Chapter 5

New certified Kenzo modules

At this moment the Kenzo system keeps on growing. Several researchers are developing new functionalities for it; for instance, spectral sequences [RRS06], resolutions [RER09], Koszul complexes [RS06] and so on.

An important point, that was already introduced in the previous chapter (see Section 4.4), in the development of new Kenzo modules is the verification of their correctness. The Kenzo system is a research tool that has got some relevant results not confirmed nor refuted by neither theoretical or computational means. Then, the question of Kenzo reliability (beyond testing) naturally arises.

In this chapter three new Kenzo modules (developed by us), their integration in the *fKenzo* system and some remarks about their verification in the ACL2 theorem prover are presented. Section 5.1 is devoted to describe our development related to simplicial complexes, a basic notion in Algebraic Topology which can be used to study digital images. Namely, we can associate a simplicial complex with a digital image and then study properties of the image by means of topological invariants (such as homology groups) of the associated simplicial complex. The framework to study digital images by means of simplicial complexes is explained in Section 5.2. Finally, explanations about the construction of the effective homology of the pushout of simplicial sets with effective homology are given in Section 5.3.

5.1 Simplicial complexes

The most elementary method to settle a connection between common "topology" and Algebraic Topology is based on the usage of simplicial complexes. The notion of topological space is too "abstract" in order to transfer it to machine universe. Simplicial complexes provide a purely combinatorial description of topological spaces which admit a triangulation. The computability of properties, such as homology groups, from a finite simplicial complex associated with a topological space is well-known and, for instance in the case of homology groups the algorithm uses simple linear algebra [Veb31]. Then, an algebraic topologist can identify a compact triangulable topological space with a finite simplicial complex, making computations easier.

Simplicial complexes are not included in the current www-available version of Kenzo. We have undertaken the task of deploying a new certified Kenzo module to work with them. In addition, we have also enhanced the *fKenzo* system to deal with simplicial complexes.

The rest of this section is organized as follows. Subsection 5.1.1 introduces the basic background about simplicial complexes and some algorithms about them; in Subsection 5.1.2 the new Kenzo module about simplicial complexes is presented; Subsection 5.1.3 explains how our framework is extended to include the functionality about simplicial complexes; moreover, the way of widening the *fKenzo* GUI to include simplicial complexes is detailed in Subsection 5.1.4. Finally, some aspects of the verification of the simplicial complexes programs are presented in Subsection 5.1.5.

5.1.1 Mathematical concepts

We briefly provide in this subsection the minimal mathematical background about simplicial complexes. We mainly focus on definitions. Many good textbooks are available for both these definitions and results about them, for instance [Mau96].

Let us start with the basic terminology. Let V be an ordered set, called the *vertex set*. An *(ordered abstract) simplex* over V is any ordered finite subset of V. An *(ordered abstract) n-simplex* over V is a simplex over V whose cardinality is equal to $n+1$. Given a simplex α over V, we call *faces* of α to all the subsets of α.

Definition 5.1. An *(ordered abstract) simplicial complex* over V is a set of simplexes $\mathcal{K}$ over V such that it is closed by taking faces (subsets); that is to say:

$$\forall \alpha \in \mathcal{K},\ if\ \beta \subseteq \alpha \Rightarrow \beta \in \mathcal{K}.$$

Let $\mathcal{K}$ be a simplicial complex. Then the set $S_n(\mathcal{K})$ of n-simplexes of $\mathcal{K}$ is the set made of the simplexes of cardinality $n+1$ of $\mathcal{K}$.

Example 5.2. Let us consider $V = (0, 1, 2, 3, 4, 5, 6)$.

The small simplicial complex drawn in Figure 5.1 is mathematically defined as the object:

$$\mathcal{K} = \left\{ \begin{array}{l} \emptyset, (0), (1), (2), (3), (4), (5), (6), \\ (0,1), (0,2), (0,3), (1,2), (1,3), (2,3), (3,4), (4,5), (4,6), (5,6), \\ (0,1,2), (4,5,6) \end{array} \right\}.$$

Note that, because the vertex set is ordered the list of vertices of a simplex is also ordered, which allows us to use a sequence notation $(\ldots)$ and not a subset notation

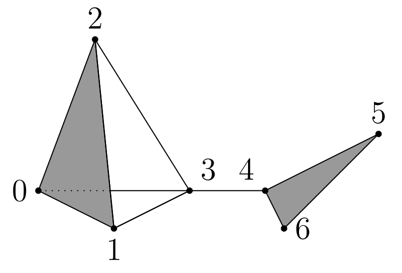


Figure 5.1: Butterfly simplicial complex

$\{\ldots\}$ for a simplex and also for the vertex set V. It is also worth noting that simplicial complexes can be infinite. For instance if $V = \mathbb{N}$ and the simplicial complex $\mathcal{K}$ is $\{(n)\}_{n\in\mathbb{N}} \cup \{(n-1,n)\}_{n\geq 1}$, the simplicial complex obtained can be seen as an infinite bunch of segments.

Definition 5.3. A *facet* of a simplicial complex $\mathcal{K}$ over V is a maximal simplex with respect to the subset relation, $\subseteq$, among the simplexes of $\mathcal{K}$.

Example 5.4. The facets of the small simplicial complex depicted in Figure 5.1 are: $\{(0,3),(1,3),(2,3),(3,4),(0,1,2),(4,5,6)\}$

Let us note that a *finite* simplicial complex can be generated from its facets taking the set union of the power set of each one of their facets. In general, we have the following definition.

Definition 5.5. Let $\mathcal{S}$ be a finite sequence of simplexes, then the set union of the power set of each one of the elements of $\mathcal{S}$ is, trivially, a simplicial complex called the *simplicial complex associated with* $\mathcal{S}$.

It is worth noting that the same simplicial complex can be generated from two different sequences of simplexes; in addition, the minimal sequence of simplexes which generates a finite simplicial complex is the sequence of its facets.

Then, the following algorithm can be defined.

Algorithm 5.6.
Input: a sequence of simplexes $\mathcal{S}$.
Output: the associated simplicial complex with $\mathcal{S}$.

Example 5.7. Let us show the way of generating the simplicial complex depicted in Figure 5.1 from its facets. Table 5.1 shows the faces of facets of the butterfly simplicial complex. If we perform the set union of all the faces, the desired simplicial complex is obtained.

In Subsection 1.1.2, we have defined the notion of simplicial set, a notion more complex than the notion of simplicial complex. Nevertheless, many common constructions

facet	faces
$(1,3)$	$\{\emptyset,(1),(3),(1,3)\}$
$(3,4)$	$\{\emptyset,(3),(4),(3,4)\}$
$(0,3)$	$\{\emptyset,(0),(3),(0,3)\}$
$(2,3)$	$\{\emptyset,(2),(3),(2,3)\}$
$(0,1,2)$	$\{\emptyset,(0),(1),(2),(0,1),(0,2),(1,2),(0,1,2)\}$
$(4,5,6)$	$\{\emptyset,(4),(5),(6),(4,5),(5,6),(4,6),(4,5,6)\}$

Table 5.1: Faces of the facets of the Butterfly simplicial complex

in topology are difficult to make explicit in the framework of simplicial complexes. It soon became clear around 1950 that the notion of simplicial set is much better.

There exists a link between these two notions which will allow us to compute homology groups of a simplicial complex by means of a simplicial set.

Definition 5.8. Let $\mathcal{SC}$ be an (ordered abstract) simplicial complex over V. Then the *simplicial set* $K(\mathcal{SC})$ *canonically associated* with $\mathcal{SC}$ is defined as follows. The set $K^n(\mathcal{SC})$ is $S_n(\mathcal{SC})$, that is, the set made of the simplexes of cardinality $n+1$ of $\mathcal{SC}$. In addition, let $(v_0,\dots,v_q)$ be a q-simplex, then the *face* and *degeneracy* operators of the simplicial set $K(\mathcal{SC})$ are defined as follows:

$$\begin{aligned}\partial_i^q((v_0,\dots,v_i,\dots,v_q)) &= (v_0,\dots,v_{i-1},v_{i+1},\dots,v_q),\\ \eta_i^q((v_0,\dots,v_i,\dots,v_q)) &= (v_0,\dots,v_i,v_i,\dots,v_q).\end{aligned}$$

That is, the face operator ∂_i^q removes the vertex in the position i of a q-simplex, and the degeneracy operator η_i^q duplicates the vertex in the position i of a q-simplex.

The proof of the fact that $K(\mathcal{SC})$ is a simplicial set is quite easy.

Then, the above definition provides us the following algorithm.

Algorithm 5.9.
Input: a finite simplicial complex $\mathcal{SC}$.
Output: the simplicial set $K(\mathcal{SC})$ canonically associated with $\mathcal{SC}$.

Definition 5.10. Given a simplicial complex $\mathcal{SC}$, the *n-homology group* of $\mathcal{SC}$, $H_n(\mathcal{SC})$, is the n-homology group of the simplicial set $K(\mathcal{SC})$:

$$H_n(\mathcal{SC}) = H_n(K(\mathcal{SC})).$$

5.1.2 Simplicial complexes in Kenzo

In the current www-available version of Kenzo, the notion of simplicial complex is not included. Then, we have developed a new Common Lisp module to enhance the Kenzo system with this notion. Our programs (with about 150 lines) implement Algorithms 5.6 and 5.9.

The following lines are devoted to explain the essential part of these programs, describing the functions with the same format as in the Kenzo documentation [DRSS98].

First of all, let us note that the vertex set V in our programs is $\mathbb{N}$; besides, we represent an n-simplex as a strictly ordered list of $n+1$ natural numbers that represent the vertices of the simplex. For instance, the 2-simplex with vertices 0, 1 and 3 is represented as the list `(0 1 3)`. Moreover, a finite simplicial complex is represented, in our system, by means of a list of simplexes.

Our first program implements Algorithm 5.6, that is, the functions which generate a simplicial complex from a sequence of simplexes. The description of the main function in charge of this task is shown here:

simplicial-complex-generator ls *[Function]*

> From a list of simplexes, `ls`, this function generates the associated simplicial complex, that is to say, another list of simplexes.

The second program implements Algorithm 5.9. It generates the simplicial set canonically associated with a simplicial complex as an instance of the `Simplicial-Set` Kenzo class (see Subsection 1.2.1). The main function is:

ss-from-sc *simplicial-complex* *[Function]*

> Build an instance of the `Simplicial-Set` Kenzo class which represents the simplicial set canonically associated with a simplicial complex, *simplicial-complex*, see Definition 5.8, using some auxiliar functions which are necessary to define simplicial sets in Kenzo.

To provide a better understanding of the new tools, an elementary example of their use is presented now. Let us consider the (minimal) triangulation of the torus presented in Figure 5.2.

The facets of the torus of Figure 5.2 are all the triangles (2-simplexes) depicted in that figure. Therefore, in our program the facets of the torus are assigned to the variable `torus-facets` as follows:

```
> (setf torus-facets  '((0 1 3) (0 1 5) (0 2 4) (0 2 6) (0 3 6) (0 4 5)
                         (1 2 5) (1 2 6) (1 3 4) (1 4 6) (2 3 4) (2 3 5)
                         (3 5 6) (4 5 6)))) ✠
((0 1 3) (0 1 5) (0 2 4) (0 2 6) (0 3 6) (0 4 5) (1 2 5) (1 2 6) (1 3 4) ...)
```

From these facets, we can construct the torus simplicial complex $S^1 \times S^1$ with our program:

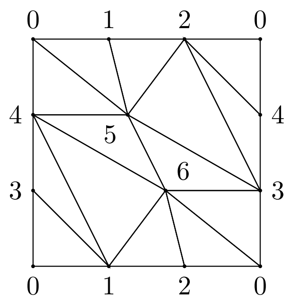


Figure 5.2: Torus triangulation

```
> (setf torus (simplicial-complex-generator torus-facets)) ✠
((0 1 3) (0 1) (0) (1) (1 3) (3) (0 1 5) (0 5) (5) (0 2 4) ...)
```

Once we have constructed this simplicial complex, we can build the simplicial set canonically associated with the torus simplicial complex by means of the instruction:

```
> (setf torus-ss (ss-from-sc torus)) ✠
[K1 Simplicial-Set]
```

Finally, we can determine its homology groups thanks to the Kenzo kernel.

```
> (homology torus-ss 0 3) ✠
Homology in dimension 0:
Component Z
Homology in dimension 1:
Component Z
Component Z
Homology in dimension 2:
Component Z
```

The result must be interpreted as stating $H_0(torus) = \mathbb{Z}$, $H_1(torus) = \mathbb{Z} \oplus \mathbb{Z}$ and $H_2(torus) = \mathbb{Z}$.

5.1.3 Integration of simplicial complexes

From the beginning of the development of our framework we strove for a system which could evolve with Kenzo. Therefore, to enhance our system with the functionality related to simplicial complexes we have developed a plug-in following the guidelines given in Subsubsection 3.1.2.

This new plug-in will allow us to construct from a sequence of simplexes $\mathcal{S}$ the simplicial set canonically associated with the simplicial complex defined from $\mathcal{S}$ (applying

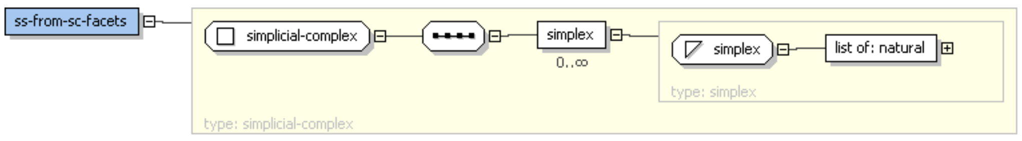


Figure 5.3: ss-from-sc-facets constructor in XML-Kenzo

Algorithms 5.6 and 5.9). The plug-in employed to include the functionality about simplicial complexes references the following resources:

```
<code id="simplicial-complexes">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> simplicial-complexes.lisp </data>
   <data format="Kf/microkernel"> simplicial-complexes-m.lisp </data>
   <data format="Kf/adapter"> simplicial-complexes-a.lisp </data>
</code>
```

As we claimed in Subsubsection 3.1.2, if we want to include new functionality related to Kenzo in our framework, all its components must be broadening. Let us explain each one of the referenced resources.

First of all, as we want to introduce a new kind of objects in our system (the simplicial sets associated with simplicial complexes generated from a sequence of simplexes), it is necessary to provide a representation for that kind of objects in our framework. Therefore, we have extended the XML-Kenzo specification (`XML-Kenzo.xsd` file) to admit the new objects related to simplicial complexes. In this specification, we have defined both the simple type called `simplex` which represents a list of natural numbers and the type called `simplicial-complex` which represents a sequence of `simplex` elements; and a new element: `ss-from-sc-facets` (see Figure 5.3), whose value is an element whose type is `simplicial-complex`. The `ss-from-sc-facets` element is defined as an element of the `SS` type; so, it can be used as any other element of this group.

As we have explained in Subsection 5.1.2 a simplex is implemented in our programs as an ordered list of natural numbers; however, in the XML-Kenzo specification we can only constrain the value of the `simplex` type to be a list of natural numbers. Therefore, the programs included in the microkernel will be in charge of checking the restriction of being ordered; here, we have an example of a functional dependency of a compound argument. As we explained in Subsubsection 3.1.2 the external server evolves when the `XML-Kenzo.xsd` file is upgraded. Then, when the XML-Kenzo.xsd file is modified to include the new elements, the external server is able to check the restrictions against the XML-Kenzo specification of requests such as:

```
<constructor>
    <ss-from-sc-facets>
        <simplex>0 1 2</simplex>
        <simplex>3 4 5</simplex>
    </ss-from-sc-facets>
</constructor>
```

The `simplicial-complexes.lisp` file includes the functionality explained in Subsection 5.1.2 to extend the Kenzo system. Moreover, this file adds code that enhances the `xml-kenzo-to-kenzo` function of the internal server to process the construction of objects from the `ss-from-sc-facets` XML-Kenzo construction requests. For instance, if the internal server receives the above request, the following instruction is executed in the Kenzo kernel.

```
(ss-from-sc (simplicial-complex-generator '((0 1 2) (3 4 5))))
```

As a result an object of the `Simplicial-Set` Kenzo class is constructed, and the identifier of that object is returned using an `id` XML-Kenzo object.

The `simplicial-complexes-m.lisp` file defines a new construction module for the microkernel called `simplicial-complex`. The procedure implemented in this module follows the guidelines explained in Subsubsection 2.2.3.3. In this case, the procedure implemented in this module checks that the values of `simplex` elements of a `ss-from-sc-facets` request are ordered lists since this constraint cannot be imposed in the XML-Kenzo specification (remember that functional dependencies cannot be defined in the XML-Kenzo specification), and therefore is not checked in the external server. Moreover, this file enhances the interface of the microkernel in order to be able to invoke the new `simplicial-complex` module.

Finally, we have extended the `SS` Content Dictionary by means of the definition of a new object: `ss-from-sc-facets`. Therefore, the `simplicial-complexes-a.lisp` file raises the functionality of the adapter, which is now able to convert from the new OpenMath objects, devoted to simplicial complexes, to XML-Kenzo requests. Namely, we have extended the Phrasebook by means of a new parser in charge of this task. Then, for instance, the XML-Kenzo request presented previously is generated by the adapter when the following OpenMath request is received:

```
<OMOBJ>
   <OMA>
      <OMS cd="SS" name="ss-from-sc-facets"/>
      <OMA>
        <OMS name="simplex"/>
          <OMI>0</OMI><OMI>1</OMI><OMI>2</OMI>
      </OMA>
      <OMA>
        <OMS name="simplex"/>
          <OMI>3</OMI><OMI>4</OMI><OMI>5</OMI>
      </OMA>
   </OMA>
</OMOBJ>
```

5.1.4 Integration of simplicial complexes in the ***fKenzo*** GUI

The current subsection is devoted to present the necessary resources to extend the *fKenzo* GUI in order to handle simplicial complexes.

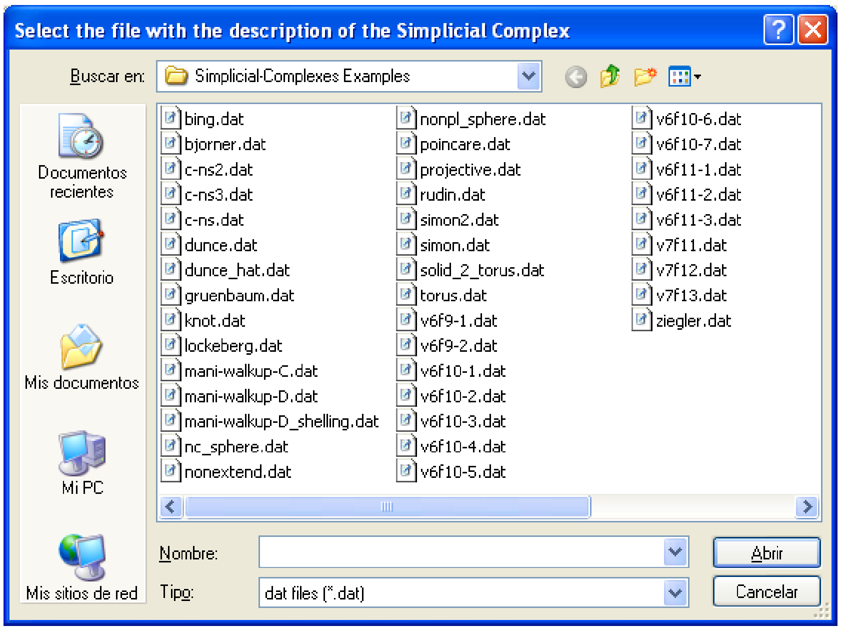
As we presented in Section 3.2 one of the modules which customizes *fKenzo* is the *Simplicial Set* module. This module contains the elements that represent simplicial set constructors of Kenzo: options to construct spaces from scratch (spheres, Moore spaces, finite simplicial sets, and so on) and from other spaces (for instance, cartesian products). We have enhanced this module by means of a new constructor related to simplicial complexes.

The original *Simplicial Set* module referenced two files: `simplicial-sets-structure` (which defined the structure of the graphical constituents of the module) and `simplicial-sets-functionality` (which provided the functionality related to the graphical constituents). Both files have been upgraded in order to tackle the use of simplicial complexes in the *fKenzo* GUI. Moreover, the new *Simplicial Set* module also references the plug-in described in the previous subsection. In this way, when the *Simplicial Set* module is loaded, the whole *fKenzo* system is ready to allow the construction of simplicial sets coming from simplicial complexes.

We have defined three new graphical elements, using the XUL specification language, in the `simplicial-sets-structure` file:

- A new menu option called `Load Simplicial Complex` included in the `Simplicial Sets` menu.
- A window called `Load Simplicial Complex` (see Figure 5.4).
- A window called `SS-from-SC-name` (see Figure 5.5).

The functionality stored in the `simplicial-sets-functionality` document related



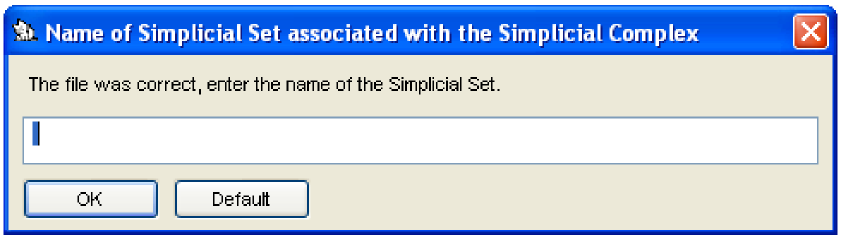
Buscar en: Simplicial-Complexes Examples

Tipo: dat files (*.dat)

Figure 5.4: `Load Simplicial Complex` window

to these components works as follows. A function acting as event handler is associated with the `Load Simplicial Complex` menu option; this function shows the window `Load Simplicial Complex` (see Figure 5.4) which allows the user to choose a file which contains a list of simplexes. It is worth noting that the user does not introduce each one of the simplexes of the sequence manually: he selects a file with that information (a file which can be either generated by a computer program or build manually with a text editor). Currently, a folder, called *Simplicial-Complexes Examples*, with several examples of simplicial complexes defined from their a sequence of simplexes is included in the distribution of *fKenzo*. At this moment it contains 39 examples such as the torus, the projective space or the duncehat (several of those examples were extracted from [Hac01] keeping its original format, which is the one used in our system).

Once the user has selected a file, the system processes the file to construct an `ss-from-sc-facets` OpenMath request with the sequence of simplexes obtained from the file. Subsequently, the framework constructs the simplicial set associated with the simplicial complex defined from the sequence of simplexes and returns its identification number (an `id` XML-Kenzo object). Afterwards, the system asks a name for the new simplicial set by means of the window `SS-from-SC-name` (see Figure 5.5); if the name given by the user is correct (was not used previously), the system stores it, otherwise it indicates that the name was previously used and asks again a name. Eventually, when the user has given a valid name, the system builds a new `FKENZO-OBJECT-NAME` instance (see Subsubsection 4.1.5.1) and, also, adds the new object to the list of constructed spaces (located at the left side of the main tab of the *fKenzo* GUI). Figure 5.6 shows the control and navigation submodel (with the Noesis notation) describing the construction of the simplicial set canonically associated with a simplicial complex from


Name of Simplicial Set associated with the Simplicial Complex

The file was correct, enter the name of the Simplicial Set.

Figure 5.5: *fKenzo* asking a name for a simplicial set coming from a simplicial complex

the `Load Simplicial Complex` menu option.

5.1.4.1 Execution flow of the *fKenzo* GUI for simplicial complexes

In order to clarify the execution flow followed by *fKenzo*, let us retake the example presented in Subsection 5.1.2, that is, the computation of the first homology groups of the torus. When the Simplicial Set module is loaded in *fKenzo*, the option *Load Simplicial Complex* is available in the *Simplicial Sets* menu. From this option, the user can select the file which contains the facets of the torus; this file is called `torus.dat` and is located at the folder *Simplicial-Complexes Examples*, see Figure 5.4. An extract of the data stored in that file is:

```
#
0 1 3
0 1 5
...
```

which corresponds with the facets of the triangulation of the torus of Figure 5.2. From that file, *fKenzo* constructs the following OpenMath request:

```
<OMOBJ>
   <OMA>
      <OMS cd="SS" name="ss-from-sc-facets"/>
      <OMA>
        <OMS cd="SS" name="simplex"/>
          <OMI>0</OMI><OMI>1</OMI><OMI>3</OMI>
      </OMA>
      <OMA>
        <OMS cd="SS" name="simplex"/>
          <OMI>0</OMI><OMI>1</OMI><OMI>5</OMI>
      </OMA>
      ...
   </OMA>
</OMOBJ>
```

which is sent to the framework. From this request, the simplicial set associated with

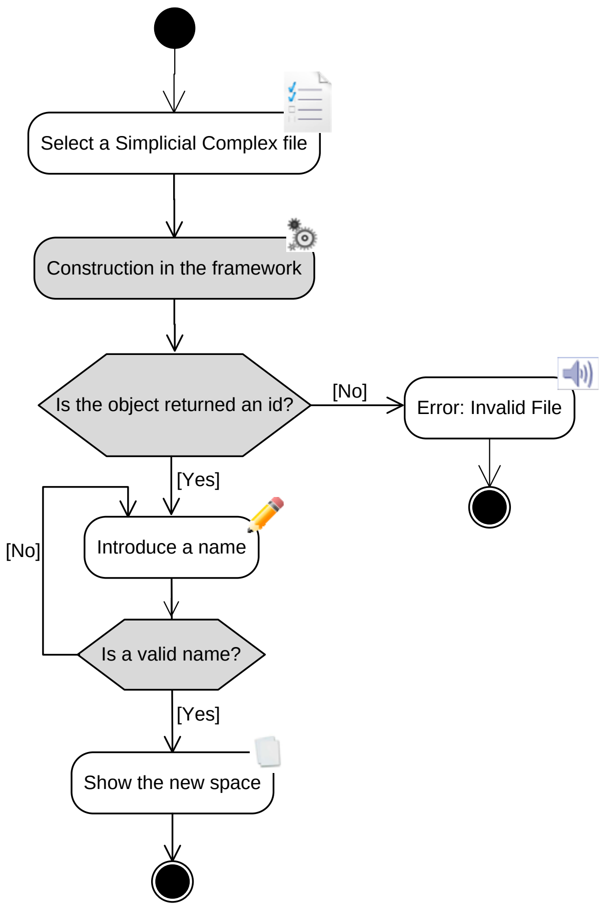
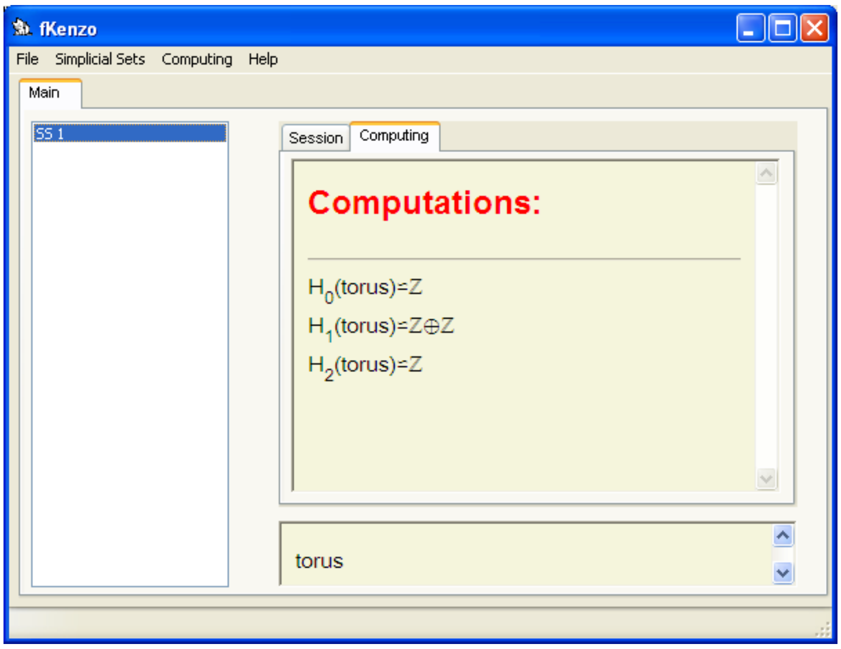

Select a Simplicial Complex file
Construction in the framework
Is the object returned an id?
[No]
Error: Invalid File
[Yes]
[No]
Introduce a name
Is a valid name?
[Yes]
Show the new space

Figure 5.6: Construction of a simplicial set from a simplicial complex in *fKenzo*



Computations:

H_0(torus)=Z

H_1(torus)=Z⊕Z

H_2(torus)=Z

torus

Figure 5.7: Homology groups of the torus

the torus simplicial complex is built and its identification number is returned. Then, *fKenzo* asks the user a name for the object (in this case the given name is "torus"). Subsequently, the new object is added to the list of constructed spaces which is shown in the left side of the *fKenzo* GUI. When, the user selects this object, the name that the user provided previously is shown (see Figure 5.7).

Finally, the user can ask *fKenzo* to compute the homology groups of the torus using the `homology` option of the *Computing* menu, the results are shown, as usual, in the Computing tab, see Figure 5.7.

5.1.5 Formalization of Simplicial Complexes in ACL2

As we said at the beginning of this chapter, we are interested not only in developing new tools for the Kenzo system but also in verifying the correctness of these new tools. This subsection is devoted to present the certification of the correctness of our implementation of Algorithm 5.6. The certification of the correctness of the implementation of Algorithm 5.9 belongs to a more general case (the certification of the implementation of Kenzo simplicial sets) and we will cope with it in the next chapter.

5.1.5.1 Main definitions and properties

As we have just said, we want to formalize in ACL2 the correctness of the `simplicial-complex-generator` function; that is to say, our implementation of Algorithm 5.6. Since both Kenzo and ACL2 are Common Lisp programs we can verify the correctness of the `simplicial-complex-generator` function in ACL2.

First of all, let us present the definition of the `simplicial-complex-generator` function. This program can be decomposed in three steps. First of all, we have defined the `powerset` function which generates the powerset of a simplex.

```
(defun map-cons (x s)
  (if (endp s)
      nil
    (cons (cons x (car s)) (map-cons x (cdr s)))))

(defun powerset (l)
  (if (endp l)
      (list nil)
    (append (powerset (cdr l)) (map-cons (car l) (powerset (cdr l))))))
```

Subsequently, we have defined the `simplicial-complex-generator-aux` function which builds a list gathering the powerset of every simplex of a sequence of simplexes `ls`.

```
(defun simplicial-complex-generator-aux (ls)
  (if (endp ls)
      nil
    (append (powerset (car ls)) (simplicial-complex-generator-aux (cdr ls)))))
```

Eventually, we have implemented the `simplicial-complex-generator` function which removes the duplicates elements (thanks to the ACL2 function `remove-duplicates-equal`) of the list generated by `simplicial-complex-generator-aux`, getting the looking for simplicial complex.

```
(defun simplicial-complex-generator (ls)
  (remove-duplicates-equal (simplicial-complex-generator-aux ls)))
```

This design follows simple recursive schemas, which are suitable for the induction heuristics of the ACL2 theorem prover.

From now on, we define the necessary functions to prove the correctness of our program. First of all, we need some auxiliary functions which define the necessary concepts to prove our theorems. These definitions are based on both Algorithm 5.6 and Definition 5.1. Namely, we need to define the notions of *simplex*, *list of simplexes*, *set of simplexes*, *face* and *member* in ACL2.

As we said in Subsection 5.1.2, a *simplex* in our programs is a strictly ordered list of natural numbers. The `simplex-p` function is a predicate that given a list `list` returns `t` if the list represents a simplex and `nil` otherwise.

```
(defun simplex-p (list)
  (if (endp list)
      (equal list nil)
    (if (endp (cdr list))
        (and (equal (cdr list) nil) (natp (car list)))
      (and (natp (car list)) (natp (cadr list)) (< (car list) (cadr list))
           (simplex-p (cdr list))))))
```

From this notion of simplex, we can easily define the notion of *list of simplexes* by means of the `list-of-simplex-p` predicate which returns `t` if its argument is a list of simplexes and `nil` otherwise:

```
(defun list-of-simplexes-p (ls)
  (if (endp ls)
      (equal ls nil)
    (and (simplex-p (car ls)) (list-of-simplexes-p (cdr ls)))))
```

A *set of simplexes* is a *list of simplexes* without duplicate elements. This is modeled with the `set-of-simplexes-p` predicate (a test function, called `without-duplicates-p`, that checks if a list does not have duplicate elements has been defined as auxiliar function).

```
(defun set-of-simplexes-p (ls)
  (and (list-of-simplexes-p ls) (without-duplicates-p ls)))
```

Finally, to define the relations "be a face of" (between two simplexes) and "be in" (between a simplex and a set of simplexes or between a simplex and a list of simplexes) we have used two already defined ACL2 functions that are `subsetp-equal` and `member-equal` respectively.

Therefore, we have the framework to prove the correctness of our programs. The following theorems state the basic properties that the `simplicial-complex-generator` function must satisfy. First of all, we prove that `simplicial-complex-generator` constructs a simplicial complex. Therefore, we need to prove the following two ACL2 lemmas.

ACL2 Lemma 5.11. Let ls be a list of simplexes, then (`simplicial-complex-generator` ls) builds a set of simplexes.

```
(defthm simplicial-complex-generator-constructs-simplicial-complex-1
  (implies (list-of-simplexes-p ls)
           (set-of-simplexes-p (simplicial-complex-generator ls))))
```

ACL2 Lemma 5.12. Let x be a simplex and ls be a list of simplexes, if x belongs to (`simplicial-complex-generator` ls) and y is a face of x, then y belongs to (`simplicial-complex-generator` *ls*).

```
(defthm simplicial-complex-generator-constructs-simplicial-complex-2
  (implies (and (simplex-p s1)
                (simplex-p s2)
                (list-of-simplexes-p ls)
                (member-equal s1 (simplicial-complex-generator ls))
                (subsetp-equal s2 s1))
           (member-equal s2 (simplicial-complex-generator ls))))
```

Once we have proved these two theorems we can claim that the `simplicial-complex-generator` function constructs an abstract simplicial complex when a list of simplexes is provided as argument.

In addition, we need to prove that the simplicial complex constructed by the `simplicial-complex-generator` function is the one that we are looking for. This means that we need to prove the following lemma.

ACL2 Lemma 5.13. Let `ls` be a list of simplexes and let `s` be an element of the simplicial complex constructed with the `simplicial-complex-generator` function taking as argument `ls`; then, `s` is a face of some of the simplexes of `ls`.

```
(defthm simplicial-complex-generator-correctness
  (implies (and (list-of-simplexes-p ls)
                (member-equal s (simplicial-complex-generator ls))
           (face-of-some-p s ls)))
```

The proof of the above lemmas, in spite of involving some auxiliary results, can be proved without any special hindrance due to the fact that, as we said previously, our programs follow simple inductive schemas that are suitable for the ACL2 heuristics. Then, we have the following theorem.

ACL2 Theorem 5.14. Let *ls* be a list of simplexes, then (`simplicial-complex-generator` *ls*) constructs the simplicial complex associated with *ls*.

The interested reader can consult the complete development in [Her11].

5.1.5.2 Two equivalent programs

The implementation of the `simplicial-complex-generator` function is suitable for the induction heuristics of the ACL2 theorem prover. However, it is an inefficient design, so, it can produce undesirable situations. For instance, if we try to build a simplicial complex from a list of 11613 simplexes, an error message will be shown:

```
> (simplicial-complex-generator ...) ✠
Error: Stack overflow (signal 1000)
[condition type: SYNCHRONOUS-OPERATING-SYSTEM-SIGNAL]
```

This kind of error occurs when too much memory is used on the data structure that stores information about the active computer program.

In order to overcome this drawback, an efficient algorithm called `optimized-simplicial-complex-generator` has been implemented. This new program relies on the *memoization* technique. Let us remember that memoization is used primarily to speed up computer programs. A memoized function "remembers" the results corresponding to some set of specific inputs. Subsequent calls with remembered inputs return the remembered result rather than recalculating it.

However, the `optimized-simplicial-complex-generator` program can not be implemented in ACL2 (remember that ACL2 is an applicative subset of Common Lisp). In order to deal with this pitfall we have based on the work presented in [ALR07], where the authors coped with a similar problem, but related to already implemented Kenzo code fragments.

Let us enumerate the characteristics of our situation:

- `simplicial-complex-generator` program is
  - specially designed to be proved;
  - programmed in ACL2 (and, of course, Common Lisp);
  - not efficient;
  - tested;
  - proved in ACL2.
- `optimized-simplicial-complex-generator` program is
  - specially designed to be efficient;
  - written in Common Lisp;
  - efficient;
  - tested;
  - unproved.

In our approach, `simplicial-complex-generator` is *supposed to be equivalent* to `optimized-simplicial-complex-generator`. But we do not pretend to prove this equivalence: this option would lead us to a form of ill-founded recursion. Our aim should be to use the *highly reliable* `simplicial-complex-generator` to perform automated testing of the *efficient* `optimized-simplicial-complex-generator`.

The following toy program will illustrate this idea:

```
(defun automated-testing ()
  (let ((cases (generate-test-cases 100000)))
    (dolist (case cases)
      (if (not (equal-as-sc (simplicial-complex-generator case)
                     (optimized-simplicial-complex-generator case)))
          (report-on-failure case)))))
```

With this intensive testing, it is hoped that `simplicial-complex-generator` accurately models `optimized-simplicial-complex-generator`, and then our strategy could be safely applied.

A really interesting work, in the same line, was presented [G$^+$08], where a method to permit the user of a mathematical logic to write elegant logical definitions while allowing sound and efficient execution was described. Those features afford dual applications: on the one hand, formal proof; on the other hand, execution. In particular, they allow the user to install, in a logically sound way, alternative executable counterparts for logically-defined functions. These alternatives are often much more efficient than the logically equivalent terms they replace. Unfortunately, in order to use the tool presented in [G$^+$08] both programs must be developed in ACL2, which is not our case.

5.2 Applications of simplicial complexes: Digital Images

Algebraic Topology is a complex and abstract mathematical subject; however, some of its techniques can be applied to different contexts such as coding theory [Woo89], data analysis [Her03], robotics [Mac03] or digital image analysis [GDMRSP05, GDR05] (in this last case, in particular in the study of medical images [SGF03]).

Here, we are going to focus on the application of Algebraic Topology to the study of binary digital images. In the Algebraic Topology framework, binary digital images can be studied using simplicial complexes. This section is devoted to present a technique based on simplicial complexes to study binary digital images. In particular, we study monochromatic (black and white) digital images by means of the algorithms related to simplicial complexes explained in Subsection 5.1.1 and new algorithms explained later.

Explanations about how we use simplicial complexes to study digital images are given in the rest of this section. Subsection 5.2.1 presents the technique and the algorithms that we apply to analyze digital images by means of simplicial complexes. The algorithms presented in Subsection 5.2.1 are implemented as an enhancement of the simplicial complex Kenzo module (presented in Subsection 5.1.2) in Subsection 5.2.2 for the cases of 2D and 3D digital images. The way of widening our framework and the *fKenzo* GUI to include digital images is explained in Subsection 5.2.3 and 5.2.4 respectively.

Finally, some remarks about the formalization of our algorithms for digital images are provided in Subsection 5.2.5.

5.2.1 The framework to study digital images

Let n be any positive integer. An *n-xel* q in an Euclidean n-space, $\mathbb{R}^n$, is a closed unit n-dimensional (hyper)cube $q \subset \mathbb{R}^n$ whose 2^n vertices have natural coordinates (more precisely, an *n-xel* in $\mathbb{R}^n$ is a cartesian product like $[i_1, i_1+1] \times [i_2, i_2+1] \times \ldots \times [i_n, i_n+1]$). In this memoir, a *pixel* is a 2-xel in $\mathbb{R}^2$. We define an *n-dimensional binary image* or *nD-image*, to be a finite set of n-xels in $\mathbb{R}^n$.

An nD-image $\mathcal{I}$ can, of course, be represented by a finite n-dimensional array of 1's and 0's in which each 1 represents an n-xel in $\mathcal{D}$ and each 0 represents an n-xel that is not in $\mathcal{D}$. Let us focus on the study of nD-images by means of simplicial complexes. Firstly, we present the study for the cases of 2D-images and eventually the general case.

As we have just said, a 2D-image $\mathcal{D}$ can be represented by a finite 2-dimensional array of 1's and 0's in which each 1 represents a pixel in $\mathcal{D}$ and each 0 represents a pixel that is not in $\mathcal{D}$ (in a monochromatic 2D-image $\mathcal{D}$, black pixels are represented by 1's, on the contrary white pixels are represented by 0's).

Let $\mathcal{D}$ be a 2D-image codified as a 2-dimensional array of 1's and 0's. We want to associate a simplicial complex with $\mathcal{D}$. It is worth noting that the most natural and efficient approach to study digital images by topological means consists in using *cubical complexes*, see [KMM04]. However we have preferred to analyse digital images through simplicial complexes, because we can reuse the certified simplicial complex Kenzo module presented in Section 5.1.

From a digital image, there are several ways of constructing a simplicial complex (see [ADFQ03]). The approach that we have followed here consists in obtaining from $\mathcal{D}$ the facets of one of its associated simplicial complexes. Subsequently, applying Algorithm 5.6 (the algorithm which constructs a simplicial complex from a sequence of simplexes), we obtain a simplicial complex associated with $\mathcal{D}$.

The process that we have followed to obtain the facets from a 2D-image $\mathcal{D}$ is as follows. Let $V = (\mathbb{N}, \mathbb{N})$ be the vertex set, that is, a vertex, in this case, is a pair of natural numbers. Let $p = (a, b)$ be the coordinates of a pixel in $\mathcal{D}$ (that is, the position of the pixel in the 2-dimensional array associated with $\mathcal{D}$). From p we can obtain two 2-simplexes that are two facets of the simplicial complex associated with $\mathcal{D}$. Namely, from $p = (a, b)$ we obtain the following facets: the triangles $((a, b), (a+1, b), (a+1, b+1))$ and $((a, b), (a, b+1), (a+1, b+1))$. If we repeat the process for the coordinates of all the pixels in $\mathcal{D}$, we obtain the facets of a simplicial complex associated with $\mathcal{D}$, that will be denoted by $\mathcal{K}_{2D}(\mathcal{D})$.

Therefore, we can define the following algorithm.

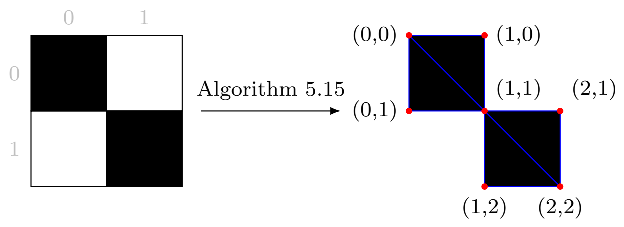


Figure 5.8: On the left, a digital image; on the right, its simplicial complex representation

Algorithm 5.15.
Input: a 2D-image $\mathcal{D}$ represented by means of a 2-dimensional array of 1's and 0's.
Output: the facets of $\mathcal{K}_{2D}(\mathcal{D})$, a simplicial complex associated with $\mathcal{D}$.

Example 5.16. Consider the 2D-image depicted in the left side of Figure 5.8. This image can be codified by means of the 2-dimensional array: $((1,0),(0,1))$, then, the coordinates of the black pixels are $(0,0)$ and $(1,1)$. Therefore, applying Algorithm 5.15 we obtain the facets of $\mathcal{K}_{2D}(\mathcal{D})$:

$$(((0,0),(0,1),(1,1)),((0,0),(1,0),(1,1)),((1,1),(1,2),(2,2)),((1,1),(2,1),(2,2))).$$

Once we have the simplicial complex associated with the digital image, we can compute the homology groups of the image from the simplicial complex. As we said previously, several simplicial complexes can be associated with a digital image, but all of them are homeomorphic (see [ADFQ03]); then, we can define the homology groups of a 2D-image as follows:

Definition 5.17. Given a 2D-image $\mathcal{D}$, the *n-homology group* of $\mathcal{D}$, $H_n(\mathcal{D})$ is the n-homology group of the simplicial complex $\mathcal{K}_{2D}(\mathcal{D})$:

$$H_n(\mathcal{D}) = H_n(\mathcal{K}_{2D}(\mathcal{D})).$$

Subsequently, we can interpret properties about the digital image from its homology groups. 2D-images are embedded in $\mathbb{R}^2$ then its homology groups vanish for dimensions greater than 2 and they are torsion-free from dimensions 0 to dimension 1; that is, their homology groups are either null or a direct sum of $\mathbb{Z}$ components in dimensions 0 and 1. The number of $\mathbb{Z}$ components of the homology groups of dimension 0 and 1 measures respectively the number of connected components and the number of holes of the image.

The method presented here for 2D-images can be generalized to nD-images with $n \geq 2$. An nD-image can be represented by a finite n-dimensional array of 1's and 0's in which each 1 represents an n-xel in $\mathcal{D}$ and each 0 represents an n-xel that is not in $\mathcal{D}$.

Let $\mathcal{D}$ be an nD-image, from the coordinates of each n-xel in $\mathcal{D}$ (its position in the n-dimensional array associated with $\mathcal{D}$), we can obtain a triangulation by means of n-simplexes, see [OS03], which are facets of a simplicial complex associated with $\mathcal{D}$. If we repeat the process for the coordinates of all the n-xels in $\mathcal{D}$, we obtain the facets of a simplicial complex associated with $\mathcal{D}$. Then, applying Algorithm 5.6, we can obtain the simplicial complex associated with $\mathcal{D}$. Therefore, the two following algorithms can be defined.

Algorithm 5.18.
Input: the coordinates of an n-xel.
Output: a triangulation of the n-xel by means of n-simplexes.

Algorithm 5.19.
Input: an nD-image $\mathcal{D}$ represented by means of a n-dimensional array of 1's and 0's.
Output: the facets of $\mathcal{K}_{nD}(\mathcal{D})$, a simplicial complex associated with $\mathcal{D}$.

It is worth noting that these two last algorithms are not implemented for the general case, just for 2D-images and 3D-images.

5.2.2 Enhancing the simplicial complex Kenzo module

Algorithms 5.18 and 5.19 explained in Subsection 5.2.1 have been implemented for the cases $n = 2, 3$ as an enhancement for the simplicial complex Kenzo module explained in Subsection 5.1.2. The set of programs that we have developed (with about 600 lines) allows the construction of the facets of the simplicial complexes $\mathcal{K}_{2D}(\mathcal{D}_1)$ and $\mathcal{K}_{3D}(\mathcal{D}_2)$ from a 2D-image, $\mathcal{D}_1$, and a 3D-image, $\mathcal{D}_2$. Moreover, thanks to the simplicial complex module and the Kenzo kernel, we can construct the simplicial complex defined by the list of facets, the simplicial set associated with that simplicial complex, and, finally, compute the homology groups of the simplicial set obtaining properties of the original image.

The rest of this subsection is devoted to present the essential part of the programs which implement algorithms 5.18 and 5.19 for the cases $n = 2, 3$, describing the functions with the same format as in the Kenzo documentation [DRSS98]. Moreover, we will see some examples of the use of these programs.

We have written several functions that allow us to cope with 2D-images and 3D-images; but most of them are auxiliary functions; so we just describe the main ones:

generate-facets-image-2d 2da *[Function]*

This function takes as argument the 2-dimensional array (a 2-dimensional array is represented by means of a list of lists) of 1's and 0's, `2da`, which determines a 2D-image and returns the list of facets of the simplicial complex $\mathcal{K}_{2D}$ associated with the 2D-image (Algorithm 5.15). The facets are returned as a list of simplexes over $V = (\mathbb{N}, \mathbb{N})$.

generate-facets-image-3d 3da *[Function]*

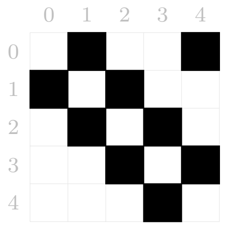


Figure 5.9: Small 2D-image

This function takes as argument the 3-dimensional array (a 3-dimensional array is represented by means of a list of lists of lists) of 1's and 0's, `3da`, which determines a 3D-image and returns the list of facets of the simplicial complex $\mathcal{K}_{3D}$ associated with the 3D-image (Algorithm 5.19 case $n = 3$). The facets are returned as a list of simplexes over $V = (\mathbb{N}, \mathbb{N}, \mathbb{N})$.

transform-lolol-to-lol lolol *[Function]*

This function transforms a list of simplexes, `lolol`, over $V = (\mathbb{N}, \mathbb{N})$ or $V = (\mathbb{N}, \mathbb{N}, \mathbb{N})$ to a list of simplexes over $V = \mathbb{N}$. This function is necessary because the programs related to simplicial complexes presented in Subsection 5.1.2 work with simplexes over $V = \mathbb{N}$ and the two above functions return list of simplexes over $V = (\mathbb{N}, \mathbb{N})$ and $V = (\mathbb{N}, \mathbb{N}, \mathbb{N})$ respectively. Then, we need a transformation between the format of the output of the functions `generate-facets-image-2d` and `generate-facets-image-3d` to the format of the input of the `simplicial-complex-generator` function (defined in Subsection 5.1.2). The `transform-lolol-to-lol` function assigns a unique natural number to each vertex over $V = (\mathbb{N}, \mathbb{N})$ or $V = (\mathbb{N}, \mathbb{N}, \mathbb{N})$ of the list `lolol` (this function is a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ and also from $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$). The natural numbers are assigned based on the lexicographical order of the lists of simplexes `lolol`. For instance, from the list of simplexes $(((0,0),(0,1),(1,1)),((0,0),(1,0),(1,1)))$ over $V = (\mathbb{N}, \mathbb{N})$ (where we have the following ordination of the simplexes based on the lexicographical order: $(0,0) < (0,1) < (1,0) < (1,1)$) we obtain the list of simplexes $((0,1,3),(0,2,3))$ over $V = \mathbb{N}$.

To provide a better understanding of the new tools, some examples of their use are presented. Let us consider the 2D-image depicted in Figure 5.9. This image can be represented by the following 2-dimensional array (a list of lists) which is assigned to the variable `small-2d-image`:

```
> (setf small-2d-image '((0 1 0 0 1) (1 0 1 0 0) (0 1 0 1 0) (0 0 1 0 1) (0 0 0 1 0)))
✠
((0 1 0 0 1) (1 0 1 0 0) (0 1 0 1 0) (0 0 1 0 1) (0 0 0 1 0))
```

From the 2-dimensional array of the image we can generate the list of facets over $V = (\mathbb{N}, \mathbb{N})$ of $\mathcal{K}_{2D}$(`small-2d-image`):

```
> (setf facets-image (generate-facets-image-2d small-2d-image)) ✠
(((1 0) (2 0) (2 1)) ((1 0) (1 1) (2 1)) ((4 0) (5 0) (5 1)) ((4 0) (4 1) (5 1))
 ((0 1) (1 1) (1 2)) ((0 1) (0 2) (1 2)) ((2 1) (3 1) (3 2)) ((2 1) (2 2) (3 2))
 ((1 2) (2 2) (2 3)) ((1 2) (1 3) (2 3)) ...)
```

Now, we can transform these facets to the suitable format for the `simplicial-complex-generator` function:

```
> (setf facets-image-nat (transform-lolol-to-lol facets-image)) ✠
((0 3 4) (0 1 4) (2 6 7) (2 3 7) (4 8 9) (4 5 9) (7 11 12) (7 8 12) (9 13 14)
 (9 10 14) ...)
```

Then, from these facets we can construct the associated simplicial complex:

```
> (setf image-2d-sc (simplicial-complex-generator facets-image-nat)) ✠
((0 3 4) (3 4) (0 4) (0 3) (4) (3) (0) (0 1 4) (1 4) (0 1) ...)
```

Subsequently, the simplicial set canonically associated with the simplicial complex `image-2d-sc` can be built:

```
> (setf image-2d-ss (ss-from-sc image-2d-sc)) ✠
[K1 Simplicial Set]
```

We obtain as result a `Simplicial-Set` object. Then, we can ask for its homology groups.

```
> (homology image-2d-ss 0 2) ✠
Homology in dimension 0:
Component Z
Component Z
Homology in dimension 1:
Component Z
Component Z
Component Z
```

The results must be interpreted as stating that the image of Figure 5.9 has 2 connected components and 3 holes.

Analogously we can consider an example of a 3D-image, namely, the one depicted in Figure 5.10. This image can be represented by the following 3-dimensional array (a list of lists of lists) which is assigned to the variable `small-3d-image`:

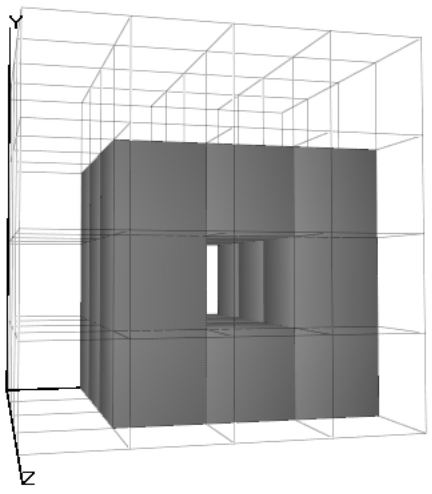


Figure 5.10: 3D cube with a tunnel

```
> (setf small-3d-image '(((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0))
                         ((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0))
                         ((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0))
                         ((0 0 0 0) (0 0 0 0) (0 0 0 0) (0 0 0 0)))) ✠
(((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0)) ((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0))
 ((0 1 1 1) (0 1 0 1) (0 1 1 1) (0 0 0 0)) ((0 0 0 0) (0 0 0 0) (0 0 0 0) (0 0 0 0))))
```

From the 3-dimensional array of an image we can generate the list of facets over $V = (\mathbb{N}, \mathbb{N}, \mathbb{N})$ of $\mathcal{K}_{3D}$(`small-3d-image`):

```
> (setf facets-image-3d (generate-facets-image-3d small-3d-image)) ✠
(((1 0 0) (2 0 0) (2 0 1) (2 1 1)) ((1 0 0) (2 0 0) (2 1 0) (2 1 1))
 ((1 0 0) (1 0 1) (2 0 1) (2 1 1)) ((1 0 0) (1 0 1) (1 1 1) (2 1 1))
 ((1 0 0) (1 1 0) (2 1 0) (2 1 1)) ((1 0 0) (1 1 0) (1 1 1) (2 1 1))
 ((2 0 0) (3 0 0) (3 0 1) (3 1 1)) ((2 0 0) (3 0 0) (3 1 0) (3 1 1))
 ((2 0 0) (2 0 1) (3 0 1) (3 1 1)) ((2 0 0) (2 0 1) (2 1 1) (3 1 1)) ...)
```

Now, we can transform these facets to the suitable format for the `simplicial-complex-generator` function:

```
> (setf facets-image-3d-nat (transform-lolol-to-lol facets-image-3d)) ✠
((0 16 17 21) (0 16 20 21) (0 1 17 21) (0 1 5 21) (0 4 20 21) (0 4 5 21) (16 32 33 37)
 (16 32 36 37) (16 17 33 37) (16 17 21 37) ...)
```

Then, from these facets we can construct the associated simplicial complex:

```
> (setf image-3d-sc (simplicial-complex-generator facets-image-3d-nat)) ✠
((0 16 17 21) (16 17 21) (0 17 21) (0 16 21) (0 16 17) (17 21) (16 21) (16 17)
 (0 21) ...)
```

Subsequently, the simplicial set canonically associated with the simplicial complex `image-3d-sc` can be built:

```
> (setf image-3d-ss (ss-from-sc image-3d-sc)) ✠
[K10 Simplicial Set]
```

We obtain as result a `Simplicial-Set` object. Then, we can ask for its homology groups.

```
> (homology image-3d-ss 0 3) ✠
Homology in dimension 0:
Component Z
Homology in dimension 1:
Component Z
Homology in dimension 2:

```

The results must be interpreted as stating that the image of Figure 5.10 has 1 connected component, 1 tunnel and 0 cavities.

Up to now, we have presented the programs that, from the 2-dimensional array associated with a 2D-image or the 3-dimensional array associated with a 3D-image, obtain the facets of a simplicial complex associated with them. However, few common 2D-image formats (such as "*jpeg*", "*bmp*", "*png*", "*pbm*" and so on) or 3D-image formats (such as "*byu*", "*jvx*", "*obj*" and so on) encode images as 2-dimensional or 3-dimensional arrays respectively.

Namely, in the case of 2D-images, the only format which codifies a image as a 2 dimensional array is *pbm*. Then, we have implemented, as a Common Lisp program, an interpreter which converts from an image in the *pbm* format to the 2-dimensional array in the format of our programs (that is to say, a list of lists).

The *pbm* images that our program can process are codified in the following way:

- The two characters "P1" which indicate that we work with plain *pbm* files (there are other kinds of *pbm* files; but we only deal with the simpler one).
- The width (w) in pixels of the image and the height (h) in pixels of the image, formatted as ASCII characters in decimal.
- A *raster* of h rows, in order from top to bottom. Each row has w bits. Each bit represents a pixel: 1 is black, 0 is white. The order of the pixels is left to right.

For instance, the image depicted in Figure 5.9 is codified in a *pbm* file as follows.

```
P1
5 5
01001
10100
01010
00101
00010
```

The above code must be read as follows. We have a plain *pbm* file which has a raster of five rows and five columns of pixels. The pixels of the coordinates $(0, 1)$, $(1, 0)$, $(1, 2)$, $(2, 1)$, $(2, 3)$, $(3, 2)$, $(3, 4)$, $(4, 0)$ and $(4, 3)$ are black and the rest of pixels are white.

From a *pbm* file we can easily obtain the 2-dimensional array of the image in the suitable format for our programs. Then, we have the following function.

pbm-to-loc-path pbm-path *[Function]*

This function takes as argument a *pbm* image stored in the path `pbm-path`, and returns the 2-dimensional array of the image as a list of lists of 1's and 0's.

To provide a better understanding of this new function, an example of its use is presented. Let us consider a *pbm* file called `simple.pbm` that codifies the image depicted in Figure 5.9 as was presented previously.

From that file, we can obtain the 2-dimensional array of the image in the desirable format for our program as follows:

```
> (setf small-2d-image (pbm-to-loc-path "simple.pbm")) ✠
((0 1 0 0 1) (1 0 1 0 0) (0 1 0 1 0) (0 0 1 0 1) (0 0 0 1 0))
```

Now, we can proceed as in the previous examples. Then, to process 2D-images codified in different image formats with our programs, we only need to transform the original codification to the *pbm* format (a task that can be performed with different tools, for instance GIMP [Pec08] allows us to convert an image stored in, practically, any format to the *pbm* one).

To sum up, an interpreter for 2D-images stored in "*pbm*" format has been developed. However, in the case of 3D-images, we have not found any 3D-image format which codifies a 3D-image by means of a 3-dimensional array; and, at this moment, we have not developed an interpreter which converts from 3D-image formats to the suitable input format of our programs; that is, the 3-dimensional array associated with an image. This task remains as further work.

Nevertheless, we have developed and interpreter which converts from a 3-dimensional array, codified as a list of lists of lists, to the "*obj*" 3D format [Tec] in order to be able to show our images in usual 3D-image renders. The information stored in the files which use this format consists of the position of the tetrahedrons of the image; since it works with tetrahedrons instead of working with voxels. Then, we have developed a Common Lisp program which from the 3-dimensional array of a 3D-image returns the 3D-image codified in this format. So, if we save the generated data in an "*obj*" file, we can visualize the image in a 3D viewer.

3da-to-obj 3da *[Function]*

This function takes as argument a 3-dimensional array, codified as a list of lists of lists, of a 3D-image, and returns the image codified in the "*obj*" format.

For instance, if we invoke the `3da-to-obj` function with the 3-dimensional array declared in `small-3d-image` as argument; the content of the file which allows us to show the image of Figure 5.10 is generated.

```
> (3da-to-obj small-3d-image) ✠
v 0 1 1
v 0 0 1
v 1 0 1
...
f 1 2 3 4
f 8 7 6 5
...
```

To sum up, we have enhanced the Kenzo system with a new module which allows us to study monochromatic 2D-images and 3D-images.

5.2.3 Digital images in our framework

We have enhanced our framework with the functionality related to digital images by means of a new plug-in. This plug-in will allow us to construct the simplicial set canonically associated with a simplicial complex defined by means of a list of facets extracted from the 2-dimensional array of a 2D-image (applying algorithms 5.19 (case $n = 2$), 5.6 and 5.9) and also the simplicial set canonically associated with a simplicial complex defined by means of a list of facets extracted from the 3-dimensional array of a 3D-image (applying algorithms 5.19 (case $n = 3$), 5.6 and 5.9).

This plug-in, which includes the functionality about digital images in our framework, references the following resources which are used by the plug-in framework to extend the functionality of all the components.

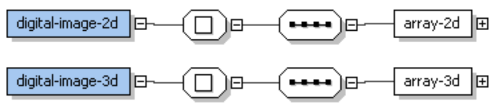


Figure 5.11: Digital images elements in XML-Kenzo

```
<code id="digital-images">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> simplicial-complexes.lisp </data>
   <data format="Kf/internal-server"> digital-images.lisp </data>
   <data format="Kf/microkernel"> digital-images-m.lisp </data>
   <data format="Kf/adapter"> digital-images-a.lisp </data>
</code>
```

Let us explain each one of the referenced resources. As we have said previously, the `XML-Kenzo.xsd` file specifies the XML-Kenzo language. We want to introduce two new kind of objects in our system (the simplicial sets associated with simplicial complexes generated from the sequence of facets of 2D-images and 3D-images); then, it is necessary to provide a representation for those objects in our framework.

In the XML-Kenzo specification, we have defined two new simple types: `array-2d`, which represents a 2-dimensional array of 1's and 0's that will be used to provide the 2-dimensional array of a 2D-image, and `array-3d`, which represents a 3-dimensional array of 1's and 0's that will be used to provide the 3-dimensional array of a 3D-image. In addition, two new elements have been defined: `digital-image-2d`, whose value is an element of the type `array-2d`; and, `digital-image-3d`, whose value is an element of the type `array-3d` (see Figure 5.11). Both `digital-image-2d` and `digital-image-3d` elements are defined as elements of the `SS` type because they are used to represent the simplicial sets associated with the simplicial complexes defined from 2D-images and 3D-images. Due to the fact that they are defined as elements of the `SS` type they can be used as any other element of this type.

When upgrading the `XML-Kenzo.xsd` the external server evolves and is able to check the constraints against the XML-Kenzo specification of requests related to digital images such as:

```
<constructor>
    <digital-image-2d>
        <array-2d>
           <line> 1 0 </line>
           <line> 0 1 </line>
        </array-2d>
    </digital-image-2d>
</constructor>
```

The `simplicial-complexes.lisp` file includes the Kenzo functionality about simplicial complexes presented in Subsection 5.1.2, since if we want to use the functionality about digital images in Kenzo, it is necessary to load the functionality about simplicial complexes.

The `digital-images.lisp` file includes the functionality explained in Subsection 5.2.2 to extend the Kenzo system. Moreover, this file includes the functionality that enhances the `xml-kenzo-to-kenzo` function of the internal server to process requests devoted to the construction of objects from the elements `digital-image-2d` and `digital-image-3d`. For instance, if the internal server receives the above XML-Kenzo request. The instruction

```
(ss-from-sc (simplicial-complex-generator (transform-lolol-to-lol
  (generate-facets-image-2d '((1 0) (0 1)))))
```

is executed in the Kenzo kernel. As a result an object of the `Simplicial-Set` Kenzo class is constructed, and the identifier of that object is returned.

The `digital-images-m.lisp` file defines two new construction modules for the microkernel called `digital-images-2d` and `digital-images-3d`. Each one of these modules implements a procedure following the guidelines explained in Subsubsection 2.2.3.3 to construct respectively the simplicial sets associated with 2D-images and 3D-images in the microkernel. Neither of the new modules have to check if the XML-Kenzo requests related to digital images are correct, since all the restrictions about these requests are controlled in the external server thanks to the XML-Kenzo specification (this is due to the fact that we only have the restriction of being an array of 1's and 0's, and this is an independent argument restriction).

Finally, we have extended the `SS` Content Dictionary by means of the definition of two new objects: `digital-images-2d` and `digital-images-3d`. Therefore, the `digital-images-a.lisp` file contains the necessary functions to raise the functionality of the adapter to be able to convert from the new OpenMath requests, devoted to digital images, to XML-Kenzo requests. Namely, we have extended the Phrasebook by means of a new parser in charge of this task. Then, for instance, the previous XML-Kenzo request is generated by the Adapter when the following OpenMath request is received.

```
<OMOBJ>
   <OMA>
      <OMS cd="SS" name="digital-image-2d"/>
      <OMA>
        <OMS name="array-2d"/>
        <OMA> <OMS name="line"/> <OMI>1</OMI><OMI>0</OMI> </OMA>
        <OMA> <OMS name="line"/> <OMI>0</OMI><OMI>1</OMI> </OMA>
      </OMA>
   </OMA>
</OMOBJ>
```

5.2.4 Digital images in the *fKenzo* GUI

This subsection is devoted to present the necessary resources to extend the *fKenzo* GUI to support the interaction with digital images. We have defined a fresh *fKenzo* module to enhance the GUI with support for digital images. The new OMDoc module references three files: `digital-images-structure` (that defines the structure of the graphical constituents), `digital-images-functionality` (which provides the functionality related to the graphical constituents) and the plug-in introduced in the previous subsection.

We have defined six graphical elements, using the XUL specification language, in the `digital-images-structure` file:

- A menu called `Digital Images` which contains two options: `Load Digital Image 2D` and `Load Digital Image 3D`.
- A selection window called `select-2D-image`.
- A selection window called `select-3D-image`.
- A window called `show-2D-image` with an image viewer as sole component.
- A window called `show-3D-image` with a browser as sole component.
- A window called `SS-from-DI-name`.

The functionality stored in the `digital-images-functionality` document related to these components works as follows. A function acting as event handler is associated with the `Load Digital Image 2D` menu option; this function shows the `select-2D-image` window which allows the user to choose an image from a *pbm* file. Currently, a folder, called *Digital Images Examples*, with several examples of *pbm* files is included in the distribution of *fKenzo*.

Once the user has selected a *pbm* image, the system checks if the file is a *pbm* file which can be processed in our system; otherwise it informs the user with a warning message. Subsequently, if the file is valid, it invokes our framework with a `digital-image-2d` OpenMath request with the information obtained from the selected file through the `pbm-to-loc-path` function, which is also included in the `digital-images-functionality` document. Subsequently, the simplicial set associated with the simplicial complex defined from the list of simplexes obtained from the 2-dimensional array of the image is constructed and an identification number is returned. Afterwards, the system asks a name for the new simplicial set by means of the window `SS-from-DI-name`; if the name given by the user is correct (was not used previously), the system stores it, otherwise it indicates that the name was previously used and asks again a name. Eventually, if the result returned by the was an identification number; the system adds to the list of constructed spaces (situated in the left side of the main tab of *fKenzo*) the new object.

Moreover, the `digital-images-functionality` file increases the behavior of the event handler associated with the left list of the *fKenzo* GUI. As we explained in Subsection 3.2.1 when a space is selected from the list of constructed spaces, its standard notation appears at the bottom part of the right side of the *fKenzo* GUI. However, if the space selected is a simplicial set constructed from a digital image, the information that appears at the bottom part of the right side of the *fKenzo* GUI is the name given by the user. In addition, the *pbm* file is used to show the image associated with the simplicial set in the window `show-2D-image`. To show the images in the `show-2D-image` window, we use a graphical component which allows us to show 2D-images.

The case of 3D-images is analogous. In this case, examples of 3D digital images are included in the folder *Digital Images Examples 3D*; these images are stored in files with extension "3d", it is our own format to store the 3-dimensional array of an image. Moreover, in this case the image viewer of 3D-images shows the 3D-image in a browser component by means of the JavaView applet [P$^+$02]. To be able to show the 3D-images, we use the `loc-to-obj` function (explained in Subsection 5.2.2) which is included in the `digital-images-functionality` file, and which transforms the 3-dimensional array of an image to its "*obj*" codification (an image format which can be rendered by the JavaView applet).

5.2.4.1 New objects in the *fKenzo* GUI

As we have just said the behavior of the event handler associated with the left list of the *fKenzo* GUI has been modified. In Subsubsection 4.1.5.1 the management of objects in the *fKenzo* GUI and the functionality associated with the left list of the *fKenzo* GUI were explained. To handle the new behavior presented for objects associated with 2D-images and 3D-images we have included the following definitions.

As we commented in Subsubsection 4.1.5.1 we have two subclasses of the `FKENZO-OBJECT` class, `FKENZO-OBJECT-NAME` and `FKENZO-OBJECT-FILE`. Now, we have specialized the `FKENZO-OBJECT-FILE` to represent objects associated with 2D-images and 3D-images. Moreover, the new classes are also an specialization of the `FKENZO-OBJECT-NAME` class. Namely, we have defined the following two classes:

```
(DEFCLASS FKENZO-OBJECT-IMAGE-2D (FKENZO-OBJECT-NAME FKENZO-OBJECT-FILE))
```

```
(DEFCLASS FKENZO-OBJECT-IMAGE-3D (FKENZO-OBJECT-NAME FKENZO-OBJECT-FILE))
```

Both `FKENZO-OBJECT-IMAGE-2D` and `FKENZO-OBJECT-IMAGE-3D` are subclasses of both the `FKENZO-OBJECT-NAME` and `FKENZO-OBJECT-FILE` classes without adding any additional slot. An instance of the `FKENZO-OBJECT-IMAGE-2D` class is constructed for simplicial sets associated with 2D-images, the `name` slot is given by the user and the `file` slot is the path of the 2D-image. Analogously for the `FKENZO-OBJECT-IMAGE-3D`.

As we wanted a different behavior for the event handler, associated with the left list of the *fKenzo* GUI for the new objects, to the one presented before, we need some new concrete methods. In particular, we have defined the following two methods:

```
(DEFMETHOD show-object ((object FKENZO-OBJECT-IMAGE-2D))
 (show-image-2d (file object))
 (call-next-method))
```

```
(DEFMETHOD show-object ((object FKENZO-OBJECT-IMAGE-3D))
 (show-image-3d (file object))
 (call-next-method))
```

The method associated with the objects of the `FKENZO-OBJECT-IMAGE-2D` class shows the window with the 2D-image stored in the path indicated by the `file` slot; subsequently, it calls the method associated with the `FKENZO-OBJECT-NAME` class; and therefore the *fKenzo* GUI shows the name of the object at the bottom part of the right side of the *fKenzo* GUI. Analogously for the objects of the `FKENZO-OBJECT-IMAGE-3D` but showing the 3D-image stored in the path indicated by the `file` slot after the conversion to the *obj* format.

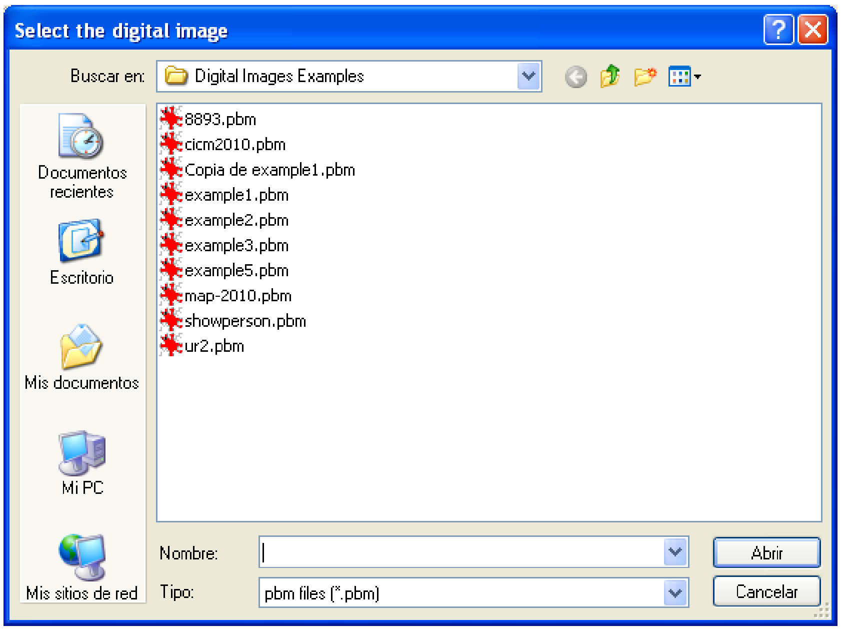
The `call-next-method` function of these methods allow them to refer to a less specific method, namely the one associated with the `FKENZO-OBJECT-NAME` class. This means that if we have selected an object in the left list of the *fKenzo* GUI that corresponds with a `FKENZO-OBJECT-IMAGE-2D` or a `FKENZO-OBJECT-IMAGE-3D` object, the GUI shows the name of the object at the bottom part of the right side of the *fKenzo* GUI.

In this way, the behavior of the left list of the GUI is enhanced without touching the main code.

5.2.4.2 Execution flow of the *fKenzo* GUI for digital images

In order to clarify the execution flow followed by *fKenzo*, let us present two examples. When the Digital Images module is loaded in *fKenzo*, a new menu called *Digital Images*, with two options; `Load Digital Image 2D` and `Load Digital Image 3D`, becomes available. From the option `Load Digital Image 2D`, the user can select a *pbm* image from the folder *Digital Images Examples* with the `select-2D-image` window, see Figure 5.12. For instance, the `8893.pbm` file is the image of a small cat.

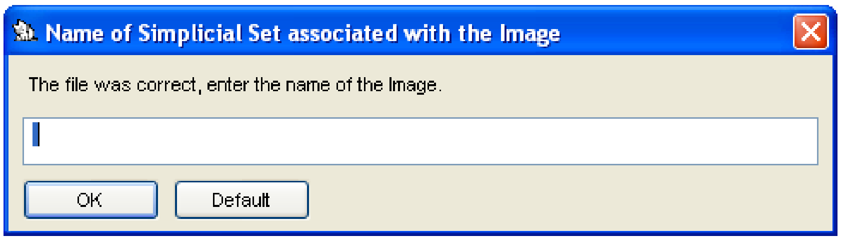
From the `8893.pbm` file and using the `pbm-to-loc` function, *fKenzo* constructs the OpenMath request which is sent to our framework. From this request the simplicial set associated with the simplicial complex defined from the facets obtained from the coordinates of black pixels of the image is built, and the identification number of the constructed object is returned. Then, *fKenzo* asks the user a name for the object by means of the window `SS-from-DI-name` (see Figure 5.13), in this case the name given is



8893.pbm
cicm2010.pbm
Copia de example1.pbm
example1.pbm
example2.pbm
example3.pbm
example5.pbm
map-2010.pbm
showperson.pbm
ur2.pbm

Nombre:
Tipo: pbm files (*.pbm)

Figure 5.12: Window to load a 2D-image



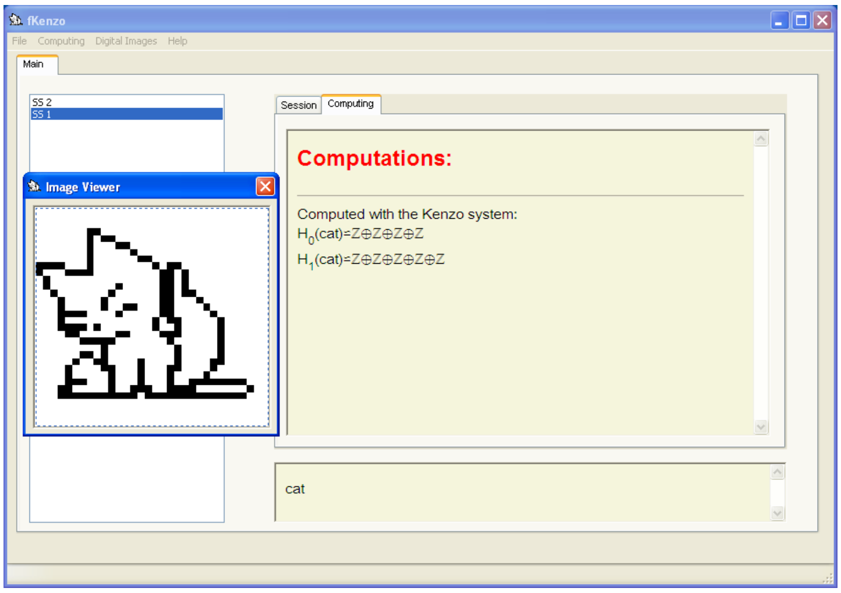
The file was correct, enter the name of the Image.

Figure 5.13: *fKenzo* asking a name for a simplicial set coming from a digital image

"cat". Subsequently, the new object is added to the list of constructed spaces shown in the left side of the *fKenzo* GUI. When the user selects this object, both the name that the user provided previously and the image are shown (see Figure 5.14).

Finally, the user can ask *fKenzo* to compute the homology groups of the image, and the results are shown, as usual, in the Computing tab. Figure 5.14 shows the computation of the homology groups of the small cat which has 4 connected components and 5 holes. These properties are interpreted from the homology groups of the image.

Analogously for the `Load Digital Image 3D` option. Figure 5.15 shows the computation of homology groups of the letters "MAP" in 3D; this image has 3 connected components, 2 tunnels and 0 cavities. As in the case of 2D-images, these properties are interpreted from the homology groups of the image.

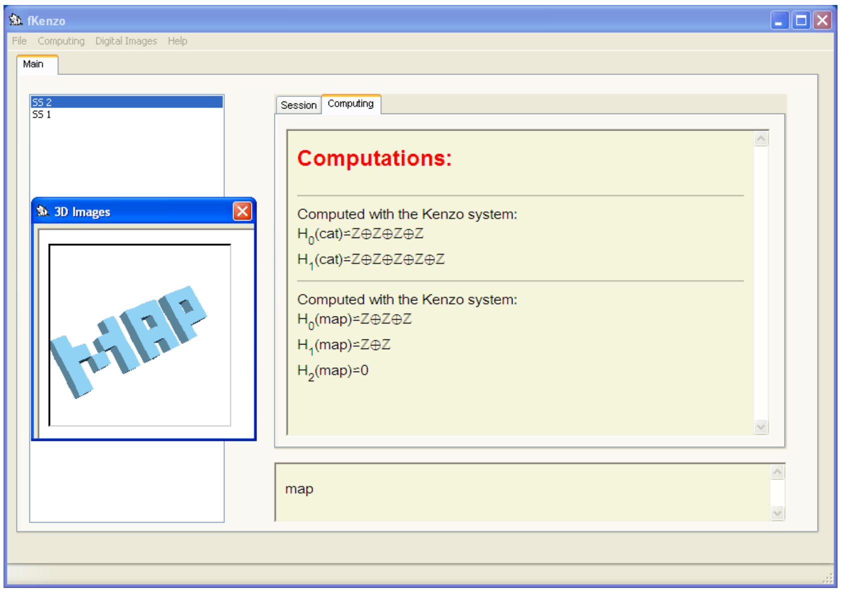


Computations:

Computed with the Kenzo system:
H_0(cat)=Z⊕Z⊕Z⊕Z
H_1(cat)=Z⊕Z⊕Z⊕Z⊕Z

cat

Figure 5.14: Homology groups of the small cat



Computations:

Computed with the Kenzo system:
H_0(cat)=Z⊕Z⊕Z⊕Z
H_1(cat)=Z⊕Z⊕Z⊕Z⊕Z

Computed with the Kenzo system:
H_0(map)=Z⊕Z⊕Z
H_1(map)=Z⊕Z
H_2(map)=0

map

Figure 5.15: Homology groups of the letters MAP in 3D

5.2.5 Formalization of Digital Images in ACL2

As we already claim, the verification of programs is an important issue, but specially in the case of the programs for digital images. If we want to use our programs in real life problems (for instance, in the study of medical images), we must be completely sure that the results produced by our programs are correct. Therefore, the formal verification of our programs with a Theorem Prover (in our case, ACL2) is significant.

This subsection is devoted to present the certification of the correctness of Algorithm 5.15 which from a 2D-image constructs a list of simplexes (which represents the facets of the simplicial complex associated with the 2D-image). The task of verifying the correctness of Algorithm 5.19 for the case $n = 3$ is analogous; then, in spite of having both developments, we just focus on the verification of the correctness of Algorithm 5.15.

5.2.5.1 Main definitions and properties

As we just said, we want to formalize in ACL2 the correctness of Algorithm 5.15; namely, our implementation of that algorithm by means of the `generate-facets-image-2d` function. Since both Kenzo and ACL2 are Common Lisp programs we can verify the correctness of that Kenzo function in ACL2.

From now on, we define the necessary functions to establish the correctness of our program. First of all, we need some auxiliary functions which define the necessary concepts to prove our theorems. These definitions are based on both Algorithm 5.15 and the notions for digital images. Namely, we need to define the notion of 2D-image.

As we said in Subsection 5.2.2, a 2D-image $\mathcal{D}$ is represented by means of a finite 2-dimensional array (that is a list of lists) of 1's and 0's where each 1 represents a pixel in $\mathcal{D}$ and each 0 represents a pixel that is not in $\mathcal{D}$. The `2d-imagep` function is a function that checks if its argument is a list of lists of 1's and 0's. This function uses the `list-0-1-p` function that checks if its argument is a list of 1's and 0's.

```
(defun list-0-1-p (list)
  (if (endp list)
      (equal list nil)
    (if (endp (cdr list))
        (and (equal (cdr list) nil)
             (or (equal (car list) 0) (equal (car list) 1)))
      (and (or (equal (car list) 0) (equal (car list) 1))
           (list-0-1-p (cdr list))))))
```

```
(defun 2d-imagep (list)
  (if (endp list)
      (equal list nil)
    (and (list-0-1-p (car list)) (2d-imagep (cdr list)))))
```

Subsequently, we define the `generate-facets-image-2d` function and all its auxiliary functions in ACL2. It is worth noting that the definition of these functions is exactly the same used in the definition of Kenzo functions (see Subsection 5.2.2). Let us show in detail these definitions.

First of all, we define the `list-up-i-j` and `list-down-i-j` functions which are used to generate from a pair of natural numbers (i, j) the simplexes $((i, j), (i+1, j), (i+1, j+1))$ and $((i, j), (i, j+1), (i+1, j+1))$ respectively.

```
(defun list-up-i-j (i j)
  (list (list i j) (list (1+ i) j) (list (1+ i) (1+ j))))
```

```
(defun list-down-i-j (i j)
  (list (list i j) (list i (1+ j)) (list (1+ i) (1+ j))))
```

From the above two functions, we can define a function, called `generate-facets-i-j` which from the pair (i, j) generates the pair of simplexes $(((i, j), (i+1, j), (i+1, j+1)), ((i, j), (i, j+1), (i+1, j+1)))$

```
(defun generate-facets-i-j (i j)
  (list (list-up-i-j i j) (list-down-i-j i j)))
```

Now, we can define the `generate-facets-image-2d` function which generates the simplexes of a list of lists of 0's and 1's `lol`.

```
(defun generate-facets-image-2d (lol)
  (generate-facets-image-aux lol 0))
```

The above function calls the more general function `generate-facets-image-aux` which takes two arguments: a list of lists of 0's and 1's `lol` and a natural number j. The function `generate-facets-image-aux` must be understood as the procedure which generates the simplexes of the list of lists of 0's and 1's `lol` which is the sublist located from position j of another list of lists of 0's and 1's, let us called it `lol-main`.

```
(defun generate-facets-image-aux (lol j)
  (if (endp lol)
      nil
     (append (generate-facets-list (car lol) 0 j)
             (generate-facets-image-aux (cdr lol) (1+ j)))))
```

For each one of the lists of 0's and 1's of `lol` and the position of that list in `lol-main`, the above function invokes the function `generate-facets-list`. The function `generate-facets-list` must be understood as the procedure which generates the simplexes of the list of 0's and 1's `list` which is the sublist located from position i of the list of position j of `lol-main`.

```
(defun generate-facets-list (list i j)
  (if (endp list)
      nil
    (if (equal (car list) 1)
        (append (generate-facets-i-j i j) (generate-facets-list (cdr list) (1+ i) j))
      (generate-facets-list (cdr list) (1+ i) j))))
```

Once we have defined our programs in ACL2 we can prove theorems about them. To be more concrete, we have proved both the correctness a the completeness of our program `generate-facets-image-2d`.

First of all we state the ACL2 theorem which ensures the completeness of `generate-facets-image-2d`.

ACL2 Theorem 5.20. Let `image` be a 2D-image represented by means of a 2 dimensional array, then, $\forall i, j \in \mathbb{N}$ such that the value of the image in position (i, j) of the array is 1, then, the simplexes $((i, j), (i+1, j), (i+1, j+1))$ and $((i, j), (i, j+1), (i+1, j+1))$ are in the list generated by the `generate-facets-image-2d` function taking as input `image`.

To state this theorem in ACL2, we need the ACL2 functions: `(natp n)`, which is a test function returning `t` if `n` is a natural number and `nil` otherwise; `(nth i ls)`, which returns the value of position `i` (a natural number) of the list `ls`; and, `(member-equal x ls)`, which returns `t` if `x` is equal to some of the elements of `ls` (a list).

```
(defthm generate-facets-image-2d-completeness
  (implies (and (2d-imagep image)
                (natp i)
                (natp j)
                (equal (nth i (nth j image)) 1))
           (and (member-equal (list-up-i-j i j) (generate-facets-image-2d image))
                (member-equal (list-down-i-j i j) (generate-facets-image-2d image)))))
```

Once we have proved the completeness of our program, we must prove its correctness. This task is handled by means of the following lemmas.

ACL2 Theorem 5.21. Let `image` be a 2D-image represented by means of a 2 dimensional array and `simplex` be an element of the output generated by `generate-facets-image-2d` taking as input `image`. Then if `simplex` is of the form $((i, j), (i + 1, j), (i + 1, j + 1))$ with i and j natural numbers, then the element $((i, j), (i, j+1), (i+1, j+1))$ is also in the output generated by `generate-facets-image-2d` taking as input `image`.

To state this theorem in ACL2 we need some auxiliary functions. Namely, `member-list-up`, which returns `t` if its input is a list of the form $((i, j), (i + 1, j), (i +$

$1, j+1))$ and `nil` otherwise; and `list-down`, which from a list of the form $((i,j),(i+1,j),(i+1,j+1))$ returns the list $((i,j),(i,j+1),(i+1,j+1))$.

```
(defthm generate-facets-image-correctness-1
  (implies (and (2d-imagep image)
                (member-equal simplex (generate-facets-image-2d image))
                (member-list-up simplex))
           (member-equal (list-down simplex) (generate-facets-image-2d image))))
```

ACL2 Theorem 5.22. Let `image` be a 2D-image represented by means of a 2 dimensional array and `simplex` be an element of the output generated by `generate-facets-image-2d` taking as input `image`. Then if `simplex` is of the form $((i,j),(i,j+1),(i+1,j+1))$ with i and j natural numbers, then the element $((i,j),(i{+}1,j),(i{+}1,j{+}1))$ is also in the output generated by `generate-facets-image-2d` taking as input `image`.

To state this theorem in ACL2 we need some auxiliary functions. Namely, `member-list-down`, which returns `t` if its input is a list of the form $((i,j),(i,j+1),(i+1,j+1))$ and `nil` otherwise; and `list-up`, which from a list of the form $((i,j),(i,j+1),(i+1,j+1))$ returns the list $((i,j),(i+1,j),(i+1,j+1))$.

```
(defthm generate-facets-image-correctness-2
  (implies (and (2d-imagep image)
                (member-equal simplex (generate-facets-image-2d image))
                (member-list-down simplex))
           (member-equal (list-up simplex) (generate-facets-image-2d image))))
```

ACL2 Theorem 5.23. Let `image` be a 2D-image represented by means of a 2 dimensional array and `simplex` be an element of the output generated by `generate-facets-image-2d` taking as input `image` of the form $((i,j),(i{+}1,j),(i{+}1,j{+}1))$ or $((i,j),(i,j+1),(i+1,j+1))$ with i and j natural numbers. Then, the element of position (i,j) of `image` is 1.

To state this theorem in ACL2 we use some ACL2 functions which have not been used previously: `caar` which returns the first element of the first element of a list and `cadar` which returns the second element of the first element of a list.

```
(defthm generate-facets-image-correctness-3
  (implies (and (2d-imagep image)
                (member-equal simplex (generate-facets-image-2d image)))
           (equal (nth (caar simplex) (nth (cadar simplex) image)) 1)))
```

Let us present some remarks about the proof of these theorems which state both the completeness and the correctness of the `generate-facets-image-2d` program.

First of all, it is worthwhile noting that the implementation of the `generate-facets-image-2d` function, and its auxiliar ones, follows simple recursive schemas, that are suitable for the induction heuristics of the ACL2 theorem prover.

Let us present now with some details two auxiliary lemmas needed in our development of the proof of the main theorems. As we have seen in the definition of `generate-facets-image-aux`, this function invokes the `generate-facets-list` function with arguments `(car list)`, `0` and `j`. Therefore, it is sensible to think that in the proof of our theorems we are going to need some auxiliary lemmas such as:

```
(thm (implies (and (list-0-1 x) (natp j)
                   (member-equal simplex (generate-facets-list x 0 j)))
              (equal (nth (caar simplex) x) 1))
```

that is to say, a lemma which involves a call to `(generate-facets-list x 0 j)`. However, ACL2 has some problems to find a proof of theorems such as the previous one, since it does not find a good inductive schema for reasoning. On the contrary, for ACL2 is much easier to find a proof of theorems such as:

```
(thm (implies (and (list-0-1 x) (natp i) (natp j)
                   (member-equal simplex (generate-facets-list x i j)))
              (equal (nth (- (caar simplex) i) x) 1))
```

that is to say, lemmas that are generalizations of the previous ones.

Taking this question into account in the development of our proofs, the certification of the completeness and correctness theorems can be done without any special trouble.

In this way, we have proved the completeness and correctness of our implementation of Algorithm 5.15 by means of the program `generate-facets-image-2d`.

In the case of 3D-digital images the development is very similar. The interested reader can consult the complete development in [Her11].

5.3 An algorithm building the pushout of simplicial sets

Many of the usual constructions in Topology are nothing but homotopy pullbacks or homotopy pushouts [Mat76]. Loop spaces, suspensions, mapping cones, wedges or joins, for instance, involve such constructions. In these cases, when the spaces are not of finite type the computation of their homology groups can be considered as a challenging task. A way of dealing with this kind of spaces consists of using the effective homology method explained in Subsection 1.1.3.

In this section we use the effective homology method to design algorithms building the pushout associated with two simplicial morphisms $f : X \to Y$ and $g : X \to Z$, where X, Y and Z are simplicial sets with *effective homology*. In addition the integration of this new tool in *fKenzo* is presented.

The rest of this section is organized as follows. Subsection 5.3.1 introduces the mathematical background about the pushout and some additional concepts about effective homology. The way of constructing the effective homology of the pushout is presented in Subsection 5.3.2. Some examples of the use of the implementation of those algorithms as a new Kenzo module are provided in Subsection 5.3.3. Subsection 5.3.4 deals with the extension of the framework to include the functionality about the pushout; and the way of widening the *fKenzo* GUI to include the pushout is detailed in Subsection 5.3.5. Finally, some remarks about the formalization of the developed algorithms are presented in Subsubsection 5.3.6.

5.3.1 Preliminaries

To explain the effective homology of the pushout is necessary to introduce the notion of pushout and also some additional notions about effective homology which were not presented in Subsection 1.1.3.

5.3.1.1 Pushout notions

First of all, let us introduce the notion of pushout. The following definitions can be found, for instance, in [Mat76, Doe98].

Definition 5.24. Consider two morphisms $f : X \to Y$, $g : X \to Z$ in a category $\mathcal{C}$. A *pushout* of (f, g) is a triple (P, f', g') where

1. P is an object of $\mathcal{C}$,
2. $f' : Y \to P$, $g' : Z \to P$ are morphism of $\mathcal{C}$ such that $f \circ g' = g \circ f'$,

and for every other triple (Q, f'', g'') where

1. Q is an object of $\mathcal{C}$,
2. $f'' : Y \to Q$, $g'' : Z \to Q$ are morphism of $\mathcal{C}$ such that $f \circ g'' = g \circ f''$,

there exists a unique morphism $p : P \to Q$ such that $f'' = p \circ f'$ and $g'' = p \circ g'$ (see the following diagram).

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, "f'"] \arrow[rdd, "f''"] & \\
Z \arrow[r, "g'"] \arrow[rrd, "g''"'] & P \arrow[rd, dashed, "p"] & \\
 & & Q
\end{tikzcd}$$

Definition 5.25. Let $f, g : X \to Y$ be two morphisms between topological spaces, then a morphism $H : X \times I \to Y$, where I is the unit interval $[0, 1]$, is a homotopy between f and g, denoted by $H : f \sim g$, if $H(x, 0) = f(x)$ and $H(x, 1) = g(x)$.

Definition 5.26. A homotopy commutative diagram:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, "f'"] \\
Z \arrow[r, "g'"] & P
\end{tikzcd}$$

equipped with $H : f' \circ f \sim g' \circ g$, is called a *homotopy pushout* when for any commutative diagram

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, "f''"] \\
Z \arrow[r, "g''"] & Q
\end{tikzcd}$$

equipped with $G : f'' \circ f \sim g'' \circ g$, the following properties hold:

1. there exists a map $p : P \to Q$ and homotopies $K : f'' \sim p \circ f'$ and $L : p \circ g' \sim g''$ such that the whole diagram

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, "f'"] \arrow[rdd, "f''"] & \\
Z \arrow[r, "g'"] \arrow[rrd, "g''"'] & P \arrow[rd, dashed, "p"] & \\
 & & Q
\end{tikzcd}$$

with all maps and homotopies above is homotopy commutative,

2. if there exists another map $p' : P \to Q$ and homotopies $K' : f'' \sim p' \circ f'$ and $L' : p' \circ g' \sim g''$ such that the diagram

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Y \arrow[d, "f'"] \arrow[ddr, "f''"] & \\
Z \arrow[r, "g'"] \arrow[drr, "g''"'] & P \arrow[dr, dashed, "p'"] & \\
 & & Q
\end{tikzcd}$$

is homotopy commutative, then there exists a homotopy $M : p \sim p'$ such that the whole diagram with all maps and homotopies above is homotopy commutative.

From now on, with an abuse of notation, we will call *(homotopy) pushout* of (f, g) to the object P.

There is a "standard" construction of the homotopy pushout of any two maps $f : X \to Y$, $g : X \to Z$ as:

$$P_{(f,g)} \cong (Y \amalg (X \times I) \amalg Z) / \sim$$

where I is the unit interval $[0, 1]$ and the equivalence relation $\sim$ is defined as follows: for every $x \in X$, $(x \times 0)$ is identified to $f(x) \in Y$ and $(x \times 1)$ is identified to $g(x) \in Z$.

5.3.1.2 Effective homology preliminaries for the pushout

Let us present now some effective homology concepts which are akin to the definition of the effective homology of the pushout. A complete study of the definitions and results presented here can be found in [RS06].

Theorem 5.27 (Direct Sum Equivalence Theorem)**.** Let C_* and D_* be two chain complexes with effective homology. Then the direct sum $C_* \oplus D_*$ is a chain complex with effective homology.

The *cone constructor* is an important construction in Homological Algebra, we present here the definition and the most elementary properties of this construction.

Definition 5.28. Let C_* and D_* be two chain complexes and $\phi : D_* \to C_*$ be a chain complex morphism. Then the cone of ϕ denoted by $Cone(\phi)$ is the chain complex $Cone(\phi) = A_*$ defined as follows. First $A_n := C_{n+1} \oplus D_n$; and the boundary operator is given by the following matrix:

$$d_{A_*} := \begin{bmatrix} d_{C_*} & \phi \\ 0 & -d_{D_*} \end{bmatrix}$$

In the previous definition, if both C_* and D_* are chain complexes with effective homology, then, the following theorems ensure that we can construct an equivalence between $Cone(\phi)$ and an effective chain complex, therefore, $Cone(\phi)$ is also a chain complex with effective homology.

Theorem 5.29 (Cone Reduction Theorem [RS06]). Let $\rho = (f, g, h) : C_* \Rrightarrow D_*$ and $\rho' = (f', g', h') : C'_* \Rrightarrow D'_*$ be two reductions and $\phi : C'_* \to C_*$ a chain complex morphism. Then these data define a canonical reduction:

$$\rho'' : Cone(\phi) \Rrightarrow Cone(f\phi g')$$

An extension of the Cone Reduction Theorem is the following result.

Theorem 5.30 (Cone Equivalence Theorem [RS06]). Let $\phi : C'_{*,EH} \to C_{*,EH}$ be a chain complex morphism between two chain complexes with effective homology. Then $Cone(\phi)$ is a chain complex with effective homology.

Definition 5.31. An *effective short exact sequence* of chain complexes is a diagram:

$$\begin{tikzcd} 0 & A_* \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & B_* \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_* \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

where i and j are chain complexes morphisms, ρ (retraction) and σ (section) are graded module morphisms satisfying:

- $\rho i = id_{C_*}$;
- $i\rho + \sigma j = id_{B_*}$;
- $j\sigma = id_{A_*}$.

It is an exact sequence in both directions, but to the left it is an exact sequence of chain complexes, and to the right it is only an exact sequence of graded modules.

The following theorem (see [RS06]) states that given an effective short exact sequence, if two of the chain complexes of the short exact sequence are chain complexes with effective homology, then, we can construct the effective homology of the third one.

Theorem 5.32 (SES Theorems). Let

$$\begin{tikzcd} 0 & A_* \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & B_* \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_* \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

be an effective short exact sequence of chain complexes. Then three general algorithms are available:

$$\begin{aligned} SES_1 &: (B_{*,EH}, C_{*,EH}) \mapsto A_{*,EH} \\ SES_2 &: (A_{*,EH}, C_{*,EH}) \mapsto B_{*,EH} \\ SES_3 &: (A_{*,EH}, B_{*,EH}) \mapsto C_{*,EH} \end{aligned}$$

producing the effective homology of one chain complex when the effective homology of both others is given.

The following two lemmas are used in the proof of the above theorem and will be important in the development of the effective homology of a pushout.

Lemma 5.33. Let

$$\begin{tikzcd} 0 & A_* \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & B_* \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_* \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

be an effective short exact sequence of chain complexes. Then the effective exact sequence produces a reduction $Cone(i) \Rightarrow\!\!\!\!\Rightarrow A_*$.

To state the second lemma we need an auxiliary definition.

Definition 5.34. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ be a chain complex. The *suspension functor* applied to C_* is the chain complex $C_*^{[1]} = (M_n, d_n)_{n\in\mathbb{Z}}$ such that, $M_n = C_{n-1}$ and $d_n = -d_{C_{n-1}}$, $\forall n \in \mathbb{Z}$.

Moreover, we can define the effective homology version of the suspension functor.

Theorem 5.35 (Suspension Functor Equivalence Theorem)**.** Let C_* be a chain complex with effective homology. Then $C_*^{[1]}$ is a chain complex with effective homology.

Now, we can provide the second lemma associated with Theorem 5.32.

Lemma 5.36. Let

$$\begin{tikzcd} 0 & A_* \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & B_* \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_* \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

be an effective short exact sequence of chain complexes. Then the effective exact sequence generates a connection chain complex morphism $\chi : A_* \to C_*^{[1]}$. Besides, B_* is canonically isomorphic to $Cone(\chi)$.

Once these concepts have been introduced, we can undertake our goal of defining the effective homology of a pushout.

5.3.2 Effective Homology of the Pushout

5.3.2.1 Main algorithms

The definitions related to the pushout given in Subsubsection 5.3.1.1 come from standard topology, from now on we switch to the simplicial framework where we can formulate the following algorithm.

Algorithm 5.37.
Input: two simplicial morphisms $f : X \to Y$ and $g : X \to Z$ where X, Y and Z are simplicial sets.
Output: the simplicial set $P_{(f,g)}$.

The above algorithm is based on the standard pushout construction presented in Subsubsection 5.3.1.1, in particular given $f : X \to Y$ and $g : X \to Z$ simplicial morphisms, then we define $P_{(f,g)}$ as the simplicial set $(Y \amalg (X \times \Delta^1) \amalg Z)/\sim$, where Δ^1 is the unit interval $[0,1]$ in the simplicial framework: the simplicial set Δ^1 has two 0-simplexes $(0),(1)$ and the non-degenerate 1-simplex $(0,1)$; and $\sim$ is the equivalence relation such that for every $x \in X$, $(x \times (0))$ is identified to $f(x) \in Y$ and $(x \times (1))$ is identified to $g(x) \in Z$.

Now, if X, Y and Z are simplicial sets with effective homology, then $P_{(f,g)}$ is also an object with effective homology; in particular we can formulate the following algorithm.

Algorithm 5.38.
Input: two morphisms $f : X \to Y$ and $g : X \to Z$ where X, Y and Z are simplicial sets with effective homology.
Output: the effective homology of $P_{(f,g)}$.

The construction of the effective homology of $P_{(f,g)}$ involves several steps. In a nutshell, the construction of the effective homology of $P_{(f,g)}$ is based on applying the case SES_2 of Theorem 5.32 to a short exact sequence which is produced by the description of C_*P (the chain complex coming from $P_{(f,g)}$):

$$\begin{tikzcd} 0 & M_* \arrow[l] \arrow[r, shift left, "\sigma"] & C_*P \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_*Y \oplus C_*Z \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

where M_* is the chain complex associated with the simplicial set $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled (that is to say, the simplexes like $(x,(0)) \in X \times (0)$, $(x,(1)) \in X \times (1)$ and their degeneracies are removed from $X \times \Delta^1$), the morphisms σ and i are inclusions and the morphisms j and ρ are projections.

To apply the case SES_2 of Theorem 5.32 to the above short exact sequence, the effective homology of M_* and $C_*X \oplus C_*Y$ must be available. Then, the main steps to construct the effective homology of the pushout $P_{(f,g)}$ are the following ones.

Step 1. From $f : X \to Y$ and $g : X \to Z$ simplicial morphisms, $P_{(f,g)}$ and its associated chain complex C_*P are constructed.

Step 2. The effective homology of M_* is constructed.

Step 3. The effective homology of $C_*X \oplus C_*Y$ is constructed.

Step 4. Eventually, from C_*P, the effective homology of M_*, the effective homology of $C_*X \oplus C_*Y$ and applying case SES_2 of Theorem 5.32 to the above short exact sequence, the effective homology of the pushout $P_{(f,g)}$ is constructed.

A complete description of the algorithm will be provided in Subsubsection 5.3.2.3.

5.3.2.2 Auxiliar algorithms

Several sub-algorithms have been required in the process to define Algorithm 5.38, in particular, we have needed the following ones.

Algorithm 5.39 (Definition 1.26)**.**
Input: two simplicial sets X and Y.
Output: the simplicial set $X \times Y$.

Algorithm 5.40 (Eilenberg-Zilber Theorem, see [May67])**.**
Input: two simplicial sets X and Y with effective homology.
Output: the effective homology of $X \times Y$.

Algorithm 5.41 (Definition 1.8)**.**
Input: two simplicial sets X and Y.
Output: the direct sum chain complex $C_*X \oplus C_*Y$ where C_*X and C_*Y are the chain complexes associated with X and Y respectively.

Algorithm 5.42 (Theorem 5.27)**.**
Input: two simplicial sets X and Y with effective homology.
Output: the effective homology of $C_*X \oplus C_*Y$.

Algorithm 5.43 (Definition 5.28)**.**
Input: a morphism i between two chain complexes A_* and B_*.
Output: the chain complex $Cone(i)$.

Algorithm 5.44 (Theorem 5.30)**.**
Input: a morphism i between two chain complexes A_* and B_* with effective homology.
Output: the effective homology of $Cone(i)$.

Algorithm 5.45.
Input: a simplicial set X.
Output: the chain complex coming from $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled.

Algorithm 5.46.
Input: a simplicial set X with effective homology.
Output: the effective homology of the simplicial set $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled.

Algorithm 5.47 (Definition 5.34)**.**
Input: a chain complex A_*.
Output: the chain complex $A_*^{[1]}$.

Algorithm 5.48 (Theorem 5.35)**.**
Input: a chain complex A_* with effective homology.
Output: the effective homology of $A_*^{[1]}$.

Algorithm 5.49 (Lemma 5.33)**.**
Input: A, B and C simplicial sets with effective homology, i, j, σ and ρ morphism which determine the short exact sequence $\begin{tikzcd} 0 & C_*A \arrow[l, "0"'] \arrow[r, shift left, "\sigma"] & C_*B \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_*C \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$ where C_*A, C_*B and C_*C are the chain complex canonically associated with A, B and C respectively.
Output: the reduction $Cone(i) \Rrightarrow C_*A$.

Algorithm 5.50 (Proposition 1.40)**.**
Input: a reduction $B_* \Rrightarrow A_*$ and the effective homology of B_*.
Output: the effective homology of A_*.

Some of these algorithms are already implemented in Kenzo (namely, algorithms 5.39, 5.40 and 5.50). On the contrary, it has been necessary to implement the rest of them.

5.3.2.3 A complete description of the algorithm

In this subsubsection, we are going to present a detailed description of the 4 main steps to construct the effective homology of the pushout of two simplicial morphisms $f : X \to Y$ and $g : X \to Z$ where X, Y and Z are simplicial sets with effective homology.

As we said previously, the construction of the effective homology of $P_{(f,g)}$ is based on applying the case SES_2 of Theorem 5.32 to the short exact sequence which is produced by the description of C_*P (the chain complex associated with $P_{(f,g)}$):

$$\begin{tikzcd} 0 & M_* \arrow[l] \arrow[r, shift left, "\sigma"] & C_*P \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_*Y \oplus C_*Z \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd}$$

where M_* is the chain complex associated with the simplicial set $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled, the morphisms σ and i are inclusions and the morphisms j and ρ are projections. To apply the case SES_2 of Theorem 5.32, the effective homology of M_* and $C_*X \oplus C_*Y$ must be available. Then, the main steps to construct the effective homology of the pushout are as follows.

Step 1. From $f : X \to Y$ and $g : X \to Z$ simplicial morphisms, $P_{(f,g)}$ and its associated chain complex C_*P are constructed.

To this aim, we simply apply Algorithm 5.37.

Step 2. We construct the effective homology of the simplicial set $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled.

The construction of the effective homology of this simplicial set is a bit tricky and needs several sub-steps that are now explained.

1. We construct the chain complex associated with $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled, that will be denoted by M_*, applying Algorithm 5.45.

   Briefly, the construction of the effective homology of M_* is obtained from the application of the case SES_1 of Theorem 5.32 to the short exact sequence produced by the description of the chain complex M_*:

$$\begin{tikzcd} 0 & M_* \arrow[l] \arrow[r, shift left, "\sigma2"] & C_*(X \times \Delta^1) \arrow[l, shift left, "j2"] \arrow[r, shift left, "\rho2"] & C_*(X \times (0)) \oplus C_*(X \times (1)) \arrow[l, shift left, "i2"] & 0 \arrow[l] \end{tikzcd}$$

   where the morphisms $\sigma2$ and $i2$ are inclusions and the morphisms $j2$ and $\rho2$ are projections. To apply the case SES_1 of Theorem 5.32, the effective homology of $C_*(X \times \Delta^1)$ and $C_*(X \times (0)) \oplus C_*(X \times (1))$ must be provided.

2. We build the effective homology of $C_*(X \times \Delta^1)$.

   Since both X and Δ^1 are simplicial sets with effective homology (Δ^1 is an effective object, then it has trivially effective homology), we can construct, applying Algorithm 5.40, the effective homology of $C_*(X \times \Delta^1)$.

3. We construct the effective homology of $C_*(X \times (0)) \oplus C_*(X \times (1))$.

   Since both $X \times (0)$ and $X \times (1)$ are simplicial sets with effective homology (applying Algorithm 5.40 since X, (0) and (1) are simplicial sets with effective homology), we can construct, applying Algorithm 5.42, the effective homology of $C_*(X \times (0)) \oplus C_*(X \times (1))$.

4. We construct the effective homology of $Cone(i2)$.

   Since $i2$ is a chain complex morphism between two objects with effective homology, $C_*(X \times \Delta^1)$ and $C_*(X \times (0)) \oplus C_*(X \times (1))$, we can construct, applying Algorithm 5.44, the effective homology of $Cone(i2)$.

5. The reduction $M_* \Lleftarrow Cone(i2)$ is constructed applying Algorithm 5.49.

6. The effective homology of M_* is constructed.

   Since $Cone(i2)$ is a chain complex with effective homology and we have the reduction $M_* \Lleftarrow Cone(i2)$, we can construct, applying Algorithms 5.50, the effective homology of M_*.

Therefore, we have constructed the effective homology of the simplicial set $X \times \Delta^1$ but with the simplexes of $X \times (0)$ and $X \times (1)$ cancelled. The above process produces Algorithm 5.46.

Step 3. We construct the effective homology of $C_*X \oplus C_*Y$.

Since both Y and Z are simplicial sets with effective homology we can construct, applying Algorithm 5.42, the effective homology of $C_*Y \oplus C_*Z$.

Step 4. The effective homology of the pushout $P_{(f,g)}$ is constructed.

Let us present how we proceed to complete this construction.

1. We define the following short exact sequence:

$$\begin{tikzcd} 0 & M_* \arrow[l] \arrow[r, shift left, "\sigma"] & C_*P \arrow[l, shift left, "j"] \arrow[r, shift left, "\rho"] & C_*Y \oplus C_*Z \arrow[l, shift left, "i"] & 0 \arrow[l] \end{tikzcd} .$$

   where the morphisms σ and i are inclusions and the morphisms j and ρ are projections.

2. We construct the effective homology of $(C_*Y \oplus C_*Z)^{[1]}$.

   Since $C_*Y \oplus C_*Z$ is a chain complex with effective homology, applying Algorithm 5.48, we can construct the effective homology of $(C_*Y \oplus C_*Z)^{[1]}$.

3. We define the morphism $shift : C_*Y \oplus C_*Z \to (C_*Y \oplus C_*Z)^{[1]}$ which assigns every element of dimension n of $C_*Y \oplus C_*Z$ to the same element in dimension $n+1$ of $(C_*Y \oplus C_*Z)^{[1]}$.

4. We define the chain complex morphism $\chi : M_* \to (C_*Y \oplus C_*Z)^{[1]}$ as the composition $\chi = shift \circ \rho \circ d_{C_*P} \circ \sigma$.

5. We construct the effective homology of $Cone(\chi)$.

   Since χ is a chain complex morphism between two chain complexes with effective homology, M_* and $(C_*Y \oplus C_*Z)^{[1]}$, we can construct, applying Algorithm 5.44, the effective homology of $Cone(\chi)$.

6. Finally, applying Lemma 5.36, C_*P is isomorphic to $Cone(\chi)$, therefore, the effective homology of C_*P is obtained.

In this way, the effective homology of the pushout of two simplicial morphisms $f : X \to Y$ and $g : X \to Z$ where X, Y and Z are simplicial sets with effective homology is obtained. Therefore, we can state the following theorem.

Theorem 5.51. Let $f : X \to Y$ and $g : X \to Z$ be two simplicial morphisms where X, Y and Z are simplicial sets with effective homology. Then $P_{(f,g)}$ is a simplicial set with effective homology.

The proof of this theorem is the above construction which produces Algorithm 5.38.

5.3.2.3.1 By-product algorithms Once we have Algorithm 5.38, it can be used in several interesting cases. For instance, as we commented at the beginning of this section, many of the usual constructions in Topology can be built from the pushout. Let us consider two particular cases: the wedge and the join of simplicial sets with effective homology.

Definition 5.52. Given X and Y 1-reduced spaces with base points $x_0 \in X$ and $y_0 \in Y$, then the *wedge* $X \vee Y$ is the quotient of the disjoint union $X \amalg Y$ obtained by identifying x_0 and y_0 to a single point.

Consider the pushout diagram

$$\begin{tikzcd} X \arrow[r, "i_1"] \arrow[d, "i_2"] & Y \arrow[d] \\ Z \arrow[r] & P \end{tikzcd}$$

where X is the one-point simplicial set and Y, Z are 1-reduced simplicial sets and i_1, i_2 are morphism from X to the base point of Y and Z respectively. Then, $P_{(i_1,i_2)}$ is the wedge $Y \vee Z$.

Algorithm 5.53.
Input: two 1-reduced simplicial sets X and Y with effective homology.
Output: the effective homology of $X \vee Y$.

Definition 5.54. Given X and Y spaces, one can define the space of all lines segments joining points in X to points in Y. This is the *join* $X \bowtie Y$, the quotient space of $X \times Y \times I/\sim$, under the identifications $(x, y_1, 0) \sim (x, y_2, 0)$ and $(x_1, y, 1) \sim (x_2, y, 1)$. Thus we are collapsing the subspace $X \times Y \times (0)$ to X and $X \times Y \times (1)$ to Y.

Consider the pushout diagram

$$\begin{tikzcd} X \times Y \arrow[r, "p_X"] \arrow[d, "p_Y"] & X \arrow[d] \\ Y \arrow[r] & P \end{tikzcd}$$

where p_X and p_Y are the projections. Then, $P_{(p_X,p_Y)}$ is the join $X \bowtie Y$.

Algorithm 5.55.
Input: two simplicial sets X and Y with effective homology.
Output: the effective homology of $X \bowtie Y$.

5.3.2.3.2 Implementation The algorithms explained throughout this subsection have been implemented as a new module for the Kenzo system. The set of programs we have developed (with about 1600 lines) allows the computation of the pushout of two simplicial morphisms $f : X \to Y$ and $g : X \to Z$, and the construction of its version with effective homology when the effective homologies of X, Y and Z are available.

In the development of the new module for Kenzo that allows one to construct the pushout associated with two simplicial morphisms, the first step has consisted in implementing the algorithms presented in Subsubsection 5.3.2.2 which were not available in Kenzo (to be more concrete, algorithms 5.37, 5.41, 5.42, 5.43, 5.44, 5.45, 5.46, 5.47, 5.48 and 5.49). Subsequently, applying the construction previously explained, we have implemented Algorithm 5.38, that is to say, a program that allows us to construct the effective homology of the pushout. Eventually, we have used that program in order to implement algorithms 5.53 and 5.55.

Among the bunch of implemented functions, we highlight the following ones.

pushout *f* *g* *[Function]*

Build a simplicial set with effective homology, namely the pushout of the simplicial morphisms f: $X \to Y$ and g: $X \to Z$ where X, Y and Z are simplicial sets with effective homology.

wedge *smst1* *smst2* *[Function]*

Build a simplicial set with effective homology, namely the wedge of *smst1* and *smst2*, that are two simplicial sets with effective homology.

join *smst1* *smst2* *[Function]*

Build a simplicial set with effective homology, namely the join of *smst1* and *smst2*, that are two simplicial sets with effective homology.

Several examples are provided in the following subsection to provide a better understanding of the new tools.

5.3.3 Examples

In this subsection we present four examples of application of the programs we have developed to build the pushout of two simplicial morphisms. In the first case, we consider the particular case of the wedge of Eilenberg MacLane spaces ($K(\mathbb{Z}, 2) \vee K(\mathbb{Z}, 2)$). As a second example, we will show the join of two spheres. A sophisticated example giving a geometrical construction of $P^2(\mathbb{C})$ is presented, too. Finally, a way of computing some homotopy groups of the suspension of the classifying space of the group $SL_2(\mathbb{Z})$ is given.

We consider a fresh Kenzo session, where our pushout Kenzo module has been loaded.

Wedge of $K(\mathbb{Z},2)$ and $K(\mathbb{Z},2)$

Let us present here an example of the use of the wedge of spaces. In this case we want to compute the first six homology groups of the wedge of $K(\mathbb{Z},2)$ and $K(\mathbb{Z},2)$. First of all, we build the Eilenberg-MacLane space $K(\mathbb{Z},2)$.

```
> (setf bkz (k-z 2)) ✠
[K13 Abelian-Simplicial-Group]
```

We construct the wedge of $K(\mathbb{Z},2)$ and $K(\mathbb{Z},2)$.

```
> (setf bkzwbkz (wedge bkz bkz)) ✠
[K41 Simplicial-Set]
```

Finally, homology groups can be computed as usual.

```
> (homology bkzwbkz 0 6) ✠
Homology in dimension 0 :
Component Z
Homology in dimension 1 :

Homology in dimension 2 :
Component Z
Component Z
Homology in dimension 3 :

Homology in dimension 4 :
Component Z
Component Z
Homology in dimension 5 :

```

Join of S^2 and S^3

Let us present here an example of the use of the join of spaces. In this case we want to compute the first seven homology groups of the join of the spheres S^2 and S^3. Let us note that the join of the spheres S^n and S^m is the sphere S^{n+m+1}; so, in our case the only non null homology groups should be 0 and 6. We construct the spheres S^2 and S^3.

```
> (setf s2 (sphere 2)) ✠
[K331 Simplicial-Set]
> (setf s3 (sphere 3)) ✠
[K336 Simplicial-Set]
```

We construct the join $S^2 \bowtie S^3$.

```
> (setf s2js3 (join s2 s3)) ✠
[K353 Simplicial-Set]
```

Lastly, homology groups can be computed getting the expected result.

```
> (homology s2js3 0 7) ✠
Homology in dimension 0 :
Component Z
Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Homology in dimension 4 :

Homology in dimension 5 :

Homology in dimension 6 :
Component Z
```

$P^2(\mathbb{C})$

This example which gives a geometrical construction of $P^2(\mathbb{C})$ [Ser10] was suggested by Francis Sergeraert. Take S^2 and construct the first stage of the Whitehead tower. We access to the current definition of the sphere of dimension 2 stored in the variable `s2`.

```
> s2 ✠
[K331 Simplicial-Set]
```

We build the fundamental cohomology class.

```
> (setf ch2 (chml-clss s2 2)) ✠
[K547 Cohomology-Class on K331 of degree 2]
```

We construct the fibration over the sphere canonically associated with the above cohomology class.

```
> (setf f2 (z-whitehead s2 ch2)) ✠
[K548 Fibration K331 -> K1]
```

Finally, the total space from the fibration is generated.

```
> (setf x3 (fibration-total f2)) ✠
[K554 Simplicial-Set]
```

Then `x3` has the homotopy type of the 3-sphere S^3. More precisely `x3`$= s2\times_{\mathtt{f2}}K(Z,1)$ with `f2` an appropriate twisting function producing S^3 as a total space. It is easy to deduce a projection $f : X3 \rightarrow S2$.

```
> (setf f (build-smmr ; the function to construct a simplicial morphism
            :sorc x3 ; the source simplicial set
            :trgt s2 ; the target simplicial set
            :degr 0  ; the degree of the morphism
            :sintr #'(lambda (dmns gmsm) ; the map
                              (declare (ignore dmns))
                            (absm (dgop1 gmsm) (gmsm1 gmsm)))
            :orgn `(proj ,x3 ,s2))) ✠
[K559 Simplicial-Morphism K554 -> K331]
```

Taking the pushout of this f and the map $g : X3 \rightarrow *$ (where $*$ is the simplicial set with just one vertex),

```
> (setf unipunctual (build-finite-ss '(x))) ✠
[K25 Simplicial-Set]
>(setf g (build-smmr ; the function to construct a simplicial morphism
            :sorc x3 ; the source simplicial set
            :trgt unipunctual ; the target simplicial set
            :degr 0  ; the degree of the morphism
            :sintr #'(lambda (dmns gmsm) ; the map
                            (if (and (equal dmns 0)
                                     (equal gmsm (bsgn x3)))
                              'x
                             nil))
            :orgn `(proj ,x3 ,unipunctual))) ✠
[K560 Simplicial-Morphism K554 -> K25]
```

then the pushout is $P^2(\mathbb{C})$.

```
> (setf p (pushout f g)) ✠
[K566 Simplicial-Set]
```

It can be checked that our programs obtain the right homology groups that are $(\mathbb{Z}, 0, \mathbb{Z}, 0, \mathbb{Z}, 0, 0, \ldots)$:

```
> (homology p 0 7) ✠
Homology in dimension 0 :
Component Z
Homology in dimension 1 :

Homology in dimension 2 :
Component Z
Homology in dimension 3 :

Homology in dimension 4 :
Component Z
Homology in dimension 5 :

Homology in dimension 6 :
```

$SL_2(\mathbb{Z})$

This example allows us to compute some homotopy groups of the suspension of $SL_2(\mathbb{Z})$ (the group of 2×2 matrices with determinant 1 over $\mathbb{Z}$, with the group operations of ordinary matrix multiplication and matrix inversion), the interest in computing homotopy groups of this kind of spaces can be seen in [MW10]. In [Ser80], it was explained that $SL_2(\mathbb{Z})$ is isomorphic to the amalgamated sum $\mathbb{Z}_4 *_{\mathbb{Z}_2} \mathbb{Z}_6$. Then, this is a pushout in the category of groups, and, in this case, applying the functor $K(-,1)$ we obtain also a pushout (see [Bro82]):

$$\begin{tikzcd}
K(\mathbb{Z}_2,1) \arrow[r, "i_1"] \arrow[d, "i_2"] & K(\mathbb{Z}_4,1) \arrow[d] \\
K(\mathbb{Z}_6,1) \arrow[r] & P
\end{tikzcd}$$

Then, we can compute the homology groups of $SL_2(\mathbb{Z})$ thanks to the programs developed to construct the pushout of simplicial sets and the programs presented in [RER09] which allow us to construct $K(G,1)$ for G a cyclic group (in this case $K(\mathbb{Z}_2,1)$, $K(\mathbb{Z}_4,1)$ and $K(\mathbb{Z}_6,1)$). Namely, we proceed as follows. First of all, we construct the spaces $K(\mathbb{Z}_2,1)$, $K(\mathbb{Z}_4,1)$ and $K(\mathbb{Z}_6,1)$.

```
> (setf kz2 (k-zp-1 2)) ✠
[K2 Abelian-Simplicial-Group]
> (setf kz4 (k-zp-1 4)) ✠
[K15 Abelian-Simplicial-Group]
> (setf kz6 (k-zp-1 6)) ✠
[K28 Abelian-Simplicial-Group]
```

Subsequently, we define the two simplicial morphisms f and g by means of a function called `kzps-incl` which represents the inclusion between $K(\mathbb{Z}_n, 1)$ and $K(\mathbb{Z}_m, 1)$.

```
(defun kzps-incl (n m)
  (declare (number n m))
  (let ((i (/ m n)))
    (build-smmr ; the function to construct a simplicial morphism
     :sorc (k-zp-1 n) ; the source simplicial set
     :trgt (k-zp-1 m) ; the target simplicial set
     :degr 0 ; the degree of the simplicial morphism
     :sintr #'(lambda (dmns gmsm) ; the map
                      (absm 0 (make-list dmns :initial-element i)))
     :orgn  `(inclusion between (k-g-1 ,n) and (k-g-1 ,m)))))
```

```
> (setf kz2-kz4 (kzps-incl 2 4)) ✠
[K40 Simplicial-Morphism K2 -> K15]
> (setf kz2-kz6 (kzps-incl 2 6)) ✠
[K41 Simplicial-Morphism K2 -> K28]
```

Eventually, we construct the pushout of `kz2-kz4` and `kz2-kz6`.

```
> (setf p (pushout kz2-kz4 kz2-kz6)) ✠
[K52 Simplicial-Set]
```

Due to the fact that the abelianization map $SL_2(\mathbb{Z}) \rightarrow \mathbb{Z}/12\mathbb{Z}$ induces an isomorphism on integral homology, see [Knu], then, the homology groups of $SL_2(\mathbb{Z})$ are the same that the ones of $K(\mathbb{Z}/12\mathbb{Z}, 1)$. It can be checked that our programs obtain the right homology groups that are $(\mathbb{Z}, \mathbb{Z}/12\mathbb{Z}, 0, \mathbb{Z}/12\mathbb{Z}, 0, \mathbb{Z}/12\mathbb{Z}, \ldots)$:

```
> (homology p 0 7) ✠
Homology in dimension 0 :
Component Z
Homology in dimension 1 :
Component Z/12Z
Homology in dimension 2 :

Homology in dimension 3 :
Component Z/12Z
Homology in dimension 4 :

Homology in dimension 5 :
Component Z/12Z
Homology in dimension 6 :

```

Therefore, we have been able to compute the homology groups of $SL_2(\mathbb{Z})$. Now, we apply the suspension constructor to the pushout, see Definition 1.28. Subsequently, we can compute, using the Kenzo system, some homotopy groups of the suspension of $SL_2(\mathbb{Z})$ thanks to the algorithm explained in [Rea94] and using the programs developed in [RER09]. Namely, the first homotopy groups of the suspension of $SL_2(\mathbb{Z})$, denoted by $\Sigma(SL_2(\mathbb{Z}))$ are $\pi_2(\Sigma(SL_2(\mathbb{Z}))) = \mathbb{Z}/12\mathbb{Z}$, $\pi_3(\Sigma(SL_2(\mathbb{Z}))) = \mathbb{Z}/12\mathbb{Z}$ and $\pi_4(\Sigma(SL_2(\mathbb{Z}))) = \mathbb{Z}/12\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z}$. The file with the procedure to compute these homotopy groups can be seen in [Her11].

5.3.4 Integration of the pushout in our framework

In the last two examples of the previous subsection, the definition of simplicial morphisms for the construction of the pushout has been shown. Namely, to define a simplicial morphism an instance of the class `SIMPLICIAL-MRPH` (which is a subclass of the `MORPHISM` class) is necessary.

The relevant slots of a `SIMPLICIAL-MRPH` instance are `sorc`, the source object of type `SIMPLICIAL-SET`; `trgt`, the target object of type `SIMPLICIAL-SET`; `degr`, the degree of the morphism; `sintr`, the internal lisp function defining the effective mapping between simplicial sets; and `orgn`, used to keep a record of information about the object.

The definition of the Lisp function installed in the `sintr` slot is the main hindrance in order to integrate the whole functionality of the pushout in *fKenzo*. The definition of this lisp function is ad-hoc in most of the cases (although in some particular cases, such as the wedge or the join, can be defined in general) and involves a study of the internal structure of the source and target simplicial sets and a knowledge of the definition of Lisp functions, and such a meticulous study is difficult to integrate in our framework, at least in an easy and *usable* way (i.e. without giving access to the internal Common Lisp code).

Nevertheless, some of the functionality developed in the new module can be integrated in *fKenzo*, namely the functionality related to construct the wedge and the join of simplicial sets, since it does not involve, at least in an explicit way, the construction of simplicial morphisms and we can use them like other Kenzo constructors such as cartesian or tensor products. This subsection is devoted to explain the necessary steps to incorporate both the wedge and join functionality to the *Kenzo* framework.

To enhance the framework with the functionality related to wedge and join constructions, we have developed a plug-in following the guidelines given in Subsubsection 3.1.2.

This new plug-in will allow us to construct the wedge and the join of simplicial sets and use them as any other object of the framework. The plug-in used to include the functionality about these constructors references the following resources:

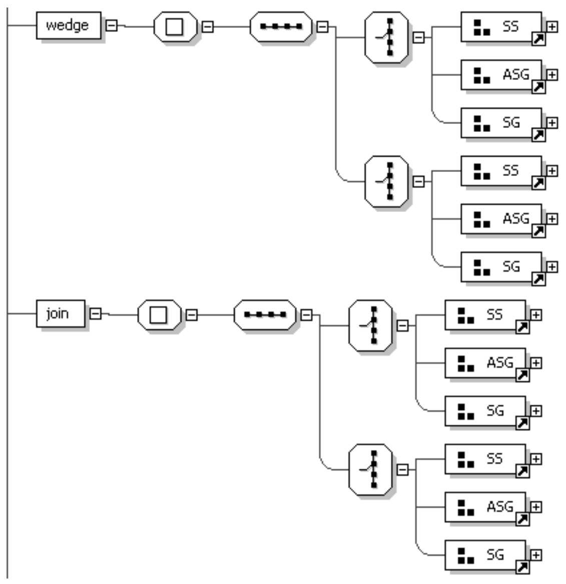


Figure 5.16: join and wedge elements in XML-Kenzo

```
<code id="pushout">
   <data format="Kf/external-server"> XML-Kenzo.xsd </data>
   <data format="Kf/internal-server"> pushout-list.lisp </data>
   <data format="Kf/microkernel"> pushout-m.lisp </data>
   <data format="Kf/adapter"> pushout-a.lisp </data>
</code>
```

As we claimed in Subsubsection 3.1.2, if we want to include new functionality in the Kenzo internal server and make it available outside the framework, all the components must be broadening. Let us explain each one of the referenced resources.

We want to introduce two new kinds of objects in our system (the wedge and the join of simplicial sets), then, it is necessary to provide a representation for that objects in our system. To that aim, we have defined two new elements: `wedge` and `join`. Both elements are defined as elements of the `SS` type, so, they can be used as any other element of this group, and have two children of some of the types `SS`, `SG` or `ASG` (see Figure 5.16).

As we explained in Subsection 3.1.2 the external server evolves when the `XML-Kenzo.xsd` file is upgraded. Then, when the `XML-Kenzo.xsd` file is modified to include the join and the wedge elements, the external server is able to check the correctness of requests such as:

```
<constructor>
    <join>
      <wedge>
        <sphere>3</sphere>
        <sphere>4</sphere>
      </wedge>
      <k-z>2</k-z>
    </join>
</constructor>
```

The `pushout-list.lisp` file includes the functionality defined in Subsection 5.3.2 to extend the Kenzo system. Moreover, this file includes the functionality that enhances the `xml-kenzo-to-kenzo` function of the internal server to process the construction of objects from the `join` and `wedge` elements. For instance, if the internal server receives the above request. The instruction

```
(join (wedge (sphere 3) (sphere 4)) (k-z 2))
```

is executed in the Kenzo kernel. As a result an object of the `Simplicial-Set` Kenzo class is constructed, and the identifier of that object is returned.

The `pushout-m.lisp` file defines two new construction modules for the microkernel called `join` and `wedge`. The procedures implemented in this module follows the guidelines explained in Subsubsection 2.2.3.3. In this case neither of the modules checks any property to construct objects when they are activated since the requests coming from the external server related to wedge and join are always safe; that is, all the requests related to wedge and join constructions that can produce errors are stopped in the external server and never arrive to the microkernel (since the constraints related to these constructors are type restrictions which are handled in the XML-Kenzo specification).

Finally, we have extended the `SS` Content Dictionary by means of the definition of two new objects: `join` and `wedge`. Therefore, the `pushout-a.lisp` file contains the necessary functions to raise the functionality of the adapter to be able to convert from these new OpenMath objects, devoted to the wedge and the join, to XML-Kenzo requests. Namely, we have extended the Phrasebook by means of a new parser in charge of this task. Then, for instance, the previous XML-Kenzo request is generated by the adapter when the following OpenMath request is received.

```
<OMOBJ>
   <OMA>
      <OMS cd="SS" name="join"/>
      <OMA>
        <OMS cd="SS" name="wedge"/>
          <OMA> <OMS cd="SS" name="sphere"/> <OMI>3</OMI> </OMA>
          <OMA> <OMS cd="SS" name="sphere"/> <OMI>4</OMI> </OMA>
      </OMA>
      <OMA> <OMS cd="ASG" name="k-z"/> <OMI>2</OMI> </OMA>
   </OMA>
</OMOBJ>
```

5.3.5 Integration of the pushout in the *fKenzo* GUI

In this subsection we are going to present the necessary resources to extend the *fKenzo* GUI with the functionality about `wedge` and `join` constructors. As we presented in Section 3.2 one of the modules which customizes *fKenzo* is the Simplicial Set module. This module contains the elements that represent the simplicial set constructors of Kenzo: options to construct spaces from scratch (spheres, Moore spaces, finite simplicial sets, simplicial sets coming from simplicial complexes and so on) and from other spaces (for instance, cartesian products). We have enhanced this module by means of the necessary ingredients to construct joins and wedges of simplicial sets.

In Section 3.2 a *Simplicial Set* module with two files, `simplicial-sets-structure` (which defined the structure of the graphical constituents) and `simplicial-sets-functionality` (which provided the functionality related to the graphical constituents), was presented.

In Subsection 5.1.4, the original Simplicial Set module was improved to include the functionality about simplicial complexes. In particular, we modified the files `simplicial-sets-structure` and `simplicial-sets-functionality` referenced by the original module and we also included a reference to the plug-in about simplicial complexes.

Now, we have extended this module to include the functionality about the wedge and the join of simplicial sets. Namely, it also references the plug-in presented in the previous subsection and, in addition, both `simplicial-sets-structure` and `simplicial-sets-functionality` documents have been upgraded in order to allow the use of wedge and join constructors in the *fKenzo* GUI.

We have defined four new graphical elements, using the XUL specification language, in the `simplicial-sets-structure` file:

- A new menu option called `Join` included in the Simplicial Sets menu.
- A new menu option called `Wedge` included in the Simplicial Sets menu.

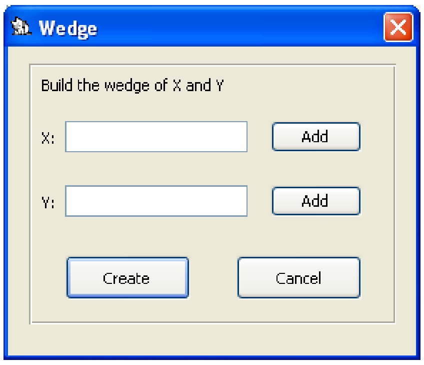


Figure 5.17: Wedge window

- A window called `Join` (very similar to the window defined to construct the cartesian product of two simplicial sets).
- A window called `Wedge` as above.

The functionality stored in the `simplicial-sets-functionality` document related to these components works as follows. The functionality is analogous for wedge and join options; so let us explain just the wedge one. A function acting as event handler is associated with the `Wedge` menu option; this function shows the `Wedge` window (see Figure 5.17) if at least a simplicial set, a simplicial group or an abelian simplicial group was built previously in *fKenzo*; otherwise, it informs the user that at least one object of one of those types must be constructed before using the `Wedge` option.

From the `Wedge` window, the user must select two objects from the two `Add` buttons; the lists of objects shown by the `Add` buttons only contain the objects of types simplicial set, simplicial group and abelian simplicial group.

Once the user has selected two objects $X1$ and $X2$, when it presses the `create` button of the `Wedge` window, an OpenMath request is generated. Subsequently, our framework is invoked with the OpenMath request. The wedge of $X1$ and $X2$ is constructed and its identification number is returned. Eventually, *fKenzo* adds to the list of constructed spaces (situated in the left side of the main tab of the *fKenzo* GUI) the new simplicial set. Figure 5.18 shows the control and navigation submodel (with the Noesis notation) describing the construction of the wedge of two simplicial sets from the `Wedge` menu option.

Moreover, we have also modified the stylesheet used to present in the *fKenzo* GUI the spaces constructed, using their mathematical notation (see Subsubsection 3.2.4.3). This stylesheet is modified in the folder of the *fKenzo* distribution to present with standard notation both wedge and join spaces.

Let us present an example, for instance if we have constructed the spaces S^3 and $M(\mathbb{Z}/2\mathbb{Z},4)$ in a *fKenzo* session, we can build the space $S^3 \vee M(\mathbb{Z}/2\mathbb{Z},4)$ and subsequently the space $(S^3 \vee M(\mathbb{Z}/2\mathbb{Z},4)) \bowtie (S^3 \vee M(\mathbb{Z}/2\mathbb{Z},4))$ from `Wedge` and `Join` menus

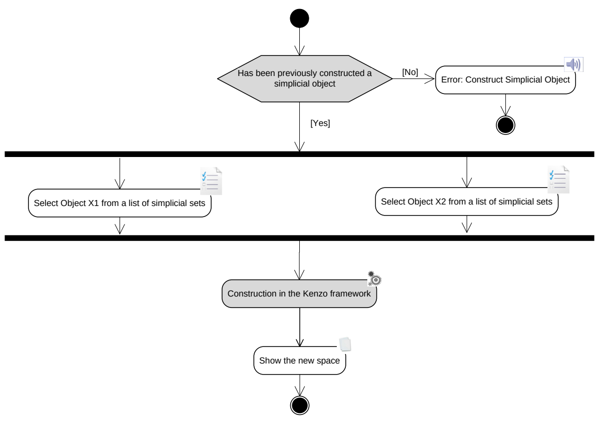

Has been previously constructed a simplicial object
[No]
Error: Construct Simplicial Object
[Yes]
Select Object X1 from a list of simplicial sets
Select Object X2 from a list of simplicial sets
Construction in the Kenzo framework
Show the new space

Figure 5.18: Control graph for the construction of the wedge of two simplicial sets

respectively. Finally, the user can ask *fKenzo* to compute the homology groups of these spaces using the `homology` option of the `Computing` menu. The results are shown, as usual, in the `Computing` tab, see Figure 5.19.

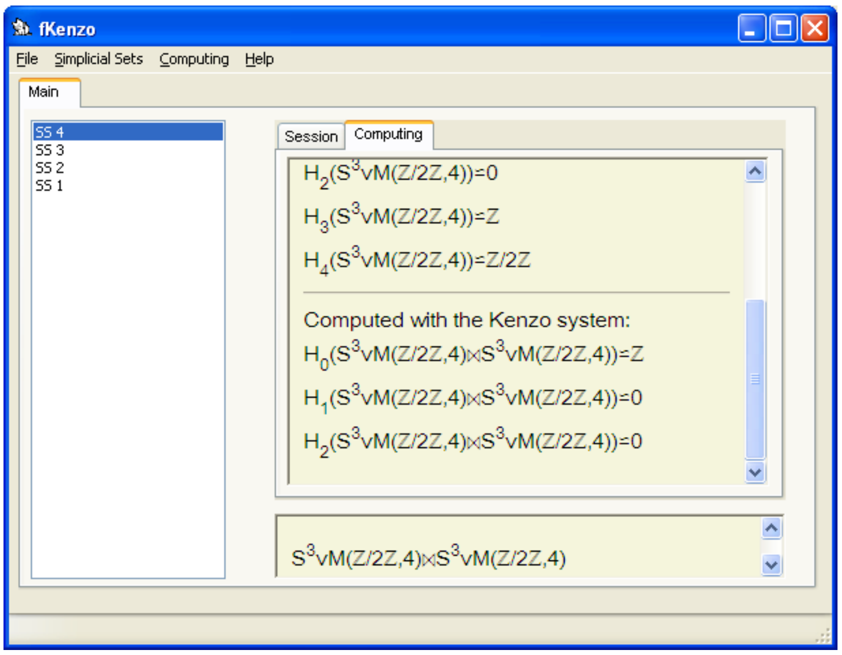
5.3.6 Formalization of the pushout in ACL2

In the same way that we certified our programs for the case of simplicial complexes and digital images, we are interested in verifying the correctness of the implementation of the algorithms which construct the pushout of simplicial sets in ACL2. This task is an ongoing work, and at this moment just some of the algorithms are verified. To be more concrete, the implementation of Algorithms 5.41 (direct sum), 5.43 (cone), 5.47 (suspension functor) and the algorithms associated with SES1 and SES3 cases of Theorem 5.32 have been formalized in ACL2.

The verification of the correctness of those algorithms and also the rest of algorithms presented is part of a wider project which tries to verify the implementation of algorithms which construct new spaces by applying topological constructions. This formalization will be presented in Section 6.2 of the following chapter.

In that section, a methodology to prove the correctness of the construction of new spaces by applying topological constructions is presented. This methodology copes with the higher-order nature of those constructors by means of the simulation of high order logic by means of ACL2 encapsulates.

Therefore, we postpone the presentation of the formalization of the implementation of algorithms of this section to Section 6.2.



H_2(S^{3}vM(Z/2Z,4))=0

H_3(S^{3}vM(Z/2Z,4))=Z

H_4(S^{3}vM(Z/2Z,4))=Z/2Z

Computed with the Kenzo system:

H_0(S^{3}vM(Z/2Z,4)⋈S^{3}vM(Z/2Z,4))=Z

H_1(S^{3}vM(Z/2Z,4)⋈S^{3}vM(Z/2Z,4))=0

H_2(S^{3}vM(Z/2Z,4)⋈S^{3}vM(Z/2Z,4))=0

S^{3}vM(Z/2Z,4)⋈S^{3}vM(Z/2Z,4)

Figure 5.19: Homology groups of $S^3 \vee M(\mathbb{Z}/2\mathbb{Z}, 4)$ and $(S^3 \vee M(\mathbb{Z}/2\mathbb{Z}, 4)) \bowtie (S^3 \vee M(\mathbb{Z}/2\mathbb{Z}, 4))$