Milnor and Michael Atiyah, Annals of Mathematics Studies, No. 88. 173

[Küh86] Wolfgang Kühnel. Higher dimensional analogues of Császár's torus. *Results Math.*, 9:95–106, 1986. 19, 52

[Küh94] Wolfgang Kühnel. Manifolds in the skeletons of convex polytopes, tightness, and generalized Heawood inequalities. In *Polytopes: abstract, convex and computational (Scarborough, ON, 1993)*, volume 440 of *NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci.*, pages 241–247. Kluwer Acad. Publ., Dordrecht, 1994. 18

[Küh95] Wolfgang Kühnel. *Tight polyhedral submanifolds and tight triangulations*, volume 1612 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1995. 15, 18, 19, 159

[Kui84] Nicolaas H. Kuiper. Geometry in total absolute curvature theory. In *Perspectives in mathematics*, pages 377–392. Birkhäuser, Basel, 1984. 18

[Lut] Frank H. Lutz. The Manifold Page. `http://www.math.tu-berlin.de/diskregeom/stellar`. 2, 78

[Lut03] Frank H. Lutz. *Triangulated Manifolds with Few Vertices and Vertex-Transitive Group Actions*. PhD thesis, TU Berlin, 2003. 2, 7, 65, 78, 79

[Lut05] Frank H. Lutz. Triangulated Manifolds with Few Vertices: Combinatorial Manifolds. `arXiv:math/0506372v1 [math.CO]`, Preprint, 37 pages, 2005. 11

[MP14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014. 2, 83

[Pac87] Udo Pachner. Konstruktionsmethoden und das kombinatorische Homöomorphieproblem für Triangulierungen kompakter semilinearer Mannigfaltigkeiten. *Abh. Math. Sem. Uni. Hamburg*, 57:69–86, 1987. 102, 113, 114, 143, 144

[PS14] J. Paixao and J. Spreer. Probabilistic collapsibility testing and manifold recognition heuristics, 2014. In preparation, 6 pages. 170, 172, 173, 174

[RÏ3] Marc Röder. GAP package polymaking. `http://www.gap-system.org/Packages/polymaking.html`, 2013. 182

[Rin74] Gerhard Ringel. *Map color theorem*. Springer-Verlag, New York, 1974. Die Grundlehren der mathematischen Wissenschaften, Band 209. 18

[RS72] Colin P. Rourke and Brian J. Sanderson. *Introduction to piecewise-linear topology*. Springer-Verlag, New York, 1972. Ergebnisse der Mathematik und ihrer Grenzgebiete, Band 69. 11, 18

[Sch94] Christoph Schulz. Polyhedral manifolds on polytopes. *Rend. Circ. Mat. Palermo (2) Suppl.*, (35):291–298, 1994. First International Conference on Stochastic Geometry, Convex Bodies and Empirical Measures (Palermo, 1993). 19

[SK11] Jonathan Spreer and Wolfgang Kühnel. Combinatorial properties of the K3 surface: Simplicial blowups and slicings. *Experiment. Math.*, 20(2):201–216, 2011. 7, 16, 19, 20, 156, 158

[Soi12] Leonard H. Soicher. GRAPE - GRaph Algorithms using PErmutation groups. `http://www.gap-system.org/Packages/grape.html`, 2012. Version 4.6.1. 2, 7, 83, 194

[Spa56] Edwin H. Spanier. The homology of Kummer manifolds. *Proc. AMS*, 7:155–160, 1956. 19

[Spa99] Eric Sparla. A new lower bound theorem for combinatorial $2k$-manifolds. *Graphs Combin.*, 15(1):109–125, 1999. 58

[Spr11a] J. Spreer. *Blowups, slicings and permutation groups in combinatorial topology*. PhD thesis, University of Stuttgart, 2011. Ph.D. thesis. 7, 55, 59, 61, 63, 64, 65

[Spr11b] Jonathan Spreer. Normal surfaces as combinatorial slicings. *Discrete Math.*, 311(14):1295–1309, 2011. `doi:10.1016/j.disc.2011.03.013`. 15, 16, 43, 119

[Spr12] J. Spreer. Partitioning the triangles of the cross polytope into surfaces. *Beitr. Algebra Geom. / Contributions to Algebra and Geometry*, 53(2):473–486, 2012. 60

[Spr14] Jonathan Spreer. Combinatorial 3-manifolds with transitive cyclic symmetry. *Discrete Comput. Geom.*, 51(2):394–426, 2014. 80, 81

[Wee99] Jeff Weeks. SnapPea (Software for hyperbolic 3-manifolds), 1999. `http://www.geometrygames.org/SnapPea/`. 8

[Wil96] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the Twenty-eighth Annual ACM Symposium on the Theory of Computing (Philadelphia, PA, 1996)*, pages 296–303. ACM, New York, 1996. 171

[Zie95] Günter M. Ziegler. *Lectures on polytopes*, volume 152 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1995. 11, 102, 113, 114

Index