Certifying homological algorithms to study biomedical images

María Poza López de Echazarreta

Dissertation submitted for the degree
of Doctor of Philosophy

Supervisors: Dr. D. César Domínguez Pérez
Dr. D. Julio Rubio García

Universidad de La Rioja
Departamento de Matemáticas y Computación

Logroño, April 2013

This work has been partially supported by the project MTM2009-13842-C02-01 from the Spanish Ministerio de Educación y Ciencia, and by the FORMATH project, nr. 243847 of the FET program within the 7th Framework program of the European Commission.

Acknowledgments

To Álvaro
To my parents and my brother

First of all, I would like to thank my supervisors César Domínguez and Julio Rubio for your support and dedication along this adventure. Thank you for the time that you have spent on me, I am sure that without your suggestions and corrections it would not have been possible for me to make this thesis. Moreover, I cannot forget Jónathan Heras. I would like to thank you for your help and unconditional support as at a scientific as personal level. Above all, I would like to underline your understanding and patience with me. Your thesis has been a point of reference and yourself a good example for me.

On the other hand, I would like to thank the rest of my group partners for the fantastic atmosphere that has been between us during this period of time. From the beginning you have encouraged me to feel part of the group, and in this way, I begun playing paddel tennis. In addition, I would like to show all my gratitude to my office partner Clara Jimenez because of her company especially along the last year. Besides, I also would like to be grateful that my partner and friend Gadea Mata has given to me the necessary encouragement and also our meals and cafes at the sun, even when there was not sun.

To conclude, I would like to thank to my parents and my brother for their patience during the development of this thesis because sometimes I have not been the best company for them during this period of time. In addition, thanks to Álvaro because he has stayed close to me when I needed him, for his support and his big confidence in me. Thanks to my friends because they did not know

exactly what was I doing but they were always asking me for the development of it.

To all of you, thanks.

Contents

Introduction

Scientific computing is an outstanding tool to assist researchers in experimental sciences. When applied to biomedical problems, the accuracy and reliability of the computations are particularly important. Thus, the possibility of increasing the trust in scientific software by means of mechanized theorem proving technologies becomes an interesting area of research.

Interactive proof assistants are software tools designed to help researchers in the development of formal proofs. These systems require the cooperation of human beings and machines. Namely, the user is in charge of designing the proofs, giving big steps as usual in mathematics, while the machine, with the help of the human, fills the gaps. There are several examples of proof assistants such as HolLight [Har09], Isabelle/HOL [NPW02], ACL2 [KM], and Coq [BC04, BGBP08]. Proof assistant tools are mature enough to tackle interesting problems in the formalization of mathematics and also to verify the correctness of software and hardware. Some appealing examples are the formalizations of the Four Color Theorem [Gon08], the Fundamental Theorem of Algebra [GWZ02], the Kepler Conjecture [Hal05a], the verification of a C compiler [Ler09], or the formalization of the correctness of an AMD microprocessor [BKM96]. Some projects have been launched with the unique purpose of ensuring the correctness of a concrete mathematical proof, as in the case of the Flyspeck project [Hal05b], devoted to formalize Hales' proof of the Kepler conjecture [Hal05a].

In our case, we use the Coq proof assistant, which is based on Calculus of Inductive Constructions [CH88]. This system has an interesting feature which allows us to extract programs from constructive proofs. In addition, we have used SSReflect [GM10], an extension for Coq. The SSReflect tactic language

and library were initially designed to prove the Four Color Theorem and has been later improved in order to tackle the Feit-Thompson Theorem (also known as Odd Order Theorem) [Mat12].

In this work, we use Coq with the aim of certifying some image processing procedures. The images are taken at the beginning of our investigation from neuron cultures using microscopical devices [C+11]. The technique that we use to process these images is based on Computational Algebraic Topology.

Computational Algebraic Topology is an emerging field which attracts the interest of researchers, in both theoretical and industrial aspects (see, for instance, [EH10]). Even if the interest is growing in recent years, the history of Computational Algebraic Topology is long (at least, with respect to the standards in computer algebra and scientific computing). One of the first theories in this field is called *effective homology* [RS02], created by F. Sergeraert, and which produced the *Kenzo* software system [DRSS98]. Kenzo is devoted to compute homology and homotopy groups of topological spaces, and some of its calculations produced unknown results [Ser92] or even corrected previously published theorems [RR12]. Therefore, mathematicians should trust Kenzo results. To reinforce this trust, several contributions have been made to apply formal methods to the study of Kenzo and its underlying algorithms (see, among others, [ABR08, ABR10, AD09, AD10, DLR07, DR10, DR11]).

This is the frame where the research presented in this memoir is located. Some algorithms by Romero and Sergeraert [RS10] were implemented in Kenzo to be applied on digital images. Here, we undertake the verification of those algorithms in Coq, emulating the Kenzo processes.

The foundations for a homological processing of digital images are based on the discipline called *Digital Topology* [ADFQ03]. Digital images must be interpreted as topological spaces in a combinatorial way. The most elementary method to settle a connection between General Topology and Combinatorial Topology is based on the use of simplicial complexes. The notion of topological space is too "abstract" in order to transfer it to a computational universe. Simplicial complexes provide a purely combinatorial description of topological spaces which admit a triangulation. The computability of invariants, 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

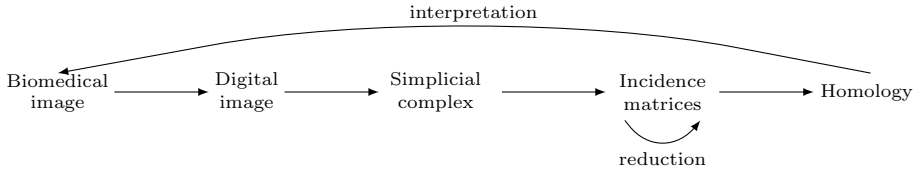


Figure 1: Computing homology from a digital image

simple linear algebra [Veb31]. Then, an algebraic topologist can identify a compact triangulable topological space (as a "continuous" interpretation of a digital image) with a finite simplicial complex, making computations possible. The role of Algebraic Topology in digital imaging is well-known (see for instance the series of conferences called *Computational Topology in Image Context*).

In a very rough manner, the process to be studied in this memoir is described in Figure 1. Putting it into words, after preprocessing a biomedical image, a monochromatic picture is obtained; then, from the black pixels of that monochromatic image a simplicial complex is created (by means of a triangulation procedure); subsequently, from the simplicial complex, its *boundary (or incidence) matrices* are constructed, and finally, *homology* can be computed. If we work with coefficients over a field (and it is well-known that it is enough to take as coefficients the field $\mathbb{Z}_2$, when we work with 2D and 3D digital images) and if only the *dimensions* of the homology groups (as vector spaces) are looked for, then having a program able to compute the rank of a matrix is sufficient to accomplish the whole task.

This architecture is particularized in this memoir with a real problem that appeared in an industrial application and with the Coq proof assistant as programming and verifying tool. The biological problem (namely, the number of synapses in a picture of a neuron) can be identified with the computation of a topological invariant (the rank of a homology group). Then, all our efforts are concentrated on computing, in a certified manner, such an invariant.

Since the size of real-life biomedical images is too big to deal with them in a direct way, we propose a *reduction strategy* (see the reduction step in Figure 1) allowing us to work with smaller data structures but preserving all their homological properties. To this aim, we use the notion of *discrete vector field* [For98], following very closely the approach presented by Romero and Sergeraert in [RS10].

In order to verify the correctness of these procedures, it is needed to formalize a certain amount of mathematics. The most significant piece of mathematics formalized in this work is the so-called *Basic Perturbation Lemma* (or BPL, in short). The proof of this theorem has been already implemented in the Isabelle/HOL proof assistant [ABR08]. The BPL formalization presented in this work is much shorter and compact than that of [ABR08]. There are two reasons for this improvement of the formal proof. The first one is that in this work we have followed a new and shorter proof of the BPL (due again to Romero and Sergeraert [RS12]). The second reason is that we have built our formal proof on the powerful library SSReflect of Coq [GM10] (on the contrary, much of the infrastructure required was defined from scratch in [ABR08]).

Apart from the efficiency in the writing of proofs, using SSReflect also has other consequences. Since SSReflect is designed to deal only with *finite structures*, the proof of the BPL presented here only applies over *finitely generated groups* (the proof formalized in [ABR08] has not this constraint). Furthermore, dealing with finite structures, and inside the constructive logic of Coq, eases the executability of the proofs, and thus the generation of certified programs (the same tasks in Isabelle/HOL pose more difficulties; see [ABR10]).

In order to prove the correctness of the generated programs, we must establish, and keep, a link among the initial digital image, and the final smaller data structure where the homological calculations are carried out. This implies a big amount of processing, and does not allow us to execute all the steps *inside* Coq (the full path has been traveled, but only in *toy* examples). Then, we have appealed to a programming language, Haskell [Hut07] in our case, to integrate computation and deduction.

Haskell appears in two different steps of our methodology. In the early development stages, Haskell prototypes of the algorithms are systematically tested using the QuickCheck tool [CH00]. This permits us to discharge many small and common errors, which could hinder the proving process in Coq. In the final computational step, Haskell is used as an *oracle* for Coq. The most hard parts of the calculation (in our case, an important bottleneck is the computation of inverse matrices) are delegated to Haskell programs; the *results* of these Haskell programs are then *proved* correct within Coq.

With this hybrid technique, we have reached the objective of computing, in a

certified way, the homology of actual biomedical images coming from neurological experiments.

The structure of the memoir is the following one.

In Chapter 1, preliminaries for our research are introduced. It includes both mathematical and theorem proving aspects. The mathematics to formalize are also described there.

Chapter 2 is devoted to the implementation and verification of Romero-Sergeraert's algorithm to compute a discrete vector field associated with a digital image. The results covered in this chapter have been partially published in the paper [HPR12].

In Chapter 3 and 4, we describe respectively the construction of the algebraic reduction defined by a discrete vector field, and a formal proof of the BPL. The results of these two chapters are the subject of the forthcoming paper [PDHR13].

Chapter 5 deals with the application of the previous results to materialize the path represented in Figure 1. Concretely, we implement chain complexes and incidence matrices (partially published in [HPDR11]) and compute homology inside Coq (see our paper [HDM$^+$12]). Then, the certified reduction strategy is integrated in this general schema to improve performance.

In Chapter 6, some computational experiments carried out around our work (with Kenzo, with Haskell, or in Coq) are reported, showing the strengths and weaknesses of our approach.

The memoir ends with a section devoted to conclusions and further work, and the bibliography.

The interested reader can consult the code presented throughout this memoir in [For].

Chapter 1

Preliminaries

In this chapter, we introduce the context and the tools that we will use in the rest of this memoir. The first section is devoted to the mathematical notions employed in this work. Then, a brief introduction to the Coq system and its SSReflect library is provided in Section 1.2. Afterwards, the methodology which we have followed in our development is presented in Section 1.3. Finally, we introduce as a summary the concepts and results which we are going to formalize in Section 1.4.

1.1 Mathematical background

In the first sections, we briefly provide the minimal standard mathematical background related to Homological Algebra and Simplicial Topology which will be necessary in the rest of the text. Many good textbooks are available for these definitions and results about them (see for instance, [Mau96] or [Mac63]). Then, we introduce the relations between both simplicial complexes and chain complexes and digital images and simplicial complexes. In this way, we can obtain the homology groups of a chain complex which give us properties of the corresponding image.

Moreover, we present some more notions about reductions and an algebraic setting of Discrete Morse Theory which allows us to reduce the amount of information without changing the homological properties [For98].

1.1.1 Chain complexes

Definition 1.1. Let R be a ring with a unit element $1 \neq 0$. A *left R-module* M is an additive abelian group together with a map $p : R \times M \to M$, denoted by $p(r, m) \equiv rm$, such that for every $r, r' \in R$ and $m, m' \in M$

$$\begin{aligned}(r + r')m &= rm + r'm \\ r(m + m') &= rm + rm' \\ (rr')m &= r(r'm) \\ 1m &= m\end{aligned}$$

A similar definition can be given for a *right R-module*.

If $R = \mathbb{Z}$ (the integer ring), a $\mathbb{Z}$-module M is simply an abelian group. The map $p : \mathbb{Z} \times M \to M$ is given by

$$p(n, m) = \begin{cases} m + \overset{n}{\cdots} + m & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ (-m) + \overset{-n}{\cdots} + (-m) & \text{if } n < 0 \end{cases}$$

Definition 1.2. Let R be a ring and M and N be R-modules. An *R-module morphism* $\alpha : M \to N$ is a function from M to N such that for every $m, m' \in M$ and $r \in R$

$$\begin{aligned}\alpha(m + m') &= \alpha(m) + \alpha(m') \\ \alpha(rm) &= r\alpha(m) \\ \alpha(0_M) &= 0_N\end{aligned}$$

Definition 1.3. Given a ring R, a *graded module* M is a family of left R-modules $(M_n)_{n\in\mathbb{Z}}$.

Definition 1.4. Given a pair of graded modules M and M', a *graded module morphism* f of degree k between them is a family of module morphisms $(f_n)_{n\in\mathbb{Z}}$ such that $f_n : M_n \to M'_{n+k}$ for all $n \in \mathbb{Z}$.

Definition 1.5. Given a graded module M, a *differential* $(d_{M_n})_{n\in\mathbb{Z}}$ is a family of module endomorphisms of M of degree -1 such that $d_{M_{n-1}} \circ d_{M_n} = 0$ for all $n \in \mathbb{Z}$.

From the previous definitions, the notion of chain complex is introduced as follows.

Definition 1.6. A *chain complex* C_* is a pair of families $(C_n, d_{C_n})_{n\in\mathbb{Z}}$ where $(C_n)_{n\in\mathbb{Z}}$ is a graded module and $(d_{C_n})_{n\in\mathbb{Z}}$ is a differential map on $(C_n)_{n\in\mathbb{Z}}$.

The module C_n is called the module of *n-chains*. The image $B_n = \operatorname{im} d_{C_{n+1}} \subseteq C_n$ is the (sub)module of n-boundaries. The kernel $Z_n = \ker d_{C_n} \subseteq C_n$ is the (sub)module of *n-cycles*.

In many situations the ring R is the integer ring, $R = \mathbb{Z}$. In this case, a chain complex C_* is given by a graded abelian group $(C_n)_{n\in\mathbb{Z}}$ and a graded group morphism of degree -1, $(d_{C_n} : C_n \to C_{n-1})_{n\in\mathbb{Z}}$, satisfying $d_{C_{n-1}} \circ d_{C_n} = 0$ for all n.

Let us present some examples of chain complexes.

Example 1.7. The *unit chain complex* has a unique non-null module, namely a $\mathbb{Z}$-module in degree 0 generated by a unique generator, and the differential is the null map.

Example 1.8. A chain complex to model the *circle* is defined as follows. This chain complex has two non-null modules, namely a $\mathbb{Z}$-module in degree 0 generated by an unique generator and a $\mathbb{Z}$-module in degree 1 generated by another generator; and the differential is the null map.

We can construct chain complexes from other ones, applying constructors such as the direct sum.

Definition 1.9. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{Z}}$ be chain complexes. The *direct sum* of C_* and D_* is the chain complex $C_* \oplus D_* = (M_n, d_n)_{n\in\mathbb{Z}}$ such that, $M_n = (C_n, D_n)$ and the differential map is defined on the generators (x, y) with $x \in C_n$ and $y \in D_n$ by $d_n(x, y) = (d_{C_n}(x), d_{D_n}(y))$ for all $n \in \mathbb{Z}$.

Let us present now one of the most important invariants used in Homological Algebra. Given a chain complex $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$, the identities $d_{C_{n-1}} \circ d_{C_n} = 0$ are equivalent to the inclusion relations $B_n \subseteq Z_n$: every boundary is a cycle but the converse is not generally true. B_n represents the image of $d_{C_{n+1}}$ and Z_n the kernel of d_{C_n}. Thus, the next definition makes sense.

Definition 1.10. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ be a chain complex of R-modules. For each degree $n \in \mathbb{Z}$, the *n-homology module* of C_* is defined as the quotient

$$H_n(C_*) = \frac{Z_n}{B_n}$$

It is worth noting that the homology modules of a space X are the ones of its associated chain complex $C_*(X)$; the way of constructing the chain complex associated with a space X is explained, for instance, in [Mau96]. In an intuitive sense, homology groups measure "n-dimensional holes" in topological spaces. H_0 measures the number of connected components of a space. The homology groups H_n measure higher dimensional connectedness. For instance, the n-sphere, S^n, has exactly one n-dimensional hole and no m-dimensional holes if $m \neq n$.

Moreover, let us highlight that homology groups are an *invariant*, see [Mau96]. That is to say, given two topological spaces if their homology groups are different then they are not homeomorphic.

The computation of homology groups of chain complexes is one of the central tasks in Homological Algebra. A general computing strategy consists in replacing a given chain complex by another simpler one, but with the same homological information. This point will be dealt with in this memoir in detail.

Let us finish this subsection with some additional definitions related to chain complexes.

Definition 1.11. A chain complex $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ is *acyclic* if $H_n(C_*) = 0$ for all n, that is to say, if $Z_n = B_n$ for every $n \in \mathbb{Z}$.

Definition 1.12. A chain complex $C_* = (C_n, d_{C_n})_{n\in\mathbb{N}}$ of $\mathbb{Z}$-modules is said to be *free* if C_n is a free $\mathbb{Z}$-module (a $\mathbb{Z}$-module which admits a basis) for each $n \in \mathbb{N}$.

Definition 1.13. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{Z}}$ be two chain complexes, a *chain complex morphism* between them is a family of module morphisms $(f_n)_{n\in\mathbb{Z}}$ of degree 0 between $(C_n)_{n\in\mathbb{Z}}$ and $(D_n)_{n\in\mathbb{Z}}$ such that $d_{D_n} \circ f_n = f_{n-1} \circ d_{C_n}$ for each $n \in \mathbb{Z}$.

Definition 1.14. An isomorphism between two chain complexes C_* and D_* is a chain complex morphism $f : C_* \to D_*$ such that there is a chain complex morphism inverse $f^{-1} : D_* \to C_*$ with the properties $f^{-1}f = id_{C_*}$ and $ff^{-1} = id_{D_*}$.

Definition 1.15. Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ be a chain complex. A chain complex $D_* = (D_n, d_{D_n})_{n\in\mathbb{Z}}$ is a *chain subcomplex* of C_* if

- D_n is a submodule of C_n, for all $n \in \mathbb{Z}$
- $d_{D_n} = d_{C_n}\,|_{D_*}$

The condition $d_{D_n} = d_{C_n} \mid_{D_*}$ means that the boundary operator of the chain subcomplex is just the differential operator of the larger chain complex restricted to its domain. We denote $D_* \subset C_*$ if D_* is a chain subcomplex of C_*.

As we said chain complexes are indexed over $\mathbb{Z}$. But in many situations we will work only with chain complex concentrated on three consecutive degrees.

$$C_0 \xleftarrow[d_1]{} C_1 \xleftarrow[d_2]{} C_2 \tag{1.1}$$

That is to say, three modules C_0, C_1, and C_2 plus two homomorphism $d_1 : C_1 \to C_0$ and $d_2 : C_2 \to C_1$ satisfying $d_2 d_1 = 0$.

From this data, we can complete an actual chain complex by adding null modules and null differential maps. Even if in standard mathematical texts these two concepts are usually identified, in our formalization setting, where types are essential, it is convenient to distinguish both notions. Therefore, we will call in the sequel 3-*truncated chain complex* to the first concept summarized in Expression 1.1.

Analogously, the concept is extended to chain complex morphisms (using the name 3-*truncated chain complex morphisms*). Nevertheless, if no confusion can arise we will use the terminology 'chain complex' to denote also 3-truncated chain complexes and the like.

1.1.2 Simplicial complexes

The notion of simplicial complex gives rise to the most elementary method to settle a connection between General Topology and Algebraic Topology. The notion of topological space is too *abstract* to perform computations. Simplicial complexes provide a purely combinatorial description of topological spaces which admit a triangulation. The computability of properties, such as homology groups, from a simplicial complex associated with a topological space is well-known and the algorithm uses simple linear algebra [Veb31]. Then, an algebraic topologist can construct from every sensible space (that is to say, a topological space which admits a triangulation) an equivalent simplicial complex, making computations easier. A complete description of these notions of Algebraic Topology (simplicial complexes, chain complexes, and so on) can be seen in [Koz07].

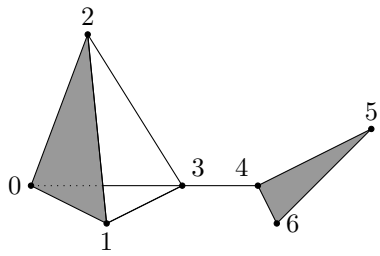


Figure 1.1: Butterfly Simplicial Complex

Let us start with some basic terminology. Let V be an ordered set, called the *vertex set*. An *(abstract) simplex* over V is any finite subset of V. An *(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 the subsets of α *faces*.

Definition 1.16. An *(ordered abstract) simplicial complex* over V is a set of simplices $\mathcal{K}$ over V such that it is closed by taking faces (subsets); that is to say, if $\alpha \in \mathcal{K}$ all the faces of α are in $\mathcal{K}$, too.

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

Example 1.17. Let us consider $V = (0, 1, 2, 3, 4, 5, 6)$.

The small simplicial complex drawn in Figure 1.1 is mathematically defined as the object:

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

and for instance, $S_2(\mathcal{K}) = \{(0,1,2), (4,5,6)\}$.

It is worth noting that simplicial complexes can be infinite. For instance if $V = \mathbb{N}$ and the simplicial complex $\mathcal{K}$ is $\{(n)\}_{n\in\mathbb{N}} \cup \{(n, n+1)\}_{n\in\mathbb{N}}$, the simplicial complex obtained can be seen as a simplicial representation of the real line.

Definition 1.18. A *facet* of a simplicial complex $\mathcal{K}$ over V is a maximal simplex with respect to the subset order $\subseteq$ among the simplices of $\mathcal{K}$.

Example 1.19. The facets of the simplicial complex depicted in Figure 1.1 are:

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

To construct the simplicial complex associated with a sequence of facets $\mathcal{F}$, we generate all the faces of the simplices of $\mathcal{F}$. Subsequently, if we perform the set union of all the faces we obtain the simplicial complex associated with $\mathcal{F}$.

Since the vertex set V is ordered, each simplex $(v_0, \ldots, v_n)$ defines uniquely a list $< v_0, \ldots, v_n >$, where $v_0 < v_1 < \ldots < v_n$. If no confusion can arise, we will denote both the simplex and the corresponding list by $(v_0, \ldots, v_n)$.

Definition 1.20. Let $\mathcal{K}$ be a simplicial complex over V. Let n and i be two integers such that $n \geq 1$ and $0 \leq i \leq n$. Then the *face operator* ∂_i^n is the linear map $\partial_i^n : S_n(\mathcal{K}) \to S_{n-1}(\mathcal{K})$ defined by:

$$\partial_i^n(v_0, \ldots, v_n) = (v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$$

the i-th vertex of the simplex is removed, so that an (n-1)-simplex is obtained.

Example 1.21. Let us show the behavior of the face operator using the simplicial complex of Figure 1.1. For instance, if we apply the face operator over the 2-simplex (0 1 2) (analogously for the rest of the simplices) we will obtain:

$$\partial_i(0,1,2) = \begin{cases} (1,2) & \text{if } i = 0 \\ (0,2) & \text{if } i = 1 \\ (0,1) & \text{if } i = 2 \end{cases}$$

Let us note that the face operator applied over the 2-simplex $(0, 1, 2)$ produces simplices with geometrical meaning (that are the three edges of the triangle $(0, 1, 2)$).

1.1.3 From simplicial complexes to chain complexes

Once we have defined the notions of simplicial complex and chain complex, we can define the link between them considering $\mathbb{Z}$ as the ring R; the most common case in Algebraic Topology.

Definition 1.22. Let $\mathcal{K}$ be a simplicial complex over V. Then *the chain complex* $C_*(\mathcal{K})$ *canonically associated with* $\mathcal{K}$ is defined as follows. The chain group $C_n(\mathcal{K})$ is the free $\mathbb{Z}$ module generated by the n-simplices of $\mathcal{K}$. In addition, let $(v_0, \ldots, v_n)$ be an n-simplex of $\mathcal{K}$, the differential map of this simplex is defined as:

$$d_{C_n}(v_0, \ldots, v_n) := \sum_{i=0}^{n} (-1)^i \partial_i^n(v_0, \ldots, v_n)$$

and then extended to any element of $C_n(K)$ by linearity.

In order to clarify the notion of chain complex canonically associated with a simplicial complex, let us present an example.

Example 1.23. Let $\mathcal{K}$ be the simplicial complex defined in Figure 1.1. The chain complex $C_*(\mathcal{K})$ canonically associated with $\mathcal{K}$ is:

$$\cdots \to 0 \to C_2(\mathcal{K}) \xrightarrow{d_2} C_1(\mathcal{K}) \xrightarrow{d_1} C_0(\mathcal{K}) \to 0 \to \cdots$$

where there are 3 associated chain groups:

- $C_0(\mathcal{K})$, the free $\mathbb{Z}$-module on the set of 0-simplices (vertices) $\{(0), (1), (2), (3), (4), (5), (6)\}$.
- $C_1(\mathcal{K})$, the free $\mathbb{Z}$-module on the set of 1-simplices (edges) $\{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3), (3,4), (4,5), (4,6), (5,6)\}$.
- $C_2(\mathcal{K})$, the free $\mathbb{Z}$-module on the set of 2-simplices (triangles) $\{(0,1,2), (4,5,6)\}$.

The elements of either of those groups C_p are linear integer combinations of the corresponding basis (set of σ_i's), i.e. elements of the form $\sum \lambda_i \sigma_i$, $\lambda_i \in \mathbb{Z}$.

The differential homomorphism is:

$$d_{C_n}(v_0, \ldots, v_n) := \sum_{i=0}^{n} (-1)^i (v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \tag{1.2}$$

For instance, $d_2(0,1,2) = (1,2) - (0,2) + (0,1)$.

From the previous definition, we can introduce a very useful concept for the computation of homology groups of simplicial complexes.

Definition 1.24. Let $\mathcal{K}$ be a simplicial complex, $C_n(\mathcal{K})$ is a free module and the n-simplices of $\mathcal{K}$ form the standard basis of it. Then, given an order, for all n we can represent the differential map $d_n : C_n(\mathcal{K}) \to C_{n-1}(\mathcal{K})$ relative to the standard basis of the chain groups as a $\mathbb{Z}$ matrix. Such a matrix is called the *n-th incidence matrix* of a simplicial complex.

It is worth mentioning that the entries of incidence matrices are 0's, 1's or -1's.

Example 1.25. If we impose a lexicographical order on the simplices of the same dimension of the simplicial complex depicted in Figure 1.1 (if $v = (a_0, \ldots, a_n)$ and $w = (b_0, \ldots, b_n)$ are n-simplices of the simplicial complex, then $v < w$ if $a_0 < b_0$, or if $a_0 = b_0$ and $a_1 < b_1$, or if $a_0 = b_0$ and $a_1 = b_1$ and $a_2 < b_2, \ldots$, or if $a_0 = b_0, \ldots, a_{n-1} = b_{n-1}$ and $a_n < b_n$), then its first incidence matrix is:

$$
\begin{array}{c}
 \\
(0) \\
(1) \\
(2) \\
(3) \\
(4) \\
(5) \\
(6)
\end{array}
\begin{array}{c}
\begin{array}{cccccccccc}
(0,1) & (0,2) & (0,3) & (1,2) & (1,3) & (2,3) & (3,4) & (4,5) & (4,6) & (5,6)
\end{array} \\
\left(\begin{array}{cccccccccc}
-1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}\right)
\end{array}
$$

The relevance of the incidence matrices of simplicial complexes lies in the fact that they can be used to compute the homology groups of the simplicial complex by means of a diagonalization process, as explained for instance in [Veb31]. In the literature, one of the algorithms to diagonalize matrices is the Smith Normal Form.

1.1.4 From digital images to simplicial complexes

The definitions presented in the previous subsections are classical definitions from Algebraic Topology. However, since our final goal consists in working with mathematical objects coming from digital images, we need to show how Algebraic Topology may be used in this context. There are several methods to construct a simplicial complex from a digital image [ADFQ03]. We are going to explain one of them. Roughly speaking, the chosen method obtains a sequence of facets from a digital image. Then, as we have explained in the previous subsections, we can generate the simplicial complex associated with the facets. So, we only need to explain how the facets from a digital image are obtained.

We work with two dimensional monochromatic images (2D images). An image can be represented by a finite 2-dimensional array of 1's and 0's in which the black pixels are represented by 1's and white pixels by 0's.

Let $\mathcal{I}$ be an image encoded as a 2-dimensional array of 1's and 0's. The simplicial complex associated with $\mathcal{I}$ will have as vertex set $V = (\mathbb{N}, \mathbb{N})$. The vertex

set V could be smaller and dependent on the concrete image, but $V = (\mathbb{N}, \mathbb{N})$ is general enough to deal with all the possible images. Let $p = (a, b)$ be the coordinates of a black pixel in $\mathcal{I}$. For each p we can obtain two 2-simplices which are two facets of the simplicial complex associated with $\mathcal{I}$. Namely, for each $p = (a, b)$ we obtain the following facets: $((a, b), (a + 1, b), (a + 1, b + 1))$ and $((a, b), (a, b + 1), (a + 1, b + 1))$. If we repeat the process for the coordinates of all the black pixels in $\mathcal{I}$, we obtain the facets of a simplicial complex associated with $\mathcal{I}$, let us called it $\mathcal{K}_{\mathcal{I}}$.

Example 1.26. Consider the image depicted in Figure 1.2. This image, $\mathcal{I}$, can be encoded by means of the 2-dimensional array: ((1,0),(0,1)). Then, with the previously explained process, we can obtain the facets of $\mathcal{K}_{\mathcal{I}}$. The coordinates of the black pixels are $(0, 0)$ and $(1, 1)$, so the facets that we obtain are:

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

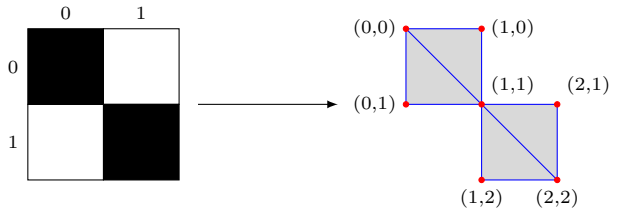


Figure 1.2: A digital image and its simplicial complex representation

We have presented a method to obtain a simplicial complex associated with a 2D image, this process can be generalized to higher-dimensional images [OS03].

Moreover, we can interpret properties about the digital image from its homology groups. 2D images can be interpreted as embedded in $\mathbb{R}^2$ (see [ADFQ03]); then its homology groups vanish for dimensions greater than 2 and they are torsion-free in 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. As the torsion groups are null, it is not necessary to work with $\mathbb{Z}$ and it is enough to take coefficients over the field $\mathbb{Z}_2$. This approach is usually followed when algebraic topology methods are applied to the study of digital images, see [GDMRSP05, GDR05]. Then, we work with a different definition of the

incidence matrices. In particular, since coefficients are in $\mathbb{Z}_2$, we do not have to deal with orientations of faces.

Thus, $\mathcal{K}$ will denote a simplicial complex over a finite set V and n an integer such that $n \geq 1$. The incidence matrix is now defined in the following way.

Definition 1. Let $\mathcal{K}$ be a simplicial complex with a concrete order on the simplices of the same dimension. Then, for all n, the *n-th incidence matrix of* $\mathcal{K}$ over the ring $\mathbb{Z}_2$, denoted by $M_n(\mathcal{K})$, is a matrix of size $m \times p$, where m is the cardinality of $S_{n-1}(\mathcal{K})$ (the number of $(n-1)$ simplices of $\mathcal{K}$) and p is cardinality of $S_n(\mathcal{K})$. Its coefficients $[a_i^j]$ are 1 if the i-th $(n-1)$-simplex is a face of the j-th n-simplex and 0 otherwise.

Note that each entry in the *n-th incidence matrix of* $\mathcal{K}$ over the ring $\mathbb{Z}_2$ is the absolute value of the corresponding entry in the *n-th incidence matrix of* $\mathcal{K}$ over the ring $\mathbb{Z}$.

Example 1.27. If we impose a lexicographical order on the simplices of the same dimension of the simplicial complex depicted in Figure 1.1, then its first incidence matrix over the ring $\mathbb{Z}_2$ is:

$$
\begin{array}{c}
 \\
(0) \\
(1) \\
(2) \\
(3) \\
(4) \\
(5) \\
(6)
\end{array}
\begin{array}{c}
\begin{array}{cccccccccc}
(0,1) & (0,2) & (0,3) & (1,2) & (1,3) & (2,3) & (3,4) & (4,5) & (4,6) & (5,6)
\end{array} \\
\left(\begin{array}{cccccccccc}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}\right)
\end{array}
$$

The incidence matrices of simplicial complexes come from the differential maps of the chain complexes canonically associated with them. These differentials satisfy the boundary condition $(d_{n-1} \circ d_n = 0)$.

Theorem 1. The product of the n-th incidence matrix of $\mathcal{K}$ over the ring $\mathbb{Z}_2$, $M_n(\mathcal{K})$, and the $(n+1)$-incidence matrix of $\mathcal{K}$ over the ring $\mathbb{Z}_2$, $M_{n+1}(\mathcal{K})$ is equal to the null matrix.

In order to compute the dimension of the homology groups associated with those chain complexes, we apply a diagonalization process to the incidence matrices, see [Veb31].

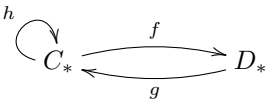
To sum up, our method to study digital images consists in using a well known Algebraic Topology tool: homology groups. As we have explained in the previous subsections, homology groups measure higher dimensional connectedness of topological spaces. In the case of 2D images, homology groups provides both the number of connected components and of holes of digital images. For instance, if $H_0 = \mathbb{Z} \oplus \mathbb{Z}$ then the images has 2 connected components. The first step to compute the homology groups of a digital image is the building of the simplicial complex associated with the image. Afterwards, we construct a chain complex from the simplicial complex. Finally, we obtain the homology groups of the previous chain complex which give us the properties of the initial image. As we work with chain complexes coming from images, their chain groups are finitely generated, then, the differential maps between them can be represented by matrices and the computation of homology groups is reduced to a diagonalization process.

1.1.5 Reductions

As we have seen, a central problem in our context consists in computing *homology groups* of topological spaces. By definition, the homology group of a space X is the one of its associated chain complex $C_*(X)$. If $C_*(X)$ can be described as a graded free abelian group with a *finite* number of generators at each degree; then, computing each homology group can be translated to a problem of diagonalizing certain matrices (see [Veb31]). So, we can assert that homology groups are computable in this finite type case.

Although we can compute the homology groups, in some cases, this can be very time consuming. We can think in reducing the chain complex preserving the homological properties. To this aim, we introduce the notion of reduction. This also allows us to study infinite spaces through finite ones (see [Ser87] and [Ser94]).

Definition 1.28. A *reduction* ρ between two chain complexes C_* and D_*, denoted by $\rho : C_* \Rightarrow D_*$, is a triple $\rho = (f, g, h)$



where f and g are chain complex morphisms, h is a graded group automorphism of degree $+1$ called homotopy operator, and the following relations are satisfied:

1) $f \circ g = \mathrm{id}_{D_*}$;

2) $d_C \circ h + h \circ d_C = \mathrm{id}_{C_*} - g \circ f$;

3) $f \circ h = 0; \quad h \circ g = 0; \quad h \circ h = 0.$

A particular reduction can be obtained from an isomorphism where $g = f^{-1}$ and the homotopy operator is null.

The importance of reductions lies in the following fact. Let $C_* \Rightarrow D_*$ be a reduction, then C_* is the direct sum of D_* and an acyclic chain complex; therefore the graded homology groups $H_*(C_*)$ and $H_*(D_*)$ are canonically isomorphic.

Very frequently, the top chain complex C_* is huge and the computation of its homology groups can take a lot of time. On the contrary, the bottom chain complex D_* has a reasonable size, then we can compute its homology groups much faster than in the case of C_*.

The composition of two reductions can be easily constructed.

Proposition 1.29. Let $\rho = (f, g, h) : C_* \Rightarrow D_*$ and $\rho' = (f', g', h') : D_* \Rightarrow E_*$ be two reductions. Another reduction $\rho'' = (f'', g'', h'') : C_* \Rightarrow E_*$ is defined by:

$$\begin{aligned} f'' &= f' \circ f \\ g'' &= g \circ g' \\ h'' &= h + g \circ h' \circ f \end{aligned}$$

Two of the most fundamental results dealing with reductions are the two *perturbation lemmas*. The main idea of both lemmas is that given a reduction, if we *perturb* one of the chain complexes then it is possible to perturb the other one; so, we obtain a new reduction between the *perturbed* chain complexes. The Easy Perturbation Lemma is easily obtained. The Basic Perturbation Lemma is not trivial at all. It was discovered by Shih Weishu [Shi62], although the abstract modern form was given by Ronnie Brown [Bro67].

Definition 1.30. Let $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$ be a chain complex. A *perturbation* δ of the differential map d is a collection of group morphisms $\delta = \{\delta_n : C_n \to C_{n-1}\}_{n\in\mathbb{Z}}$ such that the sum $d + \delta$ is also a differential map, that is to say, $(d + \delta) \circ (d + \delta) = 0$.

The perturbation δ produces a new chain complex $C'_* = (C_n, d_n + \delta_n)_{n\in\mathbb{Z}}$; called the *perturbed chain complex*.

Theorem 1.31 (Easy Perturbation Lemma, EPL)**.** Let $C_* = (C_n, d_{C_n})_{n\in\mathbb{Z}}$ and $D_* = (D_n, d_{D_n})_{n\in\mathbb{Z}}$ be two chain complexes, $\rho = (f, g, h) : C_* \Rightarrow D_*$ a reduction, and δ_{D_*} a perturbation of d_{D_*}. Then a new reduction $\rho' = (f', g', h') : C'_* \Rightarrow D'_*$ can be constructed where:

1) C'_* is the chain complex obtained from C_* by replacing the old differential map d_C by a perturbed differential map $d_C + g \circ \delta_D \circ f$;

2) the new chain complex D'_* is obtained from the chain complex D_* only by replacing the old differential map d_D by $d_D + \delta_D$;

3) $f' = f$;

4) $g' = g$;

5) $h' = h$.

The perturbation δ_{D_*} of the *small* chain complex D_* is naturally transferred (using the reduction ρ) to the *big* chain complex C_*, obtaining in this way a new reduction ρ' between the perturbed chain complexes.

If we consider a perturbation d_{C_*} of the top chain complex C_*, in general it is not possible to perturb the small chain complex D_* in such a way that there is a reduction between the perturbed chain complexes. As we will see, we need an additional hypothesis which is defined below.

Definition 1.32. An endomorphism $f : C \to C$ is *locally nilpotent* if given $x \in C$ there exists $m \in \mathbb{N}$ such that $f^m(x) = 0$.

Theorem 1.33 (Basic Perturbation Lemma, BPL [Bro67])**.** Let us consider a reduction $\rho = (f, g, h) : C_* \Rightarrow \widehat{C}_*$ between two chain complexes (C_*, d) and $(\widehat{C}_*, \widehat{d})$, and δ a perturbation of d. Furthermore, the composite function δh is assumed *locally nilpotent*. Then, a perturbation $\widehat{\delta}$ can be defined for the differential map $\widehat{d}$ and a new reduction $\rho' = (f', g', h') : (C_*, d+\delta) \Rightarrow (\widehat{C}_*, \widehat{d}+\widehat{\delta})$ can be constructed.

Following with the notation presented at the end of Subsection 1.1.1, it is similarly defined a 3-*truncated reduction* between 3-truncated chain complexes. Moreover, in order to define a reduction from a 3-truncated reduction identity maps

are added to the 3-truncated chain morphism and null maps to the 3-truncated homotopy operator.

In the following section we introduce a tool to build some reductions.

1.1.6 An algebraic setting of Discrete Morse Theory

In this section, we present admissible discrete vector fields, an instrumental notion in Discrete Morse theory. This theory was developed by Robin Forman in the 1920s [For98]. Discrete Morse theory is a tool for determining equivalences between topological spaces arising from discrete mathematical structures.

In our work we focus on studying properties of biomedical images. One of the problems of working with medical images is their size, so in this work, we reduce the image but preserving the homological properties. Namely, we focus on reducing the chain complex associated with the image through discrete vector fields. There are many ways to obtain a discrete vector field associated with a chain complex. Let us show the general idea of discrete vector fields with an example.

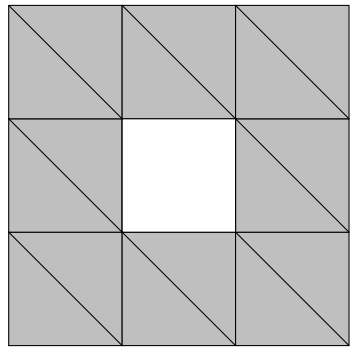


Figure 1.3: A simple digital image

Figure 1.3 shows an image with eight black pixels which has been triangulated as explained in Subsection 1.1.4 (two triangles for each black pixel), in this way we have built a simplicial complex. The bases of the corresponding simplicial complex are made of 16 vertices, 32 edges, and 16 triangles. So, the chain complex

associated with that simplicial complex is the following one.

$$0 \leftarrow \mathbb{Z}^{16} \leftarrow \mathbb{Z}^{32} \leftarrow \mathbb{Z}^{16} \leftarrow 0$$

We can reduce the amount of information using admissible discrete vector fields. Vector fields are a tool to cancel "useless" information. If we want to design an admissible vector field, we can decide that the unique allowed vectors are oriented leftward or downward because it is enough to avoid loops. In the example, the vector field can be seen in the left side of Figure 1.4. In that figure, only two critical cells remain, one vertex and one edge. So, the reduced chain complex is $0 \leftarrow \mathbb{Z} \leftarrow \mathbb{Z} \leftarrow 0$ with the null map between both copies of $\mathbb{Z}$. Therefore, $H_0 = \mathbb{Z}$ (one connected component) and $H_1 = \mathbb{Z}$ (one hole).

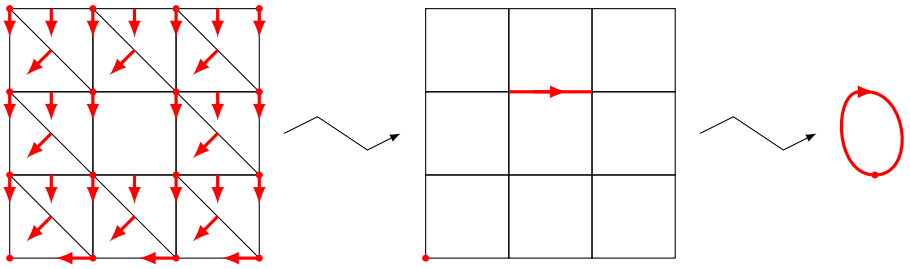


Figure 1.4: Discrete vector field

1.1.6.1 Algebraic discrete vector fields

The definitions and results introduced in this subsection have been extracted literally from [RS10, Chapter 2]. Let us start with some definitions and state the theorem of the reduction which is generated by an admissible discrete vector field.

Definition 1.34. An *algebraic cellular complex* (*ACC*) is a family $C = (C_p, d_p, \beta_p)_{p\in\mathbb{Z}}$ of free $\mathbb{Z}$-modules and boundary maps. Every C_p is a *chain group* and is provided with a distinguished $\mathbb{Z}$-basis β_p; every basis component $\sigma \in \beta_p$ is called a *p-cell*. The boundary map $d_p : C_p \to C_{p-1}$ is a $\mathbb{Z}$-linear map connecting two consecutive chain groups. The usual boundary condition $d_{p-1}d_p = 0$ is satisfied for every $p \in \mathbb{Z}$.

Henceforth, as we are working in a finite context, an ACC can be seen as a chain complex with a distinguished basis in every chain group. In addition, most often we omit the index of the differential map.

Definition 1.35. Let C be an ACC. A (p-1)-cell σ is said to be a *face* of a p-cell τ if the coefficient of α in $d\tau$ is not null. It is a *regular face* if this coefficient is 1 or -1.

Let us note that the regular property is relative because σ can be a regular face of τ but also a non-regular face of another simplex τ'.

Definition 1.36. A *discrete vector field* V on an algebraic cellular complex $C = (C_p, d_p, \beta_p)_{p\in\mathbb{Z}}$ is a collection of pairs $V = \{(\sigma_i, \tau_i)\}_{i\in\beta}$ satisfying the conditions:

1. Every σ_i is some p-cell, in which case the other corresponding component τ_i is a $(p+1)$-cell. The degree p depends on i and in general is not constant.
2. Every component σ_i is a regular face of the corresponding component τ_i.
3. A cell of C appears at most one time in the vector field: if $i \in \beta$ is fixed, then $\sigma_i \neq \sigma_j$, $\sigma_i \neq \tau_j$, $\tau_i \neq \sigma_j$, and $\tau_i \neq \tau_j$ for every $i \neq j \in \beta$.

Moreover, if $v = (\sigma_i, \tau_i)$ is a component of a vector field V, we call σ_i the source of v and τ_i the target of v.

Definition 1.37. A cell χ which does not appear in a discrete vector field $V = \{(\sigma_i, \tau_i)\}_{i\in\beta}$ is called a *critical cell*. A component (σ_i, τ_i) of the vector field V is a *p-vector* if σ_i is a p-cell.

In case of an ACC coming from a topological cellular complex, a vector field cancels "useless" cells in the underlying space, respecting the homotopy type. For instance $\partial\Delta_2$, which represents the boundary of a triangle, and the circle have the same homotopy type, which is described in Figure 1.5.

Example 1.38. The initial complex is made of three 0-cells 0, 1, and 2, and three 1-cells 01, 02, and 12. The drawn vector field is $V = \{(1, 01), (2, 02)\}$ and this vector defines a homotopy equivalence between $\partial\Delta_2$ and the minimal triangulation of the circle as a simplicial set. The last triangulation is made of

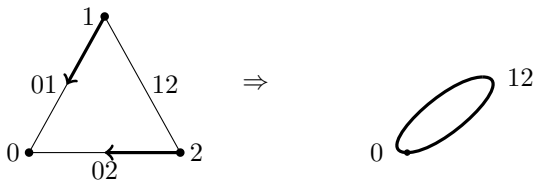


Figure 1.5: Reduction of $\partial\Delta_2$

the critical cells 0 and 12. It would be enough with knowing the homological properties of the last triangulation for knowing the properties of $\partial\Delta_2$ because these properties are the same.

However, not all the vector fields can be used to reduce chain complexes, we need an additional property called *admissibility*. The admissibility property avoids possible loops and infinitive paths.

Definition 1.39. If $V = \{(\sigma_i, \tau_i)\}_{i\in\beta}$ is a vector field on an algebraic cellular complex $C = (C_p, d_p, \beta_p)_p$, a *$V$-path of degree p* is a sequence $\pi = ((\sigma_{i_k}, \tau_{i_k}))_{0\leq k<m}$ satisfying:

1. Every pair $((\sigma_{i_k}, \tau_{i_k}))$ is a component of the vector field V and the cell τ_{i_k} is a p-cell.

2. For every $0 < k < m$, the component σ_{i_k} is a face of $\tau_{i_{k-1}}$, non necessarily regular, but different from $\sigma_{i_{k-1}}$.

If (σ, τ) is a component of a vector field, in general the cell τ has several faces different from σ, therefore the possible paths starting from a cell generate an oriented graph. Moreover, the *length* of the path $\pi = ((\sigma_{i_k}, \tau_{i_k}))_{0\leq k<m}$ is m.

Definition 1.40. A *discrete vector field* V on an algebraic cellular complex $C = (C_p, d_p, \beta_p)_{p\in\mathbb{Z}}$ is *admissible* if for every $p \in \mathbb{Z}$, a function $\lambda_p : \beta_p \to \mathbb{Z}$ is provided satisfying the following property: every V-path starting from $\sigma \in \beta_p$ has a length bounded by $\lambda_p(\sigma)$.

Example 1.41. In Figure 1.6 we have the same picture three times. Let us analyze if the built vector field on each image is an admissible discrete vector field.

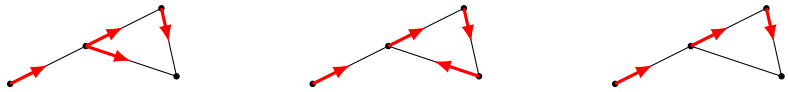


Figure 1.6: Different vector fields over a graph

In the left most image, the vector field is not a discrete vector field because there are two vectors with the same source (Property 3 of Definition 1.36 about a discrete vector field). In the center image, we can see that it is a discrete vector field but is not admissible because a cycle appears. Finally, the most right image shows an admissible discrete vector field.

1.1.6.2 Discrete vector fields over matrices

The following notions have also been extracted from [RS10, Chapter 2]. Let us note that we have defined the notion of vector field over chain complexes. In our case, the problem of computing a vector field over a chain complex is reduced to calculate a vector field over a matrix whose elements are in the field $\mathbb{Z}_2$ since we always work with finitely generated chain complex coming from 2D images.

The chain complex associated with a simplicial complex related to a 2D image has only three non-null chain groups, corresponding with the chain groups generated by 0-simplices (vertices), 1-simplices (edges), and 2-simplices (triangles).

$$\ldots \leftarrow 0 \leftarrow 0 \leftarrow C_0 \xleftarrow{d_1} C_1 \xleftarrow{d_2} C_2 \leftarrow 0 \leftarrow 0 \leftarrow \ldots$$

Then, we only have two non-null differential maps or two integer matrices that we will reduce with an admissible discrete vector field.

The following definition introduces a discrete vector field for a matrix whose elements are in $\mathbb{Z}$.

Definition 1.42. Let $M \in M_{m,n}(\mathbb{Z})$ with m rows and n columns. A vector field V for this matrix is a set of integers pairs $\{(a_i, b_i)\}_i$ satisfying these conditions:

1. $1 \leq a_i \leq m$ and $1 \leq b_i \leq n$.
2. The entry $M[a_i, b_i]$ is ± 1.
3. The indices a_i (respectively b_i) are pairwise different.

These three conditions are equivalent to the conditions of Definition 1.36 when we work with a matrix. It is worth noting that the first components of the vectors of V are row indexes of M and the second components are column indexes of M.

Using this definition, we can build a particular discrete vector field of a matrix M for a chain complex, which comes from a 2D image, where one of the differential maps is represented by M.

The following step consists in knowing when a vector field is admissible. As we work in a finite context, we only have to avoid possible loops. To this aim, a partial order between source cells is defined: the relation $a > a'$ is satisfied between source cells if and only if a V-path goes from a to a'. The partial order is built in the case of matrices as follows. Let V be a vector field for a matrix M and $1 \leq a, a' \leq m$ with $a \neq a'$, we can decide $a > a'$ if a vector (a, b) is present in V and the entry $M[a', b]$ is non-null. This means that a is a regular face and a' is an arbitrary face of b. Then the vector field V is admissible if and only if this relation transitively generates a partial order.

Once we have defined the notion of a discrete vector field and a reduction, we can state a theorem which given an admissible discrete vector field, V on an ACC C, defines a reduction where the small chain complex is generated by the critical p-cells.

Theorem 1.43. (Vector-Field Reduction Theorem) Let $C = (C_p, d_p, \beta_p)_p$ be an algebraic cellular complex and $V = \{\sigma_i, \beta_i\}_{i \in \beta}$ be an admissible discrete vector field on C. Then the vector field V defines a canonical reduction $\rho = (f, g, h) : (C_p, d_p) \Longrightarrow (C_p^c, d_p')$ where $C_p^c = \mathbb{Z}\left[\beta_p^c\right]$ is the free $\mathbb{Z}-$module generated by the critical cells.

Let us note that the larger the number of vectors which compose the vector field, the smaller the reduced chain complex.

In our case, as we work with finite type chain complexes whose differentials are represented as matrices, given an admissible discrete vector field for those matrices, we can construct new matrices taking into account the critical components. These smaller matrices define chain complexes which preserve the homological properties. This is the equivalent version of Theorem 1.43 for finite type chain complexes. A detailed description of the process can be seen in [RS10].

1.2 Coq and SSReflect

This section is devoted to provide an overview of Coq/SSReflect and the tools of this system which have been really important in our developments. In spite of being a brief introduction to this system, this description provides enough information to read the sections dedicated to Coq/SSReflect topics. A complete description of Coq/SSReflect can be found in [BC04] and [GM10].

Coq is a proof assistant based on the Calculus of Inductive Constructions (see [BC04]). Coq is not only a system for making formal proofs but also a functional programming language. This means that both the programs and their proofs of correctness can be implemented using the same language and logic. Following the Curry-Howard correspondence, types are statements of theorems and their proofs are programs. A tactic language is used to build proofs (or, equivalently, to write programs).

The SSReflect extension of Coq (see [GM10]) provides both an alternative set of tactics and a library. The library consists of definitions, lemmas and theorems which use advanced features of Coq, such as notations, implicit arguments, coercions and canonical structures. Its development was started by G. Gonthier during the formal proof of the Four Color Theorem [Gon08]. Moreover, it has been recently used to formalize the Odd Order Theorem [Mat12] which is part of the ongoing effort to formalize the classification of finite simple groups. SSReflect (for Small Scale Reflection) contains a large and well designed library of mathematical theories containing, among other things: group theory [GMR+07], algebraic structures [GGMR09], polynomials and matrices [Gon11], and so on. One of the choices made in the SSReflect library is never to rely on Coq axioms, which are statements that are admitted. However, SSReflect imposes some limitations on the user. In order to prevent definitions from being expanded during type checking some definitions are *locked* which means that computation on them are blocked. This implies that many definitions lack direct effective computation. Another limitation is that the algebraic hierarchy only captures discrete structures in order to enable equational reasoning.

In Coq and SSReflect, proofs are usually built through the interactive mode: the user writes definitions, statements and proofs in a window, and asks the software to verify them step by step. Along these verifications the agreeing and

complaining messages are shown in another window.

SSReflect introduces a new language for tactics which makes the development of proof scripts easier. The SSReflect additional tactics are few, but they can be combined with additional ones, i.e. tactic modifiers, such that one same tactic may cope with a wide range of similar situations. More details on the SSReflect tactics language and reflection techniques are presented in its manual [GM10].

A script has a linear structure composed of tactics. An example of such a script is the following

```
move=> n H.
case: n; first by done.
by rewrite muln_addr.
```

All the frequent bookkeeping operations which consists in moving, splitting, generalizing formulas from the context are regrouped in a single tactic `move`. In the first line of the example of the script, two arguments with the names `n` and `H` are moved from the conclusion to the context. The two tacticals `first` and `last` let the user restrict the application of a tactic to only the first or the last subgoal generated by the previous command. The tactic `by` is a tag closing tactics. The first case is proved with the basic closing tactic `done`. It involves other tactics as `sym_equal` (symmetry property), `trivial` and so on. It is uniquely used if the subgoal is easy to prove.

Finally, the tactic more used is `rewrite`. It comes with a concise syntax to accommodate in a single command all the possible combinations of conditional rewriting, unfolding of definition, simplifying, rewriting, and selecting specific both occurrences and patterns.

1.2.1 Inductive schemas

In the Coq system, the simplified induction principles are automatically generated for inductive types of the sort `Prop` and the maximal induction principle is automatically generated for the other inductive type definitions. The simplest inductive types are the enumerated types, used to describe finite sets. The natural numbers are built in two steps, inductively: first, zero is a natural; second, the

successor of a natural is also a natural.

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

Such a declaration defines several objects at once. First, a new `Set` is declared, with name `nat`. Then the constructors of `nat` are declared, called `O` and `S`. The constructor `O` has type `nat`, and it actually builds the first natural, i.e. zero. The second constructor of `nat`, `S`, expects a natural number, let us call it `n`, and returns a new natural number `(S n)` which is meant to be the successor of `n`. In SSReflect, `(S n)` is usually written `n.+1`.

Moreover, the system automatically adds several theorems and functions that make it possible to reason and to compute on data in this type. The usual theorem is `nat_ind`. It is also called the induction principle associated with the inductive definition.

```
nat_ind : forall P : nat -> Prop, P O ->
  (forall n : nat, P n -> P n.+1) -> forall n : nat, P n
```

In this way, when we want to apply induction (with the tactic `elim`) on a natural number the proof will be divided into two parts, if the value is zero or if it is the successor of another natural number. The inductive schemas associated with a type are not always useful to prove some properties. This type can depend on the behavior of other parameters.

In those cases, Coq allows us to define an inductive schema which fits our recursive programs. The generation of inductive schemas from recursive functions is automated in systems like ACL2 [KM]. In fact, it would not be difficult to translate the inductive schema generated in a proof from ACL2 to Coq to prove the same lemma in Coq.

In our development, we define methods using a *functional style*; that is, our programs are defined using *pattern-matching* and *recursion*. Therefore, in order to reason about our recursive functions, we need elimination principles which are fitted for them. Let us see an example to illustrate it. We define a function `subm` which takes as arguments a natural number `n`, and a sequence of sequences `M`, and removes the first `n`-rows of `M`. Let us note that `matZ2` represents a matrix as a sequence of sequences with the constraint of representing a matrix i.e., all the sequences have the same length.

```
Fixpoint subm (k: nat) (M: matZ2):=
  match k with
   |0 => M
   |S p => match M with
           |nil => nil
           |a::b => if (k == 1%N)
                    then a::b
                    else (subm p b)
         end
  end.
```

The inductive schema associated with `subm` is got as follows.

```
Functional Scheme subm_ind := Induction for subm Sort Prop.
```

Then, in order to reason about `subm`, we can apply this schema with the corresponding parameters using the instruction `functional induction`. In this schema, the natural number and the sequence of sequences are modified according to the definition of `subm`, if the natural number decreases in one unit the first row of the matrix is removed.

To define the induction principles, we use the tool presented in [BC02] which allows one to reason about complex recursive definitions since Coq does not directly generate elimination principles for complex recursive functions. Let us see how the tool presented in [BC02] works.

In some cases it is necessary to merge several inductive schemas to induct simultaneously on several variables. For instance, following with the example `subm`, let M be a matrix and M' be a submatrix of M where we have removed the $(k-1)$ first rows of M; then, we want to prove that $\forall j,\ M(i,j) = M'(i-k+1,j)$. This can be stated in Coq as follows.

```
Lemma Mij_subM (i k: nat) (M: matZ2):
  k <= i -> k != 0 ->
  let M' := (subm k M) in M i j == M' (i - k + 1) j.
```

To prove this lemma it is necessary to induct simultaneously on the parameters `i`, `k`, and `M`, but the inductive schema generated from `subm` only applies induction on `k` and `M`. Of course, the inductive schema of the type `nat` is not useful. Therefore,

we have to define a new recursive function, called `Mij_subM_rec`, to provide a proper inductive schema to prove this theorem.

```
Fixpoint Mij_subM_rec (i k: nat) (M: matZ2) :=
  match k with
  |0 => M
  |S p => match M with
          |nil => nil
          |hM::tM => if (k == 1)
                       then a::b
                     else (Mij_subM_rec p (i- 1) tM)
          end
  end.
```

Let us note that the recursive call decreases `k` and `i` in one unit and `M` deletes the first row.

1.2.2 Record types

A record type in Coq is a record structure which can store not only a bundle of objects, but also properties about such objects. In the following example the plane is defined as a set of points where each point is a pair of integer numbers which correspond with its coordinates.

```
Record plane: Set := point {abscissa : Z; ordinate : Z}.
```

In this manner, the interpretation of the various fields of the record is also more explicit that if one defines

```
Inductive plane: Set := point : Z -> Z -> plane.
```

In Coq, a record type is a generalization of sigma types: it is an inductive type with one constructor, plus a name to access the arguments of the constructor. Moreover, record types can include specifications as we can see in Figure 1.7. In this case, this type satisfies the associative property and has a two-sided identity element.

In addition, it allows us to define structures with inheritance. The identifier `monoid` is the name of the defined record. Apart from the properties and types

```
Record monoid := mkMonoid {
  car :> Type;
  dot : car -> car -> car;
  one : car;
  dot_assoc : associative dot;
  dot_one : forall x, dot one x = x;
  one_dot : forall x, dot x one = x}.
```

Figure 1.7: Example of record: `monoid`

which are defined in it, this record inherits the properties which are included in `Type`. This is got by the notation `car:> Type`. This coercion is applied to a monoid structure when a `Type` structure is expected, in a transparent way to the user. Note that here we define the first projection as a coercion from a monoid to its carrier.

1.2.3 Relevant SSReflect libraries in our development

SSReflect provides a set of libraries embedding definitions and properties for a variety of mathematical structures. In our formalization, it is worth mentioning the following libraries:

- `matrix.v`: this library formalizes matrix theory, determinant theory and matrix decompositions. In our development, this library is used to define incidence matrices, morphisms, and so on.
- `finset.v` and `fintype.v`: theory of finite sets and finite types. We use these libraries to define the basic concepts about simplicial complexes.
- `bigop.v`: generic indexed "big" operations, like $\sum_{i=0}^{n} f(i)$ or $\bigcup_{i \in I} f(i)$ and their properties, which are useful to deal with properties such as the incidence matrices product.
- `zmodp.v`: additive group and ring $\mathbb{Z}_p$, together with field properties when p is a prime. As we work with elements of the field $\mathbb{Z}_2$, we need this library.

- `vector.v`: finite dimensional abstract linear algebra. We use it to define the homology groups using linear applications.

- `ssralg.v`: main algebraic structures (Zmodule, Ring, etc) where for each one its type and its canonical properties, are defined.

- `fingraph.v`: theory of finite graphs.

1.2.4 The CoqEAL library

The CoqEAL library [DMS12b] has been built using a particular methodology on top of the SSReflect libraries. The aim of this library was the development of effective computer algebra programs with proofs of correctness. The methodology which is chosen for this goal is the following one. First, the mathematical algorithm is written using high-level data structures; this makes the proof of its correctness easier. Second, the algorithm is refined to an efficient version using again high-level data structures. Moreover, a proof of the equivalence between the two algorithms is provided. Finally, the efficient version of the algorithm is refined to simpler data structures closer to the machine representation. An example can be seen in [DMS12a]. This methodology has been also used in some of our developments.

We intensively use the `seqmatrix` file of the CoqEAL library. In this file, the low-level data type `seqmatrix` is defined to represent matrices as sequences of sequences. The relevance of using this type is that we can compute with it. This is essential for our work. This type can be transformed into an abstract matrix (`matrix`). This matrix, which is defined in the file `matrix.v`, is represented by finite functions over pairs of ordinals. The main functions are `seqmx_of_mx` and `mx_of_seqmx` which consists of changing the representation, from an abstract matrix to a sequence of sequences and the opposite. Apart from that, a huge number of functions are implemented with this structure related to the operations of matrices (`addseqmx`, `subseqmx`, `mulseqmx`, and so on). Moreover, we can easily take blocks of matrices.

1.3 A methodology to formalize algorithms

The "steep learning curve" is often mentioned as one of the big obstacles to wider adoption of Interactive Theorem Provings (ITPs) by professional mathematicians or industries like [Ben06]. Moreover, the complexity increases considerably when facing directly to this duty without any previous experience. In this work, we have used a methodology which tries to smooth such a learning curve when we formalize the correctness of algorithms in Coq.

In our developments the formally certified implementation of the algorithms have followed the methodology presented in [MÏ0]. This methodology can be split into these three steps:

1. Implement a version of our algorithms in *Haskell* [J+03], a *lazy* functional programming language.

2. Test properties about the Haskell programs using *QuickCheck* [CH00]. This tool allows one to test intensively properties about programs implemented in Haskell, in an automatic way.

3. Verify the programs using Coq [BGBP08], an *interactive* proof assistant, and its SSReflect library [GM10].

Using QuickCheck can be considered as a good starting point towards the formal verification of our programs. This provides us two advantages:

- A specification of the properties which must be satisfied by our programs is given (a necessary step in the formalization process which will be reused in Coq).

- The testing process can be useful in order to detect bugs in a quick and easy way, before trying a formal verification of our programs (a quite difficult task).

Let us illustrate this methodology with an example.

1.3.1 A Haskell program

The choice of Haskell to implement our programs was because both the code and the way of working is similar to the ones of the Coq formal proof management system which will be used to certify the correctness of the programs. The simple semantics of purely functional languages makes them easy to reason about its programs. Moreover, Haskell functions often satisfy simple algebraic properties, which can be used to prove correctness.

Let us introduce an example of implementation in Haskell. The method `swap` swaps the values `True` and `False` of a sequence of booleans.

```
swap :: [Bool] -> [Bool]
swap s = map not s
```

The function `map` receives a function f and a list. It returns a list where each element is the result of applying f to each element in the input list. In our example, the input function returns the logical negation of its boolean argument. Both `map` and `not` are already implemented into Haskell.

1.3.2 Testing with QuickCheck

As we have said, the use of QuickCheck is considered a good starting point towards the formal verification of our programs.

To illustrate its use, let us present a property about the `swap` function: given a sequence of booleans `s`, if we apply the `swap` method twice to `s`, we obtain `s` as a result. Let us represent this property as a Haskell function which is defined with the name `swap_swap`.

```
swap_swap :: [Bool] -> Bool
swap_swap s = swap (swap s) == s
```

The `==` is Haskell's equality test. The function `swap_swap` allows us to test for every possible argument if the property holds. The input parameters, in our example sequences of booleans, are generated randomly by the system and the property is tested for all them. To this aim, we use the function `quickCheck` which takes a property as a parameter and applies it to a large number of randomly generated arguments.

```
> quickCheck swap_swap
+ + + OK, passed 100 tests.
```

The above display must be read as follows. In the first line and according to the definition of `swap_swap`, we state that given a sequence of booleans, the output of `swap` fulfills the specification of the property called `swap_swap`. The second line, which is the result produced by QuickCheck when evaluating the statement of the first line, means that QuickCheck has generated 100 random values for `s`, checking that the property was true for all these cases.

In this way, we can check the properties which satisfy the output of a method. We can test our program in different ways but QuickCheck can be easily handled. Moreover, let us recall that this testing forces us to define a specification of the method which will be used also in the verification part.

In the cases where this property fails, QuickCheck reports a counter-example. For instance, if we mistakenly define that the first element of a sorted sequence is equal to the minimum element of the sequence

```
prop_minimum xs = (head (sort xs)) == minimum xs
```

then checking the property might appear

```
> quickCheck prop_minimum
* * * Failed!
  Exception: 'Prelude.head: empty list' (after 1 test): []
```

Then, we know at least that this property is not verified for all the sequences because the empty sequence does not hold it.

In general, many properties are satisfied under certain conditions. QuickCheck provides us an implication combinator to represent such preconditions. For instance, the previous property is defined correctly in the following way

```
prop_minimum xs =
  (not (null xs)) ==> (head (sort xs)) == minimum xs
```

The properties are checked by generating test cases which satisfy the precondition and checking the conclusion only for those. QuickCheck generates a limited number of candidate test cases. If 100 valid test cases among those candidates are not found, then we can see the number of successful tests.

1.3.3 Formalization in Coq/SSReflect

After testing our programs, and as final step to confirm their reliability, we can undertake the challenge of formally certify their correctness. To this aim, we must provide the closer data types and translate both the programs and the properties from Haskell to SSReflect, a task which is quite direct since these two systems are close. It is important to provide equivalent codes to ensure that the algorithm in Haskell works properly since the same algorithm is certified in SSReflect.

Let us see the implementation in SSReflect of the method shown in Subsection 1.3.1. The differences in the method `swap` comes from the own language. The functionalities of `map` and `not` are also defined in SSReflect. In this case, the corresponding `not` function in SSReflect is named `negb`. The method is defined as follows.

```
Definition swap (s:seq bool) := map negb s.
```

After the translation of the code and of the corresponding testing, we have to prove lemmas like the following one to verify the correctness of the algorithm.

```
Lemma swap_swap_seq : forall s, swap (swap s) = s.
```

This lemma corresponds with the function `swap_swap` which have been defined to test the function `swap` with QuickCheck.

Let us note that the properties specified in Haskell to test some methods will be also used in the verification step. In this way, we will be able to prove in a formal way our algorithms. In other words, we will verify the correctness of our developments, detecting and removing (previously by means of a testing process with QuickCheck) some bugs which could appear in the implementation.

1.3.4 Feedback loop

Along this section, we have introduced a methodology to formalize algorithms. At first sight, it could seem which is a linear methodology, but in some cases we need to return to the first step, the implementation of the algorithm. The reasons for this feedback loop, for instance could be, adding ending conditions in a loop function or detecting some bugs in the testing process. It is relevant to keep the Haskell code and the Coq/SSReflect one equivalent because we want

to verify formally the properties of the programs in Coq and consequently, using the Haskell code with a complete reliability. Then, this forces us to change the Haskell functions if some of them cannot be defined in SSReflect.

Let us introduce an example to see this idea. The following method receives two list of natural numbers $l1$ and $l2$ and returns a list which is the concatenation of every element of $l1$ with every element of $l2$.

```
pair_concat :: [[Int]] -> [[Int]] -> [[Int]]
pair_concat l1 [] = []
  ...
pair_concat (a:b)(c:d) = [c ++ a] ++ (pair_concat [a] d)
                              ++ (pair_concat b (c:d))
```

In order to define a recursive function the command `Fixpoint` in SSReflect is usually used. However, this case is not so simple. We cannot translate it directly because the system does not know that the method finishes. It does not exist any argument which decreases in all the cases. In some cases, the first list decreases but in other cases the second one is decreasing.

The easy way to deal with it is adding a parameter which always decreases. In this method, this iterator should be the addition of the lengths of every sequence as an initial value. This change would cause a modification in the implementation of Haskell because both codes have to be equivalent. Then, the algorithm in Haskell would have to be tested again with those changes. However, adding a new parameter is not the properest option. In order to solve it without using additional parameters in SSReflect, we can add a measure which specifies how the argument decreases. To employ this option we use the command `Function` instead of `Fixpoint` which is the usual one. Moreover, this option forces the function to having a unique input parameter. In this case, we have two sequences which are transformed into one parameter which is a pair of sequences. Finally, this method in SSReflect is defined in Figure 1.8.

Let us recall that this change has to be reflected in the Haskell code. Finally, this method in Haskell would be the function shown in Figure 1.9.

To be sure of that the methods keep behaving properly, we should test again with QuickCheck our Haskell programs to detect new bugs.

```
Function pair_concat (ords: ((seq (seq nat)) * (seq (seq nat)))
 {measure (fun ords =>((size (fst ords)) + (size (snd ords)))%N)}
  :orders:=
  match (fst ords), (snd ords) with
  | nil , _ => nil
    ...
  |a::b , c::d => ((c++a)::nil) ++ (pair_concat ((a::nil), d))
                  ++ (pair_concat (b, (c::d)))
  end.
```

Figure 1.8: `pair_concat` function in SSReflect

```
pair_concat :: ([[Int]],[[Int]]) -> [[Int]]
pair_concat (l1, []) = []
  ...
pair_concat (a:b)(c:d) = [c ++ a] ++ (pair_concat ([a], d))
                              ++ (pair_concat (b, (c:d)))
```

Figure 1.9: `pair_concat` function in Haskell

1.4 Mathematics to formalize

Applying the methodology introduced in Section 1.3 and using Coq/SSReflect presented in Section 1.2 we are going to formalize different issues:

- An algorithm to compute admissible discrete vector fields for chain complexes finitely generated (Chapter 2).
- A reduction of a finitely generated chain complex given an admissible discrete vector field (Chapter 3).
- BPL applied to finitely generated chain complexes (Chapter 4).
- Homology of digital images (Chapter 5).

Software systems which are used to study biomedical images must fulfill two requirements: efficiency and reliability. The former is required due to the huge

size of the images. The latter is necessary to ensure the correctness of the results (an important issue when dealing with biomedical applications).

In order to cope with the first requirement, our work will be based on techniques which allows us to reduce the amount of information without changing the homological properties [For98]. In order to deal with the second issue we will verify the correctness of our programs with the Coq theorem prover [BGBP08] and its extension SSReflect [GM10].

Chapter 2

Formalization of an algorithm to compute discrete vector fields

In this chapter, we formalize the Romero-Sergeraert's algorithm to compute an admissible discrete vector field from a matrix; this algorithm will be called from now on the RS algorithm. In particular, we will prove that the properties of an admissible discrete vector field are verified by the output of the RS algorithm. As the chain complex built from a digital image is of finite type, then its differential maps can be represented as matrices. So, the RS algorithm could be applied to these matrices. The admissible discrete vector field will be useful in order to reduce this chain complex. This was presented in [HPR12].

The first section is devoted to explain the RS algorithm. Following the methodology presented in Section 1.3, we proceed as follows to formalize such an algorithm. First, we have implemented the RS algorithm in the functional programming language Haskell [Hut07]. This issue is covered in Section 2.2. Second, the implementation of the algorithm will be tested in Section 2.3 by means of QuickCheck [CH00]. Finally, we certify our programs with the formal proof management system Coq [tdt10] using the SSReflect [GM10] extension of Coq as will be presented in Section 2.4. Moreover, we introduce a non deterministic

version of the reduction algorithm which provides us a high-level theory close to usual mathematics. This abstract algorithm generalizes the effective previous algorithm.

2.1 Romero-Sergeraert's algorithm (RS algorithm)

Most of the algorithms devoted to construct admissible discrete vector fields share the same goal, which is the construction of an admissible discrete vector field as big as possible, while keeping computation time reasonable; see for instance [Koz07, Chapter 11] and [LLT04]. Some of them return a vector field quickly, but others spend more time to compute it. Let us emphasize that the latter ones are notable because the search has been more thorough, so the number of vectors will be higher. We looked not only for an algorithm which gives us a big vector field but also does not spend too much time to compute it. The RS algorithm does not always build the biggest vector field possible, but experiments showed that the number of vectors is quite close to the biggest one. Furthermore, in many cases it returns the best possible vector field. On the other hand, it is fast enough to obtain the vector field in our application domain. Due to these reasons, the RS algorithm has been chosen to be used in order to make our computations.

The underlying idea of the RS algorithm is that given an admissible discrete vector field from a matrix, we try to enlarge it adding a new vector which preserves the properties of an admissible discrete vector field.

Concretely, this algorithm looks for a new vector in the following way. It consists of running the rows of the matrix $M_{m,n}$ in the usual reading order looking for entries whose value is $+1$ or -1. This entry will be a position of M, for instance (i, j), which will be a candidate vector to add to the admissible discrete vector field. This element will be selected whether it verifies the conditions given in the definition of a discrete vector field (see Definition 1.42). In particular, the third property which establishes that every row and column can appear only once in the vector field should be satisfied. The other properties are trivially verified since $M[i, j] = +1$ or -1 (this is the reason because the vector (i, j) has been selected) and $0 <= i < m$ and $0 <= j < n$ (since (i, j) is an entry of the matrix M). Moreover, in order to satisfy the admissibility property it is necessary to

avoid the creation of cycles. To this aim, we have to pay attention to the entries of the column j which are in a different position from i. For instance, if k is a position different from i of the column j whose entry is different from 0, a path from (i, j) to an element with k in the second component can be built. This will be represented through a particular relation $i > k$. In the process cycles will be avoided if an element is not related to itself through these relations, after taking its transitive closure.

If this vector satisfies the previous properties, it is added (and the generated relations are also added) and we have to continue looking for an entry whose value is $+1$ or -1 in the next row. On the contrary, if some of the conditions are not verified then a new possible vector will have to be looked for along the same row. This algorithm finishes when every row have already been visited.

We can define algorithmically the RS algorithm as follows.

Algorithm 2.1 (RS Algorithm)**.**

Input: a matrix M with coefficients in $\mathbb{Z}$.
Output: an admissible discrete vector field for M and a list of relations r.
Description:

1. Initialize the vector field V with the void vector field.
2. Initialize the relations r with the empty list, *nil*.
3. For every row i of M:
   - 3.1. Search the first entry of the row equal to $+1$ or -1, j.
     If no entry verifies this condition we return to Step 3 with the next row.
   - 3.2. `if` the properties of an admissible discrete vector field are preserved for (i, j).

     `then`:
     - Add (i, j) to V.
     - Add to r the corresponding relations generated from (i, j).
       $\forall k \neq i$ such as $M[k, j] \neq 0$, add $i > k$ to r and the necessary relations to complete the transitive closure of r.

- Go to the next row and repeat from Step 3.

`else`: look for the next entry of the row whose value is $+1$ or -1.

- `if` there is some.

  `then`: go to Step 3.2 with j the column of the entry whose value is $+1$ or -1.

  `else`: go to the next row and repeat from Step 3.

The heart of the algorithm is carried out in the point 3.2. which consists in:

3.2.1. $0 \leq i < m$ and $0 \leq j < n$;

3.2.2. $M[i,j] = \pm 1$;

3.2.3. i and j are different from the first and second components of V;

3.2.4. the relations do not generate cycles.

In order to check that the relations do not generate cycles we will proceed in the following way. In every step we add to the list of relations the transitive closure of this list. In particular, if two elements are related then they have to appear in one of the relations of the computed transitive closure. Therefore, so as to define the admissibility property will be enough with checking that no relation has repeated elements. An alternative method consists in storing only the relations and generating the transitive closure wherein it is necessary. However, every time a vector is selected to be part of the vector field this condition has to be verified. Therefore, the whole transitive closure would have to be computed many times.

In general, this algorithm can be applied over matrices with coefficients in a ring taking into account that the possible vectors will be only the elements whose value is a unit of the ring. Specifically, if we work with a field F, instead of a ring, every the non-null elements are units. In our particular case, we will work with the smallest field which is $\mathbb{Z}_2$ because homology groups of 2D digital images are torsion-free. Therefore, the selected vectors are the entries whose value is 1. From now on, we will work with $\mathbb{Z}_2$. Let us highlight that depending on the system wherein is worked the conditions or the tools are not the same. In particular, Haskell does not contain a data type for representing the $\mathbb{Z}_2$ ring. In this case, the *matrices* will be represented as a list of lists of $\mathbb{Z}$ in spite of the fact that the algorithm will be defined for a matrix over $\mathbb{Z}_2$. On the contrary, we can work

with the $\mathbb{Z}_2$ ring in SSReflect but we cannot use $\mathbb{Z}$ as the library about integer numbers of SSReflect is a work in progress as can be seen in [CM12].

In order to clarify how this algorithm works, let us construct an admissible discrete vector field from the following matrix.

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

We start with the void vector field $V = \{\}$. Running the successive rows, we find $M[0,0] = 1$, and we include the vector $(0,0)$ in V, obtaining $V = \{(0,0)\}$. Then, let us add the generated relations, in this case is just the relation $0 > 1$ because $M[1,0] \neq 0$ and the rest of entries $M[2,0]$ and $M[3,0]$ are null. So, it will be forbidden to incorporate the relation $1 > 0$ as a cycle would appear. So, we go on with the second row and find $M[1,0] = 1$. But we cannot add the vector $(1,0)$ because the row 0 and the column 0 cannot be used anymore due to the vector $(0,0)$ has been selected. Therefore, we look for the next element in the row whose value is 1 and we find the element $(1,1)$. But it cannot be incorporated as $M[0,1] \neq 0$ and then the relations $1 > 0$ and $0 > 1 > 0$ would be created. So, we continue and find the next element, $M[1,2] = 1$. This does not create any cycle, since the relations to add are $1 > 2$ and $1 > 3$. Then, we obtain $V = \{(1,2),(0,0)\}$. Running the next row, the first element equal to 1 is in the position $(2,2)$, but we cannot include it because $M[1,2] \neq 0$ therefore, the relation $2 > 1$ would be added and a cycle would appear. Therefore, we try with the last element of this row $(2,3)$. No relation is generated in this case because in this column the only non-null element is in the chosen position. So, $V = \{(2,3),(1,2),(0,0)\}$. Finally, we run the last row. The elements that could be added are $(3,1),(3,2)$, but in both cases we would have to append the relation $3 > 1$. This would generate a cycle with one of the previous restrictions, $1 > 3$. So, we obtain $V = \{(2,3),(1,2),(0,0)\}$ and the relations are: $0 > 1$, $1 > 2$, $1 > 3$, $0 > 1 > 2$, and $0 > 1 > 3$ including the relations created to obtain the transitive closure.

2.1.1 Realignment of an admissible discrete vector field

Let us mention that it is extremely relevant sorting the admissible discrete vector field because this will let us order the matrix as a previous step to reduce it. This matrix will have the feature of that the top-left matrix is a lower triangular matrix with $1's$ on the main diagonal.

For every vector (a, b), it is computed the value of the function $\lambda(a)$ which gives us the longer path from a taking into account the transitivity of the partial relations according to Definition 1.40. In our case, as we build the transitive closure, it is the maximum length of the relations which start with a. Then we sort the vector field by the values of λ in decreasing order.

In the previous example, we are reordering the vector field $V = \{(2,3),(1,2),(0,0)\}$ with the relations $0 > 1$, $1 > 2$, $1 > 3$, $0 > 1 > 2$, and $0 > 1 > 3$. For example, $\lambda(0) = 3$ because of the fact that the largest paths starting with 0 are $0 > 1 > 2$ and $0 > 1 > 3$, in any case the length is 3. In a similar way, $\lambda(1) = 2$ and $\lambda(2) = 1$. Then, the ordered vector field with the values of λ decreasingly is $V = <(0,0),(1,2),(2,3)>$.

2.2 Implementation in Haskell

Let us explain how we have implemented the algorithm presented in previous section in the lazy functional programming language Haskell. In particular, let us introduce the main data types and comment the main methods.

First, the *matrices* will be represented as a list of lists of integer numbers. Then, a *discrete vector field* will be represented as a list of pairs of natural numbers and the *partial relations* as lists of natural numbers. For instance, a partial relation $i > j$ will be represented by `[i,j]`.

Frow now on, let us define the main methods to implement Algorithm 2.1 taking into account the fact that the integer matrices are only consisted of $0's$ and $1's$. For each method, we will provide the input, the output, a brief description in some cases and a small example to clarify the meaning of the method. The first function will be in charge of selecting the entries of the matrix whose value is 1. This will be applied in the step 3.1. of the Algorithm 2.1.

firstElem1 k ls *[Function]*

Input: a natural number k and a list over $\mathbb{Z}$, *ls*, where all the elements are 0 or 1.
Output: a natural number which is the first position of the list which is higher or equal than k whose value is 1, or the size of *ls* if no element verifies the condition.

```
firstElem1 2 [1,0,0,1,1] ✠
3
```

The above display must be read as follows. The maltese cross (in fact not visible on the user screen) marks in this text the end of the Haskell statement, in this case `firstElem1 2 [1,0,0,1,1]`. The result of the evaluation appears in the next line.

Afterwards, we implement the function which checks if a vector can be added to a discrete vector field. This corresponds with the steps 3.2.1. and 3.2.3. of the Algorithm 2.1.

canAddCvd m n a b vf *[Function]*

Input: four natural numbers m, n, a, and b and a vector field *vf*.
Output: a boolean value which is `True` if (a, b) can be added to *vf*, and `False` otherwise.

The parameters m and n refer to the dimensions of the matrix M whose vector field associated is *vf*. This function checks if the vector (a, b) verifies the properties 1 and 3 of Definition 1.42. The second property is verified taking into account the way of choosing a vector from a matrix (only the elements whose value is 1 obtained by the `firstElem1` function). In order to check the first property it is needed that the vector (a, b) corresponds with an element of the matrix M, i.e., $0 \leq a < m$ and $0 \leq b < n$. Moreover, the third property is satisfied if neither a is in the first components of the pairs of *vf* nor b is in the second components. If some of these conditions are false, the vector (a, b) cannot be added, so the `canAddCvd` function returns `False`.

```
canAddCvd 3 4 2 1 [(0,2),(1,1)] ✠
False
```

```
canAddCvd 3 4 2 1 [(0,2),(1,0)] ✠
True
```

The following functions (`canAddOrder`, `canAOrder`, and `canAddOrders`) are implemented to check that the relations which are generated when a new vector is added do not create cycles, that is to say, the admissibility property is verified (step 3.2.4 of Algorithm 2.1). In our implementation, the list of relations always consists of the transitive closure of the relations. For instance, if $1 > 2$ and $2 > 4$ belong to the list of relations then the relation $1 > 2 > 4$ will be included too. Let us recall that the admissibility property checks that there are not repeated elements in any of the relations.

canAddOrder i j r *[Function]*

Input: two natural numbers i and j and a list of relations r.
Output: a boolean value which is `True` if the relation $i > j$ does not generate cycles with any of the partial relation which already exist in r, and `False` otherwise.

For every relation *r1* in r, it is checked that the new relation $i > j$ does not generate cycles, concretely, if the element j appears before the element i in *r1*. For instance, if *r1* is $k > j > l > i$ the function will return `False`.

```
canAddOrder 1 2 [[0,1],[2,3]] ✠
True
```

```
canAddOrder 2 1 [[0,3],[1,2]] ✠
False
```

But the `canAddOrder` function is not enough to be sure that this new vector does not generate cycles. For instance, if the new relation is $i > j$ and the list of relations r is composed by $j > k > l$ and $m > l > i$ then `canAddOrder` returns `True` because i and j does not appear in any partial relation at the same time. But in fact, it is not possible to add it because this new relation $m > l > i > j > k > l$

appears in the transitive closure of these three partial relations. In order to avoid this problem, we are going to check that there are not repeated elements in the concatenation of partial relations which start with j with partial relations which finish with i. To this aim, the following function has been defined.

canAOrder i j r *[Function]*

Input: two natural numbers i and j which represent a partial relation and a list of relations r.
Output: a boolean value which is `True` if the relation $i > j$ can be added to a list of relations r checking that every concatenation of a list of r which starts with j with a list of r which finishes with i has not repeated elements, and `False` otherwise.

```
canAOrder 1 2 [[0,1],[2,0]] ✠
False
canAOrder 1 2 [[0,1],[2,3]] ✠
True
```

The first example is `False` because if we could add the relation $1 > 2$, the relation $0 > 1 > 2 > 0$ would have repeated elements.

The following function involves the `canAddOrder` and `canAOrder` functions.

canAddOrders i k col r *[Function]*

Input: two natural numbers i and k, a list over $\mathbb{Z}$ (a column), *col*, where all the elements are 0 or 1, and a list of relations r. The first natural number is the first component of the selected vector and the latter one k is an iterator which will be used for running *col*.
Output: a boolean value which is `True` if the list of relations which will be generated from *col* could be added to r without generating cycles.

As we have explained in Section 2.1, when a vector is selected we pay attention to the column which it belongs to. This column will be run looking for the positions whose value is different from 0 because they imply that a new relation should be added.

```
canAddOrders 2 0 [1,0,1,1,0] [[0,1]] ✠
True
```

```
canAddOrders 2 0 [1,0,0,0,0] [[0,1],[1,2],[0,1,2]] ✠
False
```

```
canAddOrders 2 0 [0,1,0,0,0] [[0,1],[1,2],[0,1,2]] ✠
False
```

In the last example, the function returns `False` because the relation $2 > 1$ would be added to the list of relations wherein the restriction $1 > 2$ is included.

The next method consists in adding a vector to a discrete vector field. This is used when the step 3.2. returns `True`.

addcvd a b vf *[Function]*

Input: two natural numbers a and b which represent a vector (a, b), and a vector field *vf*.
Output: a vector field which is the concatenation of *vf* and (a, b). This function is called after checking that this new vector can be added by means of the function `canAddCvd`. Then the vector (a, b) can always be added.

```
addcvd 2 4 [(0,1),(1,3)] ✠
[(0,1),(1,3),(2,4)]
```

Another interesting point is the way of adding a new partial relation to a list of partial relations when it can be added (when the returned value of the `canAddOrders` function is `True`). The method consists in generating the transitive closure of the relations. This task is carried out by the `addOrder_orders`, `addOrder_concat` and `addOrders` functions.

addOrder_orders i j r *[Function]*

Input: two natural numbers i and j which represent the relation $i > j$ and a list of relations r.
Output: a list of relations r adding the concatenations of $i > j$ with every relation of r which starts with j and finishes with i.

This function will take into account the lists which start with j and finish with i. This function is used after checking that the corresponding call to the `canAddOrders` function, namely the `canAddOrder` function, returns `True`. Otherwise, the chosen vector could not be added.

```
addOrder_orders 2 3 [[0,1]] ✠
[[0,1],[2,3]]
```

```
addOrder_orders 2 3 [[0,2]] ✠
[[0,2],[2,3],[0,2,3]]
```

```
addOrder_orders 2 3 [[1,2]] ✠
[[1,2],[2,3],[1,2,3]]
```

```
addOrder_orders 2 3 [[1,2],[3,4]] ✠
[[1,2],[3,4],[1,2,3],[2,3,4]]
```

addOrder_concat l1 l2 *[Function]*

Input: two lists of relations *l1* and *l2*.
Output: the list of relations obtained from concatenating every list of *l1* with every list of *l2*.

This function adds new relations that the `addOrder_orders` function does not. For instance, if we have the following relations: $l > i$ and $j > m$ and we add the partial relation $i > j$, then the `addOrder_orders` function adds $l > i > j$ and $i > j > m$ but the relation $l > i > j > m$ would not be added. Let us note that this relation is obtained concatenating a relation which finishes with i with a relation which starts with j. The `addOrder_concat` function will be used to obtain this type of relations which is in charge of completing the computation of the transitive closure.

```
addOrder_concat [[0,1],[2,1]][[6,4],[3,5]] ✠
[[0,1,6,4],[0,1,3,5],[2,1,6,4],[2,1,3,5]]
```

The following function completes the transitive closure of the relations every time that a new relation is added. So, this function involves both the `addOrder_orders` function and `addOrder_concat` function to build it.

addOrders i k col r *[Function]*

Input: two natural numbers i and k (for running the column), a list over $\mathbb{Z}$ (a column), *col*, where all the elements are 0 or 1 and a list of relations r.
Output: a list of relations r adding the relations which come from computing the transitive closure generated by the relations in the way $i > k$ (which are built with *col*) added to r.

```
addOrders 2 0 [1,0,1,1][[0,1],[5,4]] ✠
[[0,1],[5,4],[2,0],[2,3],[2,0,1]]
```

```
addOrders 2 3 [0,1,1,1] [[0,2],[3,4]] ✠
[[0,2],[3,4],[2,1][2,3],[0,2,1],[0,2,3],[2,3,4],[0,2,3,4]]
```

Finally, we introduce the `genDvfOrders` function which corresponds with Algorithm 2.1.

genDvfOrders i j M M1 vf r *[Function]*

Input: two natural numbers i, j, two list of lists M, $M1$, a vector field vf, and a list of relations r.
Output: a pair of lists. The former one is an admissible discrete vector field. The latter one is the list of partial relations.

This function is essential and uses the previous ones. The two first parameters represent the row i and the column j of the element to be processed. M is the initial matrix and $M1$ is a submatrix of M obtained removing the i first rows from M. At the beginning, $i = 0$ so $M1$ will be equal to M. The task of looking for a possible vector in every row is carried out with $M1$. We remove the first row of this matrix every time a new vector is added (if it is possible) or if no candidate to be a vector can be selected. So, the next vector to add will be in the first row of the submatrix. Let us note that we have to keep the initial matrix M to create the partial relations since we will need the whole columns to add them (as we have explained in Section 2.1). Finally, the two last parameters are an admissible discrete vector field vf and a list of partial relations r.

Let us note that the call to the `genDvfOrders` function starts with an empty vector field and an empty list of relations. Moreover, this method starts in the entry $(0, 0)$ of M (with the two first parameters).

```
genDvfOrders 0 0 [[1,0,1,1],[0,0,1,0],[1,1,0,1]]
         [[1,0,1,1],[0,0,1,0],[1,1,0,1]] [] [] ✠
([(0,0),(1,2),(2,1)],[[0,2],[1,0],[1,0,2]])
```

The components of the pair obtained by `genDvfOrders` are projected by the following functions.

genDvf M *[Function]*

Input: a list of lists M whose elements are over $\mathbb{Z}$.
Output: an admissible discrete vector field obtained taking the first component of the output of `genDvfOrders`.

```
genDvf [[1,0,1,1],[0,0,1,0],[1,1,0,1]] ✠
[(0,0),(1,2),(2,1)]
```

genOrders M *[Function]*

Input: a list of lists M whose elements are over $\mathbb{Z}$.
Output: a list of relations. Namely, an empty list if M is empty, and the relations generated during the computation of the vector field from M (using `genDvfOrders`) otherwise.

```
genOrders [[1,0,1,1],[0,0,1,0],[1,1,0,1]] ✠
[[0,2],[1,0],[1,0,2]]
```

2.2.1 Realignment of an admissible discrete vector field

Up to now, we have functions to build both an admissible discrete vector field and their partial relations. Finally, we only have to sort this admissible discrete vector field taking into account these partial relations. The necessary definition to compute an ordered and admissible discrete vector field is introduced.

dvford M *[Function]*

Input: a list of lists M whose elements are over $\mathbb{Z}$.
Output: an ordered and admissible discrete vector field. The vector field computed by `genDvf` is sorted.

```
dvford [[1,0,1,1],[0,0,1,0],[1,1,0,1]] ✠
[(1,2),(0,0),(2,1)]
```

2.3 Testing

The previous section was mainly devoted to present an implementation of the algorithm which provides an ordered and admissible discrete vector field from a matrix over $\mathbb{Z}$. However, this implementation could contain some bugs. In order to handle this issue, it is not enough with creating some small examples which can be solved by hand and checking that the result obtained by the programs is the expected. We could use some tools similar to QuickCheck such as *HUnit* (a traditional *xUnit* testing framework for unit testing for Haskell, see [Her]) which helps us to test batteries of examples. Even if we cannot guarantee that the implementation of our programs was correct, we can increase the confidence in their results with this intensive testing.

Besides, using QuickCheck can be considered as a good starting point towards the formal verification of our programs. On the one hand, before trying a formal verification of our programs (a quite difficult task) we test the properties hold in a large number of randomly generated cases. This process can be useful in order to detect bugs. Moreover, we obtain the specification of the properties which must be verified formally in Coq/SSReflect thanks to this process. In order to test our programs, first we have to think and implement the functions which specify the properties to test.

In this section, we will show how using QuickCheck in some examples. We focus on testing the implementation of the algorithm which was presented in Section 2.1.

2.3.1 Testing with QuickCheck

QuickCheck is an easy to use framework to test programs in a battery of examples. It requires us to write a specification of our code by defining properties which our algorithm has to satisfy, as a first step towards the testing of the algorithm (against those properties). Then it generates random sample data to verify that the properties hold. This higher level view of testing has a good match with Haskell.

First of all, let us recall the conditions which any admissible discrete vector field has to verify i.e., the conditions related to the vector field and its relations:

1. $0 \leq a_i < m$ and $0 \leq b_i < n$.
2. $\forall\ i, M[a_i, b_i] = 1$.
3. $(a_i)_i$ (resp. $(b_i)_i$) are pairwise different.
4. V is ordered regarding the maximum length of the relations.
5. r does not have any cycle (admissibility property).
   - $\forall\ k\ v,\ v = (a, b) \in V \wedge M[a, k] \neq 0 \rightarrow [a, k] \in r$.
   - $\forall\ x\ y,\ last\ x = head\ y \rightarrow x + +(tail\ y) \in r$.
   - $\forall\ r1,\ r1 \in r \rightarrow uniq\ r$.

Let us note that the admissibility property is detailed with the three last conditions. The first one checks that the relations added by the vector field are in r. The second one is in charge of checking that the relations added using the transitive closure also have to belong to r. Finally, taking into account our implementation, the admissibility property (see Definition 1.40) is verified if every relation of r has not repeated elements. Let us note that in the last property, `uniq r` means that a relation `r` has not repeated elements.

Then, we introduce as an example a pair of definitions which are needed to specify the properties which we have just presented to test the RS algorithm which computes an admissible discrete vector field. For each one, we will provide the input, the property checked, and their Haskell code. As in the implementation the matrices will be represented as list of lists over $\mathbb{Z}$ since the Haskell language

is exactly the same one of QuickCheck. Let us highlight that some of these functions are similar to the functions used in the algorithm. For instance, the `canAddCvd` function fulfills the properties so that a vector belongs to a vector field. However, we have to check that every vector which belongs to a vector field verifies this property. It is worth noting that the following functions are test functions, namely, functions which return a boolean value which is true if the property is fulfilled and false otherwise. The first function is in charge of testing the second property and the other one is focused on testing the second condition of the admissibility property. Let us explain the notation for the function `nth` which appears in `compijCvd`. The function `nth` takes three parameters as input: `x0` (a default element), `s` (a sequence) and `i` (a natural number); and returns the `i`-th element of the sequence `s` if `i` is smaller than the length of `s`; otherwise, it returns `x0`.

compijCvd vf M *[Function]*

Input: a vector field vf and a a list of lists M over $\mathbb{Z}$ where all the elements are 0 or 1.
Property checked: $\forall (i,j) \in vf$, $M[i,j] = 1$.

The definition of `compijCvd` in Haskell code is as follows.

```
compijCvd dvf M =
  all (\x -> (nth 0 (nth nil M (fst x)) (snd x)) == 1).
```

prop_cat r *[Function]*

Input: a list of relations r.
Property checked: $\forall$ a, $b2$: $relation$, $b1$: nat, a $\in r \wedge (b1::b2) \in r \wedge$ last $a = b1$ then $a +\!+ b2 \in r$.

Its code in Haskell is as follows.

```
prop_cat r = all (\x -> all (\y -> (isinSeq (x ++ tail y) r))
  (filter (\y0 -> (last (tail x)) == (head y0)) r)) r
```

These functions and others which specify the properties to verify have been grouped in Haskell by means of a function called `isOrdAdmVecfield`. To test in QuickCheck that our implementation of the RS algorithm fulfills the specification

given in `isOrdAdmVecfield`, the following *property definition*, using QuickCheck terminology, is defined.

```
condOrdAdmVecfield M = isOrdAdmVecfield M (dvford M) (genOrders M)
```

Let `M` be a matrix, the definition of `condOrdAdmVecfield` states that the ordered and admissible discrete vector field and the relations from `M` produced by the outputs of `dvford` and `genOrders` fulfill the specification of the property called `isOrdAdmVecfield`.

Let us note that the matrix `M` generated randomly can be any list of lists over $\mathbb{Z}$, then it is quite likely that `M` is not only consisting of $0's$ and $1's$, or that it does not satisfy the condition of that every row has the same size. In order to deal with this drawback, it is necessary to redefine the *property definition* wherein these preconditions about the input parameters are satisfied. Moreover, you can define new functions to transform a data type in another one instead of adding other precondition over this parameter.

```
condOrdAdmVecfield M =
  let M01 = (fill_equalsize (matrixM01 M)) in
   (M01 /= [[]]) ==>
   isOrdAdmVecfield M01 (dvford M01) (genOrders M01)
```

Let us emphasize that we are not losing the randomness of the matrices, because we generate randomly lists of lists of integer numbers. The `matrixM01` function becomes a list over $\mathbb{Z}$ in a list over $\mathbb{Z}_2$, in the following way; if the integer is a even number, then it is considered as a 0 and in another case, 1. Then `fill_equalsize` is in charge of filling the list of lists with $0's$ and $1's$ randomly, so that every row has the same number of elements. In this way we can work with well built matrices over $\mathbb{Z}_2$ keeping the randomness.

Now, we can test whether `condOrdAdmVecfield` satisfies such a property.

```
> quickCheck condOrdAdmVecfield ✠
 + + + OK, passed 100 tests.
```

QuickCheck works in the following way: for every parameter which is generated randomly by QuickCheck, it is checked that the preconditions of the function to test are verified. If these conditions are fulfilled the system checks if

the property is satisfied. The system generates randomly lists of lists for the property `condOrdAdmVecfield` and checks that the precondition (`MO1 /= [[]]`) is satisfied. For every list which verifies the precondition the return value of `isOrdAdmVecfield` should be true. Finally, the result produced by QuickCheck when evaluating this statement means 100 random values for `M` have been generated, which verify the preconditions, checking that the property was true for all these cases.

2.4 Verification

After testing our programs, and as a final step to ensure their correctness, we can undertake the challenge of formally verifying them. For this task we are going to use Coq/SSReflect.

First of all, we define in the next subsection the data types related to our programs which are mainly effective matrices, vector fields, and relations. Afterwards, we translate both the programs and the properties, which were specified during the testing of the programs, from Haskell to Coq. This task is quite direct since these two systems are close (as it will be explained in Subsection 2.4.1). Subsection 2.4.2 is devoted to verify the correctness of `dvford`. Namely, we will define a `Vecfieldadm` function which receives a matrix over $\mathbb{Z}_2$, a vector field and some relations, and checks if the properties tested in Section 2.3 are verified. Then we prove that every property included in `Vecfieldadm` is verified for our admissible discrete vector field and their relations. Finally, we prove that the output produced by `dvford` satisfies the properties specified in `Vecfieldadm`.

2.4.1 Implementation in SSReflect

First of all, we define the data types of the main structures which are involved in our development. As we work with matrices over $\mathbb{Z}_2$, we have to define the field $\mathbb{Z}_2$. In SSReflect, the field $\mathbb{Z}_p$ is defined, $\forall p \in \mathbb{N}$, with the constructor `Fp_fieldType`. So, we only need an instance of it with $p = 2$.

```
Definition Z2 := Fp_fieldType 2.
```

Let us note that 0 and 1 over $\mathbb{Z}_2$ are represented as `0%R` and `1%R`, respectively. Let us highlight that the matrices are represented in SSReflect by finite functions over pairs of ordinals (the indices):

```
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}
```

This encoding makes many properties easy to derive, but it is inefficient for evaluation. Indeed, finite functions are internally represented by their graph which has to be traversed linearly whenever the function is evaluated. Moreover, having the size of matrices encoded in their type allows us to state concise lemmas without explicit side conditions. However, our main aim is computing, despite the proofs are more complicated. Following with this idea, we will keep the data types as close as possible to the Haskell ones. Therefore, a matrix will be encoded as a list of lists of $\mathbb{Z}_2$. It is worth noting that lists are called sequences in SSReflect, and we will use this terminology from now on each time that we refer to SSReflect lists. In addition, a well defined matrix satisfies the condition that every row has the same size. So, a sequence of sequences of $\mathbb{Z}_2$ is a matrix if the sequence is empty or if the sequence has a good shape. To define it, we use the `rowseqmx` function which given a matrix M and a integer i returns the row $M[i]$. This function is defined in [DMS].

```
Definition seqZ2 := seq Z2.
Record matZ2:=
  {M:> seq (seq Z2);
   m:nat;
   n:nat;
   is_matrix: M = [::] \/
     [/\ m = size M & forall i, i < m -> size (rowseqmx M i) = n]
  }.
```

Moreover, a vector field will be represented by means of a list of pairs of natural numbers.

```
Definition vectorfield:= seq (prod nat nat).
```

Finally, a relation $i > j$ will be represented as a sequence of natural numbers `(i::j::nil)`. So, the partial relations will be encoded as a sequence of sequences of natural numbers.

```
Definition rels:= seq (seq nat).
```

Let us recall that if a new relation is going to be added, for instance $j > k$, to a list of relations which consists of $i > j$ then we can concatenate both relations in the following way $i > j > k$ and get a new relation. So, the list `(i::j::k::nil)` will represent this relation in SSReflect.

The translation of the RS algorithm from Haskell to SSReflect is quite direct. Namely, several of the functions that we use have their counterpart in SSReflect, for instance `all` and `filter`. Others, such as `uniq` (checking that a sequence has not repeated elements) or `index` (giving the position of an element in a list), which have been defined *ad-hoc* in Haskell are already available in the SSReflect library. Let us see an example (the Haskell counterpart has been introduced in Section 2.2).

```
Definition compijCvd (vf:vectorfield)(M: matZ2) :=
  all [pred p| (nth 0 (nth nil M p.1) p.2) == 1] vf.
```

An important difference between Haskell and Coq/SSReflect is the definition of recursive functions. In Coq, all the functions must terminate; however this condition is not compulsory in Haskell. In order to illustrate this fact, let us present how the recursive `addOrder_concat` function (defined in Section 2.2) has had to be defined in SSReflect. As we explained in the methodology, we translate this Haskell function to Coq/SSReflect. We usually define the recursive functions using the `Fixpoint` command. But, in this case we cannot define it in this way since the system cannot determine that this function really ends. This is due to the fact that in some cases, the size of the first list decreases, but in other cases, the size of the second one decreases. This problem can be dealt with in different ways. The easiest way to be able to define this function consists in adding a new parameter, an iterator which decreases in every iteration and makes sure us that the function terminates. This option has been taken to define the `genDvfOrders` function making sure that the algorithm can run on all the elements of the matrix. But this is not the most elegant way. The alternative that we have employed for `addOrder_concat` consists in providing a measure to ensure its termination. After Coq v8.1, the system includes a new command, called `Function`, which allows us to directly encode general recursive functions. This command accepts a measure function that specifies how the argument "decreases" between recursive function calls.

In the case of the addOrder_concat function, the measure is just the sum of the size of the sequences which are the input parameters.

```
Function addOrder_concat (rs: rels * rels)
{measure (fun ords=> (size (fst rs) + size (snd rs))%N)}:=
 match (fst rs), (snd rs) with
 | nil , _ => nil
 | _ , nil => nil
 |a::nil,c::d => ((c ++ a)::nil) ++ addOrder_concat ((a::nil), d)
 |a::b,c::nil => ((c ++ a)::nil) ++ addOrder_concat (b,(c::nil)))
 |a::b,c::d => ((c ++ a)::nil) ++(addOrder_concat ((a::nil), d))
           ++ (addOrder_concat (b, (c::d)))
end.
Proof.
...
Defined.
```

The using of the Function constructor generates proof-obligations that must be proved to guaranty the termination of the function. This type of proofs are usually easy to prove using properties about natural numbers. Furthermore, the Function command generates a lot of auxiliary results related to the defined function. Some of them are powerful tools to reason about it. For instance, addOrder_concat_ind is an specialized induction principle tailored for the specific recursion pattern of the function. Moreover, this induction principle will facilitate us the task of proving theorems about addOrder_concat.

Let us note that this proof does not finish with Qed (as happens in the rest of lemmas) but with Defined. Both options are correct but if we choose the proof using Qed, the function is not computable.

Another difference between Haskell and SSReflect programs is the generalization of some programs to make the proofs easier. For instance, we can consider the function which sorts a discrete vector field (this function is called dvford). In Haskell the realignment is performed depending on the maximum length of the paths generated for each vector. However, other functions can be used to sort a vector field. For instance, we have defined the insert_sort function (which is internally called by dvford) parameterized by a function f : nat -> relations -> nat which will be in charge of sorting the vector

field. Afterwards, we instantiate it with our function to compute the maximum length of the paths associated with the vector.

2.4.2 Verification in SSReflect

In this section we are going to present the verification of the RS algorithm. In particular, we prove that our implementation constructs an ordered and admissible discrete vector field computed with `dvford`. Let us recall that this function uses `genDvfOrders` which returns both an admissible discrete vector field without reordering and the relations associated with it. This means that we have to prove that the output produced by `genDvfOrders` satisfies the properties of a discrete vector field (see Definition 1.42) and the admissibility property (see Definition 1.40). In addition the output produced by `dvford` must be sorted.

Let us introduce briefly a sketch of the steps that we have followed in the proof. First of all, the definition of an ordered and admissible discrete vector field, `Vecfieldadm` is introduced. This definition consists of a matrix M, a vector field vf, a sequence of relations r, and a proof of the fact that (vf, r) satisfies the properties of an ordered and admissible discrete vector field for M. Then, we have to construct an instance of this definition using the output produced by the functions `dvford M` (an ordered and admissible discrete vector field) and `genOrders M` (a sequence of relations taking into account `dvford M`) being M a matrix over $\mathbb{Z}_2$.

2.4.2.1 Definition of an ordered and admissible discrete vector field

Now, let us present the definition of `Vecfieldadm` (Figure 2.1) which involves conditions about the three input parameters: a well built matrix, a vector field, and a list of relations.

Let us note that the five first conditions come from the three properties of a discrete vector field (see Definition 1.42). Let M be a matrix over $\mathbb{Z}_2$ with m rows and n columns which will be denoted forward by $M : M[\mathbb{Z}_2]_{(m,n)}$ and $vf = (a_i, b_i)_i$ be a vector field from M, the first two properties establish that $\forall i, 0 \leq a_i < m$ and $0 \leq b_i < n$. The third property states that $\forall\ i\ j : nat,\ (i,j) \in vf\ \rightarrow\ M[i,j]\ =\ 1$. Another property reflects that a_i are different among them and the same happens

with b_i.

```
Definition Vecfieldadm (M: matZ2)(vf: vectorfield)(r:rels) :=
  (all [pred i | 0<= i < (M m)](getfirstseq vf))
  (all [pred i | 0<= i < (M n)](getsndseq vf)) /\
  (forall i j:nat, (i,j) \in vf -> (nth 0 (nth nil M i) j) = 1%R)
  /\ (uniq (getfirstseq vf)) /\
  (uniq (getsndseq vf)) /\
  (forall i j l:nat, (i,j) \in vf -> i!=l
    -> (nth 0 (nth nil M l) j)!= 0%R -> (i::l::nil) \in r) /\
  prop_cat2 r /\
  (all uniq r) /\
  (ordered glMax vf r).
```

Figure 2.1: `Vecfieldadm` definition

Then the three following conditions are linked with the relations. The first one gives us the link between the vector field and the relations. The condition $\forall\ i\ j\ m : nat,\ (i,j) \in vf \rightarrow i \neq m \rightarrow M[m,j] \neq\ 0 \rightarrow (i :: m :: nil) \in r$ states the completeness property of the relations which are generated. The second one verifies that we are constructing the transitive closure. And the last one states the admissibility property. It makes sure that every sequence of r has not repeated elements. Finally, it is checked that vf is ordered taking into account the `glMax` function and the relations r.

2.4.2.2 Properties to formalize about an ordered and admissible discrete vector field

In this section we are going to focus on certifying the correctness of the `dvford` function. In particular, we are going to prove that the `dvford` function builds an ordered and admissible discrete vector field (given a matrix with entries from $\mathbb{Z}_2$ as input). Internally, both this function and `genDvf` use the `genDvfOrders` function which builds and stores an admissible discrete vector field and the relations associated with the vector field. The difference between the `genDvf` and `dvford` functions is the ordering of the vector field. In the process, we prove that the output of `genDvf` satisfies both the three definitional properties of a discrete

vector field (Definition 1.42) and the admissibility property (Definition 1.40). Knowing that `dvford` is a permutation of such a vector field, these properties for `dvford` are directly obtained using the SSReflect library about permutations. For instance, if there is a permutation between `s1` and `s2` and `s1` has not repeated elements then `s2` neither (with `perm_eq_uniq`). Finally, it remains to be proved that `dvford` returns an ordered vector field.

Property 1

We are going to prove that the output produced by `genDvf` verifies the first property of Definition 1.42. It says that let $M : M[\mathbb{Z}_2]_{(m,n)}$ and $vf = (a_i, b_i)_i$ be a vector field associated with M then on the one hand, $\forall i,\ 0 \leq a_i < m$ and on the other hand, $0 \leq b_i < n$. As both cases are analogous, let us focus on the former one.

The statement of such a property in SSReflect is the following one.

```
Lemma propSizef (M:matZ2):
 (all [pred i | 0 <= i < (size M)](getfirstseq (genDvf M)))
```

The `genDvf` function gives us an admissible discrete vector field *vf* from a matrix. But we only need the first components of *vf*, so we use the `getfirstseq` function. This function returns the first components of a sequence of pairs.

This proof is not difficult and only some ideas will be outlined here. The `genDvf` function returns either an empty discrete vector field (when `M` is empty) or the first component of another function named `genDvfOrders`. So, rewriting the definition `genDvf` and removing the trivial cases, the proposition to prove with `M = a::b` is as follows.

```
P: all [pred i | 0 <= i < (size M)]
  (getfirstseq (fst (genDvfOrders (size M*size a).+1 0 0 M
      [::][::])))
```

Let us emphasize that the method `genDvfOrders` is recursive. So, we will prove its properties applying induction. In this case, we can use the schema generated by the own `genDvfOrders` function which is defined as follows:

```
Functional Scheme genDvfOrders_ind :=
  Induction for genDvfOrders Sort Prop.
```

Afterwards, we apply this schema over `P` with the corresponding parameters of `genDvfOrders` using the command `functional induction`. In order to prove the different cases we have to notice two aspects. On the one hand, the `canAddCvd` function is the responsible for that a new vector `(i,j)` which was added to the vector field verifies the condition `0 <= i < (size M)`. And on the other hand, it is also necessary to check that the vectors which are in *vf* (that have already been added previously) also satisfy the same property. But this is obtained using the inductive hypothesis.

Property 2

In this section, we will prove the second property of Definition 1.42. It says: let $M : M[\mathbb{Z}_2]_{(m,n)}$ be a matrix and $(a_i, b_i)_i$ be a vector field from M, then $\forall\, i, M[a_i, b_i] = 1$.

The equivalent statement in SSReflect of this property would be the following one.

```
Lemma v_in_genDvf_Mv1 (M: matZ2): (forall a b:nat,
((a,b) \in (genDvf M)) -> nth 0 (nth nil M a) b = 1%R).
```

Proof of Property 2

An auxiliar lemma will be introduced to prove this property, just to give an idea of the proof.

To prove `v_in_genDvf_Mv1`, we will need the following property which is the result of expanding the definition of `genDvf` and removing trivial cases.

```
Lemma inDvf_compij1 (p a b:nat) (M : matZ2):
(a,b) \in (fst (genDvfOrders p 0 0 M M [::] [::]))
-> nth 0 (nth nil M a) b = 1%R
```

As we said in the proof of the previous property, we will prove this property by means of induction since `genDvfOrders` is recursive. However, as usually happens in mathematics, it is easier to prove a generalization of a result and subsequently, particularize the result to verify in a straightforward manner the particular case. So, we are going to generalize some parameters of this function to provide an easier lemma to prove shown in Figure 2.2.

First, we generalize the two sequences `[::]`, which represent both a vector field and a sequence of relations. Moreover, the second and third parameter will

```
Lemma inDvf_compij1_general (p i j a b :nat) (M M2: matZ2)
   (vf: vectorfield)(r: rels):
 (forall k2, nth 0 (nth nil M (i + a)) k2 = nth 0 (nth nil M2 a)
     k2)
 -> (i + a,j + b) \in fst (genDvfOrders p i j M M2 vf r)
 -> nth 0 (nth nil M (i + a)) (j + b) = 1%R
```

Figure 2.2: `inDvf_compij1_general` lemma

be generalized as two natural numbers `i` and `j`. Finally, we generalize only the second matrix `M` by means of a general matrix `M2`, because only one of them will be modified in the recursion of `genDvfOrders`. So, it is a way of distinguishing them. But this change would not be enough because there is a relation between both matrices. Namely, `M2` is a submatrix of `M` where the i-first rows have been removed, that is to say, $\forall\ k, M[(i\ +\ a), k] = M2[a, k]$. Consequently, this condition will be added as a hypothesis. Then, we are interested in knowing, given a general position (i, j) instead of $(0, 0)$, what is the corresponding vector to add taking into account this new reference point. If a vector is *(a, b)* when the reference point is $(0, 0)$, consequently, the vector will be $(i{+}a, j{+}b)$ with (i, j) as a reference point.

In many cases, the induction schema associated with a data type is not enough to prove lemmas which involve complex functions. It is sometimes necessary to define an induction on different parameters at the same time. Moreover, these parameters can behave in different ways depending on the case wherein the processing is located. So, for instance, a parameter cannot always decrease but sometimes keeps fixed. This is the case of the fourth parameter *M1* of `genDvfOrders`. To this aim, an inductive schema will be defined taking into account the complex function (in our case `genDvfOrders`) to prove the concrete lemma (`inDvf_compij1_general`).

Moreover, every recursive function has associated an inductive schema based on the different cases which appear in it. This schema can be sometimes useful as we can see during the proof of the Property 1 (Section 2.4.2.2). On the contrary, you may need to prove a lemma which uses this function but the previous schema is not suitable. These situations appear if some of the parameters of

the lemma, which does not appear in the recursive function, change depending on some of the parameters of the recursive function. In these cases, we have to define a new function which allows us to build the schema associated with the function taking into account the new parameters and how they behave in the different cases of the method. To define a schema in SSReflect the instruction `Functional Scheme` is used with the new function defined. This idea has been used in many occasions to prove our lemmas with recursive functions and it was explained in detail in Section 1.2 with a small example.

Let us emphasize that it is necessary to think what schema can be useful to prove a lemma, namely, how the parameters behave when others change. In this case, the schema associated is complicated. We show a case to explain how the variables will have to be modified. This case consists of running the i-th row (the second parameter of `genDvfOrders`) with the parameter `j`. So, this parameter is increasing in a unit along the row. It is worth noting that when `j` increases in a unit, the parameter `b` decreases in a unit, so that the proposition to prove `nth 0 (nth nil M (i + a)) (j + 1 + b.-1) = 1%R` is equivalent to the previous one `nth 0 (nth nil M (i + a)) (j + b) = 1%R`. Using this schema the environment will include an inductive hypothesis which will be enough to prove the lemma.

This lemma is proved by cases. The first one satisfies that `nth 0 (nth nil M (i + a)) (j + b) = 1%R`, consequently the lemma is proved. The latter one states that `nth 0 (nth nil M (i + a))(j + b) = 0%R` but the proposition to prove means that its value is 1. Then a contradiction appears. To deal with this case we apply the following lemma which proves that if an entry of the matrix is 0 then this position cannot be selected to be a member of the vector field. However, we had as hypothesis `(i + a,b) \in fst (genDvfOrders p i j M M2 vf r)` that means that `(i + a,b)` is a member of the vector field.

- ***compij0_notinDvf***.

  The lemma says that if $v = (a, b)$ is a vector, $M : M[\mathbb{Z}_2]_{(m,n)}$, vf is an admissible discrete vector field generated from M and $M[a, b] = 0$, then $v \notin vf$. By construction, the function which creates the discrete vector field only selects some of the entries of M whose value is 1. So, if an entry

of M is 0 then the position of this entry will not belong to *vf*. The statement of Lemma `compij0_notinDvf` is the following one.

```
Lemma compij0_notinDvf (p i j a b:nat)(M: matZ2):
  nth 0 (nth nil M a) b = 0%R ->
  (a,b) \notin fst (genDvfOrders p i j M M [::] [::]).
```

To prove it, we will generalize this result and then particularize instead of proving it from scratch. In particular, we generalize in the same way that the lemma `inDvf_compij1`. The new lemma would be as follows.

```
Lemma compij0_notinDvf_general (p i j a b:nat)(M M2: matZ2)
             (vf: vectorfield)(r: rels): (forall k2,
    nth 0 (nth nil M (i+a)) k2 = nth 0 (nth nil M2 a) k2)
 -> nth 0 (nth nil M (i+a)) b = 0%R
 -> (i + a,b) \notin fst (genDvfOrders p i j M M2 vf r).
```

This lemma will be proved using induction but the schema `genDvfOrders_ind` will not be useful. So, we have to define a new schema for `genDvfOrders` which is useful for this concrete lemma. Let us explain that the parameter `a` will behave in different ways depending on the case with which one is dealing. The essential idea is that if `i` increases then `a` should decrease so that the vector `(i + a,b)` would not change: `((i + 1)+ (a - 1),b)`.

Property 3

In this section we will prove the third property of Definition 1.42. Let M be a matrix over $\mathbb{Z}_2$ and $vf = (a_i, b_i)_i$ be a vector field from M, then a_i are different among them and the same happens with b_i. Let us focus on the first part (lemma `norepfst`). The second one `norepsnd` is proved analogously.

The statement in SSReflect that establishes the first components are unique is:

```
Lemma norepfst (M: matZ2) : (uniq (getfirstseq (genDvf M))).
```

This lemma is proved rewriting the `genDvf` function, removing the trivial cases, for instance when the matrix is empty, and applying induction on the `genDvfOrders` function with the inductive schema generated by the own function.

Property 4

This property is in charge of checking that the transitive closure of the relations, which is generated during the construction of the vector field from a matrix does not have repeated elements. The statement in SSReflect is the following one.

```
Lemma norep_genOrders (M: matZ2): all uniq (genOrders M).
```

This lemma will be proved using induction and applying the following lemma since the relations which belong to the output of `genOrders` have been added with the `addOrders` function.

```
Lemma canAddOrders_addOrders (i j:nat)(col: seqZ2) (r: rels):
 all uniq r -> canAddOrders i j col r
 -> all uniq (addOrders i j col r).
```

Concretely, let `i` and `j` be natural numbers, `col` be a sequence over $\mathbb{Z}_2$ (which represents a column of a matrix), `r` be a list of relations which has not repeated elements, and the returned value of `canAddOrders` function is `True`, then we obtain that the corresponding relations which will be added to `r` will be sequences without repeated elements.

The `addOrders` function is a recursive one. In every step the parameter `j` increases in one unit, the sequence `col` deletes its first element, and `r` is modified adding new relations. The new possible relations can be:

- `[:: i,j]`
- `addOrder_orders i j r`

  The relations which can be added as a concatenation of `[:: i,j]` with any relation in r which starts with j or ends in i.

- `addOrder_concat ((getfirstE j r), (getlastE i r))`

  This function has two sequences as input parameters. The former one is a list of sequences of `r`, where every sequence starts with `j`. The latter one is another list of `r` which finishes with `i`. This function adds the relations which are the concatenation of one sequence of every list. Concretely, the new sequences are consisted of one of the sequences which finishes with

`i` (second parameter) concatenated with another one which starts with `j` (first parameter).

```
addOrder_concat [[1,7],[1,3]][[4,2],[5,6,2]] ✠
[[4,2,1,7],[4,2,1,3],[5,6,2,1,7],[5,6,2,1,3]]
```

Moreover, these different ways of adding relations will be reflected in the way of checking if new relations can be added. The function which is in charge of this duty is `canAddOrders`. In every step the checking is carried out by the following functions.

- `canAddOrder i j ord`

  This function returns true if for every element of `r`, for instance `r1`, one of the following conditions happens:

  - `i \notin r1`
  - `j \notin r1`
  - `index i r1 < index j r1`

  Then, the relation `(i::j::nil)` will not be added if a relation where first appears `j` and then `i` exists in `r`. This function checks that the relations which are created by `addOrder_orders` do not generate cycles.

- `canAOrder i j ord`

  This function returns `True` if any two relations, where one of them starts with `j` and the other one finishes with `i`, are joint the new relation will not have repeated elements. Let us note that `canAOrder` checks that the relations created using `addOrder_concat` do not generate cycles.

These ideas are reflected in the lemmas shown in Figure 2.3.

These last functions depend again on other functions to check that no cycle will appear in the construction of the relations. We have needed proving similar lemmas with the different functions for adding relations. Namely, 20 lemmas related to these functions and some other 30 lemmas about auxiliary functions have been proved previously.

```
Lemma canAddOrder_norep (i k:nat)(r: rels):
  i != k -> all uniq r ->
  canAddOrder i k r -> all uniq (addOrder_orders i k r).

Lemma norep_addOrder_concat (i k:nat)(r:rels)(rs: rels*rels):
  (i != k) -> all uniq r ->
  (forall x:seq nat, x \in (fst rest) -> head 0 x = k) ->
  (forall x:seq nat, x \in (snd rest) -> last 0 x = i) ->
  subseq (fst rest) r -> subseq (snd rest) r ->
  canAddOrder i k r -> canAOrder i k r ->
  all uniq (addOrder_concat rest).
```

Figure 2.3: `canAddOrder_norep` and `norep_addOrder_concat` lemmas

Property 5

With this property, we are interested in checking that if M is a matrix, vf is a vector field over M, m is a natural number and (i, j) is a vector satisfying that $(i, j) \in vf$ and $M[m, j] \neq 0$ then the relation $i > m$ has to belong to the relations generated during the construction of vf. This lemma in SSReflect is formalized as follows.

```
Lemma indif0_order M : forall i j m,
  (a,b) \in (genDvf M) -> a != m ->
  nth 0 (nth nil M m) b != 0%R -> (a::m::nil) \in genOrders M.
```

First of all, we introduce two lemmas which will be used to prove `indif0_order`. These are in charge of proving properties about the relations which are added with `genOrders`, and particularly with `addOrders`.

- ***invf_inaddO***.

  This lemma states that if a relation already belongs to a list of relations then the same relation is a member of the relations where other ones have been added. The lemma in SSReflect is the following one.

  ```
  Lemma invf_inaddO a i j c ords:
    a \in ords -> a \in addOrders i j c ords.
  ```

The lemma is not difficult to prove: the inductive schema associated to the own `addOrders` function is applied.

- ***compij_in_addOrders***.

  This lemma proves that the relations added from the column of a chosen vector *(i, j)*, i.e. the elements different from 0 whose position is not the *i*-th of the column, have to belong to the list of relations which are added. For instance, if the position m of the list satisfies these conditions then the relation $i > m$ has to be in the relations generated from the vector *(i, j)*.

```
Lemma compij_in_addOrders (im i:nat) l r:
  nth 0%R l im!= 0%R -> uniq (i::im::nil) ->
  (i::im::nil) \in addOrders i 0 l r.
```

  To prove it, this lemma is generalized to the following one which allows us to prove `compij_in_addOrders`. In the following lemma, the new extra parameter allows us to add the relation from the $k - nth$ component of the list instead from the first component.

```
Lemma compij_in_addOrders_general (im k i:nat) l r:
  nth 0%R l im!= 0%R -> uniq (i::(im + k)::nil) ->
  (i :: (im + k) ::nil) \in addOrders i k l r.
```

  We will use induction to prove it, but the own inductive schema of this function is not useful as the parameter k changes in different ways in different cases. The important point is that this parameter increases when im decreases so that the relation is always the same one:

```
(i::(im + k)::nil) = (i::(im - 1 + k + 1)::nil).
```

The lemma `indif0_order` will be proved using the following lemma which is more general since both `genDvf` and `genOrders` are expanded and their arguments have been generalized as it was made in the proof of Property 2.

```
Lemma indif0_order_genDvfOrders a b m p i j M M1 r vf :
  ((a, b) \in vf -> nth 0 (nth nil M m) b != 0%R
  -> (a::m::nil) \in r)
  -> (a, b)\in (genDvfOrders p i j M M1 vf r).1 -> a!= m
  -> nth 0 (nth nil M m) b != 0%R
  -> (a::m::nil) \in (genDvfOrders p i j M M1 vf r).2.
```

Let us highlight the first hypothesis verifies that every vector `(a, b)` which is in the vector field and $\forall$ `m,M [m,b]` $\neq 0$ then the relation `(a::m::nil)` will be in `r`. As we know the elements which are already in `r` satisfy the property, so we only have to check the relations which are added in consecutive steps. Finally, the previous lemmas will be used to sort out the cases which appear when the inductive schema of `genDvfOrders` is applied. Therefore, this lemma is proved in this way.

Property 6

This property checks if the transitive closure property is verified by the relations. We use the following definition instead of the implemented previously `prop_cat`.

```
Definition prop_cat2 (ords: seq (seq nat)):=
  (forall a b s p, (a::s) \in ords -> (last 0%N s = b)
  -> (b::p) \in ords -> ((a::s) ++ p) \in ords).
```

Let us note that the difference between both definitions lies in their implementations. Consequently, the lemma to prove is as follows.

```
Lemma in_in_cat (M:matZ2):
 let r:= (genOrders M) in prop_cat2 r.
```

Let us note that this property was tested with QuickCheck with the `prop_cat` function instead of `prop_cat2`. Taking into account the following lemma which relates both definitions:

```
Lemma prop_cat_prop_cat2 r: (prop_cat r) -> (prop_cat2 r).
```

we can obtain that if this property is verified by `prop_cat` then `in_in_cat` would be easy to prove, because `prop_cat` is processed as a generalization of `prop_cat2` thanks to the previous lemma. The Lemma `in_in_cat` will be proved in this way. On the other hand, `prop_cat r` will be proved by the own construction of the relations of the algorithm.

Property 7

After proving that the output of `genDvf` is an admissible discrete vector field, it is needed to prove that `dvford` returns an ordered and admissible discrete

vector field. The properties of an admissible discrete vector field about the output of `dvford` will be proved easily using the properties about the permutations of sequences. So, let us prove that the vector field obtained with `dvford` is an ordered one decreasingly according to the `glMax` function, which computes the maximum length of the paths. Let us note that `dvford M` is defined as `insert_sort glMax (genDvf M)(genOrders M)`. So, we sort the vectors which belong to `genDvf M` using the returned value of the `glMax` function which takes into account the relations `genOrders`. The lemma in SSReflect is the following one.

```
Lemma ordered_dvfg M: let dvf := (dvford M) in
  let ords:= (genOrders M) in ordered glMax dvf ords.
```

To prove it, we generalize the lemma taking `f` as the reordering function in the following way.

```
Lemma ordered_insert_sort vf ords:
  ordered (insert_sort f vf ords) ords.
```

Namely, `f` is a function which given a natural number and a sequence of relations returns a natural number (`f: nat -> rels -> nat`). Moreover, `vf` and `ords` represent a general vector field and a general list of relations, respectively. To prove it, we apply induction on `vf`.

2.4.2.3 The RS algorithm builds an ordered and admissible discrete vector field

In order to prove that the returned value of our algorithm satisfies the properties of an ordered and admissible discrete vector field, `Vecfieldadm` (defined in Section 2.4.2.1) is needed to have proved the seven properties depicted in the previous subsection over the parameters `(dvford M)` and `(genOrders M)`. So, we will only state the theorem `dvfordisVecfieldadm` (shown in Figure 2.4) and will apply the corresponding lemmas to prove the seven properties. The used lemmas which corresponds with the seven properties are `propSizef`, `propSizes`, `norepfst`, `norepsnd`, `norep_genOrders`, `v_in_genDvf_Mv1`, `indif0_order`, `in_in_cat`, and `ordered_dvfg`.

The proofs which have been detailed in this section involve 49 definitions and

109 lemmas. In general, the development takes up 3772 code lines.

```
Theorem dvfordisVecfieldadm (M:matZ2):
  Vecfieldadm M (dvford M)(genOrders M).
  Proof.
  case H1: M=>[|s1 s2]; first by done.
  have HH: (M!=[::]); first by rewrite H1.
  rewrite -H1 /Vecfieldadm.
  rewrite (propSizef_ord M) (@propSizes_ord m n M HH ismatrix)
    (norepfst_ord M)(norepsnd_ord M)(admis M).
  split; split; split; split; split.
  apply: v_in_genDvf_Mv1.
  split; first by done.
  split; first by apply: ordered_dvfg.
  split; first by apply:indif0_order1.
  apply: in_in_cat.
  Qed.
```

Figure 2.4: `dvfordisVecfieldadm` theorem

2.5 A non deterministic algorithm in SSReflect

In this section we introduce an abstract formalization of admissible discrete vector fields on matrices and a non deterministic algorithm to construct an admissible discrete vector field from a matrix[1].

SSReflect provides all the necessary tools to achieve our goal. In particular, we take advantage of the `matrix`, `ssralg` and `fingraph` libraries, which formalize, respectively, matrix theory, the main algebraic structures and the theory of finite graphs.

First of all, we are going to define an admissible discrete vector field on a matrix M with coefficients in a ring $\mathcal{R}$, and with m rows and n columns. It is

[1]Thanks are due to Maxime Dénès and Anders Mörtberg which guided us in this development.

worth noting that our matrices are defined over a generic ring instead of working with coefficients in $\mathbb{Z}$ since the SSReflect implementation of $\mathbb{Z}$, see [CM12], is not yet included in the SSReflect distributed version. The vector fields are represented by a sequence of pairs where the first component is an ordinal m and the second one an ordinal n.

```
Variable R : ringType.
Variables m n : nat.
Definition vectorfield := seq ('I_m * 'I_n).
```

Now, we can define in a straightforward manner a function, called `dvf`, which given a matrix `M` (with coefficients in a ring $\mathcal{R}$, and with m rows and n columns, `'M[R]_(m,n)`) and an object `V` of type `vectorfield` checks whether `V` satisfies the properties of a discrete vector field on `M` (Definition 1.42).

```
Definition dvf (M : 'M[R]_(m,n)) (V : vectorfield) :=
  all [pred p | (M p.1 p.2 == 1) || (M p.1 p.2 == -1)] V &&
  (uniq (map (@fst _ _) V) && uniq (map (@snd _ _) V)).
```

It is worth noting that the first condition of Definition 1.42 is implicit in the `vectorfield` type. Now, as we have explained at the end of the previous section, from a discrete vector field `V` a binary relation is obtained between the first elements of each pair of `V`. Such a binary relation will be encoded by means of an object of the following type.

```
Definition orders := (simpl_rel 'I_m).
```

Finally, we can define a function, which is called `advf`, that given a matrix `'M[R]_(m,n)`, `M`, a `vectorfield`, `V` and an `orders`, `ords`, as input, tests whether both `V` satisfies the properties of a discrete vector field on `M` and the admissibility property for the relations, `ords`, associated with the vector field, `V`. In order to test the admissibility property we generate the transitive closure of `ords`, using the `connect` operator of the `fingraph` library, and subsequently check that there is not any path between the first element of a pair of `V` and itself.

```
Definition advf (M:'M[R]_(m,n)) (V:vectorfield) (ords:orders) :=
  dvf M V && all [pred i|~~(connect ords i i)] (map (@fst _ _) V).
```

Now, let us define a non deterministic algorithm which construct an admissible discrete vector field from a matrix. First, we define a function, `gen_orders`, which

generates the relations between the elements of the discrete vector field as we have explained at the end of the previous section.

```
Definition gen_orders (M0 : 'M['F_2]_(m,n)) i j :=
  [rel x y | (x != i) && (y == i) && (M0 x j == 1)].
```

Subsequently, the function, `gen_adm_dvf`, which generates an admissible discrete vector field from a matrix is introduced. This function invokes a recursive function, `genDvfOrders`, which in each step adds a new component to the vector field in such a way that the admissibility property is fulfilled. The recursive algorithm stops when either there is not any new element whose inclusion in the vector field preserves the admissibility property or the maximum number of elements of the discrete vector field (which is the minimum between the number of columns and the number of rows of the matrix) is reached.

```
Fixpoint genDvfOrders M V (ords : simpl_rel _) k :=
  if k is l.+1 then
    let P := [pred ij | admissible (ij::V) M
                      (relU ords (gen_orders M ij.1 ij.2))] in
    if pick P is Some (i,j)
       then genDvfOrders M ((i,j)::V)
                       (relU ords (gen_orders M i j)) l
    else (V, ords)
  else (V, ords).

Definition gen_adm_dvf M :=
  genDvfOrders M [::] [rel x y | false] (minn m n).
```

Eventually, we can certify in a straightforward manner (just 4 lines) the correctness of the function `gen_adm_dvf`.

```
Lemma admissible_gen_adm_dvf m n (M : 'M[R]_(m,n)) :
 let (vf,ords) := gen_adm_dvf M in admissible vf M ords.
```

As a final remark, it is worth noting that the function `gen_adm_dvf` is not executable. On the one hand, SSReflect matrices are locked in a way that do not allow direct computations since they may trigger heavy computations during deduction steps. On the other hand, we are using the `pick` instruction, in the definition of `genDvfOrders`, to choose the elements which are added to the vector

field; however, this operator does not provide an actual method to select those elements.

Chapter 3

Reduction associated with an ordered and admissible discrete vector field

In the last chapter, we have shown an algorithm to compute an ordered and admissible discrete vector field from a matrix (Section 2.2). Moreover, a formalized proof of its correctness has been presented in Section 2.4. This vector field will be used to obtain a reduction of this matrix. The process will be the following one. The discrete vector field was sorted (see Section 2.1) since our following step will be reordering the matrix according to it. Moreover, the size of the vector field gives us an organization of the matrix into four blocks. Let us highlight that the top-left block will be a triangular matrix with $1's$ on the main diagonal, therefore, we can compute the inverse of this matrix. If this matrix represents the unique non-null differential map of a chain complex, we can directly apply the Hexagonal Lemma (Lemma 3.2). Then, the reduced matrix can be obtained using operations of matrices over the blocks of the reordered matrix. The size of this new matrix will be the size of the bottom-right block.

In our applications to image processing, we can associate a 2D monochromatic image with a 3-truncated chain complex of finite type, which consists of only two non-null differential maps. These differentials can be represented as matrices

known as incidence matrices. The first matrix is composed by the incidence relations from edges to vertices and the second one, from triangles to edges. As the composition of two consecutive differential maps is null by definition of a chain complex, the product of both matrices has to be the null matrix. It is worth noting that the size of the matrices of the chain complex associated with digital images is usually quite big. In most cases it is useful, and sometimes necessary, to reduce the matrices previously to its homological processing.

The aim is getting a reduced chain complex obtained from the initial one. For this task, our method to obtain a reduction of a matrix, based on discrete vector field, will be useful. First, the method can be applied to one of the matrices and then to the others to reduce more the chain complex. Along the section, we prove the Vector-field reduction theorem which allows us build a reduction. Let us note that if we reduce, for instance, the matrix which relates edges to vertices, some vertices and edges could be deleted. Therefore, we will also need to modify the matrix where edges and triangles are involved. Moreover, the property of the chain complex has to hold.

The structure of this chapter is the following. In Section 3.1 we detail the reduction method based on the Hexagonal Lemma and illustrate it by means of an example. Then, we explain the implementation in Haskell of the algorithm to obtain the reduced matrices of the chain complex in Section 3.2. In the rest of the chapter, we focus on proving in Coq/SSReflect that there is a reduction between the initial chain complex and the chain complex which consists of these reduced matrices. First, in Section 3.3 we introduce the basic structures which are necessary to define Reduction (Definition 1.28) and Isomorphism (Definition 1.14) in our concrete case. Afterwards, we prove there exists a reduction between the initial chain complex and the reordered one in Section 3.4. In particular, we prove that exists an isomorphism between both chain complexes. After that, we focus on proving the correctness about the reduction between the reordered chain complex and the reduced one. Finally, taking both reductions into account, we prove that the composition of these reductions builds a new reduction (Theorem 1.29). Moreover, in Section 3.5 it is proved that the reduction between both chain complexes preserves the Betti numbers. This implies that the homology groups associated with both chain complexes are isomorphic. Finally, we conclude in Section 3.6 with another type of reduction based on collapses. This reduction could be used as a preprocessing of the image with the aim of applying finally the

reduction explained in the previous sections over the new chain complex obtained.

3.1 Introduction

As we said, after computing an ordered and admissible discrete vector field from a matrix, the matrix will be sorted. Then the reduced matrix will be computed using operations over the blocks of the reordered matrix. If the initial matrix corresponds with the unique non-null differential map of a chain complex, then the reduced chain complex is consisted of an unique differential map represented by the reduced matrix.

For instance, if we compute an admissible discrete vector field V of the matrix which represents the differential map d_1 this can be divided into four blocks taking into account the size of V. The top-left block is a square matrix and the number of rows is given by the number of vectors in the vector field. The rest of the blocks are immediately determined. Besides, if the top-left block has inverse, it is possible to apply the Hexagonal Lemma (Lemma 3.2) to this case. The reordered matrix of d_1 is $d'_1 = \left(\begin{array}{c|c} \varepsilon & \varphi \\ \hline \psi & \beta \end{array}\right)$ then the reduced matrix will be obtained from the formula $\bar{d}'_1 = \beta - \psi\varepsilon^{-1}\varphi$.

On the other hand, the admissible discrete vector field computed from one of the differential maps is going to allow us to reduce a finitely generated chain complex. In this case, the differential of degree -1 and $+1$ with regard to the differential which we have computed the vector field are also affected by the reduction. For instance, if we reduce the differential in degree 1 d_1, the previous one d_0, and the following one d_2, will be also modified but using the vector field in a different way. Specifically, the reduced matrix of d_0 consists of reordering the columns with the second components of the discrete vector field $d'_0 = \left(\begin{array}{c|c} \delta & \alpha \end{array}\right)$ and deleting the first columns of this matrix $\bar{d}'_0 = (\alpha)$. Something similar happens with d_2, the reordered matrix leaves $d'_2 = \left(\begin{array}{c} \eta \\ \hline \gamma \end{array}\right)$ and the reduced one will be $\bar{d}'_2 = (\gamma)$. This matrix is reordered by rows using the first components of the vector field and then deletes the first rows. In both cases, the number of columns or rows which are removed is exactly the size of the computed admissible discrete vector field.

Example

Let us use an example to illustrate the method to reduce a chain complex using admissible discrete vector field. For this matter we are going to revisit the example of Section 2 where the only non-null differential map of a chain complex is defined by a matrix. Let us recall that the matrix was:

$$d = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

and the ordered and admissible discrete vector field computed was: $V =< (0,0),(1,2),(2,3) >$. Afterwards, we reorder the rows with the sequence `[0,1,2]` (the first components of the vector field) and the columns with `[0,2,3]` (the second ones). These sequences give us the order of the rows and columns that will be located in the beginning of the matrix. The rest of rows and columns are added at the end of the matrix. Then the reordered matrix leaves as follows.

$$d' = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 \end{array}\right)$$

Using the previous formula the reduced matrix is:

$$\bar{d}' = \beta - \psi\varepsilon^{-1}\varphi = (1)$$

3.2 Implementation in Haskell

In this section we are going to focus on the implementation in Haskell of both a way of reordering a matrix and of obtaining the reduced matrix. So as to reorder the matrix we need two sequences which encode the new positions of the rows and the columns, respectively. This is obtained with the following function which first, reorders the rows and then the columns.

```
reorderM s1 s2 M = (reorder_columns_s s2 (reorder_rows_s s1 M))
```

Let us note that `reorder_rows_s` uses the sequence `s1` as a permutation of $\{0, 1, \ldots, (\texttt{length M} - 1)\}$ to reorder the rows of `M`. Something similar happens with `s2` which reorders the columns with the function `reorder_columns_s`. Both sequences are obtained from the discrete vector field. We know that the first components of the vector field are higher or equal than 0 and lower than the size of `M`. But in some cases all the rows have not been able to be chosen, therefore, we will have to complete the sequence with the aim of obtaining a permutation of $\{0, 1, \ldots, (\texttt{length M} - 1)\}$. The function `fill` is in charge of carrying out this task. Then the function which obtains a realignment of a matrix given an admissible discrete vector field is defined in the following way.

```
reorderM_dvf dvf M = reorderM (fill (getfirstseq dvf) (length M))
                              (fill (getsndseq dvf) (length (head M))) M
```

The result of this function applied to our previous example is:

```
reorderM_dvf [(0,0),(1,2),(2,3)] [[1,1,0,0],[1,1,1,0],[0,0,1,1],[0,1,1,0]] ✠
[[1,0,0,1],[[1,1,0,1],[0,1,1,0],[0,1,0,1]]
```

In this example, the sequence to reorder the rows is `[0,1,2,3]` and to reorder the columns is `[0,2,3,1]`. In both cases, we have had to complete the sequence which comes from the components of the vector field. This happens because the length of the vector field is three and both the number of rows and the number of columns is four.

Then, it is easy to use the functions `take` and `drop` (which are Haskell primitives) to define the functions which allow us to obtain the different blocks of a matrix M: `ulsubseqmx`, `ursubseqmx`, `dlsubseqmx`, and `drsubseqmx`. For instance, `ulsubseqmx` returns the top-left submatrix of M given two natural numbers *(i, j)* i.e., the matrix with the elements at positions *(a,b)* of M where $a \leq i$ and $b \leq j$.

```
ulsubseqmx 2 3 [[1,1,0,0],[1,1,1,0],[0,0,1,1],[0,1,1,0]] ✠
[[1,1,0],[1,1,1]]
```

These functions are used to build the blocks of the reduced matrix. Namely, let *sdvf* be the size of the computed admissible discrete vector field, then the matrix ε is a square matrix with *sdvf* rows and *sdvf* columns which is computed

with the function `epsilonseq`. The functions which allow us to obtain the rest of the blocks are `phiseq`, `psiseq`, and `betaseq`.

```
epsilonseq 3 [[1,1,0,0],[1,1,1,0],[0,0,1,1],[0,1,1,0]] ✠
[[1,1,0],[1,1,1],[0,0,1]]
```

Let us recall that the reduced matrix is obtained by means of operations over matrices in the following way: $\bar{M}' = \beta - \psi\varepsilon^{-1}\varphi$. The function `matrixReduced` returns this matrix.

```
matrixReduced [[1,1,0,0],[1,1,1,0],[0,0,1,1],[0,1,1,0]] ✠
[[1]]
```

Let us note that the other matrices which define the differential maps of the chain complex in degree -1 or $+1$ affected by the reduction will be sorted using `reorder_rows_s` or `reorder_columns_s` and reduced taking or dropping some rows or columns of the matrices.

3.3 Formalization of the basic algebraic structures in SSReflect

Different ways to represent matrices exist in a system. The one used in Coq/SSReflect consists in formalizing a matrix as a function which determines every element of the matrix through two indices (for its row and its column). With this abstract representation it is very easy to formalize different operations with matrices and to prove properties of them. Indeed, an extensive library on matrices is provided in SSReflect. But this formalization is not directly computable. An alternative could be representing a matrix as a sequence of sequences. This representation allows us to define operations directly computable in the system. For this reason, it was chosen for the implementation of our algorithms in the previous chapter. But proving properties with this representation is much harder, and we do not dispose of the extensive SSReflect library with this representation.

The solution adopted by us tries to take advantage of both representations. On the one hand, we have used the concrete representation of a matrix as a sequence of sequences in the previous chapter because our main purpose was to compute an admissible discrete vector field. This representation was also used to compute the reduced matrix in the previous section. On the other hand, we will use in this chapter the abstract representation of matrices as functions to define a reduction between the initial matrix and the reduced one and actually to prove that it satisfies the reduction properties.

In order to prove that these two matrix representations are equivalent we define two morphisms, `seqmx_of_mx` from abstract to concrete matrices and `mx_of_seqmx` in the other direction (Figure 3.1), whose compositions are identities. They allow us to change the representation when required.

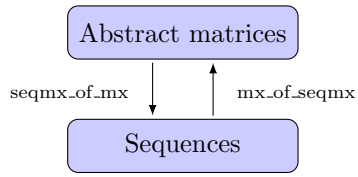


Figure 3.1: Bridges between matrix representations

In this section, we will formalize the notion of Reduction and Isomorphism of chain complexes (introduced in Section 1.1), restricting the chain complex to the one which consists of two non-null differential maps. First, we introduce the formalization of chain complexes, morphism of chain complexes, and homotopy operator with the previous constraint; working in this very particular case is justified because our main application domain will be the processing of 2D images.

A chain complex which comes from a 2D image consists in two matrices where the product of both matrices is null. This condition has to be verified for any chain complex. Moreover, we are interested in *proving* that property, and not in the computation of both matrices. This is the reason why the matrices which are involved in the product will be represented as abstract matrices instead of as sequences. Let us note that our algorithm is defined with sequences, so the two input matrices have type `matZ2`, but in order to state the property of the product and later to prove it, we use SSReflect matrices. Let us recall that

the function mx_of_seqmx builds an abstract matrix from a concrete one.

```
Definition is_chaincomplex (d1 d2: matZ2) (m n p: nat):=
  is_matrix m n d1 /\
  is_matrix n p d2 /\
  (mx_of_seqmx m n d1) *m (mx_of_seqmx n p d2) = 0.

Record chaincomplex:=
  {d1: matZ2;
   d2: matZ2;
   m: nat;
   n: nat;
   p: nat;
   chaincomplex_proof: is_chaincomplex d1 d2 m n p}.
```

Let us highlight *m denotes the matricial product. The type information of each matrix includes its size. When the product operator is applied, the type-checking ensures that the two arguments have compatible sizes. Then the system knows the expected size of the result matrix and reads 0 as the null matrix of this size.

Moreover, we need to introduce the corresponding notions of a chain complex morphism and of a homotopy operator. Again, the properties included in the definitions will not be computed but proved.

```
Definition is_chaincomplex_morphism C D p0 p1 p2:=
  ((p0 *m (mx_of_seqmx (m C) (n C)(d1 C))) =
    ((mx_of_seqmx (m D) (n D)(d1 D))) *m p1) /\
  (p1 *m (mx_of_seqmx (n C) (p C)(d2 C)) =
    ((mx_of_seqmx (n D) (p D)(d2 D)) *m p2)).

Record chaincomplex_morphism (C D: chaincomplex):=
  {p0: 'M[Z2]_(m D, m C);
   p1: 'M[Z2]_(n D, n C);
   p2: 'M[Z2]_(p D, p C);
   chaincomplex_morphism_proof: is_chaincomplex_morphism p0 p1 p2}.
```

```
Record homotopy_op (C: chaincomplex):=
  {h0: 'M[Z2]_(n C, m C);
   h1: 'M[Z2]_(p C, n C)}.
```

Finally, we can state the definition of a reduction between two chain complexes C and D. This reduction consists of two chain complex morphisms, namely, f and g, and a homotopy operator h. Moreover, a reduction has to verify the eleven properties defined in `is_reduction_CC2`.

```
Definition is_reduction_CC2 C D (f:chaincomplex_morphism C D)
  (g:chaincomplex_morphism D C)(h: homotopy_op C) :=
  (p0 f) *m (p0 g) = (1:Z2)%:M /\
  (p1 f) *m (p1 g) = (1:Z2)%:M /\
  (p2 f) *m (p2 g) = (1:Z2)%:M /\
  (p0 g) *m (p0 f) + (mx_of_seqmx (m C) (n C) (d1 C)) *m (h0 h)
     = (1:Z2)%:M /\
  (p1 g) *m (p1 f) + (mx_of_seqmx (n C) (p C) (d2 C)) *m (h1 h) +
     (h0 h) *m (mx_of_seqmx (m C) (n C) (d1 C)) = (1:Z2)%:M /\
  (p2 g) *m (p2 f) + (h1 h) *m (mx_of_seqmx(n C)(p C)(d2 C))
     = (1:Z2)%:M /\
  (p1 f) *m (h0 h) = 0 /\
  (p2 f) *m (h1 h) = 0 /\
  (h0 h) *m (p0 g) = 0 /\
  (h1 h) *m (p1 g) = 0 /\
  (h1 h) *m (h0 h) = 0.

Record reduction_CC2 (C D: chaincomplex):=
 {f: chaincomplex_morphism C D;
  g: chaincomplex_morphism D C;
  h: homotopy_op C;
  is_reduction_CC2_proof: is_reduction_CC2 f g h}.
```

Moreover, we also present the notion of isomorphism between two chain complexes. An isomorphism is stronger than a reduction, so if we have an isomorphism, we can build a reduction of it, defining the homotopy operator as the null automorphism. (In fact, we can build two reductions given an isomorphism. For instance, if we have an isomorphism between C and D then we can build a

reduction between D and C and another one between C and D.)

```
Definition is_isomorphism_CC2 C D (f:chaincomplex_morphism C D)
    (g:chaincomplex_morphism D C) :=
  (p0 f) *m (p0 g) = (1:Z2)%:M /\
  (p0 g) *m (p0 f) = (1:Z2)%:M /\
  (p1 f) *m (p1 g) = (1:Z2)%:M /\
  (p1 g) *m (p1 f) = (1:Z2)%:M /\
  (p2 f) *m (p2 g) = (1:Z2)%:M /\
  (p2 g) *m (p2 f) = (1:Z2)%:M.

Record isomorphism_CC2 (C D: chaincomplex):=
 {f_eq: chaincomplex_morphism C D;
  g_eq: chaincomplex_morphism D C;
  is_isomorphism_CC2_proof: is_isomorphism_CC2 f_eq g_eq}.
```

3.4 Reduction of a chain complex

In this section, we are going to deal with the formalization of the algorithm presented in Section 3.2 which reduces a chain complex which comes from a 2D monochromatic digital image to a simpler one. The reduced chain complex will be obtained through an admissible discrete vector field of one of the matrices which represent the differential maps of the chain complex. The lemma which we formalize is a particular case of the Vector-Field Reduction Theorem introduced in Subsection 1.1.6. Let us state this particular theorem.

Theorem 3.1. Let $C = (C_p, d_p, \beta_p)_p$ be a 3-truncated algebraic cellular complex and $V = \{\sigma_i, \beta_i\}_{i \in \beta}$ be an admissible discrete vector field on C built from one of the map of d_p. Then the vector field V defines a canonical 3-truncated reduction $\rho = (f, g, h) : (C_p, d_p) \Longrightarrow (C_p^c, d'_p)$ where $C_p^c = \mathbb{Z}\left[\beta_p^c\right]$ is the free $\mathbb{Z}$-module generated by the critical cells.

With this theorem, we are able to work with the chain complex generated by the critical cells which is smaller than the initial chain complex, knowing that homological properties are preserved. Let us note that the larger is the number of

vectors which compose the vector field the smaller is the reduced chain complex. So, we are interested in creating a vector field with many vectors as possible.

Initially, this process has been proved starting with a chain complex which consists of only one non-null differential map (as a first step in the proof). It will be also useful to become familiar with the way of proving this kind of lemmas. Then, we focus on a chain complex which comes from a 2D image, that is to say, a chain complex with two non-null differential maps. Some lemmas which are already proved for the first step will be useful. Finally, we formalize the algorithm for a chain complex composed by three non-null matrices. Thanks to the last step, we can apply the process to a finitely generated chain complex, since when a matrix is reduced due to an admissible discrete vector field the only matrices which are also reduced are the previous one and the following one. The rest of differential maps of the chain complex are not modified. Along the section, we focus on explaining the idea for a chain complex with only two non-null matrices.

The section is organized as follows. First, some details about the different definitions to reorder a matrix are introduced in Subsection 3.4.1. Then in Subsection 3.4.2 a reduction between the initial chain complex C and the reordered one D will be defined where C consists of two matrices, d_1 and d_2 and D of the ordered ones d'_1 and d'_2. Afterwards, another reduction will be defined from D to the reduced chain complex E in Subsection 3.4.3 where E is composed by the reduced matrices $\bar{d}'_1$ and $\bar{d}'_2$, which we have obtained with the algorithm presented in Section 3.2. Later, in Chapter 4, we introduce an alternative to the reduction presented in Subsection 3.4.3 (from the ordered chain complex to the reduced one) applying the BPL. Finally, in Subsection 3.4.4 we build the reduction of the initial chain complex C to the reduced one by composing the previous reductions.

The development described in this section takes up 7511 lines where 303 definitions have been stated and 361 lemmas have been proved. Namely, about 1000 lines have been used to prove properties about the realignment of the matrix (Subsection 2.1.1) and about the inverse of a lower triangular matrix (Subsection 3.4.3.4).

Example

Let us see an example to clarify the process. The digital image of Figure 3.2 can be represented through a chain complex in the following way. First, the chain

complex associated with the image consists of the incidence matrices d_1 and d_2. The first one relates edges to vertices and the other one triangles to edges.

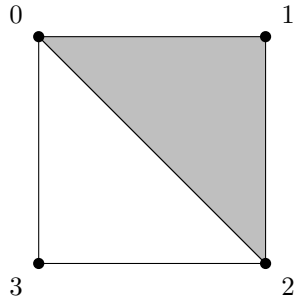


Figure 3.2: A small digital image

$$d_1 = \begin{array}{c} \\ \{0\} \\ \{1\} \\ \{2\} \\ \{3\} \end{array} \begin{array}{c} \begin{array}{ccccc} \{0,1\} & \{0,2\} & \{1,2\} & \{1,3\} & \{2,3\} \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \qquad d_2 = \begin{array}{c} \\ \{0,1\} \\ \{0,2\} \\ \{1,2\} \\ \{1,3\} \\ \{2,3\} \end{array} \begin{array}{c} \begin{array}{c} \{0,1,2\} \end{array} \\ \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \end{array}$$

As we explained in Section 2.1 we compute the ordered and admissible discrete vector field from a matrix, for instance d_1. In this case, the ordered and admissible discrete vector field obtained is $\{(0,0),(1,2),(2,4)\}$. The chain complex D is defined with the ordered matrices of d_1 and d_2 using this vector field. Then we reorder the rows of d_1 with $(0,1,2,3)$ and its columns with $(0,2,4,1,3)$. Moreover, let us reorder the rows of the matrix d_2 with the last sequence. Then the matrices which take part in the reordered chain complex remain as follows.

$$d_1' = \begin{array}{c} \\ \{0\} \\ \{1\} \\ \{2\} \\ \{3\} \end{array} \begin{array}{c} \begin{array}{ccccc} \{0,1\} & \{1,2\} & \{2,3\} & \{0,2\} & \{1,2\} \end{array} \\ \left(\begin{array}{ccc|cc} 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ \hline 0 & 0 & 1 & 0 & 1 \end{array}\right) \end{array} \qquad d_2' = \begin{array}{c} \\ \{0,1\} \\ \{1,2\} \\ \{2,3\} \\ \hline \{0,2\} \\ \{1,2\} \end{array} \begin{array}{c} \begin{array}{c} \{0,1,2\} \end{array} \\ \left(\begin{array}{c} 1 \\ 1 \\ 0 \\ \hline 1 \\ 0 \end{array}\right) \end{array}$$

Finally, as for building the reduced chain complex, we compute $\bar{d}'_1 = \beta - \psi\varepsilon^{-1}\varphi$ and $\bar{d}'_2 = \gamma$. Therefore, the obtained chain complex is the following one:

$$\bar{d}'_1 = \begin{pmatrix} 0 & 0 \end{pmatrix} \qquad \bar{d}'_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

If we had chosen d_2 instead of d_1, an ordered and admissible discrete vector field would have computed from d_2. Then, we had reordered the rows and columns of d_2 and only the columns of d_1. In a similar way, we had obtained the reduced matrices.

3.4.1 Realignment of a matrix

Let us recall that the chain complex is sorted after computing an ordered and admissible discrete vector field *vf* from one of their matrices. We reorder this matrix using the first components of *vf* to reorder rows and the second ones to reorder columns. For the other matrix, we only reorder rows using the second components of *vf* so that the product of both matrices continues making sense and being null. Let us focus on the realignment of the first matrix where the discrete vector field is computed, because the realignment of the second matrix is done using similar techniques.

In our implementation with matrices as sequences of sequences, we use auxiliary function `reorderM` which has as inputs two sequences of natural numbers, `s1` and `s2`, and a matrix `M` represented by means of sequences. Namely, `s1` will give us the new positions of the rows of `M` and `s2` the new positions of the columns of `M`. These lists are obtained from a discrete vector field. The function which is in charge of reordering the matrix given an admissible discrete vector field is the following one.

```
Definition reorderM_dvf (dvf:vectorfield) (M: matZ2):=
  reorderM (fill (getfirstseq dvf) (size M))
    (fill (getsndseq dvf) (size (head [::] M))) M.
```

On the other hand, there is a pair of libraries related to permutations in SSReflect: `perm` and `perm_seq`. These libraries include some functions about matrices represented as functions, such as `row_perm` and `col_perm`. The former one receives a permutation `s` and a matrix and returns the matrix with the rows permuted by `s`. These functions are useful to reorder a matrix.

```
Definition reorder_mx (s1:'S_m)(s2:'S_n)(M:'M[R]_(m,n)):=
  col_perm s2 (row_perm s1 M).
```

The main differences between both definitions for reordering is that the first one is applied to a sequence of sequences and it is computable, but the second one is defined over abstract matrices (SSReflect matrices) and, therefore, it is not possible to compute with it. Moreover, `reorderM` receives two sequences of natural numbers but `reorder_mx` receives two permutations. Let us stress that `'S_k` is the set of all permutation of the ordinal `k`, `'I_k`, for instance $\{0, 1, \ldots, (\mathtt{k}-1)\}$.

Following with our general way of work, we have an abstract version of the algorithm `reorder_mx` using SSReflect's structures and libraries to prove properties about them. Here, we can use the full power of dependent types when proving correctness. On the other hand, we have another more efficient definition of reordering which uses simple types which are closer types to standard implementation in traditional programming languages. Our aim is proving the correctness of these translations to get that the abstract definition and the concrete efficient one are equivalent. To this aim, we state another definition to reorder a matrix, `t_reorderM`, which is an intermediary between both definitions. The input parameters are the same that the ones used by the function `reorderM` but the structure of the implementation is closer to `reorder_mx`. This is due to the fact that the new functions which order the rows and the columns in the function `t_reorderM` are quite similar to the functions `row_perm` and `col_perm` (used by `reorder_mx`). Then, we will prove the equivalence between `reorderM` and `t_reorderM` and between `t_reorderM` and `reorder_mx`. Finally, we obtain an equivalence between `reorder_mx` and `reorderM_dvf` since the last one involves `reorderM`.

Let us introduce the statements of the two lemmas which relate the abstract definition to the concrete and efficient one. The first one receives a SSReflect matrix and the last one a sequence of sequences. Their proofs are based on applying the intermediate bridge (as we can see in Figure 3.3).

```
Variable s s1: seq nat.
Variable m n:nat.
Hypothesis H : perm_eq s (iota 0 m).
Hypothesis H1 : perm_eq s1 (iota 0 n).
```

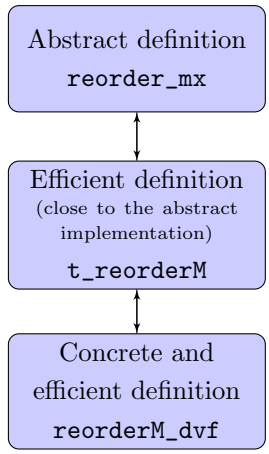


Figure 3.3: Bridges between matrix realignments

```
Lemma reorderM_E (M1:'M[Z2]_(m,n)) :
  (seqmx_of_mx M1) != [::] -> size s = m -> size s1 = n ->
  reorderM s s1 (seqmx_of_mx M1) = (seqmx_of_mx
    (reorder_mx (perm_of_seq H s) (perm_of_seq H1 s1) M1)).

Lemma reorderM2_E (M1:matZ2) : M1!= [::] ->
 size s = m -> size s1 = n -> perm_eq s (iota 0 (size s)) ->
 is_matrix m n M1 ->
 reorder_mx (perm_of_seq H s) (perm_of_seq H1 s1)
   (mx_of_seqmx m n M1) = mx_of_seqmx m n (reorderM s s1 M1).
```

We use the function `perm_of_seq` to convert a sequence of naturals `s`, in a permutation satisfying that `s` is a permutation of $\{0, 1, \ldots, (\mathtt{k}-1)\}$, with `perm_eq s (iota 0 k)`. The function `(iota m n)` builds the sequence $\{m, m+1, \ldots, m+n-1\}$ then `(iota 0 k)` corresponds with the sequence $\{0, 1, \ldots, (\mathtt{k}-1)\}$.

A similar process is followed with other definitions as `reorder_rows_s` and `reorder_columns_s` since these functions are used in the definition of `reorderM`. Let us see the concrete implementation to reorder rows in our algorithm (given a sequence of sequences).

```
Definition reorder_rows_s (s: seq nat) (M: matZ2):=
  map (nth nil M) s.
```

Then we can use the groups of permutation formalized in SSReflect where the permutation of a matrix vertically `row_perm`, is defined. This choice in the definition does not permit us compute it, but the proofs will be easier.

```
Definition row_perm (s : 'S_m) A := \matrix_(i, j) A (s i) j.
```

Our aim is proving the equivalence between both definitions, the efficient and the abstract one. Taking this into account, we can prove with the abstract one and compute using the efficient one. In order to obtain the correctness of this equivalence, we use another definition which approaches to both definitions. On the one hand, the function has the same input parameters as `reorder_rows_s`, and on the other hand, the implementation is closer to the abstract one. Let us see this new definition.

```
Definition t_reorder_rows_s (s: seq nat) (M: matZ2):=
  map (fun i => (nth [::] M (nth 0%N s i))) (iota 0 (size M)).
```

Finally, we will prove that the functions `reorder_rows_s` and `row_perm` return the same result. To get this objective, we need to prove the equivalence between `reorder_rows_s` and `t_reorder_rows_s` and the equivalence between `t_reorder_rows_s` and `row_perm`. Let us see the lemma which describes the second equivalence.

```
Lemma t_reorder_rows_sE (M1:'M[R]_(m,n)) :
  (t_reorder_rows_s s (seqmx_of_mx M1)) =
  (seqmx_of_mx (row_perm (perm_of_seq H s) M1)).
```

3.4.2 Reduction between the initial chain complex and the reordered one

In this subsection we are going to build an isomorphism between the initial chain complex and the chain complex obtained after reordering the rows and columns of the differential matrices. The process of reordering was explained in previous subsection and is carried out taking into account an admissible discrete vector

field. This is a first step in the reduction process. Obviously, an isomorphism is a particular case of reduction with the homotopy operator being null. First, we are going to enrich the chain complex structure (defined in Section 3.3) with the admissible discrete vector field obtained from its first differential.

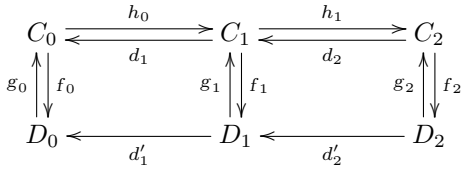
```
Record chaincomplex_advf:=
  {C:> chaincomplex;
  dvf_c :seq (nat*nat);
  ords_c: orders;
  dvfnonil_c: dvf_c!=[::];
  M2nonil_c: (d2 C)!=[::];
  vecfield_c: Vecfieldadm (d1 C) dvf_c ords_c}.

Variable chaincomplexd1d2 : chaincomplex_advf.
```

Let us recall Coq allows us to define structures with inheritance using the `Record` macro. Apart from the properties and types which are defined in `chaincomplex_advf`, this record inherits the properties which are included in the record `chaincomplex` using the notation `C:> chaincomplex`.

On the other hand, we have to define the reordered matrices d'_1 and d'_2 which are part of the reordered chain complex which will be called D.

```
Definition d1':= reorderM_dvf dvf d1.
Definition d2':= t_reorder_rows_s (s2 n dvf) d2.
```

Finally, let us show an small sketch of the steps to prove that a reduction exists between the initial chain complex and the reordered one which will be described along this subsection:



1. Building of the chain complex D which consists of the reordered matrices.
2. Construction of the chain complex morphisms $f = (f_0, f_1, f_2)$ and $g = (g_0, g_1, g_2)$.

3. Definition of an isomorphism between the chain complexes C and D.

4. Building a reduction from the previous isomorphism.

3.4.2.1 Building of the chain complex consisted of the reordered matrices

In this subsection we build a chain complex whose differential maps are the reordered matrices obtained from an admissible discrete vector field computed from the first differential map of a chain complex. For this purpose, we have to take into account two aspects. On the one hand, the sequences defined are indeed matrices, and in particular every row has the same size. On the other hand, checking that the product of both matrices, $d'_{1(m,n)}$ and $d'_{2(n,p)}$ is null. The first issue will be easy to solve since these matrices, sequence of sequences, are permutations of the initial ones, $d_{1(m,n)}$ and $d_{2(n,p)}$. Consequently, let us concentrate on proving the second issue.

First of all, let us introduce some lemmas related to the groups of permutations which are defined in SSReflect. These lemmas will be useful due to the fact that our reordered matrices are permutations of the initial ones. Let us recall that `row_perm` and `col_perm` permute a matrix vertically (rows) and horizontally (columns), respectively, given a group of permutation. Moreover, `perm_mx` is the matrix permutation, namely, it is a permutation of rows of the identity matrix. Let us show some lemmas about permutations already proved in SSReflect which will be useful in our proofs.

mul_row_perm: Let $A_{m,n}$ and $B_{n,p}$ be two matrices and s be a group of permutation then $A * row\_perm\ s\ B = (col\_perm\ s^{-1}\ A) * B$.

row_permE: Let $A_{m,n}$ be a matrix and s and t be two groups of permutation then $row\_perm\ s\ A = perm\_mx\ s * A$.

col_permE: Let $A_{m,n}$ be a matrix and s and t be two groups of permutation then $col\_perm\ s\ A = A * perm\_mx\ s^{-1}$.

Let us prove that $d'_1 * d'_2 = 0$ knowing that $d_1 * d_2 = 0$. First, we present the statement of the lemma to prove where the matrices represented by sequences are converted in SSReflect matrices with `mx_of_seqmx` giving its dimensions.

```
Lemma prod_d1'd2':
 (mx_of_seqmx (sdvf dvf + (m - (sdvf dvf)))
   (sdvf dvf + (n-sdvf dvf) d1')
 *m (mx_of_seqmx (sdvf dvf + (n - (sdvf dvf))) p d2') = 0.
```

Let us note that d'_1 and d'_2 are defined as sequences of sequences and are reordered matrices obtained from d_1 and d_2. Namely, d'_1 reorders both the rows and the columns of d_1 and d'_2 only the rows. Then we will change the view of this executable definition to an abstract one to be able to use SSReflect matrices and properties of the groups of permutations. In this way, the proof will be easier. We will use the lemmas `reorderM2_E` and `t_reorder_rows_sE` mentioned in Section 3.4.1 as a bridge between both definitions. In particular, d'_1 is a permutation of rows and another one of columns over d_1. Finally, the statement to prove (with the abstract representation) is as follows with `s1'` a group of permutations of `'I_m` and `s2'` a group of permutations of `'I_n`.

```
col_perm s2' (row_perm s1' d1) * (row_perm s2' d2) = 0
```

For the proof we apply the lemma `mul_row_perm` whose statement is let `A: 'M_(m, n)`, `B: 'M_(n, p)` and `s` be a group of permutations then

```
A *m row_perm s B = col_perm s^-1 A *m B
```

In other words, this lemma says that the product of a matrix `A` with a row permutation matrix of `B` is equal to the product of a column permutation matrix of `A` with `B`. Then we obtain:

```
col_perm (inv s2') (col_perm s2' (row_perm s1' d1)) * d2 = 0
```

Concretely, this lemma consists in applying a column permutation given by `(inv s2')` to every matrix of the equation. In this way, the matrix `0` continues being the null matrix and the matrix `(row_perm s2' d2)` is simplified to `d2`. Finally, applying `col_permE` to cancel the two consecutive permutations given by `inv s2'` and `s2'` and rewriting the lemma `row_perm_E` we obtain `perm_mx s1' * d1 * d2 = 0`. Therefore, since `d1 * d2 = 0`, the lemma `prod_d1'd2'` is proved.

The lemma `d1'd2'_chaincomplex` collects the proofs of the properties required in `is_chaincomplex`, so that D is a chain complex. Then, it is used to build the chain complex D.

```
Definition chaincomplexd1'd2' :=
  Build_chaincomplex d1'd2'_chaincomplex.
```

3.4.2.2 Definition of an isomorphism between the chain complexes C and D

In this section we deal with building two chain complex morphisms $f : C \to D$ and $g : D \to C$ which define an isomorphism between them. We focus on the definition of the first morphism f. It consists of a family of three morphisms $f = (f_0, f_1, f_2)$. These morphisms are defined as permutation matrices since the matrices of D are permutations of the matrices of C, respectively. Let us introduce the definitions of these functions. For instance, `f0` is a permutation matrix which will be created from the set of all permutations of the ordinal `m` obtained from the first components of the discrete vector field.

```
Definition f0 := (@perm_mx Z2 m
  (perm_of_seq (H vecfield dvfnonil ismatrixM1) (s1 m dvf))).
Definition f1 := (@perm_mx Z2 n
  (perm_of_seq (H1 vecfield dvfnonil ismatrixM1) (s2 n dvf))).
Definition f2: 'M[Z2]_p := (1:Z2)%:M.
```

Then we have to prove that f verifies the properties of a chain complex morphism defined in Section 3.3. In our concrete case, there are two conditions to verify: $f_0 * d_1 = d_1' * f_1$ and $f_1 * d_2 = d_2' * f_2$. Let us state the first one.

```
Lemma proof_f0_d1_d1'_f1:
(f0_eq *m (mx_of_seqmx m n d1))=((mx_of_seqmx m n d1') *m f1_eq).
```

We focus on this proof using lemmas about permutation matrices in a similar way as in the proof of the lemma `prod_d1'd2'` in the previous subsection. Let us stress that f_0 means a exchange of rows and f_1 an exchange of columns. On the another hand, we have the matrix d_1' which is a permutation of rows and columns of d_1 with respect to the same patterns, then we cancel this permutation of columns with f_1. Therefore, we obtain the same in both sides.

After proving the two previous conditions, we can define the chain complex f in SSReflect, `chaincomplex_morphism_f`. Then this development is repeated analogously to the chain complex morphism g which goes from D to C, which

is called `chaincomplex_morphism_g`. Let us define g as the inverse of f. In particular, g_0 gives us a exchange of rows and g_1 one of columns.

```
Definition g0 := (invmx f0).
Definition g1 := (invmx f1).
Definition g2 :'M[Z2]_p := (1:Z2)%:M.
```

Finally, it is easy to prove that the morphism `chaincomplex_morphism_f` and `chaincomplex_morphism_g` fulfill the conditions given in `is_isomorphism_CC2` and this allows us to define an isomorphism between C and D.

3.4.2.3 Building a reduction from the previous isomorphism

As we know, the definition of isomorphism is a particular case of the definition of reduction. Anyway, we are interested in building a reduction between C and D since we want to compose two reductions to obtain a new one.

The process is straightforward. It only consists in considering the null homotopy operator. This is defined in the following way.

```
Definition h0: 'M[Z2]_(n,m):= 0.
Definition h1: 'M[Z2]_(p,n):= 0.

Definition homotopy_op_h_eq :=
  (@Build_homotopy_op chaincomplexd1d2 h0 h1).
```

The reduction properties are also verified with no much effort taking into account the previous isomorphism properties; this allows us to build the required reduction.

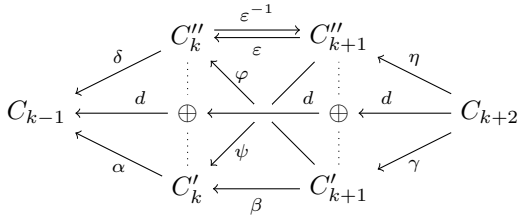
3.4.3 Reduction between the reordered chain complex and the reduced one

In this section we are going to define a reduction between the reordered chain complex D from an ordered and admissible discrete vector field and a reduced chain complex E applying Hexagonal Lemma. This is extracted from [RS10, Section 2.5]. Then, let us formalize these results. In Section 4.3 we

introduce another way of getting a reduction between the reordered chain complex and another one reduced applying the Basic Perturbation Lemma (Lemma 1.33).

3.4.3.1 Hexagonal Lemma

Proposition 3.2 (Hexagonal Lemma)**.** Let $C = (C_p, d_p)_p$ be a chain complex. For some $k \in \mathbb{Z}$, the chain groups C_k and C_{k+1} are given with decompositions $C_k = C'_k \oplus C''_k$ and $C_{k+1} = C'_{k+1} \oplus C''_{k+1}$, so that between the degrees $k-1$ and $k+2$ this chain complex is described by the diagram:



(3.1)

The partial differential $\varepsilon : C''_{k+1} \to C''_k$ is assumed to be an isomorphism. Then a canonical reduction can be defined $\rho : C \Rightarrow C'$ where C' is the same chain complex as C except between the degrees $k-1$ and $k+2$:

$$\ldots \longleftarrow C_{k-2} \longleftarrow C_{k-1} \xleftarrow{\alpha} C'_k \xleftarrow{\beta - \psi\varepsilon^{-1}\varphi} C'_{k+1} \xleftarrow{\gamma} C_{k+2} \longleftarrow C_{k+3} \longleftarrow \ldots$$

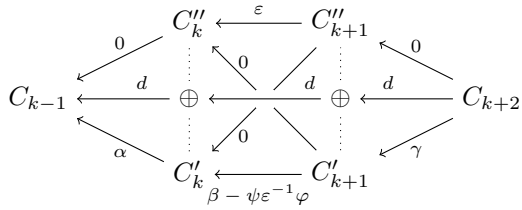
3.4.3.1.1 Proof

An integer matrix $\begin{bmatrix} \varepsilon & \varphi \\ \psi & \beta \end{bmatrix}$ is equivalent to the matrix $\begin{bmatrix} \varepsilon & 0 \\ 0 & \beta - \psi\varepsilon^{-1}\varphi \end{bmatrix}$ if $|\varepsilon| = 1$. It is the simplest case of Gauss' elimination. More generally, the following matrix relation is always satisfied:

$$\begin{bmatrix} \varepsilon & \varphi \\ \psi & \beta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \psi\varepsilon^{-1} & 1 \end{bmatrix} \begin{bmatrix} \varepsilon & 0 \\ 0 & \beta - \psi\varepsilon^{-1}\varphi \end{bmatrix} \begin{bmatrix} 1 & \varepsilon^{-1}\varphi \\ 0 & 1 \end{bmatrix}$$

The lateral matrices of the right-hand term can be considered as basis changes. These matrices define an isomorphism $\rho' : C \to \bar{C}$ between the initial chain

complex C and the chain complex $\bar{C}$ made of the same chain groups but the differentials displayed on this diagram:



(3.2)

Throwing away the component $\varepsilon : C''_{k+1} \to C''_k$ from this chain complex $\bar{C}$ produces a reduction $\rho'' : \bar{C} \Rightarrow C'$ to the announced chain complex C'. The desired reduction is $\rho = \rho''\rho' : C \Rightarrow C'$ where $\rho = (f, g, h)$ with

1. The morphism f is the identity except:
$$f_k = [-\psi\varepsilon^{-1} \;\; 1] \quad f_{k+1} = [0 \;\; 1]$$
2. The morphism g is the identity except:
$$g_k = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad g_{k+1} = \begin{bmatrix} -\varepsilon^{-1}\varphi \\ 1 \end{bmatrix}$$
3. The homotopy operator h is the null operator except:
$$h_k = \begin{bmatrix} \varepsilon^{-1} & 0 \\ 0 & 0 \end{bmatrix}$$

matrices to be interpreted via appropriate block decompositions.

Let us note the boundary components α and γ are not modified by the reduction process. Since the independent "hexagonal" decompositions are given for every degree, the process can be applied to every degree simultaneously.

3.4.3.2 Formalization

Let us recall that D is defined through the differential maps $d'_1 = \left(\begin{array}{c|c} \varepsilon & \varphi \\ \hline \psi & \beta \end{array}\right)$ and $d'_2 = \left(\begin{array}{c} \eta \\ \hline \gamma \end{array}\right)$ and E is defined through the reduced matrices $\bar{d}'_1 = \beta - \varphi * \varepsilon^{-1} * \psi$

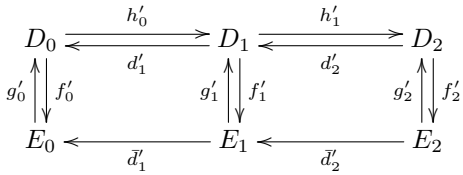
and $\bar{d}'_2 = \gamma$. Let us note that the reduced matrices are obtained using operations over blocks of matrices. In the `matrix` library of SSReflect, we can find many lemmas about properties of blocks of matrices which will be very useful in this task. Again, although our computable algorithm is defined for matrices as sequences of sequences we will change this representation to abstract SSReflect matrices (using the bridges between these representations) in order to prove the required properties in this context. In this way, the new definitions, as for instance the reduced matrices or the morphisms, will use SSReflect matrices.

In order to define this reduction we have used the formulas from the Hexagonal Lemma (3.2). Let us recall that the hypotheses are that exists a decomposition between some degrees and ϵ is an isomorphism that is to say, ϵ is invertible. We have followed a plan similar to the one presented in the previous subsection:

1. Defining the chain complex E.

2. Defining the chain complex morphisms between D and E, f' and g', and the homotopy operator h'.

3. Verifying the properties of the reduction given by (f', g', h').



The reduction generated by (f', g', h') will be named `reduction_CC2_D_E`. The different definitions and lemmas required to obtain this reduction are formalized in a similar way to the previous reduction. For this reason, we are not going to detail all the steps and we only focus on some ideas which have been instrumental for the proof. More concretely, we will present the work with block matrices, the proof about the top-left block has an inverse and the effective computation of this inverse. The last issue could be considered contrary to the idea of proving lemmas in abstract versions, but in this case, we need to *compute* the inverse to give the *definition* of one of the reduced matrices.

3.4.3.3 Block matrices

The representation of the matrices in SSReflect as finite functions allows us to express block matrices in an easy way. In this section we explain some properties of block matrices included in the `matrix` library which are useful for the representation of our matrices d'_0 and d'_1. First, we introduce some constructors of matrices using block matrices.

row_mx: Let $Al_{m,n1}$ and $Ar_{m,n2}$ be two matrices, returns the row block matrix $\left(\begin{array}{cc} Al & Ar \end{array}\right)$.

col_mx: Let $Au_{m1,n}$ and $Ad_{m2,n}$ be two matrices, returns the column block matrix $\left(\begin{array}{c} Au \\ Ad \end{array}\right)$.

block_mx: Let $Aul_{m1,n1}$, $Aur_{m1,n2}$, $Adl_{m2,n1}$ and $Adr_{m2,n2}$ be four matrices, returns the block matrix $\left(\begin{array}{cc} Aul & Aur \\ Adl & Adr \end{array}\right)$.

Let us show the last definition in SSReflect as an example.

```
Definition block_mx Aul Aur Adl Adr : 'M_(m1 + m2, n1 + n2) :=
  col_mx (row_mx Aul Aur) (row_mx Adl Adr).
```

Other interesting functions consists in slicing each matrix in four blocks. To get this aim, A has to have the type: `M_(m1+m2,n1+n2)`. The following definitions are defined to obtain every block: `ulsubmx`, `ursubmx`, `dlsubmx`, and `drsubmx`. For instance, `ulsubmx` returns the top-left matrix of `M` consisted of `m1` rows and `n1` columns.

After introducing the main definitions about block matrices, let us state a lemma in SSReflect to show the potential of working with blocks of matrices.

mul_row_block: Let $Al_{m,n1}$, $Ar_{m,n2}$, $Bul_{n1,p1}$, $Bur_{n1,p2}$, $Bdl_{n2,p1}$ and $Bdr_{n2,p2}$ be six matrices then $\left(\begin{array}{c|c} Al & Ar \end{array}\right) * \left(\begin{array}{c|c} Bul & Bur \\ \hline Bdl & Bdr \end{array}\right) = \left(\begin{array}{c|c} Al * Bul + Ar * Bdl & Al * Bur + Ar * Bdr \end{array}\right)$.

To illustrate why it is interesting to work with blocks, we introduce the lemma (with standard mathematical notation) which proves the boundary condition over the differential map $\bar{d}'$.

As hypothesis, we have that $d'_1 * d'_2 = 0$, in other words, $\left(\begin{array}{c|c} \varepsilon & \varphi \\ \hline \psi & \beta \end{array}\right) * \left(\begin{array}{c} \eta \\ \hline \gamma \end{array}\right) = 0$. Therefore, the following conditions are verified:

1. $\varepsilon * \eta + \varphi * \gamma = 0$ which implies that $\varphi * \gamma = -\varepsilon * \eta$
2. $\psi * \eta + \beta * \gamma = 0$

Expanding the proposition to prove $\bar{d}'_1 * \bar{d}'_2 = 0$ we obtain that it is necessary to prove $(\beta - \psi * \varepsilon^{-1} * \varphi) * \gamma = 0$. By the distributive property, the previous equality is equivalent to $\beta * \gamma - \psi * \varepsilon^{-1} * \varphi * \gamma = 0$. Using (1), it is converted to $\beta * \gamma - \psi * \varepsilon^{-1} * (-\varepsilon * \eta) = 0$. Then applying the fact of that ε is an invertible matrix, the proposition is reduced to $\beta * \gamma + \psi * \eta = 0$. Finally, using the commutative property of matrix addition, we exactly obtain the hypothesis (2).

Moreover, a required tool in SSReflect are *casts*. They help us to change the type of an object in another type which is not directly convertible. Let us see an example which has appeared in our development.

After computing the reordered matrix d'_1, which represents m sequences of n elements, we build the SSReflect matrix associated with it (`mx_of_seqmx m n d0'`) which consists of `m` rows and `n` columns. Then, we are interested in applying the lemma `submxK` which says that if $M : M_{m1+n1,m2+n2}$ then M can be defined as a block matrix composed by its four blocks. This statement in SSReflect is the following one:

```
block_mx (ulsubmx M)(ursubmx M)(dlsubmx M)(drsubmx M)= M.
```

But let us note that we cannot apply it directly because to use this lemma the matrix need to have $m1 + n1$ rows (instead of m) and $m2 + n2$ columns (instead of n). Then it is compulsory that d'_1 is a matrix of this type $M_{m1+n1,m2+n2}$. Namely, the top-left block of the matrix is a matrix $M_{sdvf,sdvf}$, considering $sdvf$ as the size of the discrete vector field obtained from d_1. Let us note that in order to build the SSReflect matrix of d_1 using the function `mx_of_seqmx` is necessary to provide its dimensions. Taking everything into account, we define

the reordered matrix d'_1 as an abstract matrix representing a matrix $M_{m,n}$ in the following way:

```
mx_of_seqmx (sdvf + (m - sdvf))(sdvf + (n - sdvf)) d1'
```

Let us note that we can prove that `(sdvf + (m - sdvf))= m` since `sdvf` (number of vectors of the vector field) is always lower or equal than `m`, the number of rows of d_0. If this condition does not fulfill then the equality is false since we are working over natural numbers and `(m - sdvf)` would not be a natural number.

This decision affects to other definitions, for instance to the definition of the morphism

$$h'_0 = \left(\begin{array}{c|c} \varepsilon^{-1} & 0 \\ \hline 0 & 0 \end{array}\right)$$

which is created by blocks. Then h'_0 has type $M_{sdvf+(n-sdvf),\ sdvf+(m-sdvf)}$. Taking into account the properties about a reduction which have to be proved, for instance, $g_1 * f_1 + d'_1 * h'_1 + h'_0 * d'_0 = \mathrm{id}$, we can see that $h'_0 : M_{sdvf+(n-sdvf),\ sdvf+(m-sdvf)}$ should be multiplied to $d'_0 : M_{m,n}$. Both matrices, however, cannot be multiplied since the types are not directly convertible for the system. So, we will need to cast the type of h'_0 to $M_{n,m}$ as we can see below.

```
Definition h0_CC2 :'M_(n,m):=
  (castmx (sdvf_n_sdvf, sdvf_m_sdvf)
  (block_mx (epsilon_inverse dvf M1) 0 0 0)).
```

Let us analyze this definition. Every matrix which is defined with `block_mx` is a matrix $M_{m1+n1,\ m2+n2}$ depending on the size of the blocks given to build this matrix. In our case, `(block_mx (epsilon_inverse dvf M1)0 0 0))` returns a matrix of type $M_{sdvf+(n-sdvf),\ sdvf+(m-sdvf)}$. For the previous reason, we want that this matrix is $M_{n,\ m}$. To prove it, we use the definition `castmx` to change the type of this matrix in another one that the system does not consider directly convertible. But the equality between the old dimensions and the new ones can be proved. The definition `castmx` receives as parameters: a matrix and the corresponding proofs to change the dimensions of this matrix. For instance, the first proof is the lemma `sdvf_n_sdvf` which states that `sdvf + (n - sdvf)` is equal to `n`. Paying attention to the definition `h0_CC2`, and namely the parameters of `castmx`, the matrix $M_{sdvf+(n-sdvf),\ sdvf+(m-sdvf)}$ is converted into a matrix

$M_{n,\,m}$.

The occurrence of this type of situations (where explicit casts are necessary) increases the effort in proofs that, otherwise, would be considered easy (from a strictly mathematical point of view). In order to ease these proofs we prove some lemmas about `castmx` which let us work with them in a convenient way. Let us present a simple example with its proof.

```
Lemma addmx_cast : forall (R : ringType) (m1 n1 m2 n2 : nat)
  (eq_mn1:(m1 = m2)*(n1 = n2))(A:'M[R]_(m1,n1))(B:'M[R]_(m1,n1)),
  ((castmx eq_mn1 A) + (castmx eq_mn1 B))= castmx eq_mn1 (A + B).
Proof.
  move => R m1 n1 m2 n2 [eq_mn1 eq_mn1'] A B.
  move: (eq_mn1) (eq_mn1') A B.
  rewrite eq_mn1 eq_mn1' => eq_mn_0 eq_mn_1 A B.
  by rewrite !castmx_id.
Qed.
```

This lemma means that the cast (given by `eq_mn1`) of the addition of two matrices `A` and `B`, is equal to the addition of the cast of each matrix. This type of proofs follows always the same pattern. In the first line, the hypotheses are transferred to the context. Then, the equalities given by `eq_mn1` are generalized and then they are used to rewrite our particular case. Finally, we obtain `castmx ((m2 = m2) * (n2 = n2)) M = M` and it is proved using the lemma `castmx_id` which says that if the equalities of the dimensions of the matrix, which allow us to change the type, are of the following form: `(k = k)` then the matrix is equal to the matrix applying this concrete cast.

3.4.3.4 Proving that $|\varepsilon| = 1$

We want to stress the proof of that the determinant of the top-left block matrix ε (submatrix of d'_0) is 1. In particular, we will prove that ε is a lower triangular matrix with $1's$ on the diagonal. Then the determinant of this matrix is computed as the product of the elements of the diagonal. As a result, we also obtain that ε is invertible.

The steps of the proof are the following ones. First, we will show that ε is a triangular matrix with $1's$ on the main diagonal. This is an executable matrix

(sequence of sequences) obtained from our algorithm. On the other hand, we will define the notion of an abstract lower triangular matrix M with $1's$ on the diagonal. With this definition, we will prove that $det\ M = 1$ since M is invertible: $M * M^{-1} = id$. Afterwards, the equivalence between both definitions, the abstract and the executable ones, will be proved. Finally, we will be able to prove that the SSReflect matrix built from the top-left submatrix of d'_0 verifies the abstract definition of a triangular matrix. In addition, we define an executable function which computes the inverse of a lower triangular matrix with $1's$ on the main diagonal and we prove that this definition is equivalent to the definition of the inverse of SSReflect in our particular case, a lower triangular matrix with $1's$ on the diagonal. Let us note that if we only focus on the proof it would be enough with proving that the inverse of this matrix exists on the contrary, we need to be able to compute the inverse since the reduced matrix is obtained thanks to it.

3.4.3.4.1 Executable triangular matrix

As we said, the reordered matrix from our algorithm $M = \left(\begin{array}{c|c} \varepsilon & \varphi \\ \hline \psi & \beta \end{array}\right)$ has as top-left submatrix a lower triangular matrix with $1's$ on the diagonal. Namely, this submatrix is ε and is a square matrix whose dimension is given by the size of the ordered discrete vector field. Indeed we can prove a more general result, if we have a vector field and a list of relations obtained from M which satisfy the properties of an admissible discrete vector field encoded by the definition of `Vecfieldadm` (given in Subsection 2.4.2.1), we can also prove that the reordered matrix verifies that its top-left submatrix is a lower triangular matrix with $1's$ on the main diagonal.

The definition of a lower triangular matrix above a sequence of sequences is given by two conditions. The former one is that the entries on the diagonal are 1's ($\forall\ k,\ k < size\ M \rightarrow M[k,k] = 1$) and the latter one is that the entries which are above the diagonal are 0's ($\forall\ r\ s, r < size\ M\ \wedge\ s < size\ M\ \wedge r < s \rightarrow M[r,s] = 0$). This definition in SSReflect is as follows.

```
Definition lower_triangular_seq n (M:matZ2) :=
  (forall k, k < (n+1) -> (nth 0 k (nth [::] k M)) = 1%R) /\
  (forall r s, (r < (n+1)) && (0 <= s < (n+1)) && (r < s) ->
  (nth 0 s (nth [::] r M)) = 0%R).
```

In order to prove that the top-left submatrix of the reordered matrix (using `reorderM_dvf`) verifies this definition, two lemmas are required. The first one ensures that the values of the elements which are on the diagonal are 1's.

```
Lemma diagonalis1_gen (dvf:vectorfield)(M:matZ2) rel:
  Vecfieldadm M dvf rel ->
  (forall k, k < (size dvf) ->
  nth 0 k (nth [] k (reorderM_dvf dvf M)) = 1%R).
```

A brief idea of the proof is the following one. Let us imagine that we have an admissible discrete vector field `dvf` $= ((a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n))$ sorted according to the maximum lengths of the paths. Then, we reorder the rows and columns of M with the orders $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$, respectively. Let us recall that $\forall\, i, M[a_i, b_i] = 1$ due to the second property of a discrete vector field consequently, the value of the entries of the diagonal (indexed by (a_i, b_i)) of the reordered matrix using `dvf` are $1's$.

The second lemma expresses that the values which are above the diagonal are 0's. Let us see its statement and a brief idea of its proof.

```
Lemma onDiagonal0_gen (n:nat) M dvf rel:
  let n := size dvf in Vecfieldadm M dvf rel ->
  (forall (r s:nat), (r < n) && (s < n) && (r < s) ->
  nth 0 s (nth [] r (reorderM_dvf dvf M)) = 0%R).
```

A sketch of the proof is the following one. Let us denote by `ir` the r-th element of the sequence composed by the first components of every pair of `dvf` and by `js` the s-th element of the sequence composed by the second components of every pair of `dvf`. After the realignment, the entry `(r,s)` of the reordered matrix corresponds with the entry `M[ir,js]`. Then this condition in SSReflect is:

```
(reorderM_dvf dvf M)[r, s] = M[ir,js].
```

Consequently, we will have to prove that `M[ir,js] = 0`. We divided the problem in cases, the first one, `M[ir,js]= 0` is exactly the proposition to prove. The other one, `M[ir,js]` $\neq$ `0` will produce a contradiction. As `js` $\in$ `(snd dvf)`, we can obtain that $\exists$`is`, `is` $\in$ `(fst dvf)` such as `(is,js)` $\in$ `dvf`. Then we split into two cases depending on whether a path between `ir` and `is` exists.

- There exists a path between `ir` and `is`.

  This means there exists a relation `p` generated by the vector field such as `ir` and `is` belong to `p` and `head p = ir`. For instance, `p` could be $[\texttt{ir}, \texttt{i}_1, \ldots, \texttt{i}_k, \texttt{is}]$. Due to `M[ir,js]` $\neq$ `0` and `(is,js)` $\in$ `dvf` we obtain that the relation `[is,ir]` exists using the lemma `indif0_order` presented in Subsection 2.4.2.2. So, a path exists between `is` and `ir`. Finally, it is proved that this generates a contradiction about the admissibility property. In particular, we know that if `p = [p1,p2]` then `p = [ir,p2]`, and then the relation `(ir::p2)` exists in our relations. Using the property which is in charge of generating the transitive closure we obtain that the relation `[is,ir,p2]` has to be in the set of relations. But on the other hand, `is` $\in$ `p` so, `is` $\in$ `[ir,p2]`, and therefore the relation `[is,ir,p2]` has repeated elements.

- There exists no path between `ir` and `is`.

  We have that the relation `[is,ir]` belongs to the set of relations. Consequently, let `g` be the function to compute the maximum length of the paths and `rel` be the list of relations; then we can obtain that `g ir rel < g is rel`. On the other hand, we have that `r < s`. So, the vector which is in the position `r` will have associated a higher value of `g` that what is in the position `s`. Taking into account the notation of `ir` and `is` we can conclude that `g is rel <= g ir rel`. Finally, we obtain a contradiction using the transitive property: `g is rel < g is rel`.

■

3.4.3.4.2 Abstract triangular matrix

First, we show the abstract definition of a lower triangular matrix.

```
Definition lower_triangular n (A:'M[F]_n) :=
  forall (i j : 'I_n), i <= j -> A i j = (i == j)%:R.
```

Then we will prove that the determinant of a lower triangular matrix with $1's$ on the main diagonal is equal to 1.

```
Lemma lower_triangular_det_1 : forall n (A:'M[F]_(n+1)),
  (lower_triangular A) -> \det A = 1.
```

It will be proved applying induction on n. The equality $n = 0$ implies that $det\ A = A(0,0)$, since A has only one entry. On other cases, $(n = n'.+1)$ we have as inductive hypothesis

$\forall$`A:M_n'+1, lower_triangular A` $\rightarrow$ `det A` $= 1$.

Moreover, $\det A = \det \left(\begin{array}{c|c} Aul & Aur \\ \hline Adl & Adr \end{array} \right)$ being $Aul : M_{n'}$, $Aur : M_{n',1}$, $Adl : M_{1,n'}$ and $Adr : M_{1,1}$. We have proved that $Aur = 0$ and $Adr = 1$ and A is a lower triangular matrix with $1's$ on the main diagonal. So, we have to prove that $\det \left(\begin{array}{c|c} Aul & 0 \\ \hline Adl & 1 \end{array} \right) = 1$ or equivalently to $\det Aul * 1 = 1$. Now, we could apply the inductive hypothesis if we prove `(lower_triangular Aul)`. Finally, this is true since A was a lower triangular matrix by hypothesis. With this result, we can obtain other interesting results, such as A is invertible (`lower_triangular_invertible`) and $A * A^{-1} = \mathrm{id}$ (`lower_triangular_product`).

3.4.3.4.3 Equivalence between both representations

Up to now, we have presented two definitions of a lower triangular matrix with $1's$ on the diagonal. On the one hand, we have obtained that the top-left submatrix of the reordered matrix verifies the definition of `lower_triangular_seq`. On the other hand, if a matrix M fulfills the definition `lower_triangular` then we obtain that $det\ M = 1$.

We include in this subsection the lemma that establishes that the two previous representations of a lower triangular matrix are equivalent. Thanks to this, we will be easily able to obtain properties about the first representation, such as that its determinant is 1, using the second one as we have shown before.

```
Lemma lower_triangular_seqE : forall n (M:'M[R]_(n+1)),
 lower_triangular_seq n (seqmx_of_mx M) <-> (lower_triangular M).
```

3.4.3.4.4 Implementation to compute the inverse of a lower triangular matrix with $1's$ on the main diagonal

Our aim is implementing an algorithm to compute the inverse of a lower

triangular matrix (sequence of sequences) with $1's$ on the diagonal because it is used in the computation of the reduced matrices. Moreover, it is necessary to check that this algorithm is equivalent to the abstract function `invmx` defined in SSReflect for this same task. Namely, this algorithm computes properly the inverse of a matrix if the input matrix is a lower triangular matrix with $1's$ on the diagonal. The process which we carry out to prove this equivalence is based on the methodology explained in [DMS12a]. Let us illustrate schematically this idea in Figure 3.4.

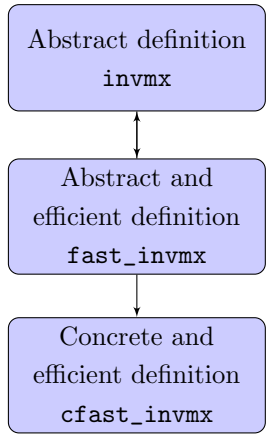


Figure 3.4: Bridges between inverse matrices

First, we introduce the abstract efficient function to compute this inverse, `fast_invmx`. This definition was implemented with the help of G. Gonthier, A. Mörtberg, and V. Siles who are members of the Formath project. Let us stress that it is a recursive function.

```
Fixpoint fast_invmx (m : nat) : 'M[R]_m -> 'M[R]_m :=
  match m return 'M[R]_m -> 'M[R]_m with
  | S p => fun (M : 'M[R]_(1 + p)) =>
        let: N := fast_invmx (drsubmx M) in
        block_mx 1%:M 0 (- N *m dlsubmx M) N
  | 0 => fun _ => 1%:M
end.
```

Then it is proved the following lemma which establishes that `fast_invmx` and `invmx` obtain the same result when the matrix satisfies the condition `lower_triangular`.

```
Lemma fast_invmxP m (M : 'M[R]_m) (H : lower_triangular M) :
  fast_invmx M = invmx M.
```

Therefore, we are sure that the function `fast_invmx` computes properly the inverse of a lower triangular matrix with $1's$ on the diagonal. But this definition is still not executable, so let us define a computable function, using functions defined over sequences instead of over SSReflect matrices. These functions appear in [DMS].

```
Fixpoint cfast_invmx (m : nat) (M : seqmatrix R) :=
  match m with
  | S p => let: N := cfast_invmx p (drsubseqmx 1 1 M) in
           block_seqmx (seqmx1 _ 1) (seqmx0 _ 1 p)
           (mulseqmx (oppseqmx N) (dlsubseqmx 1 1 M)) N
  | O => seqmx1 _ O
end.
```

As we can see both definitions are quite similar, the differences coming only from the used data types. Finally, we prove a morphism lemma linking the effective definition to its abstract counterpart.

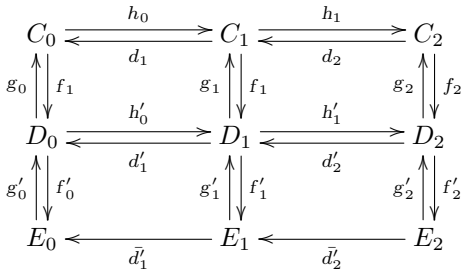
```
Lemma cfast_invmxP : forall (m : nat),
  {morph (@seqmx_of_mx R m m):
    M / fast_invmx M >-> cfast_invmx m M}.
```

3.4.4 Composing reductions

Up to now, we have built two reductions, a reduction of the initial chain complex to the reordered one (`reduction_CC2_C_D`) in Section 3.4.2 and the other one from the ordered chain complex to the reduced one (`reduction_CC2_D_E`) in Section 3.4.3. In this section we will build a reduction from the initial chain complex to the reduced one.

3.3.4.1 Construction of a reduction from two reductions

We are actually to prove a more general result (Proposition 1.29) where a reduction $\rho'' = (f'', g'', h'') : C \Rightarrow E$ is built from two reductions: $\rho = (f, g, h) : C \Rightarrow D$ and $\rho' = (f', g', h') : D \Rightarrow E$ where $f'' = f' \circ f$, $g'' = g' \circ g$ and $h'' = h + g \circ h' \circ f$. Taking into account the corresponding diagram:



We consider three chain complexes which will name C, D, and E and two reductions $reduct\_eq : C \Rightarrow D$ and $reduct : D \Rightarrow E$.

```
Variables C D E:chaincomplex.
Variable reduct_eq : reduction_CC2 C D.
Variable reduct : reduction_CC2 D E.
```

We prove Theorem 1.29 which says that the composition of two reductions is a reduction. In this way, we can define a reduction between C and E. This result will be proved mainly applying the particular properties given by the reductions ρ and ρ', using the command `rewrite` of SSReflect. Then, it will be easy to obtain that the chain complex morphisms f'' and g'' and the homotopy operator h'' satisfy the properties of a reduction, `is_reduction_red_red`. Finally, we build the reduction with the definition `Reduction_red_red`.

```
Lemma is_reduction_red_red:
  is_reduction_CC2 chaincomplex_morphism_f''
    chaincomplex_morphism_g'' homotopy_op_h''.

Definition Reduction_red_red :=
  Build_reduction_CC2 is_reduction_red_red.
```

We have seen that we can define a reduction $\rho'' : C \Rightarrow E$ from two reductions: $\rho : C \Rightarrow D$ and $\rho' : D \Rightarrow E$. Therefore, we can apply this result to our particular case. Let us use the definition `Reduction_red_red` with the two reductions which have been built in Subsection 3.4.2 and in Subsection 3.4.3.

```
Definition Reduction_red_red_rs :=
  Reduction_red_red reduction_CC2_C_D reduction_CC2_D_E.
```

3.5 The homology groups of the chain complexes in a reduction are isomorphic

In this section, we introduce an abstract definition of reduction `reduction_VS` between two chain complexes C and D. These chain complexes consist of two homomorphisms or linear applications between vector spaces instead of two matrices (namely, sequences of sequences) as in the definition of `chaincomplex` presented in Section 3.3. This new definition is going to allow us to define the concept of homology. We will prove from the previous definition that the dimension of the homology is the same for both chain complexes. Then we prove that any two vector spaces over a field F having the same dimension are isomorphic. This has been carried out thanks to A. Mörtberg, V. Siles, and J. Heras. Finally, we will prove that the reduction obtained in the previous subsection `Reduction_red_red_rs` (verifying the properties of `reduction_CC2`) satisfies the definition of `reduction_VS`.

3.5.1 A reduction preserves the Betti numbers

First of all, we define an abstract reduction $\rho = (f, g, h)$ between two chain complexes C and D which consists of three vector spaces and two homomorphisms which represent the differential maps. A chain complex is consisted of two homomorphisms of vector spaces satisfying the boundary condition. In SSReflect the definition of an abstract chain complex is as follows.

```
Definition is_ChainComplex_VS (K : fieldType)
  (V0 V1 V2 : vectType K)(d2 : 'Hom(V2,V1))(d1 : 'Hom(V1,V0)):=
  (d1 \o d2 = \0)%VS.
```

```
Record ChainComplex_VS (K : fieldType) :=
 { V0 : vectType K;
   V1 : vectType K;
   V2 : vectType K;
   d2 : 'Hom(V2,V1);
   d1 : 'Hom(V1,V0);
   CC_VS_proof: is_ChainComplex_VS d2 d1
  }.
```

Let us note that we use the type `vectType` to define a vector space structure. Moreover, the differential maps are functions which are defined as homomorphisms between two vector spaces. Apart from this definition, we will need to define a chain complex morphism (`ChainComplexMorphism_VS`) and a homotopy operator (`HomotopyOperator_VS`) to be able to introduce the abstract concept of reduction (`Reduction_VS`).

```
Record Reduction_VS (K : fieldType) (C D : ChainComplex_VS K) :=
 { f : ChainComplexMorphism_VS C D;
   g : ChainComplexMorphism_VS D C;
   h : HomotopyOperator_VS C;
   Reduction_VS_proof: is_Reduction_VS f g h
 }.
```

Let us show diagrammatically the situation reflected in the previous definition:

$$\begin{array}{ccccc} C_0 & \underset{d_1}{\overset{h_0}{\rightleftarrows}} & C_1 & \underset{d_2}{\overset{h_1}{\rightleftarrows}} & C_2 \\ {\scriptstyle g_0}\uparrow\downarrow{\scriptstyle f_0} & & {\scriptstyle g_1}\uparrow\downarrow{\scriptstyle f_1} & & {\scriptstyle g_2}\uparrow\downarrow{\scriptstyle f_2} \\ D_0 & \xleftarrow[d'_1]{} & D_1 & \xleftarrow[d'_2]{} & D_2 \end{array}$$

Then, the aim of this section is proving that if we have a reduction between two chain complexes C and D then C and D have the same Betti numbers. As we explained in Subsection 1.1.1 the homology of a chain complex C, denoted by $H(C)$, is the quotient between the kernel and the image. In our case, $H(C) = \ker d_1 / \operatorname{im} d_2$. The Betti numbers are the dimensions of the homology vector spaces, because we are working with coefficients over a field. This pair of

definitions in SSReflect is as follows. Let us note the quotient is denoted in SSReflect as `:\:`.

```
Variables (K : fieldType) (C : ChainComplex_VS K).

Definition Homology := ((lker (d1 C)) :\: (limg (d2 C)))%VS.
Definition Betti := \dim Homology.
```

Then, we will introduce different properties which are necessary to prove the final result which establishes that the homology of both chain complexes are isomorphic.

```
Lemma reduction_preserves_betti : Betti C = Betti D.
```

Since `\dim (A :\: B)= \dim A - \dim B`. The previous lemma is equivalent to prove that `\dim (lker (d1 C))- \dim (limg (d2 C))= \dim (lker (d1 D))- \dim (limg (d2 D))`.

The mathematical sketch of the proofs of these lemmas are followed step by step during the proof in SSReflect. Most of these steps are given applying lemmas which are already proved in the library of vector spaces in SSReflect.

1. $f(ker\ d_1)$ is a subspace of $ker\ d'_1$

   This lemma is proved as follows: let $x \in ker\ d_1$ then $d_1 x = 0$. We have to prove that $fx \in (ker\ d'_1)$ i.e., $d'_1(fx) = 0$. Using that f is a chain complex morphism we obtain that $d'_1(fx) = f(d_1\ x)$. Finally, as $d_1 x = 0$ then $f(d_1\ x) = 0$

2. $f(\mathrm{im}\, d_2)$ is a subspace of $im\ d'_2$

   Let $x \in (\mathrm{im}\, d_2)$ then $\exists y$ such as $d_2\ y = x$; thus $f(d_2\ y) = fx$. Applying that f is a chain complex morphism we obtain $d'_2(fy) = fx$. On the other hand, $fx \in \mathrm{im}\, d'_2$ if $\exists z$ such as $d'_2\ z = fx$. The result is proved taking as value z the element fy.

3. $dim\ (f(\ker d_1)) = dim\ (\ker d'_1)$

   Using (1) we obtain $dim\ (f(\ker d_1)) \leq dim\ (\ker d'_1)$. On the other hand, we have that $g(\ker d'_1) \leq \ker d_1$ in a similar way. As f is an application we apply it in this result in both sides: $f\ g\ (\ker d'_1) \leq f\ (\ker d_1)$ (`limg_monotone`

from the SSReflect library is used here). Therefore, $dim\ (f\ g\ (\ker d'_1)) \leq dim\ (f\ (\ker d_1))$. Moreover, $fg = \mathrm{id}$ is one of the reduction properties; then $dim\ (\ker d'_1) \leq dim\ (f\ (\ker d_1))$. Finally, we can apply the Sandwich Lemma and obtain $dim\ (f\ (\ker d_1)) = dim\ (\ker d'_1)$.

4. $dim\ (f\ (\mathrm{im}\, d_2)) = dim\ (\mathrm{im}\, d'_2)$

   This lemma is proved in an analogous way to the previous one.

5. $Betti(C) = Betti(D) + dim\ (\ker d_1 \cap \ker f_1) - dim\ (\mathrm{im}\, d_2 \cap \ker f_1)$

   By expanding the definition of $Betti(D)$, we get $Betti(C) = dim\ (\ker d'_1) - dim\ (\mathrm{im}\, d'_2) + dim\ (\ker d_1 \cap \ker f_1) - dim\ (\mathrm{im}\, d_2 \cap \ker f_1)$.

   First, we define two equalities between the dimension of $\ker d_1$ with the dimension of $\ker d'_1$ and about $\mathrm{im}\, d_2$ and $\mathrm{im}\, d'_2$. We have that $dim\ (\ker d_1) = dim\ (\ker d_1 \cap \ker f_1) + dim\ (f\ (\ker d_1))$ using the lemma `limg_ker_dim` already proved in SSReflect. Using (3) $dim\ (\ker d_1) = dim\ (\ker d_1 \cap \ker f_1) + dim\ (\ker d'_1)$. With an analogous argument, we obtain $dim\ (\mathrm{im}\, d_2) = dim\ (\mathrm{im}\, d_2 \cap \ker f_2)\ + dim\ (\mathrm{im}\, d'_2)$. Then, the result is obtained subtracting the previous equalities.

6. $dim\ (\mathrm{im}\, d_2 \cap \ker f_1) \leq dim\ (\ker d_1 \cap \ker f_1)$

   Since C is a chain complex then $d_1 d_2 = 0$, therefore $\mathrm{im}\, d_2$ is a subspace of $\ker d_1$. Applying the intersection of $\ker f$ to this inequality,

   $$\mathrm{im}\, d_2 \cap \ker f_1 \leq \ker d_0 \cap \ker f_1$$

   Therefore,

   $$dim\ (\mathrm{im}\, d_2 \cap \ker f_1) \leq (dim\ (\ker d_1 \cap \ker f_1))$$

7. $dim\ (\mathrm{im}\, d_2 \cap \ker f_1) \geq dim\ (\ker d_1 \cap \ker f_1)$

   Let us see that $(\ker d_1 \cap \ker f_1)$ is a subspace of $(\mathrm{im}\, d_2 \cap \ker f_1)$. If $x \in (\ker d_1 \cap \ker f_1)$ then $d_1 x = fx = 0$. On the other hand, if $x \in (\mathrm{im}\, d_2 \cap \ker f_1)$, $\exists\ y$ such that $d_2 y = x \wedge fx = 0$. Let us prove that $\exists\ y$ such as; $d_2 y = x$. Using the condition of the reduction $d_2 h_1 + h_0 d_1 + g_1 f_1 = id$ applied to the element x, we obtain

   $$d_2 h_1 x + h_0 d_1 x + g_1 f_1 x = id x$$

This can be simplified since $h_0 d_1 x = 0$ (due to $d_1 x = 0$) and $g_1 f_1 x = 0$ (due to $f_1 x = 0$). Therefore,

$$d_1 h_1 x = x.$$

Thus, we can define $y = h_1 x$. Finally, as $\ker d_0 \cap \ker f_1$ is a subspace of $\operatorname{im} d_1 \cap \ker f_1$ then $dim\ (\ker d_1 \cap \ker f_1) \leq dim\ (\operatorname{im} d_2 \cap \ker f_1)$.

8. $dim\ (\operatorname{im} d_2 \cap \ker f_1) = dim\ (\ker d_1 \cap \ker f_1)$

   This lemma follows from by (6) and (7).

Finally, we can prove the lemma:

$$Betti\ C = Betti\ D.$$

Since $Betti\ C = Betti\ D + dim\ (\ker d_1 \cap \ker f_1) - dim\ (\operatorname{im} d_2 \cap \ker f_1)$ by (5) and $dim\ (\operatorname{im} d_2 \cap \ker f_1) = dim\ (\ker d_1 \cap \ker f_1)$ by (8), we obtain $Betti\ C = Betti\ D$.

■

3.5.2 Two vector spaces with the same dimension are isomorphic

To complete the proof that the homology groups of C and D are isomorphic, we conclude proving the following result: any two vector spaces of the same dimension are isomorphic. The idea of the mathematic proof is as follows: let X and Y be vector spaces satisfying that $|X| = |Y|$ then we have $xs = \{x_1, \ldots, x_n\}$ a basis for X and $ys = \{y_1, \ldots, y_n\}$ for Y. Afterwards, we define a linear application f which consists in $\forall\ i, f(x_i) = y_i$. This is extended by linearity and it is proved that exists a linear isomorphism between X and Y.

However, we cannot prove this lemma in SSReflect following this same idea because a vector space X is defined inside a bigger structure U which is a `vectType` structure. The type `vectType` is defined over a field K and X is a subspace of it, `{vspace U}`. Furthermore, we cannot define a linear application from X to Y in SSReflect. Then we have to define one between U and V (where U (respectively V) includes X (Y)). In this way, we can obtain an image in Y of the elements which belong to X. However, the elements which are in the

complement of X ($U \backslash X$) cannot be defined in the same way. Consequently, it is not likely to define an isomorphism in this way.

To overcome this difficulty, we define the notion of isomorphism as a bijective application f from V to V verifying that the image of X applying f is Y. Let us note that both X and Y are in V. In SSReflect, the notion of isomorphism (a bijective homomorphism) is not defined but the definition of homomorphism does exist (`'Hom(U,V)`). But we will not use this notion to define the isomorphism because f has to be defined from V to itself. Then we use the notation `f:'End(V)` to define this isomorphism. Moreover, this function has to be also bijective and verifies that the image of $f(X)$ is Y.

```
Variable (K : fieldType).
Variable (V : vectType K).

Definition isomorphic (X Y : {vspace V}) := exists (f : 'End(V)),
  bijective f /\ (f @: X == Y)%VS.
```

Thanks to this notion, we can state the lemma to prove.

```
Lemma same_dim_isomorphic (V1 V2 : {vspace V})
  (hdim : \dim V1 = \dim V2): isomorphic V1 V2.
```

This proof consists of providing two applications f and g where g is the inverse of f so that the lemma is verified. Concretely, f is going to be the linear application associated with the change of basis which we define as follows.

```
Definition base_change m (M M' : 'M[K]_m):=
  invmx (row_ebase M) *m row_ebase M'.
```

The way to define a change of basis is using the bases of the vector spaces. In SSReflect, a set of vectors $\{v_j\}_{j \in J}$ of V is a basis if this set is linearly independent and spans V. Let us note that in the definition `base_change`, we see the vector spaces as matrices and in this way we define the application as a matrix, too.

3.5.3 The computed reduction is a `reduction_VS`

Up to now, we have introduced an abstract definition to the notion of reduction, `reduction_VS`. Then, we have proved that the dimensions of the homology of both chain complexes are the same from this definition. The purpose of this subsection is proving that the Betti number are the same for the chain complexes which are in the reduction given by the definition `reduction_CC2` presented in Section 3.3. In other words, we want to build a reduction (`reduction_VS`) from a reduction defined with `reduction_CC2`. Let us show the sketch of this process in Figure 3.5.

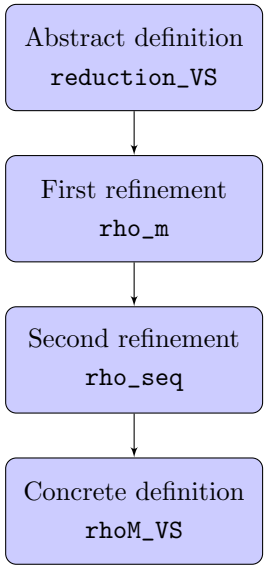


Figure 3.5: Refinements of the reduction concept

3.5.3.1 First refinement

In a first refinement we change the general finite vector spaces for the type `matrixVectType` which is represented by its matrix since every linear map between finite dimensional based vector spaces can be represented as a matrix. As the chain complex is finitely generated, the functions which appear in the definition of a reduction are represented as SSReflect matrices `'M[K]_(m,n)` with

`m` and `n` being their corresponding dimensions. For instance, the chain groups of the bigger chain complex are `C0`, `C1`, and `C2`. Then, this chain complex is defined by `d1` and `d2`. So, the morphisms required in the reduction are also matrices of similar types.

```
Variable (K : fieldType) (v0 v1 v2 :nat).

Definition C0 := (matrixVectType K 1 v0).
Definition C1 := (matrixVectType K 1 v1).
Definition C2 := (matrixVectType K 1 v2).

Variables (d1: 'M[K]_(vdim C0,vdim C1))
          (d2: 'M[K]_(vdim C1,vdim C2)).
```

Then, we have to build the necessary abstract chain complexes, homomorphisms and the homotopy operator using the definitions given in Subsection 3.3 in order to define an abstract reduction. The definition `is_Reduction_VS_rho` is going to involve the properties to verify, which come from the elements which have been built. Therefore, we define the reduction in the following way.

```
Definition rho_m := Build_Reduction_VS is_Reduction_VS_rho.
```

Let us remark that, for instance, the function which gives the properties of a chain complex is `is_ChainComplex_VS` and this function receives two homomorphisms (two differential maps), but in this case we work with SSReflect matrices. Therefore, we will have to build the linear application given for every matrix using `LinearApp`. Let us see how the chain complex `C` is built in an easy way.

```
Lemma is_ChainComplex_VS_C :
  is_ChainComplex_VS (LinearApp d1) (LinearApp d2).
Definition C_m := Build_ChainComplex_VS is_ChainComplex_VS_C.
```

3.5.3.2 Second refinement

In a second refinement we define our differential maps as sequences of sequences over a general field K, i.e., as executable structures since we want to reduce these differential maps. The rest of the other morphisms which appear in the

reduction are not going to redefine because we do not want to obtain explicitly these matrices but only proving properties about them. Let us note that only the structures which we want to compute are redefined in an executable way. Then, we only show the changes with relation to the previous refinement.

```
Variable (K : fieldType).
Variables (d1 d2 d'1 d'2: seqmatrix K).
```

Finally, we build a new reduction given the differential maps as sequences. The necessary conditions to construct it are given as hypotheses by the definition `rho_m`, then most of the lemmas are already proved. Therefore, the new reduction is expressed in the following way.

```
Definition rho_seq :=
  rho_m boundary_C boundary_D f0d1 f1d2 g0d'1 g1d'2
        f0g0 f1g1 f2g2 h1h0 f1h0 f2h1 h0g0 h1g1 d2h1_h0d1_g1f1.
```

3.5.3.3 Final refinement

We want to define a reduction with the previous definition `rho_seq` from two concrete chain complexes defined as sequences of sequences of $\mathbb{Z}_2$ (which were used to compute the reduction introduced in Section 3.4).

```
Variables (C D: chaincomplex)(rho: reduction_CC2 C D).
```

In this case, we will have to prove the properties about a reduction included in `rho_seq` over our reduction `rhoM_VS`. These proofs will be easy since they belong to the properties verified by `reduction_CC2`.

```
Definition rhoM_VS := rho_seq boundary_C boundary_D f0d1 f1d2
  g0d'1 g1d'2 f0g0 f1g1 f2g2 h1h0 f1h0 f2h1 h0g0 h1g1
  d2h1_h0d1_g1f1.
```

Finally, using these chain of refinements we can build a reduction between abstract chain complexes from a reduction between computable chain complexes. On the other hand, we have proved in previous section that the dimension of the homology groups for both chain complexes are the same. In this way, we can ensure that the homology groups (homology vector spaces, in our case) of the

initial concrete chain complex and the reduced one have the same dimension, and as a consequence they are isomorphic.

The formalization of this development takes up 394 lines. Concretely, it involves 12 definitions and only 17 lemmas since many of the lemmas which we have needed have been proved in an easy way applying lemmas already included in SSReflect.

3.6 Another reduction: Collapses

In this chapter, we have proved that if we have an admissible discrete vector field from a matrix which represents a differential map of the whole chain complex then we are going to be able to reduce the chain complex. On the other hand, we can build an admissible discrete vector field in many different ways, for instance, looking for the elements in the columns instead of the rows or even with other algorithms. In this section, we explain how to reduce a chain complex using collapses as a method to obtain an admissible discrete vector field.

Definition 3.3. A simplicial collapse of a cellular complex K is the deleting of a pair of cells (σ, τ), of dimensions respectively p-1 y p where σ is not face of any other cell in dimension p of K except τ.

The result of a simplicial collapse is another cellular complex, which is a subset of the original one preserving the homology.

In this case, every reduction is generated by an admissible discrete vector field consisted of only one vector. If a vector is selected is because the row which belongs is composed by only one element equal to 1. Geometrically, the vector (i, j) relates an $(n+1)$-simplex to an n-simplex, so this reduction consists of deleting the rows or columns of the matrices which correspond with these n-simplex and $(n+1)$-simplex. In general, we use (geometric) collapses to obtain a simplicial complex from another one.

3.6.1 Example

Let us see an example computing some of the consecutive reductions of the image of Figure 3.6. Afterwards, we show the matrices of the different reductions which

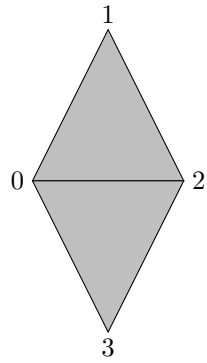


Figure 3.6: A triangled rhombus

consist of the chain complex associated with the image.

$$d_1 = \begin{array}{c} \\ \{0\} \\ \{1\} \\ \{2\} \\ \{3\} \end{array} \begin{array}{c} \begin{array}{ccccc} \{0,1\} & \{0,2\} & \{0,3\} & \{1,2\} & \{2,3\} \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix} \end{array} \quad d_2 = \begin{array}{c} \\ \{0,1\} \\ \{0,2\} \\ \{0,3\} \\ \{1,2\} \\ \{2,3\} \end{array} \begin{array}{c} \begin{array}{cc} \{0,1,2\} & \{0,2,3\} \end{array} \\ \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \end{array}$$

Let us note that in this moment, we cannot reduce the matrix d_1 since every row is composed by more of an element equal to 1. Therefore, we will focus on reducing d_2. The first row has only one element equal to 1, so this row will be selected and namely, the entry $(0,0)$. Then we will reduce the matrices deleting the edge $\{0,1\}$ and the triangle $\{0,1,2\}$, so both differential maps are modified. Let us recall that we have to order the matrix according to the admissible discrete vector field, in this case $\{(0,0)\}$. With this vector the image is as we can see in Figure 3.7 and the matrices associated with it as follows.

$$d_1' = \begin{array}{c} \\ \{0\} \\ \{1\} \\ \{2\} \\ \{3\} \end{array} \begin{array}{c} \begin{array}{cccc} \{0,2\} & \{0,3\} & \{1,2\} & \{2,3\} \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{array} \quad d_2' = \begin{array}{c} \\ \{0,2\} \\ \{0,3\} \\ \{1,2\} \\ \{2,3\} \end{array} \begin{array}{c} \begin{array}{c} \{0,2,3\} \end{array} \\ \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \end{array}$$

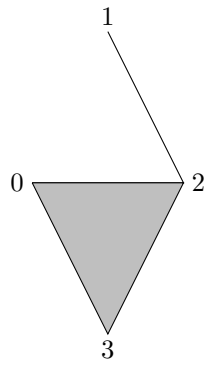


Figure 3.7: First collapse

Going on with this reduction, let us note that an element can be selected both of d'_1 and of d'_2. For instance, we pay attention to d'_1. In this case the only element of the matrix which can be selected is the vector $(1, 2)$. In other words, we delete the vertex $\{1\}$ and the edge $\{1, 2\}$. Afterwards, we will have to order the rows and the columns according to the vector field $\{(1, 2)\}$ and we obtain the reduced matrix (see Figure 3.8).

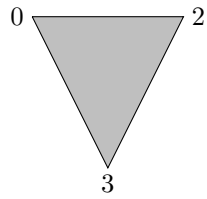


Figure 3.8: Second collapse

$$d''_1 = \begin{array}{c} \\ \{0\} \\ \{2\} \\ \{3\} \end{array} \begin{array}{c} \begin{array}{ccc} \{0,2\} & \{0,3\} & \{2,3\} \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \end{array} \qquad d''_2 = \begin{array}{c} \\ \{0,2\} \\ \{0,3\} \\ \{2,3\} \end{array} \begin{array}{c} \begin{array}{c} \{0,2,3\} \end{array} \\ \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \end{array}$$

Finally, following with the same method, Image 3.6 is reduced to a vertex where the matrices of the chain complex are as follows. Let us note that $\bar{d}_1$ and

$\bar{d}_2$ are empty matrices but $\bar{d}_1$ consist of one row but no column.

$$\bar{d}_1 = () \quad \bar{d}_2 = ()$$

3.6.2 Formalization of the reduction using collapses

Let us stress that many functions which were necessary to implement the RS algorithm (see Section 2.2) have been useful here, too. One of the differences between both reductions is the way of selecting a vector. In the RS algorithm (Section 2.1) we select any element whose value is 1; however, using collapses we can only select the elements whose value is 1 and it is the unique one in the row. Consequently, the function which chooses is as follows.

```
Definition find_collapse_rows (s : matZ2) :=
  search (fun i => (count (@pred1 Z2 1) i) = 1%N) s.
```

The implemented function `search` is in charge of selecting a row which must satisfy a condition. In this case, it looks for the rows with an unique element whose value 1. Moreover, we need to obtain its position.

```
Definition position_collapse_aux (s : matZ2) (l : seqZ2) :=
  (pair (index l s) (index 1 l)).
```

Then we have to generate the relations as we have explained in Section 2.2 looking for the elements of the column whose value is 1. In this case, it would not be necessary to store these relations because we use the relations to order the vectors of the vector field but in this case, we only work with an unique vector in each vector field. But in this way, we will be able to prove that this vector field is admissible and therefore, to apply the obtained result in the Section 3.4. This will be easy to prove as the vector field from a matrix consisted of $\{(i, j)\}$, the possible relations are of the form: $i > k$ with k lower and equal to the number of rows of the matrix. Therefore, it is impossible that these relations generate cycles. Finally, we reorder the matrix and obtain the reduced matrix (see Section 3.2). Let us recall that the reordered matrix has the following form $M' = \left(\begin{array}{c|c} \varepsilon & \varphi \\ \hline \psi & \beta \end{array}\right)$. In our particular case, this matrix is always as follows: $M' = \left(\begin{array}{c|c} 1 & 0 \\ \hline \psi & \beta \end{array}\right)$. Therefore, the reduced matrix will be computed as $\bar{M}' = \beta$ instead of $\bar{M}' = \beta - \psi\varepsilon^{-1}\varphi$.

Then the formalization of the reduction can be obtained as a particular case of the reduction using the RS algorithm. Every time which this matrix is reduced (deletes a row and a column) the previous and the following matrix of the chain complex will also be reduced as we said in Section 3.2. This process can be repeated until there are not more rows of any matrices of the chain complex composed by only one non-null element.

In this case, we simply have to prove that this function returns a vector field which verifies the properties of `Vecfieldadm`. This development only requires 7 definitions and 13 lemmas. After that, every lemma proved in previous sections over a `Vecfieldadm` can be transferred to the output of this new algorithm.

Chapter 4

Formalization of the Basic Perturbation Lemma (BPL)

In this chapter, we introduce a formalization of the Basic Perturbation Lemma (in short, *BPL*). This is an essential result in Computational Algebraic Topology [RS02]. In the literature, there are several ways of proving this lemma (see for instance, [Gug72], [BL91], and [RS97]). Moreover, there are works related to the formalization of the BPL. The non-graded case of this lemma was proved in *Isabelle/HOL* [ABR08]. Furthermore, a particular case of the BPL was also proved in Coq using bicomplexes [DR11]. Now, we show a formalization of the general case in SSReflect but with finitely generated structures. Let us recall that SSReflect only works with finite types; so, it can seem that this technology can restrict our development. Nevertheless, it is enough because we are interested in applying this lemma to the computation of the homology of a 2D digital image, a case where every chain group is finitely generated. Indeed, in this context, this lemma is applied to a reduction where most of the differential maps are null. An alternative, in the case of 2D digital images, consists of proving a very particular case of the BPL removing in the chain complexes all the differential maps except two of them. This particular lemma can be easily obtained from the general case.

In the next section, we introduce a mathematical proof of the BPL. Then, its formalization in SSReflect is presented. Finally, we show a proof of the Vector-

Field Reduction Theorem for 3-truncated structures (Theorem 3.1) applying the BPL. This proof is an alternative to the one introduced in Chapter 3. Let us recall the statement of the BPL introduced in Subsection 1.1.5.

Theorem 4.1 (Basic Perturbation Lemma, BPL [Bro67])**.** Let us consider a reduction $\rho = (f, g, h) : C_* \Rightarrow \widehat{C}_*$ between two chain complexes (C_*, d) and $(\widehat{C}_*, \widehat{d})$, and δ a perturbation of d. Furthermore, the composite function δh is assumed *locally nilpotent*, in other words, given $x \in C_*$ there exists $m \in \mathbb{N}$ such that $(\delta h)^m(x) = 0$. Then a perturbation $\widehat{\delta}$ can be defined for the differential map $\widehat{d}$ and a new reduction $\rho' = (f', g', h') : (C_*, d+\delta) \Rightarrow (\widehat{C}_*, \widehat{d}+\widehat{\delta})$ can be constructed.

4.1 Mathematical proof of the BPL

The proof of the BPL presented in [RS12] is based on two results. The former named Decomposition Theorem (Subsection 4.1.1), builds a decomposition of a chain complex from a reduction of it. The latter is a Generalization of the Hexagonal Lemma (Subsection 4.1.2). First, we introduce the Decomposition Theorem.

4.1.1 Decomposition Theorem

Theorem 4.2 (Decomposition Theorem)**.** Let $\rho = (f, g, h) : (C_*, d) \Rightarrow (\widehat{C}_*, \widehat{d})$ be a reduction. This reduction is equivalent to a decomposition: $C_* = A_* \oplus B_* \oplus C'_*$ where:

1. $C_* \supset C'_* = \operatorname{im} g$.
2. $A_* \oplus B_* = \ker f$.
3. $C_* \supset A_* = \ker f \cap \ker h$.
4. $C_* \supset B_* = \ker f \cap \ker d$.
5. The chain complex morphisms f and g are inverse isomorphisms between C'_* and $\widehat{C}_*$.
6. The arrows d and h are module isomorphisms between A_* and B_*.

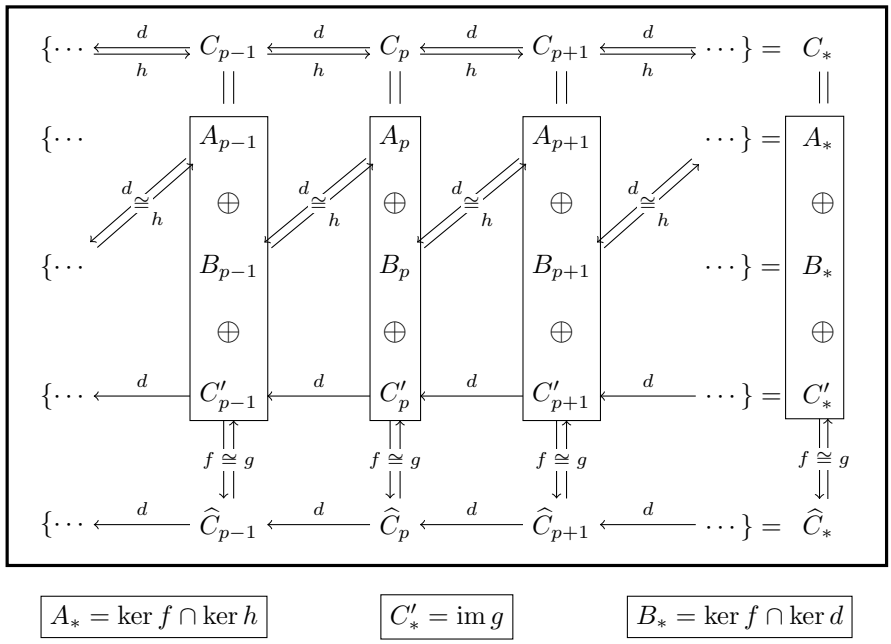


Figure 4.1: Diagram of Decomposition Theorem

This decomposition is represented in Figure 4.1.

Proof.

First, we prove that $C_* = \ker f \oplus \operatorname{im} g$. Given $x \in C_*$, $gf(x)+dh(x)+hd(x) = x$ by Property 2 of the reduction ρ. Then,

- $gf(x) \in \operatorname{im} g$.
- $dh(x) + hd(x) \in \ker f$.

  This is proved applying the morphism f, $f(dh(x) + hd(x)) = fdh(x) + fhd(x) = dfh(x) + fhd(x) = d(0) + 0 = 0$ by Property 3 of ρ.
- $\operatorname{im} g \cap \ker f = \{0\}$.

  Let $x \in \operatorname{im} g$ be an element different from 0, then $\exists y \neq 0$ such that $g(y) = x$. Therefore, $f(x) = f(g(y)) = y$ because $fg = \mathrm{id}$ by Property 1 of ρ. Finally, $x \notin \ker f$.

Second, let us see that $\ker f$ can be split into two disjoint subsets. Given $x \in \ker f$, the equality of Property 2 of ρ: $gf(x) + dh(x) + hd(x) = x$ is reduced to $dh(x) + hd(x) = x$. Then,

- $dh(x) \in \operatorname{im} d_{n+1} \subset \ker d_n$ by the boundary condition of d. It implies that $dh(x) \in \ker f \cap \operatorname{im} d \subset \ker f \cap \ker d$. Given $x \in \ker d$ we have $d(x) = 0$. As $dh(x) + hd(x) = x \Rightarrow dh(x) = x \Rightarrow x \in \operatorname{im} d$. Finally, $dh(x) \in \ker f \cap \operatorname{im} d = \ker f \cap \ker d = B_*$.
- $hd(x) \in \operatorname{im} h_n \subset \ker h_{n+1}$ by Property 5 of ρ. It implies that $hd(x) \in \ker f \cap \operatorname{im} h \subset \ker f \cap \ker h$. Given $x \in \ker h$ we have $h(x) = 0$. As $dh(x) + hd(x) = x \Rightarrow hd(x) = x \Rightarrow x \in \operatorname{im} h$. Then, $hd(x) \in \ker f \cap \operatorname{im} h = \ker f \cap \ker h = A_*$.
- $A_* \cap B_* = \{0\}$.
  Let $x \in \ker h$ verifying $x \neq 0$ then, $x \notin \ker d$ due to Property 2 of ρ.

Finally, we focus on proving the isomorphisms shown in Figure 4.1.

- $f|_{C'_*} = g^{-1}|_{C'_*}$ where $f|_{C'_*} : C'_* \to \widehat{C}_*$.
  Due to Property 1 of ρ, we have $fg = \mathrm{id}$, then we only need to prove $gf = \mathrm{id}$.

  Given $x \in \operatorname{im} g$, then $\exists\, y \in C_*$ such as $x = g(y)$. Then, by Property 2 of ρ:

$$\begin{aligned} & gf(x) + dh(x) + hd(x) = x \\ \Rightarrow\ & gf(x) + dhg(y) + hdg(y) = x \\ \Rightarrow\ & gf(x) + dhg(y) + hgd(y) = gf(x) + d(0) + 0 = x \\ \Rightarrow\ & gf(x) = x \end{aligned}$$

  Here, we used Property 4 of ρ and that g is a morphism.

- $d|_{A_*} = h^{-1}|_{A_*}$ where $d|_{A_*} : A_* \to B_*$.
  - If $x \in A_*$, let us prove that $hd(x) = x$. As $x \in A_* = \ker f \cap \ker h$, then Property 2 of a reduction $gf(x) + hd(x) + dh(x) = x$ is reduced to $hd(x) = x$
  - If $x \in B_*$, we have to prove that $dh(x) = x$. As $x \in B_* = \ker f \cap \ker d$, then the condition $gf(x) + hd(x) + dh(x) = x$ is reduced to $dh(x) = x$.

In this way, we obtain the diagram of Figure 4.1 where the components which do not appear are null. ■

4.1.2 Generalization of the Hexagonal Lemma

The Hexagonal Lemma (Lemma 3.2) allows us to reduce only a chain group of a chain complex in a particular degree. It is possible to generalize this lemma applying the reduction to every degree simultaneously.

Theorem 4.3. Let $C = (C_p, d_p)_p$ be a chain complex. We assume that every chain complex is decomposed $C_p = A_p \oplus B_p \oplus C'_p$. The boundary maps d_p are then decomposed in 3×3 block matrices $[d_{p,i,j}]_{1 \leq i,j \leq 3}$. If every component $d_{p,2,1} : A_p \to B_{p-1}$ is an isomorphism, then the chain complex can be canonically reduced to a chain complex (C'_p, d'_p).

Proof.

Applying the formulas produced by the Hexagonal Lemma in each component, the desired reduction is obtained. Its components are:

$$d'_p = d_{p,3,3} - d_{p,3,1} d_{p,2,1}^{-1} d_{p,2,3} \qquad f_p = \begin{bmatrix} 0 & -d_{p,3,1} d_{p,2,1}^{-1} & 1 \end{bmatrix}$$

$$g_p = \begin{bmatrix} -d_{p,2,1}^{-1} d_{p,2,3} \\ 0 \\ 1 \end{bmatrix} \qquad h_{p-1} = \begin{bmatrix} 0 & d_{p,2,1}^{-1} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

It is not difficult to check the shown formulas satisfy the relations of a reduction stated in Definition 1.28. The components $d_{p,1,1}$, $d_{p,1,2}$, $d_{p,1,3}$, $d_{p,2,2}$ and $d_{p,3,2}$ do not play any role in the homological nature of C, but these components are not independent from the others, because of the relation $d_{p-1} d_p = 0$. ■

4.1.3 Proof of the BPL

The hypotheses of the BPL are:

1. A reduction $\rho = (f, g, h) : (C_*, d) \Rightarrow (\widehat{C}_*, \widehat{d})$.

2. A perturbation δ of the differential d.

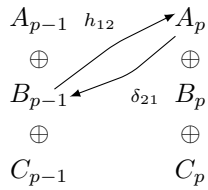
3. δh is locally nilpotent.

With these hypotheses it is necessary to build a new reduction $\rho' = (f', g', h') : (C_*, d+\delta) \Rightarrow (\widehat{C}_*, \widehat{d}+\widehat{\delta})$.

As we have a reduction $\rho : C_* \Rightarrow \widehat{C}_*$, we can apply the Decomposition Theorem. We obtain the diagram of Figure 4.1 where every chain group $C_* = A_* \oplus B_* \oplus C'_*$. Afterwards, we introduce a perturbation δ of the differential d, i.e. a morphism such that $(d+\delta)(d+\delta) = 0$. Then it is clear that $(d+\delta)$ can be depicted in nine blocks. If the component $(d+\delta)_{21} : A_* \to B_*$ is invertible, then the Generalization of the Hexagonal Lemma can be applied and the BPL proved.

Since $h_{12}d_{21} = \mathrm{id}$, $(d+\delta)_{21} = d_{21} + \delta_{21} = d_{21} + \delta_{21}h_{12}d_{21} = (\mathrm{id} + \delta_{21}h_{12})d_{21}$. As d_{21} is invertible (h_{21} is its inverse), we focus on proving that the other member of the product $(\mathrm{id} + \delta_{21}h_{12})$ is invertible.

As δh is locally nilpotent, i.e. for every $x \in C_*$, there exists $n \in \mathbb{N}$ satisfying $(\delta h)^n(x) = 0$, then $(\delta_{21}h_{12})^n(x) = 0$ since the unique non-null component of h is $h_{12} : B_* \to A_*$.



The nilpotency of $\delta_{21}h_{12}$ gives us an inverse of $(\mathrm{id} + \delta_{21}h_{12})$, Φ'.

$$\begin{aligned}\Phi' &= (\mathrm{id} + \delta_{21}h_{12})^{-1} = \mathrm{id} - \delta h + (\delta h)^2 - (\delta h)^3 + \ldots + (-1)^n(\delta h)^n + \cdots \\ &= \sum_{i=1}^{\infty} (-1)^i (\delta_{21}h_{12})^i\end{aligned}$$

Let us check that forall $x \in C_*$, $(\mathrm{id} + \delta_{21}h_{12})(\mathrm{id} + \delta_{21}h_{12})^{-1}(x) = x$.

$$
\begin{aligned}
&(\mathrm{id} + \delta_{21}h_{12})(\mathrm{id} - \delta h + (\delta h)^2 - (\delta h)^3 + \ldots + (-1)^n(\delta h)^n)(x) \\
&= (\mathrm{id} - \delta h + \ldots + (-1)^n(\delta h)^n + \delta h + \ldots - (-1)^n(\delta h)^n \\
&\quad + (-1)^{n+1}(\delta h)^{n+1} + \ldots)(x) \\
&= \mathrm{id}(x) + 0 = x.
\end{aligned}
$$

The last step is true because some of the addends are cancelled pairwise and the rest of them are null because of the nilpontency property. Therefore, $(d + \delta)_{21} = (\mathrm{id} + \delta_{21}h_{12})d_{21} : A \to B$ is invertible with $\Phi' h_{12}$ as inverse.

4.2 Formalization of the proof

The formalization of the proof of the BPL in SSReflect requires restricting the data structures to finitely generated chain complexes. These structures are presented in Subsection 4.2.2. Before introducing the BPL formalization, we give a brief explanation of the formalization of the kernel of a map in SSReflect. The representation chosen for this well-known notion in mathematics has important consequences in the rest of structures used in the formalization of the proof. Then, the Decomposition Lemma and the Generalization of the Hexagonal Lemma are formalized in Subsection 4.2.3 and in Subsection 4.2.4, respectively. These are the key ingredients used in the formalization of the BPL included in Subsection 4.2.5.

Before explaining the details of this formalization let us highlight the use of casts in our development. The rigid typing strategy of Coq makes sometimes difficult to translate mathematical intuitions. For instance, even if the expressions `i` and `i+1-1` are provable equal with `i` an integer number, they are not directly convertible in Coq. When integers are used as indexes (for instance, in degrees of chain complexes), problems increase. In order to show an example about it, we introduce some notations.

The differential of a chain complex C in degree i is usually defined as a morphism $d_i : C_i \to C_{i-1}$. In Coq, we can define d_i as `(diff C i)` whose type is a matrix `'M_(m C i, m C (i-1))` where `(m C i)` gives us the number of generators of C in degree `i`. A homotopy operator h in degree i in C is a morphism

$h_i : C_i \to C_{i+1}$. In Coq, we can define h_i as (`Ho H i`) whose type is a matrix `'M_(m C i, m C (i+1))`.

Then, if we compose both morphisms $d_{i+1}h_i : C_i \to C_{i+1-1}$, we will obtain in Coq a matrix of type `'M_(m C i, m C (i+1-1))`. This matrix is not directly convertible by Coq to `'M_(m C i)`. The standard solution in SSReflect consists in using casts, in this case `(Ho H i)*m (diff C (i+1))` is equal to:

```
castmx (cast1, cast2) ((Ho H i) *m (diff C (i+1)))
```

where `cast1` and `cast2` are the following lemmas.

```
Lemma cast1: m C i = m C i.
Lemma cast2: m C (i+1-1) = m C i.
```

In this way, we change the type of `(diff C (i+1-1))`. The pair of equalities of the first argument of `castmx` provides the change of the dimensions of `(diff C (i+1-1))`.

Being a general solution, the explicit using of casts makes notations cumbersome. In order to avoid these casts, at least in the main definitions of our development, we can take profit from the freedom of dealing with structures indexed over the integers. As an illustration, instead of working with the usual definition of the differential, we defined $d_i : C_{i+1} \to C_i$. In this way, the composition $d_{i+1}h_i$ is translated with the new definition into $d_ih_i : C_i \to C_i$. Moreover, the type of the obtained matrix is `'M_(m C i)`.

We will use systematically this kind of tricks. To this aim, we need a terminology expressing this shift of indexes. We will call *n-suspended up* or *n-suspended down* chain complex for a chain complex which is built moving up or down n degrees of a chain complex. Concretely, if we have a chain complex C and define n-suspended up chain complex C' from C, the differential in degree k of C is the same object than the differential in degree $k + n$ in C'.

This terminology is extended naturally to other graded structures such as chain morphisms, reductions, and so on.

4.2.1 The kernel of a map

The kernel of a finite map is defined by the kernel of the matrix which represents this map. The kernel of a matrix `A` in SSReflect is defined using `kermx A`, which is the row kernel of `A`. In other words, it is a square matrix whose row space consists of all `u` such that `u *m A = 0`. The row space of a matrix is the set of all possible linear combinations of its row vectors. Let us show the definition of the kernel in SSReflect.

```
Definition kermx m n (A: 'M_(m,n)): 'M_m :=
  copid_mx (\rank A) *m invmx (col_ebase A).
```

The kernel of a matrix `A` is defined as the inverse of `col_ebase` (the extended column basis of `A`), with the top rank `A` rows zeroed out. The kernel (`kermx A`) is defined as the product of two matrices. The first one, `copid_mx (\rank A)` is a square diagonal matrix with 0's in the first top (`\rank A`) rows and 1's in the rest of the diagonal. This matrix converts the first (`\rank A`) rows of the second one into zeros.

Two lemmas, which are included in the library, help us to understand better this definition:

```
Lemma mulmx_ker m n (A : 'M_(m, n)) : kermx A *m A = 0.

Lemma mulmxKV_ker m n p (A : 'M_(n, p)) (B : 'M_(m, n)) :
  B *m A = 0 -> B *m col_ebase A *m kermx A = B.
```

The first lemma establishes that the elements of the kernel of a matrix are made null when `A` is applied to the left. The second lemma establishes that `kermx A` is a partial right inverse of `col_ebase A`. Let us note that `col_ebase A *m kermx A` is the right identity if `B *m A = 0`, in other words, if `B` is included in `kermx A`.

In addition to the lemmas about the kernel proved in the library, we need to define other ones; for instance, the following one related to the intersection.

```
Lemma ker_intersection m1 n1 n2 (A:'M[K]_(m1,n1))
  (B:'M[K]_(m1,n2)):
  (kermx A :&: kermx B :=: kermx (row_mx A B))%MS.
```

This lemma states that the intersection of kernels of two matrices generates the same space that the kernel of a row matrix composed by both matrices.

As we have previously said, the kernel of a matrix `A` has the top (`\rank A`) rows zeroed out. In our development, we chose to delete these null rows; because if we worked with the original definition, we would obtain partial identities in some proofs. Due to this fact, we define a new kernel denoted by `ker_min`.

```
Definition ker_min (m n : nat) (M : 'M_(m,n)) :=
 (castmx ((Logic.eq_refl (m-\rank M)),
 (\rank M) + (m-(\rank M))) = m)
 (row_mx (@const_mx _ (m-\rank M)(\rank M)0) 1%:M)) *m (kermx M).
```

It consists of the product of a row matrix with the kernel (`kermx`). The row matrix is composed by a block of zeros with (`m-\rank M`) rows and (`\rank M`) columns and an identity matrix of (`\rank M`) rows and columns. This definition allows us to obtain the elimination required. Let us note that the cast in the definition is necessary to define the product properly. Then, the lemmas about the kernel are more thorny because some casts are also required. In this way, we delete the null rows of the matrix obtained from `kermx`. Anyway, both definitions generate the same space as we can see in the following lemma.

```
Lemma ker_min_kermx (m n : nat) (M : 'M_(m,n)) :
  (kermx M :=: (ker_min M))%MS.
```

The proof of this lemma can be divided into two parts using the inclusion of subsets (`A:=:B <-> A ⊆ B /\ B⊆ A`). Both lemmas are proved in the same way, applying a change of view with the lemma `submxP` (`A ⊆ B <-> ∃D, A = DB`). We obtain a proposition of this type:

```
exists D0 : 'M_(m - \rank M, m), ker_min M = D0 *m kermx M
```

This is proved giving a matrix `D0` which fulfills the proposition. In this case, `D0` is the row matrix given in the own definition of `ker_min`.

Let us recall that the kernel of A in SSReflect consists of the elements u which $uA = 0$. However, in our previous mathematical definitions the product is null when is applied to the right. Therefore, we have decided to represent the matrices with their transposes in SSReflect. For instance, if M is a matrix of dimension $m \times n$ this variable in SSReflect is defined with the following type.

```
Variable M: 'M[K]_(n,m).
```

This implies that the rows and columns in the matrix are swapped. Moreover, the product of matrices is also affected because it is reversed.

Then, along this chapter we will alternate both representations using the most suitable in each moment. This decision will be carried out to ease the reader the understanding of some topics. In our mathematical representation, given two matrices $M_{m,n}$ and $N_{n,l}$ their product is denoted as MN, whereas in Coq we define the matrices as `M:'M[K]_(n,m)` and `N:'M[K]_(l,n)` and the product is represented as `N *m M`.

4.2.2 Main mathematical structures

Following the methodology explained in Section 1.3, we have used SSReflect matrices to prove mathematical results; and we only use sequences to represent matrices when we want to compute. Therefore, we will use abstract structures to prove the BPL. Let us define a finitely generated chain complex with a countable list of chain groups.

```
Variable K : fieldType.

Record FGChain_Complex :=
  { m : Z -> nat;
    diff : forall i:Z, 'M[K]_(m (i + 1), m i);
    boundary : forall i:Z, (diff (i + 1)) *m (diff i) = 0}.
```

Some comments about this definitions are necessary. The chain complex definition contains a function denoted by `m` which obtains the number of generators for each degree as we said at the beginning of the section. Then, we can define the differentials using the matrix representation of these maps `forall i:Z, 'M[K]_(m (i + 1), m i)`. Two important design decisions have been included in this definition. Due to the definition of the kernel of a matrix in SSReflect we will work with transposed matrices due to the definition of kernel mentioned in Subsection 4.2.1. This implies that the product is also reversed. Furthermore, the degrees of the differentials have been increased in one unit to avoid using casts.

Now, the definitions of chain complex morphism and homotopy operator are introduced.

```
Record FGChain_Complex_Morphism (A B : FGChain_Complex) :=
  { M : forall i:Z, 'M[K]_((m A i),(m B i));
    M_well_defined : forall i:Z,
              (diff A i) *m (M i) = (M (i+1)) *m (diff B i)}.

Record FGHomotopy_operator (A : FGChain_Complex) :=
  { Ho : forall i:Z, 'M[K]_(m A i, m A (i+1)%Z)}.
```

With these previous definitions, we can define the notion of a reduction for a finitely generated chain complex. In the definition of reduction of a 3-truncated chain complex presented in Subsection 3.3, we needed to define the same property twice, one for each degree. Now, we define the properties in each degree at once.

```
Record FGReduction :=
  { C : FGChain_Complex;
    D : FGChain_Complex;
    F : FGChain_Complex_Morphism C D;
    G : FGChain_Complex_Morphism D C;
    H : FGHomotopy_operator C;
    ax1 : forall i:Z, (M G i) *m (M F i) = 1%:M;
    ax2 : forall i:Z, (M F (i+1)) *m (M G (i+1)) +
                      ((Ho H (i+1)) *m (diff C (i+1))) +
                      ((diff C i) *m (Ho H i)) = 1%:M;
    ax3 : forall i:Z, (Ho H i) *m (M F (i+1)) = 0;
    ax4 : forall i:Z, (M G i) *m (Ho H i) = 0;
    ax5 : forall i:Z, (Ho H i) *m (Ho H (i+1)) = 0}.
```

4.2.3 Formalization of the Decomposition Theorem

In this subsection, we focus on proving the Decomposition Theorem (Theorem 4.2): a reduction $\rho = (f, g, h) : C_* \Rightarrow \widehat{C}_*$ is equivalent to a decomposition $C_* = A_* \oplus B_* \oplus C'_*$ where $A_* = \ker\ f \cap \ker\ h$, $B_* = \ker\ f \cap \ker\ d$ and $C'_* = \operatorname{im} g$ satisfying the chain complex morphisms f and g are inverse isomorphisms between C'_* and $\widehat{C}_*$ and d and h are module isomorphisms between A_*

and B_*.

First, we focus on the decomposition of the chain groups. Let us denote the morphisms f and g of the reduction ρ in SSReflect as `F_rho` and `G_rho` and the homotopy operator as `H_rho`. Moreover, the differential of the chain complex C_* is `D_rho` and the differential of C'_* is `d_rho`.

We are going to define the decomposition through an isomorphism between the chain groups of C_* and $A_* \oplus B_* \oplus C'_*$. To build this isomorphism we have to define two functions which we call Fi_isom and Gi_isom satisfying for every degree i:

$$Fi\_isom\ i * Gi\_isom\ i\ = \mathrm{id} \wedge\ Gi\_isom\ i * Fi\_isom\ i = \mathrm{id} \tag{4.1}$$

These functions can be understood as a change of basis between C_* (where we work with the canonical basis) and $A_* \oplus B_* \oplus C'_*$. The function Fi_isom can be defined directly taking into account how every chain group is divided (see Figure 4.1).

$$Fi\_isom(i : Z) = \begin{pmatrix} ker\_min(F\_rho(i+1)) \cap ker\_min(H\_rho(i+1)) \\ ker\_min(F\_rho(i+1)) \cap ker\_min(D\_rho(i+1)) \\ G\_rho(i+1) \end{pmatrix}$$

Let us show the definition of Fi_isom in SSReflect.

```
Definition Fi_isom (i : Z):=
 (col_mx (ker_min (row_mx (F_rho (i+1)) (H_rho (i+1))))
         (col_mx (ker_min (row_mx (F_rho (i+1)) (D_rho i)))
         (G_rho (i+1)))).
```

Let us simply emphasize two aspects in the previous definition. First, the direct sum $A \oplus B$ generates the same space that the matrix `col_mx A B` defined in SSReflect. It is a square matrix whose row space is the concatenation of the row spaces of A and B. Second, the intersection $A \cap B$ is equivalent to `row_mx A B`. Moreover, the first block of the column matrix in both definitions,

$$ker\_min(F\_rho(i+1)) \cap ker\_min(H\_rho(i+1))$$

and

```
(ker_min (row_mx (F_rho (i+1))(H_rho (i+1))))
```

generate the same subspace, taking into account the lemma `ker_intersection` introduced in Subsection 4.2.1.

The following task is the definition of Gi_isom knowing that $\forall i, Gi\_isom(i)$ is a row matrix composed by three blocks ($m1\ m2\ m3$) satisfying the conditions in (4.1). Let us see the process followed to define Gi_isom.

$$\mathrm{id} = (Gi\_isom\ i)(Fi\_isom\ i) = (m1\ m2\ m3) \begin{pmatrix} ker\_min(F\_rho(i+1), H\_rho(i+1)) \\ ker\_min(F\_rho(i+1), D\_rho(i+1)) \\ G\_rho(i+1) \end{pmatrix}$$

$$\begin{aligned} & m1 * ker\_min(F\_rho(i+1), H\_rho(i+1)) + \\ & m2 * ker\_min(F\_rho(i+1), D\_rho(i+1)) + \\ & m3 * G\_rho(i+1) = \mathrm{id} \end{aligned} \tag{4.2}$$

Now, we try to link Equation 4.2 with Property 2 of the reduction ρ. In the formalization, this exactly corresponds with `ax2` of the definition `FGReduction` introduced in Subsection 4.2.2. In this case, `ax2` is as follows.

```
(F_rho (i+1)) *m (G_rho (i+1)) + ((H_rho(i+1)) *m (D_rho (i+1)))
 + ((D_rho i) *m (H_rho i)) = 1%:M
```

Then, our aim is associate each addend of this property with one of Equation 4.2. In this way, we will be able to obtain the values $m1$, $m2$ and $m3$ that satisfy the equality. Let us note that the first addend corresponds with $m3 * G\_rho(i+1)$ if $m3 = F\_rho(i+1)$. In addition, ker_min appears in the other addends of Equation 4.2. Then, we will have to apply some of the lemmas related to ker_min to obtain $m1$ and $m2$. In particular, there exists the lemma `mulmxKV_ker` (defined in Subsection 4.2.1) which will be useful in this case. It allows us to cancel the ker_min part. For instance, in order to obtain $m2$ we check that

```
(H_rho(i+1))*m (D_rho (i+1))*m row_mx(F_rho (i+1))(D_rho i)= 0
```

we can apply this lemma and obtain `HD *m col_ebase(FD)*m kermx (FD)= HD` where `HD` is `(H_rho(i+1)) *m (D_rho (i+1))` and `FD` is `row_mx(F_rho (i+1)) (D_rho i)`. In this way, we obtain the required simplification of `kermx`. Consequently, we can define

```
m2 = (H_rho(i+1)) *m (D_rho (i+1)) *m
     col_ebase(F_rho (i+1),D_rho i) *m castFiDi i
```

The component (`castFiDi i`) appears because we are working with `ker_min` instead of `kermx`. We recall that some casts are necessary with this new definition. Now, we prove why the previous condition `HD * FD = 0` is true. Multiplying both matrices, the condition is the following:

$$\begin{pmatrix} (H\_rho\ (i+1)) * (D\_rho\ (i+1)) * (F\_rho\ (i+1)) \\ (H\_rho\ (i+1)) * (D\_rho\ (i+1)) * (D\_rho\ i) \end{pmatrix} = 0$$

The upper block is 0 using the properties of the morphism F_rho and applying Property 3 of the reduction ρ. The lower block is null due to the boundary condition of D_rho.

The other block of Gi_isom is obtained in a similar way.

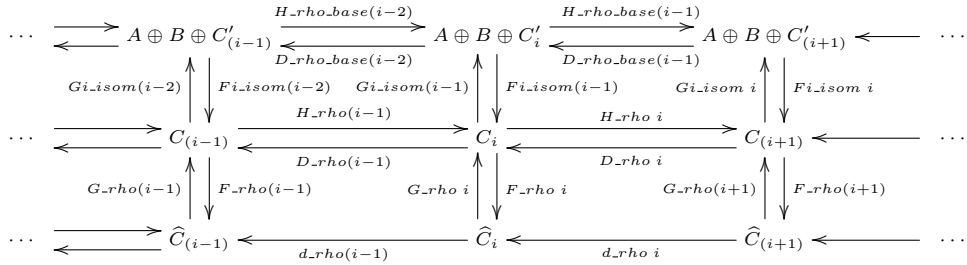
```
m1 = ((D_rho i) *m (H_rho i)) *m
     col_ebase(F_rho (i+1), H_rho (i+1)) *m castFiHi (i+1)
```

4.2.3.1 Conditions of the decomposition

The two functions Fi_isom and Gi_isom can be understood as a change of basis between the initial representation of the chain groups of C_* and the decomposed one $A_* \oplus B_* \oplus C'_*$. Now, this change of basis is useful to define the new representation of the differentials for $A_* \oplus B_* \oplus C'_*$. Also, this change of basis can be applied to the reduction ρ as a whole in order to obtain a reduction from $A_* \oplus B_* \oplus C'_*$ to $\widehat{C}_*$.

```
Definition D_rho_base (i:Z):=
 (Fi_isom (i+1) *m (D_rho (i+1)) *m (Gi_isom i)).

Definition H_rho_base (i:Z):=
 (Fi_isom i) *m (H_rho (i+1)) *m (Gi_isom (i+1)).

Definition F_rho_base (i:Z) := Fi_isom (i) *m (F_rho (i+1)).

Definition G_rho_base (i:Z) := G_rho (i+1) *m Gi_isom (i).
```

These definitions are represented in the following diagram.



It is not difficult to prove that the previous definitions are:

$$\texttt{F\_rho\_base i} = \begin{pmatrix} 0 \\ 0 \\ \mathrm{id} \end{pmatrix} \qquad \texttt{G\_rho\_base i} = \begin{pmatrix} 0 & 0 & \mathrm{id} \end{pmatrix}$$

$$\texttt{H\_rho\_base i} = \begin{pmatrix} 0 & 0 & 0 \\ \texttt{H\_aux i} & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{4.3}$$

$$\texttt{D\_rho\_base i} = \begin{pmatrix} 0 & \texttt{D\_aux i} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \texttt{d\_rho (i+1)} \end{pmatrix}$$

With this representation we directly obtain that $f = g^{-1}$ on the third component of the decomposition. In this case, these maps correspond with the third block both of (`F_rho_base i`) and of (`G_rho_base i`).

The other condition $d = h^{-1}$ is defined between the block with coordinates $(1, 2)$ of `(D_rho_base i)` and the block with coordinates $(2, 1)$ of `(H_rho_base i)` (taking into account that the matrices associated with the maps have been transposed). These blocks have been denoted by `(D_aux i)` and `(H_aux i)`, respectively. Then, the two lemmas to prove in SSReflect are:

```
Lemma D_aux_H_aux : forall i:Z, (D_aux i) *m (H_aux i) = 1%:M.
Lemma H_aux_D_aux :
  forall i:Z, (H_aux (i+1)) *m (D_aux (i+1)) = 1%:M.
```

The proof of these lemmas is based on the fact that a reduction can be defined from the decomposed chain complex $A_* \oplus B_* \oplus C_*$ to $\widehat{C}_*$ (a composition of an

isomorphism and a reduction). This reduction includes properties such as the following lemma `gf_Dh_hD_base`.

```
Lemma gf_Dh_hD_base : forall i:Z,
  F_rho_base (i+1) *m G_rho_base(i+1) +
  H_rho_base (i+1) *m D_rho_base(i+1) +
  D_rho_base i *m H_rho_base i = 1%:M.
```

Both lemmas are obtained from this property. We can see in (4.3) that both `H_aux` and `D_aux` are included in the definitions of `H_rho_base` and `D_rho_base`, respectively. Concretely, if we multiply the matrix `H_rho_base` with `D_rho_base` (we obtain a matrix composed by 3×3 blocks) where the block $(1,1)$ and $(2,2)$ is the product of `D_aux` and `H_aux` and the one of `H_aux` and `D_aux`, respectively, as we can see below.

$$\texttt{H\_rho\_base (i+1)*m D\_rho\_base(i+1)} =$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & \texttt{H\_aux (i+1)*m D\_aux (i+1)} & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Furthermore, if we simplify the previous lemma `gf_Dh_hD_base` we obtain the following equalities.

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathrm{id} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & \texttt{H\_aux (i+1)*m D\_aux (i+1)} & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} \texttt{D\_aux i *m H\_aux i} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} \texttt{D\_aux i *m H\_aux i} & 0 & 0 \\ 0 & \texttt{H\_aux (i+1)*m D\_aux (i+1)} & 0 \\ 0 & 0 & \mathrm{id} \end{pmatrix} = \mathrm{id}$$

4.2.4 Formalization of the Generalization of the Hexagonal Lemma

Let us recall that Theorem 4.3 establishes that given a chain complex where each chain group C_k can be decomposed into 3 parts: $A_k \oplus B_k \oplus C'_k$, where the block $(2,1)$ of every differential of C_k is an isomorphism, then a canonical reduction can be defined $\rho : C_k \Rightarrow C'_k$.

The first step towards the formalization of this lemma is defining its hypotheses. As every chain group is divided into three parts, every differential consists of nine blocks.

```
Variables (m1 m2 m3 : Z -> nat).
Variable Di : forall i:Z,
  'M[K]_(m1(i+1)+(m2(i+1)+ m3(i+1)),((m1 i)+ ((m2 i) + (m3 i)))).
Hypothesis boundary_Di : forall i:Z, Di (i+1) *m Di i = 0.

Definition CH := Build_FGChain_Complex boundary_Di.
```

In this way, we have a chain complex `CH` where each differential (`Di i`) consists of nine blocks. Let us denote the blocks of each differential in the following way.

$$\texttt{Di i} = \begin{pmatrix} \texttt{(d11 i)} & \texttt{(d12 i)} & \texttt{(d13 i)} \\ \texttt{(d21 i)} & \texttt{(d22 i)} & \texttt{(d23 i)} \\ \texttt{(d31 i)} & \texttt{(d32 i)} & \texttt{(d33 i)} \end{pmatrix}$$

As we are working with transposed matrices, the hypothesis over the block $(2,1)$ of every differential is defined over the block $(1,2)$ of its transpose. This means that there exists matrices for these blocks, which are defined as `d12_1`, satisfying the condition given in `d12_invertible`.

```
Variable d12_1 : forall i : Z, 'M[K]_(m2 i, m1 (i + 1)).
Hypothesis d12_invertible :
  forall i:Z, (d12 (i+1)) *m (d12_1 (i+1)) = 1%:M
  /\ (d12_1 (i+1)) *m (d12 (i+1)) = 1%:M.
```

Afterwards, we define the morphisms `fi` and `gi` and the homotopy operator `hi` to build a reduction of the chain complex `CH`. These maps are detailed in the proof of Theorem 4.3 of Subsection 4.1.2. First,

`gi i` $= \begin{pmatrix} -(\texttt{d32 i}) * (\texttt{d12\_1 i}) & \texttt{0} & \texttt{1} \end{pmatrix}$. We remark that (`gi i`) has type `'M_(m3(i+1), m1(i+1)+m2(i+1)+m3(i+1))`. In this definition, we increase a degree in the chain morphism to avoid the use of casts.

Taking this into account, `(fi i)` is defined according to the definition of (`gi i`).

$$\texttt{fi i} = \begin{pmatrix} \texttt{0} \\ \texttt{(- (d12\_1 (i+1))*m (d13 (i+1))} \\ \texttt{1} \end{pmatrix}$$

In a similar way, the rest of the matrices are defined as follows.

$$\texttt{di\_up i} = \texttt{(d33 (i+1))} - \texttt{((d32 (i+1)) *m (d12\_1 (i+1))) *m (d13 (i+1))}$$

$$\texttt{Di\_up i} = \texttt{(diff CH (i+1))}$$

$$\texttt{hi\_up i} = \begin{pmatrix} \texttt{0} & \texttt{0} & \texttt{0} \\ \texttt{(d12\_1 (i+1))} & \texttt{0} & \texttt{0} \\ \texttt{0} & \texttt{0} & \texttt{0} \end{pmatrix}$$

We can note that `Di_up` is the differential of the 1-suspended up chain complex from `CH`, `CH_up`. Let us represent the components of the reduction in the following diagram.

$$\begin{array}{ccccccccc} \cdots & \rightleftarrows & A \oplus B \oplus C'_i & \underset{Di\_up\ (i-1)}{\overset{hi\_up(i-1)}{\rightleftarrows}} & A \oplus B \oplus C'_{(i+1)} & \underset{Di\_up\ i}{\overset{hi\_up\ i}{\rightleftarrows}} & A \oplus B \oplus C'_{(i+2)} & \rightleftarrows & \cdots \\ & & gi(i-1) \uparrow \downarrow fi(i-1) & & gi\ i \uparrow \downarrow fi\ i & & gi(i+1) \uparrow \downarrow fi\ (i+1) & & \\ \cdots & \leftarrow & \widehat{C}_i & \xleftarrow[di\_up\ (i-1)]{} & \widehat{C}_{(i+1)} & \xleftarrow[di\_up\ i]{} & \widehat{C}_{(i+2)} & \leftarrow & \cdots \end{array}$$

Finally, the proof that the components of the previous diagram define a reduction is an exercise based on using rewriting tactics. In conclusion, the reduction defined as `rhoHL` is a 1-suspended up reduction because we have not built from the initial chain complex `CH`, but `CH_up`.

4.2.5 Formalization of the BPL

First, we define the objects which appear in the BPL. These are a reduction and a perturbation of the top chain complex of the reduction. Moreover, the

boundary condition of the chain complex where the differentials are the sum of the differential of `(C rho)` and the perturbation is hold.

```
Variable K: fieldType.
Variable rho : FGReduction K.
Variable delta : forall i:Z, 'M[K]_(m (C rho)(i+1), m (C rho) i).
Hypothesis boundary_dp :
  forall i:Z, ((diff (C rho)(i+1) + delta (i+1))
              *m ((diff (C rho)i + delta i) = 0.
```

In addition, we will assume the nilpotency hypothesis. Let us remember that a function $f : X \to X$ satisfies the nilpotency property if $\forall x,\ x \in X,\ \exists n$ such that $f^n(x) = 0$. Since we are working in a finite context, we can consider n as the maximum of the values which verify the nilpotency property for all the elements of the function. Then, our function is not only locally nilpotent but also globally nilpotent. Let us note that `pot_matrix` is a function which we have defined to formalize the power of a matrix.

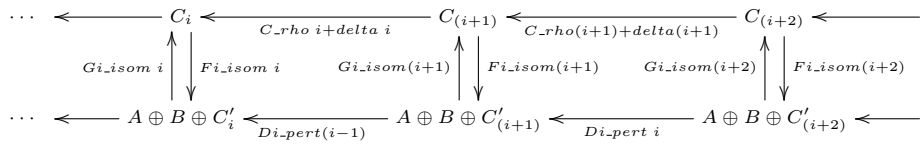
```
Variable (n : nat).
Hypothesis nilpotency_hp : forall i:Z,
  (pot_matrix (delta i *m (Ho (H rho) i)) n = 0).
```

With these hypotheses, we define a reduction from the chain complex whose differentials are: `(diff (C rho)i)+ delta i` and `delta`. To this aim, we apply the Decomposition Lemma, introduced in Subsection 4.2.3, to `(C rho)`. We can use the isomorphism given by `Fi_isom` and `Gi_isom`. This isomorphism allows us to decompose the chain group into three parts and consequently the matrices are divided into nine blocks as we saw in Subsection 4.2.3.1. Let us show the new differential in SSReflect.

```
Definition Di_pert(i:Z):=
  (Fi_isom rho (i+1)) *m (diff (C rho) (i+1) + delta (i+1))
  *m (Gi_isom rho i).
```

In this definition, we move up again one degree the chain complex to avoid the use of casts in the last component, `(Gi_isom rho i)`. Then `(Di_pert i)` is a matrix which is defined from the degree `(i+2)` to the degree `(i+1)`. In the same way, the perturbation is also increased in one degree.

```
Definition delta_new (i:Z):=
  (Fi_isom rho (i+1)) *m (delta (i+1)) *m (Gi_isom rho i).
```



The hypotheses assumed for the initial chain complex are transferred, i.e. proved over the decomposed chain complex.

```
Lemma boundary_dp_new :
  forall i:Z, Di_pert (i+1) *m Di_pert i = 0.
Lemma nilpotency_hp_new : forall i:Z,
  pot_matrix ((delta_new i) *m (H_rho_base rho i)) n = 0.
```

Afterwards, we will apply the Generalization of the Hexagonal Lemma presented in Subsection 4.1.2 and formalized in Subsection 4.2.4. In this step, every differential is divided into nine blocks. Then, the first hypothesis of the lemma which says that every chain complex is decomposed $C_p = A_p \oplus B_p \oplus C'_p$ is already verified. Afterwards, we focus on proving the other hypothesis of the lemma in SSReflect; i.e., that the block $(1,2)$ of `Di_pert` is invertible. Then, a series of equalities about the block $(1,2)$ is shown where we denote `M_12` as the block $(1,2)$ of a matrix `M`.

```
Di_pert_12 = (Fi_isom * (C rho + delta)* Gi_isom)_12
           = (Fi_isom * C_rho * Gi_isom)_12 +
             (Fi_isom * delta * Gi_isom)_12
           = D_rho_base_12 + delta_new_12                  (4.4)
           = D_aux + delta_new_12
           = D_aux + delta_new_12 * (H_aux * D_aux)        (4.5)
           = (id + delta_new_12 * H_aux)* D_aux
```

The first step consists in rewriting the definition of `Di_pert` and using the distributive property. Let us note that the two addends of (4.4) correspond with the definition of the differential and the perturbation in the decomposed chain complex. Concretely, the block `D_rho_base_12` is `D_aux`. We can see the

definition given in SSReflect of `D_rho_base` in Subsection 4.2.3.1. Moreover, we also know that `H_aux * D_aux = id` (lemma `H_aux_D_aux`) then it can be introduced in (4.5). Finally, we reorder the equation by taking `D_aux` as a common factor.

With these relations, `Di_pert_12` has an inverse if both `D_aux` and (id `+ delta_new_12 * H_aux`) have inverse. As we have said, thanks to the lemmas `D_aux_H_aux` and `H_aux_D_aux`, the inverse of `D_aux` is `H_aux`. Then, we focus on proving that id `+ delta_new_12 * H_aux` has an inverse. To this aim, the following lemma is useful. If $M^m = 0$ then the inverse of $\mathrm{id} + M$ is $\sum_{0 \leq i < m} (-M)^i$. One of the lemmas used to prove this is the following one.

$$M^m = 0 \rightarrow \sum_{0 \leq i < m} (-M)^i * (\mathrm{id} + M) = \mathrm{id}$$

Its statement in SSReflect is as follows.

```
Lemma inverse_I_plus_M_big : (pot_matrix M m = 0)
  -> (\sum_(0<=i<m) (pot_matrix (- M) i)) *m (1%:M + M) = 1%:M.
```

Instead of proving this lemma directly, we will focus on a generalization of it.

```
Lemma inverse_I_minus_M_big (M : 'M[R]_n): (pot_matrix M m = 0)
  -> (\sum_(0<=i<m) (pot_matrix M i)) *m (1%:M - M) = 1%:M.
```

First, we apply induction on the dimension of the matrix `M`, `n`. The first case is trivial since all the matrices which are involved in the statement are empty matrices. Namely, the lemmas `thinmx0` and `flatmx0` of SSReflect are in charge of every matrix which has no rows or columns. These particular matrices are considered as empty matrices represented by `0`. To deal with the inductive case, we apply induction again on the exponent of the power, `m`. In the first case, we obtain as hypothesis that $M^0 = 0$ then $\mathrm{id} = 0$. The system automatically does not find the contradiction; but, if we use properties already proved in SSReflect about the determinant, we obtain the equality $1 = 0$ and finally, the contradiction is solved by the system. Now, we focus on the general case ($m = n1 + 1$). As hypothesis, we have $M^{n1+1} = 0$ and the proposition to be proved is

$$\sum_{0 \leq i < n1+1} M^i * (\mathrm{id} - M) = \mathrm{id}$$

Then, we introduce the steps which have been followed in this proof.

$$\begin{aligned}
\sum_{0\leq i<n1+1} M^i * (\mathrm{id} - M) &= \sum_{0\leq i<n1+1} M^i - (M^i * M) \\
&= \sum_{0\leq i<n1+1} M^i - \sum_{0\leq i<n1+1} (M^i * M) \\
&= \sum_{0\leq i<n1+1} M^i - \sum_{0\leq i<n1+1} (M^{i+1}) \\
&= M^0 + \sum_{0\leq i<n1} \cancel{M^{i+1}} - M^{n1+1} - \sum_{0\leq i<n1} \cancel{(M^{i+1})} \\
&= M^0 - M^{n1+1} = M^0 = \mathrm{id}
\end{aligned}$$

These steps are easily achieved using the powerful *bigop* library of SSReflect [BGBP08].

Finally, if we prove that the block `H_aux * delta_new_12` to the power of `n` is 0; then, we can apply the lemma `inverse_I_plus_M_big`. The lemma to be proved is introduced below.

```
Lemma H_aux_delta_i : forall i:Z,
  pot_matrix ((lsubmx (ursubmx (delta_new i)))
  *m (H_aux rho i)) n = 0.
```

This lemma is proved thanks to the `nilpotency_hp_new` lemma. Then, the inverse of (id `+ delta_new_12 * H_aux`) is $\sum_{0\leq i<m} (-(\texttt{delta\_new\_12 * H\_aux}))^i$.

Consequently, it is proved a lemma `inverse_dp_12_inverse` which ensures that the inverse of `D_aux *` (id `+ delta_i_new_12 * H_aux`) is the following one.

```
Definition inverse_dp_12 (i:Z) :=
 (H_aux rho i) *m (\sum_(0<=j<n)(pot_matrix
   (-(lsubmx (ursubmx (delta_new i)) *m (H_aux rho i)))) j).
```

Now, applying the Generalization of the Hexagonal Lemma (where the reduction `rhoHL` is defined) we build a reduction, called `quasi_bpl` (see Figure 4.2), from the decomposed and perturbed chain complex.

The Generalization of the Hexagonal Lemma proved in Subsection 4.2.3 builds a reduction of a chain complex. But, the chain complex is an 1-suspended up

```
Definition quasi_bpl :=
  (rhoHL (Di:= Di_pert) (boundary_Di := boundary_dp_new)
      inverse_dp_12_inverse).
```

Figure 4.2: quasi_bpl definition

chain complex of the original one. Moreover, the chain complex which involves `Di_pert` is also an 1-suspended up one. To sum up, we have built a reduction from the following perturbed chain complex:

$$\cdots \longleftarrow A \oplus B \oplus C'_{(i+1)} \xleftarrow[Di\_pert\ (i-1)]{} A \oplus B \oplus C'_{(i+2)} \xleftarrow[Di\_pert\ i]{} A \oplus B \oplus C'_{(i+3)} \longleftarrow \cdots$$

Besides, the Decomposition Lemma allows us to build an isomorphism between C_* and $A_* \oplus B_* \oplus C'_*$. Using a composition of both results, we obtain a reduction from the 2-suspended up perturbed chain complex.

```
Definition Di_BPL (i:Z):= (diff (C rho) (i+1+1) + delta (i+1+1)).
```

Let us introduce graphically this situation.

$$\begin{array}{ccccccc}
\longleftarrow & C_{(i+1)} & \xleftarrow[Di_BPL(i-1)]{} & C_{(i+2)} & \xleftarrow[Di_BPL\ i]{} & C_{(i+3)} & \longleftarrow \\
 & {\scriptstyle Fi_isom}\uparrow\downarrow{\scriptstyle Gi_isom} & & {\scriptstyle Fi_isom}\uparrow\downarrow{\scriptstyle Gi_isom} & & {\scriptstyle Fi_isom}\uparrow\downarrow{\scriptstyle Gi_isom} & \\
\longleftarrow & A \oplus B \oplus C'_{(i+1)} & \underset{Di_pert(i-1)}{\overset{H(quasi_BPL)}{\rightleftarrows}} & A \oplus B \oplus C'_{(i+2)} & \underset{Di_pert\ i}{\overset{H(quasi_BPL)}{\rightleftarrows}} & A \oplus B \oplus C'_{(i+3)} & \longleftarrow \\
 & {\scriptstyle G(quasi_BPL)}\uparrow\downarrow{\scriptstyle F(quasi_BPL)} & & {\scriptstyle G(quasi_BPL)}\uparrow\downarrow{\scriptstyle F(quasi_BPL)} & & {\scriptstyle G(quasi_BPL)}\uparrow\downarrow{\scriptstyle F(quasi_BPL)} & \\
\longleftarrow & C'_{(i+1)} & \xleftarrow[Di'_BPL(i-1)]{} & C'_{(i+2)} & \xleftarrow[Di'_BPL\ i]{} & C'_{(i+3)} & \longleftarrow
\end{array}$$

Finally, we focus on obtaining a reduction from the initial chain complex C_* perturbed by δ. Then, we have to define the 2-suspended down reduction from the reduction `quasi_bpl`. In this way, we obtain the expected reduction of applying the BPL to the initial reduction ρ (defined at the beginning of this same subsection). Since the new definition is analogous, we introduce as an example the definition of `Di_BPL_down`.

```
Definition Di_BPL_down(i:Z):=
  castmx(cast3 i,cast4 i)(Di_BPL(i-1-1)).
```

Afterwards, we build the morphisms and the homotopy operator and the reduction `rho_BPL`. In this moment the cast are required, but we have avoided using casts until the last step of the proof.

4.3 Using the BPL to reduce a chain complex associated with a digital image

Two different proofs of the Vector-Field Reduction Theorem (Theorem 1.43) appear in [RS10]. The most direct one is based on the BPL. In this section, we are going to formalize this proof of the theorem when we work with a 3-truncated chain complex. This is the particular case of the chain complex associated with a 2D image where the admissible discrete vector filed is computed using the RS algorithm (explained in Subsection 2.1). The Chapter 3 includes the other proof of this theorem introduced in [RS10], which uses the Hexagonal Lemma. Now, we are going to follow the same structure of that proof since we want to reuse most of the work presented in that section. Nevertheless, this new proof includes some minor extra steps to apply the BPL; for instance, to convert a 3-truncated chain complex into a chain complex or to transpose the matrices.

The structure of the proof is the following one. Let us consider a chain complex which comes from a 2D image and an admissible discrete vector field associated with it. Then, the proof is divided into the same parts than the one included in Chapter 3. First, the admissible discrete vector field is used to reorder the differentials of the chain complex. Then, an isomorphism is defined between the chain complex and the reordered one. Second, a reduction of the reordered chain complex is built. Finally, the desired reduction is obtained thanks to the composition of the two previous reductions. In this process, the proofs of the first and third step will be reused from the ones detailed in Subsection 3.4.2 and Subsection 3.4.4. Then, we only focus on obtaining a reduction from the ordered chain complex using the BPL.

The second part of the proof begins with an ordered chain complex `chaincomplexd1'd2'` where the differentials `d'1` and `d'2` have been sorted according to an admissible discrete vector field. This vector field is computed from `d1` which is one of the differentials of the initial chain complex `chaincomplexd1d2` (defined in Subsection 3.4.2). It is also proved that the sorting in `d'1` implies that the top-left block is a lower triangular matrix. In our case, `d'1` is a matrix `'M_(m1+m2,m1+n2)` where `m1=sdvf`, `m2=n-sdvf` and `n2=m-sdvf`, then `d'2` is a matrix `'M_(m1+m2,l)`. Let us recall that `n` is the number of rows of `d'1`, `m` is the number of columns of `d'1`, `l` is the number of columns of `d'2`, and finally, `sdvf`

is the size of the admissible discrete vector field obtained from `d1`.

The required reduction is obtained directly from a chain complex which verifies the hypotheses of the BPL. But we need previously to build a `FGChain_Complex`, denoted by $D_* = (D_n, d_{D_n})$, from our `chaincomplexd1'd2'`. It consists of obtaining a finitely generated chain complex from a 3-truncated chain complex in the way represented in the following diagram.

$$\cdots \xleftarrow{} D_{-1} \xleftarrow[0]{} D_0 \xleftarrow[d'^T_1]{} D_1 \xleftarrow[d'^T_2]{} D_2 \xleftarrow[0]{} D_3 \xleftarrow[0]{} \cdots$$

Let us note that the number of generators of D_0 is `m1+m2`, of D_1 is `m1+n2`, and of D_2 is `l`.

This process requires to translate sequences to SSReflect matrices, to transpose the matrices, and to complete the 3-truncated structure to define a chain complex. In this way, the two non-null components are:

```
Definition d'1_trmx :=
 trmx (mx_of_seqmx (sdvf + (m - sdvf))(sdvf + (n -sdvf)) d'1)).

Definition d'2_trmx :=
 trmx (mx_of_seqmx (sdvf + (n - (sdvf))) 1 d'2).
```

The condition $d'_1 d'_2 = 0$ which comes from the chain complex `chaincomplexd1'd2'` is converted into $d'^T_2 d'^T_1 = 0$ (lemma `prodd1d2'_trmx`). Moreover, the top-left block of d'^T_1 is an upper triangular matrix because the top-left block of d'_1 is a lower triangular matrix. This lemma, which we will name `daa_triangular_upper`, is proved applying properties of the transpose of a matrix and the following lemma which relates a lower triangular matrix to an upper triangular one.

```
Lemma lower_triangular_trmx n (A:'M[F]_(n)) :
  lower_triangular A -> upper_triangular A^T.
```

Then, we define a reduction of the chain complex $D_* = (D_n, d_{D_n})$ where the top-left block of d_{D_n} in degree 1 is an upper triangular matrix with 1's on the diagonal.

Let us start with a `FGChain_Complex`, $C_* = (C_n, d_{C_n})$, a reduction $\rho = (f, g, h) : (C_n, d_{C_n}) \Rightarrow (E_n, d_{E_n})$, and a perturbation δ which verifies the

nilpotency condition ($\exists m \in \mathbb{N}$ such that $(\delta h)^m = 0$) such that $d_{C_n} + \delta = d_{D_n}$. Then, the BPL directly produces the required reduction of the chain complex D_*.

Briefly, let C_* be a `FGChain_Complex` composed only of the vectors of the vector field (i.e., the only non-null differential consists of the identity in the top-left block and null in the rest of the blocks) and $\delta = d_{D_n} - d_{C_n}$ which is clearly a perturbation whose top-left block is strictly upper diagonal. This can be seen easily according to these definitions.

$$d_{C_n} = \left(\begin{array}{ccc|c} 1 & & 0 & 0 \\ & \ddots & & \\ 0 & & 1 & \\ \hline 0 & \cdots & 0 & 0 \end{array}\right) \qquad d_{D_n} = \left(\begin{array}{c|c} A & 0 \\ \hline 0 & 0 \end{array}\right)$$

where

$$A = \begin{pmatrix} 1 & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ 0 & 1 & a_{23} & a_{24} & \cdots & a_{2n} \\ 0 & 0 & 1 & \ddots & \ddots & a_{3n} \\ 0 & 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 1 & a_{mn} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Then, we obtain the nilpotency hypothesis taking a trivial reduction where h is the transpose of d_{C_n} (which also has an identity in the top-left block and null in the rest of the blocks). In order to have an idea of the proof about the nilpotency condition, the following equations are going to be used.

$$(d_{D_n} - d_{C_n}) \cdot h = \begin{pmatrix} 0 & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ 0 & 0 & a_{23} & a_{24} & \cdots & a_{2n} \\ 0 & 0 & 0 & \ddots & \ddots & a_{3n} \\ 0 & 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 & a_{mn} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

So, δh has a top-left block strictly upper diagonal and the lower and top-right

blocks are null. Then, the nilpotency condition is easily proved. We are going to detail this proof in the rest of the section.

4.3.1 The initial reduction

Let us start with a 3-truncated reduction where the top 3-truncated chain complex has the same chain groups than `chaincomplexd1'd2'`; i.e., D_0, D_1 and D_2 with $m1+m2$, $m1+n2$ and l generators, respectively. The differential between D_0 and D_1 is defined by $\widehat{d}_1$ whose top-left block is an identity matrix of dimension $m1$, and null the rest of the blocks. The differential between D_1 and D_2 is null. From a notational point of view, the differential of this 3-truncated chain complex is denoted by $< \widehat{d}_1, 0 >$ where

$$\widehat{d}_1 = \left(\begin{array}{ccc|c} 1 & & & 0 \\ & \ddots & & \\ & & 1 & \\ \hline 0 & \cdots & 0 & 0 \end{array}\right)$$

It is worth mentioning here that $m1$ is the number of vectors in the admissible discrete vector field obtained from d'_1, and $m2$ the number of critical cells of d'_1.

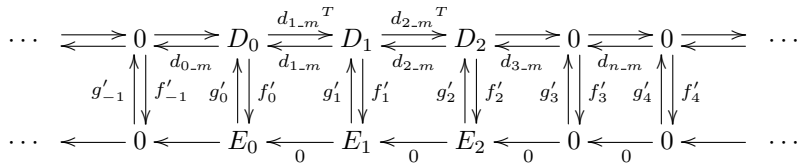
The reduced 3-truncated chain complex consists of non-null chain groups E_0, E_1 and E_2 with, respectively, $m2$, $n2$ and l generators and null differentials. The rest of the morphisms and the homotopy operator of the reduction are included in the following diagram.

$$\begin{array}{ccccc} D_0 & \underset{\widehat{d}_1}{\overset{\widehat{d}_1^{\,T}}{\rightleftarrows}} & D_1 & \underset{0}{\overset{0^T}{\rightleftarrows}} & D_2 \\ {\scriptstyle g'_0}\uparrow\downarrow{\scriptstyle f'_0} & & {\scriptstyle g'_1}\uparrow\downarrow{\scriptstyle f'_1} & & {\scriptstyle g'_2}\uparrow\downarrow{\scriptstyle f'_2} \\ E_0 & \underset{0}{\longleftarrow} & E_1 & \underset{0}{\longleftarrow} & E_2 \end{array}$$

where $f'_0 = \begin{pmatrix} 0 \\ \mathrm{id} \end{pmatrix}$, $f'_1 = \begin{pmatrix} 0 \\ \mathrm{id} \end{pmatrix}$, $f'_2 = \mathrm{id}$, $g'_0 = \begin{pmatrix} 0 & \mathrm{id} \end{pmatrix}$, $g'_1 = \begin{pmatrix} 0 & \mathrm{id} \end{pmatrix}$, $g'_2 = \mathrm{id}$ and $h_0 = \widehat{d}_1^{\,T}$. It is easy to prove that these components define a reduction.

4.3.2 From a 3-truncated reduction to a reduction

We want to apply the BPL to the reduction defined in the previous subsection. But, let us recall that the BPL is proved over chain complexes not over 3-truncated chain complexes. Then, we are going to define a general reduction from the 3-truncated reduction. The process consists in adding null components.



(4.6)

According to the previous subsection, in our particular case, d_{1_m} corresponds with $\widehat{d_1}$ and d_{2_m} with 0.

The top chain complex of the reduction (4.6) consists of two unique non-null maps, the rest of the maps are null. However, there are two particular null maps: the one in dimension -1, with rows but no columns, and the one in dimension 2, with columns but no rows. In our development, we have built a general definition to build a chain complex from a 3-truncated chain complex. This definition is explained as follows. First, it is necessary to define a function `m_m`, of type: `Z -> nat`, which provides the number of generators for each chain group. Let us note that the number of generators is 0 in all the degrees but in degrees 0, 1 and 2. The corresponding definition in SSReflect is as follows.

```
Definition m_m: Z -> nat:= fun i:Z => match i with
  |0%Z => c0
  |1%Z => c1
  |2%Z => c2
  |_ => 0%nat
end.
```

As the types of d_{1_m} and d_{2_m} are known thanks to $\widehat{d_1}$ and 0, respectively, we provide the type of the rest of maps of differential. These maps are represented as empty matrices. We have to highlight the differentials d_{0_m} and d_{3_m} because they are matrices with rows and no columns or with columns and no rows. The rest of maps will be matrices with no rows and no columns.

```
Variable (d3_m : 'M[K]_(0,c2)).
```

```
Variable (d0_m : 'M[K]_(c0,0)).
Variable (dn_m : 'M[K]_(0,0)).
```

Afterwards, we define the function `diff_m` which involves the whole differential.

```
Definition diff_m:= fun i:Z => match i as z
  return ((fun z0 : Z => 'M_(m_m (z0 + 1), m_m z0)) z) with
   |(-1)%Z => d0_m
   |0%Z => d1_m
   |1%Z => d2_m
   |2%Z => d3_m
   |3%Z => dn_m
   |_ => dn_m
end.
```

Finally, we prove the boundary condition of the differentials which allows us to build the finitely generated chain complex `Dcc`.

```
Lemma boundary_diff_m:
  forall i:Z, (diff_m (i+1)) *m (diff_m i) = 0.
```

This lemma is proved applying induction and proving the corresponding lemmas of the boundary condition of `diff_m` for each degree, as for instance:

```
Lemma d2d1: d2_m *m d1_m = 0.
Lemma d1d0: d1_m *m d0_m = 0.
Lemma d0dn: d0_m *m dn_m = 0.
```

Due to the fact that most of the matrices are empty matrices, this process is simplified using the lemmas `thinmx0` and `flatmx0` which transform a matrix without rows or columns into a null matrix.

The morphisms `F_m_m` and `G_m_m` and the homotopy operator `H_m_m` and the reduced chain complex `Dcc'` generated by the chain groups E_n are defined in a similar way. Finally, we build the reduction with the previous definitions.

```
Definition reductionFG_gen :=
  Build_FGReduction (C:=Dcc)(D:= D'cc)(F:=F_m_m)(G:=G_m_m)
                    (H:=H_m_m) GF_m FGHDDH_m HF_m GH_m HH_m.
```

The proof of the properties required by the `FGReduction` definition is quite direct.

4.3.3 Applying the BPL

Now, we can apply the BPL over the reduction `reductionFG_gen`, which have just been defined, and which is depicted in (4.6). Previously, we need to prove the following properties:

P1. A perturbation δ_{Dcc} for the differential of the chain complex `Dcc`.

P2. $\delta_{Dcc}h$ is locally nilpotent.

Let us start proving Property P1. First, we define a perturbation for `Dcc`. We define its components in degree 1 and 2 as $\delta_1 = {d'_1}^T - \widehat{d}_1$ and $\delta_2 = {d'_2}^T$. The rest of the components are null. In this way, the perturbed chain complex is the one we want to reduce (thanks to the BPL); i.e., it is composed by the non-null differentials given by $(\widehat{d}_1 + ({d'_1}^T - \widehat{d}_1)) = {d'_1}^T$ and $(0 + {d'_2}^T) = {d'_2}^T$. The map $\widehat{d}_1$ is defined in SSReflect using the function `hat_d1`, and the components of the perturbation, called `delta_m`, are the following ones.

```
Definition delta1 := d'1_trmx - hat_d1.
Definition delta2 := d'2_trmx.
Definition delta3 : 'M[K]_(0,1):=0.
Definition delta0 : 'M[K]_(m1+n2,0):=0.
Definition deltan : 'M[K]_(0,0):=0.
```

Now, we can prove that `delta_m` is a perturbation of the chain complex `Dcc` (defined by `(C reductionFG_gen)`).

```
Lemma boundary_dp_m: forall i : Z,
  (diff (C reductionFG_gen) (i + 1) + delta_m (i + 1)) *m
  (diff (C reductionFG_gen) i + delta_m i) = 0.
```

In the proof of this lemma, most of the cases are easy since most of the differentials and perturbations are null. The unique interesting case involves the differential in degrees 1 and 2 which is proved directly from the boundary condition of `Dcc` in these degrees. Let us recall that the differential in degree 1 (`d1_m`) corresponds with `hat_d1`. The statement in SSReflect is as follows.

```
Lemma boundary_d1d2_pert :
  (d2_m K (m1 + m2) l + delta2) *m (hat_d1 + delta1) = 0.
```

In mathematic notation this is equivalent to:

$$(\widehat{d}_1 + ({d'_1}^T - \widehat{d}_1))(0 + {d'_2}^T) = {d'_1}^T {d'_2}^T = 0$$

This is proved thanks to the boundary condition of `chaincomplexd1'd2'` ($d'_2 d'_1 = 0$).

Property P2 is stated in SSReflect in the following way. In our case, the number m which verifies the nilpotency property is $m1 + 2$ where m1 is the size of the vector field associated with d_1.

```
Lemma nilpotency_hp_m :
  forall i:Z, (pot_matrix
    (delta_m i *m (Ho (H reductionFG_gen) i)) (m1.+2) = 0).
```

In the proof of this lemma most of the cases are proved easily. We only focus on the part associated with the homotopy operator in degree 0: $(\delta_1 h_0)^{m1+2} = 0$. Let us recall that $\delta_1 = {d'_1}^T - \widehat{d}_1$ and $h_0 = \widehat{d}_1^T$ and expanding the definitions, we obtain

$$\begin{aligned}
(\delta_1 h_0)^{m1+2} &= \left(\left(\begin{pmatrix} d11 & d12 \\ d21 & d22 \end{pmatrix} - \begin{pmatrix} \mathrm{id} & 0 \\ 0 & 0 \end{pmatrix}\right)\begin{pmatrix} \mathrm{id}^T & 0 \\ 0 & 0 \end{pmatrix}\right)^{m1+2} \\
&= \left(\begin{pmatrix} d11 - \mathrm{id} & d12 \\ d21 & d22 \end{pmatrix}\begin{pmatrix} \mathrm{id} & 0 \\ 0 & 0 \end{pmatrix}\right)^{m1+2} \qquad (4.7) \\
&= \begin{pmatrix} d11 - \mathrm{id} & 0 \\ d21 & 0 \end{pmatrix}^{m1+2} \qquad (4.8)
\end{aligned}$$

Then, we prove different lemmas for every block of the power of a block matrix which will be applied over (4.8).

- `ulsubmx` $\begin{pmatrix} a & 0 \\ b & 0 \end{pmatrix}^k = a^k$
- `ursubmx` $\begin{pmatrix} a & 0 \\ b & 0 \end{pmatrix}^k = 0$

- $0 < k$, $\texttt{dlsubmx}\begin{pmatrix} a & 0 \\ b & 0 \end{pmatrix}^k = b * a^{k-1}$
- $\texttt{drsubmx}\begin{pmatrix} a & 0 \\ b & 0 \end{pmatrix}^k = 0$

Using the previous lemmas we obtain:

$$(\delta_1 h_0)^{m1+2} = \begin{pmatrix} (d11 - \mathrm{id})^{m1+2} & 0 \\ d21(d11 - \mathrm{id})^{m1+1} & 0 \end{pmatrix} = \begin{pmatrix} (d11 - \mathrm{id})(d11 - \mathrm{id})^{m1+1} & 0 \\ d21(d11 - \mathrm{id})^{m1+1} & 0 \end{pmatrix}$$

If we prove that $(d11 - \mathrm{id})^{m1+1} = 0$, then `nilpotency_hp_m` will be proved. Let us recall that $d11$ is an upper triangular matrix of type `'M_m1` (where `m1` is the length of the admissible discrete vector field associated with d_1). Then, we define the notion `upper_triangular_up_to_k` which receives an integer k and a matrix M and returns true if the matrix is an upper triangular matrix with $0's$ below the diagonal matrix and the first k diagonals above it. Consequently, if $k = 0$ the matrix M is an upper triangular matrix with $0's$ on the main diagonal and below it. This definition is represented in SSReflect as follows.

```
Definition upper_triangular_up_to_k n k (A:'M[F]_(n)) :=
  forall (i j : 'I_n), j <= i + k -> A i j = 0.
```

As $d11$ is an upper triangular matrix; then, $d11 - \mathrm{id}$ is an upper triangular matrix with $0's$ on the main diagonal and below it. This property is shown in the following lemma.

```
Lemma upper_triangular_minus_1 n (A:'M[F]_n):
  upper_triangular A -> upper_triangular_up_to_k 0 (A - 1%:M).
```

Then, in order to prove $(d11 - \mathrm{id})^{m1+1} = 0$ we focus on the following generalization of this lemma.

```
Lemma upper_triangular_pot_matrix_n n (A:'M[F]_n) :
  upper_triangular_up_to_k 0 A -> (pot_matrix A n.+1) = 0.
```

To prove this result, we introduce several auxiliary lemmas. The first one says that if A is an upper triangular matrix with $0's$ on and below the main diagonal, and B is an upper triangular matrix with $0's$ below the main diagonal and the

first k diagonals above it then, the product of A and B is an upper triangular matrix with $0's$ below the main diagonal and the first $(k+1)$ diagonals.

```
Lemma upper_triangular_mulmx_AB n k (A B:'M[F]_n) :
  upper_triangular_up_to_k 0 A ->
  upper_triangular_up_to_k k B ->
  upper_triangular_up_to_k k.+1 (A *m B).
```

The previous lemma is instrumental to prove the following one.

```
Lemma upper_triangular_pot_matrix n k (A:'M[F]_n) :
  upper_triangular_up_to_k 0 A ->
  upper_triangular_up_to_k k (pot_matrix A k.+1).
```

It is proved by applying induction on the parameter `k`. Since the first case is trivial, we pay attention to the inductive one. We have as hypotheses that `upper_triangular_up_to_k 0 A` and `upper_triangular_up_to_k 0 A -> upper_triangular_up_to_k k (pot_matrix A k.+1)`. We want to prove that `upper_triangular_up_to_k (k.+1)(pot_matrix A k.+2)`. Let us observe that `(pot_matrix A k.+2)= A *m (pot_matrix A k.+1)`. Then, the matrices involved in this product satisfies the conditions of the lemma `upper_triangular_mulmx_AB`. In this way, we obtain the expected result.

Finally, we also need the proof of a lemma which states that given an upper triangular matrix with $0's$ below the main diagonal and the first n diagonals about it (with n the size of the matrix) this implies that the matrix is a null matrix.

```
Lemma upper_triangular_up_to_dimension n (A:'M[F]_n):
  upper_triangular_up_to_k n A -> A = 0.
```

This lemma is proved applying induction on the size of the matrix. The interesting case is the inductive one. As inductive hypothesis we have `upper_triangular_up_to_k n A -> A = 0`, and we want to prove that if `A` is a matrix `'M[F]_n.+1` satisfying the condition `upper_triangular_up_to_k n+1 A` then the matrix is null. The type of the matrix can be written in this way: `'M[F]_1+n` where the bottom-right block is a matrix `'M[F]_n`. In this way, proving `A=0` is equivalent to prove that every block of `A` is null. The inductive hypothesis is used to prove this property over the bottom-right block.

After having defined and proved the conditions to apply the BPL, we can define a new reduction using `rho_BPL` defined at the end of Subsection 4.2.5.

```
Definition red_BPL := (rho_BPL boundary_dp_m nilpotency_hp_m).
```

In this way, we have built the reduction `red_BPL` using the BPL. We show this reduction in the following diagram where the number of generators of D_0, D_1, and D_2 are $m1+m2$, $m1+n2$, and l, respectively. The number of generators of E_0, E_1, and E_2 are $m2$, $n2$, and l, respectively. Let us recall that $m1$ is the number of vectors of the admissible discrete vector field built from the initial chain complex. Then, the reduction depends on the size of the vector field.

$$\begin{array}{ccccccccccc}
\cdots & \rightleftarrows & 0 & \underset{0}{\rightleftarrows}^{h'_0} & D_0 & \underset{d'_1{}^T}{\rightleftarrows}^{h'_1} & D_1 & \underset{d'_2{}^T}{\rightleftarrows}^{h'_2} & D_2 & \rightleftarrows & \cdots \\
 & & \Big\uparrow g'_{-1} & \Big\downarrow f'_{-1} & \Big\uparrow g'_0 & \Big\downarrow f'_0 & \Big\uparrow g'_1 & \Big\downarrow f'_1 & \Big\uparrow g'_2 & \Big\downarrow f'_2 & \\
\cdots & \leftarrow & 0 & \underset{0}{\leftarrow} & E_0 & \underset{\bar{d}'^T_1}{\leftarrow} & E_1 & \underset{\bar{d}'^T_2}{\leftarrow} & E_2 & \leftarrow & \cdots
\end{array} \tag{4.9}$$

4.3.4 From a reduction to a 3-truncated reduction

Up to now, we have obtained a reduction defined as `red_BPL` where the non-null differentials of the top 3-truncated chain complex are ${d'_1}^T$ and ${d'_2}^T$ as we can see in (4.9). But we want to obtain a 3-truncated reduction where the maps of the differentials of the top chain complex are d'_1 and d'_2, denoted as $< d'_1, d'_2 >$.

$$\begin{array}{ccccc}
D_0 & \underset{d'_1}{\rightleftarrows}^{{h'_1}^T} & D_1 & \underset{d'_2}{\rightleftarrows}^{{h'_2}^T} & D_2 \\
\Big\uparrow {g'_0}^T & \Big\downarrow {f'_0}^T & \Big\uparrow {g'_1}^T & \Big\downarrow {f'_1}^T & \Big\uparrow {g'_2}^T \quad \Big\downarrow {f'_2}^T \\
E_0 & \underset{\bar{d}'_1}{\leftarrow} & E_1 & \underset{\bar{d}'_2}{\leftarrow} & E_2
\end{array}$$

In order to obtain this desired 3-truncated reduction we have focused on two points. First, we extract from `red_BPL` the necessary maps to be able to build a 3-truncated reduction. In other words, we only take the two non-null differentials with the corresponding consecutive modules. Second, it is necessary to apply the transpose operation to all the components of the reduction. In this way, we obtain the expected differentials $d'_1 = {d'_1}^{TT}$ and $d'_2 = {d'_2}^{TT}$. Let us show some details of this formalization.

The definition of the bottom 3-truncated chain complex involves the application of the transpose operation to the differentials of the reduced chain complex of `red_BPL`.

```
Definition diff1D:= seqmx_of_mx (trmx (diff (D red_BPL) 0)).
Definition diff2D:= seqmx_of_mx (trmx (diff (D red_BPL) 1)).
```

These matrices are defined as sequences transposing the corresponding differentials. We define a 3-truncated chain complex `ccD` checking that these sequences are matrices which verify the boundary condition. It is necessary to highlight that the obtained 3-truncated chain complex is composed by the differential $< {d'_1}^{TT}, {d'_2}^{TT} >$ instead of the 3-truncated chain complex `chaincomplexd1'd2'` defined by $< d'_1, d'_2 >$. We deal with this issue using casts. Let us see the way of checking that the differentials of the initial chain complex of the reduction `red_BPL` are exactly the same that the reordered matrices `d'1` and `d'2`. To this aim, it is necessary to prove the following lemmas, and other similar ones.

```
Lemma b0:
  m (C red_BPL) 0 = (definitions_typesCC2.m chaincomplexd1'd2').
Lemma b1:
  m (C red_BPL) 1 = (definitions_typesCC2.n chaincomplexd1'd2').

Lemma M1C:
  (mx_of_seqmx (definitions_typesCC2.m chaincomplexd1'd2')
               (definitions_typesCC2.n chaincomplexd1'd2') d'1)
  = (castmx (b0,b1) (trmx (diff (C red_BPL) 0))).
```

Finally, we define the 3-truncated reduction, which we call `red_BPL_CC2`, from the `chaincomplexd1'd2'` to `ccD`.

Let us recall that our objective was to obtain a reduction of a 3-truncated chain complex $< d_1, d_2 >$ (`chaincomplexd1d2`) obtained from a digital image. We have started with a 3-truncated reduction `reduct_eq` (defined in Subsection 3.4) from `chaincomplexd1d2` to the ordered chain complex defined with the differential $< d'_1, d'_2 >$ (`chaincomplexd1'd2'`). Then, we have built a 3-truncated reduction `red_BPL_CC2` from the ordered 3-truncated chain complex to a reduced 3-truncated chain complex. Finally, using the lemma `Reduction_red_red`, both reductions are composed and we obtain a 3-truncated reduction of the chain complex `chaincomplexd1d2`.

Chapter 5

Homological processing of digital images

In this chapter we present an application in Digital Algebraic Topology of our formal development about the reduction of chain complexes using admissible discrete vector fields. Digital Topology, and more specifically, the *computation of homology groups* from digital images is mature enough (see, for instance, [ZA02], one among many good references) to go one step further and investigate the possibility of a *certified computation* (i.e., formally verified by proving correctness using an *interactive* proof assistant).

In a rough manner, the process to be verified is depicted in Figure 5.1. Putting it into words, from the black pixels of a monochromatic digital image a simplicial complex is obtained (by means of a triangulation procedure). Subsequently, from the simplicial complex, its *boundary (or incidence) matrices* are constructed. Finally, the *homology* of the chain complex defined by those matrices can be computed. If we work with coefficients over a field and if only the *dimensions* of the homology groups (as vector spaces) are looked for (which determines the groups of homology), then having a program able to compute the rank of a matrix is sufficient to accomplish the whole task. However, incidence matrices associated with a digital image usually have a considerable size and, therefore, homology computation can take a lot of time. In order to overcome this drawback

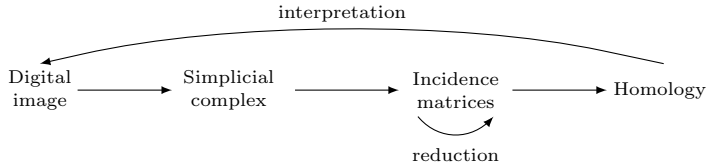


Figure 5.1: Computing homology from a digital image

we reduce those matrices applying the procedures explained in the two previous chapters. In this way, we can work with smaller matrices but with the same homology.

The rest of this chapter is organized as follows. First, we briefly explain in Section 5.1 how the Haskell code has been tested to check that the computation of the homology using the reduced matrices was exactly the same than using the initial matrices. Then, we focus on the formalization of the computation of the homology in SSReflect in general. We introduce an abstract formalization ranging from simplicial complexes to homology in Section 5.2. However, this does not allow us to effectively compute the homology of a simplicial complex since for instance, some definitions used for the definition of homology are based on abstract structures of the `bigop` library of SSReflect. Then, we define again the same structures but in an executable way in Section 5.3. The equivalence between both representations is proved in Section 5.4. Then, in Section 5.5, we focus on building the connection between digital images and simplicial complexes. Section 5.6 is devoted to present an example of computation of homology associated with a digital image in SSReflect (directly obtained, i.e., without using reductions). Finally, in Section 5.7 we also compute homology but using the reduced matrices which have been obtained thanks to an ordered and admissible vector field.

5.1 Semi-automated testing

As we have previously commented, our final aim consists in computing the homology groups of digital images. In the process, huge matrices, namely the incidence matrices, are used. The algorithm presented in Section 3.1 reduces these matrices in order to make the computations quicker. In this process we should check that the homology groups computed with the initial incidence matrices are

the same that the homology groups computed with the reduced matrices. If both results coincide, this is a proof that the ordered and admissible discrete vector field, computed to obtain the reduced matrices, is at least coherent from a homological point of view. However, it is not yet a proof that it is actually a correct ordered and admissible discrete vector field.

To this aim, a file will be generated with a battery of pairs of incidence matrices coming from random 2D monochromatic images and the homology groups associated with them. This file is carried out with a Computer Algebra System devoted to Algebraic Topology named Kenzo [DRSS98] and other auxiliary programs implemented in Common Lisp. First of all, randomly images composed by 0's and 1's are generated. Subsequently, the simplicial complex associated with every image is built. Then, the chain complex associated with this simplicial complex is constructed. Finally, we can compute the homology groups of the image thanks to the chain complex which give us properties of the given image. Next sections can be understood as an attempt to formalize all this process (but not exactly the same programs) in SSReflect.

With this file, we can take each pair of matrices, d_0 (the incidence matrix between degrees 1 and 0) and d_1 (between degrees 2 and 1) together with the homology groups H_0 and H_1 already computed. On the other hand, we can reduce both matrices using our algorithms (presented in Section 3.1), to get $\bar{d}'_0$ and $\bar{d}'_1$ and to obtain H'_0 and H'_1 in a similar way.

Finally, we create a new file with the values of H_0 and H_1 and the values of H'_0 and H'_1. If the computed values coincide with the expected values, the programs seem to work properly. This process has been applied to several batteries of examples and the obtained results have been the expected ones in all the cases.

5.2 Abstract formal development

In this section we focus on defining the concepts presented in Figure 5.1 using the SSReflect style. Some of the developments included in this section were written in [HPDR11].

5.2.1 Simplicial complexes

First of all, we define the notions related to simplicial complexes included in Subsection 1.1.2. The vertices are represented by a finite type `V`. Then, a simplex is defined as a finite set of vertices. Finally, the definition of a simplicial complex as a set of simplices closed under inclusion is straightforward:

```
Variable V : finType.
Definition simplex := {set V}.
Definition simplicial_complex (c : {set simplex}) :=
  forall x, x \in c -> forall y: simplex, y \subset x -> y \in c.
```

As we have seen in Definition 1.18 a natural way of determining a simplicial complex is by means of its facets. Therefore, given a list of simplices (which can be considered that include all the facets) we want to be able to construct the simplicial complex associated with it. To that aim, we need to define a function which obtains all the subsets from a simplex. The function `powerset` is in charge of this task. Then, we can define the function which creates a simplicial complex given a sequence of simplices, just performing the set union of the powerset of each one of them. To define such a function, we use the `bigop` library, namely the `bigcup` function which is used for iterated unions.

```
Variable (T : finType)(A : {set T}).
Definition powerset D := [set A : {set T} | A \subset D].
Definition create_sc (s : seq simplex) : {set simplex} :=
  \bigcup_(sp <- s) powerset sp.
```

Finally, to be sure that this function really creates a simplicial complex, we have to prove that the output of the function `create_sc` satisfies the property specified in the definition of a simplicial complex.

```
Lemma create_sc_correct:
  forall s, simplicial_complex (create_sc s).
```

On the other hand, it is easy to obtain the facets of any list of simplices.

```
Definition facet (s : seq simplex) (x : simplex) :=
  [set y : simplex | (x \subset y) && (y \in s)] == [set x].
Definition facets (s : seq simplex) := filter (facet s) s.
```

Then, we show the lemma which ensure that given a sequence `s` of simplices, the simplicial complex obtained from `s` is the same that the one computed from the facets of `s`.

```
Lemma facets_sc_same:
  forall s, create_sc s =i create_sc (facets s).
```

5.2.2 Abstract incidence matrices

Let us introduce the notion of incidence matrices associated with a simplicial complex. Since we can work with $\mathbb{Z}_2$ as a ground ring, we define a face operator as a set difference (we remove a vertex from a simplex) and the boundary as the image of a simplex by the face operator.

```
Definition face_op (S : simplex) (x : V) := S :\ x.
Definition boundary (S : simplex) := (face_op S) @: S.
```

We prove the correctness of our definition of boundary by showing that it is equivalent to a subset relation with constraints on cardinality:

```
Lemma boundaryP: forall (S : simplex) (B : simplex),
  reflect (B \subset S /\ #|S| = #|B|.+1) (B \in boundary S).
```

The statement `reflect P b` expresses an equivalence between a proposition `P` and a boolean expression `b`. This allows us to take advantage of the decidability of some propositions by going back and forth from their logical expressions (useful for reasoning) to their boolean counterparts (well suited for computations). Let us recall that this is the heart of SSReflect.

A key argument for our proof is the injectivity of the face operator above, which we establish as a lemma:

```
Lemma face_op_inj2: forall (S : simplex),
  {in S &, injective (face_op S)}.
```

The notation `{in S &, P}` performs localization of predicates: if `P` is of the form `forall x y, (Q x y)` then `{in S &, P}` means `forall x y, x \in S -> y \in S -> (Q x y)`. In our case, `injective f` stands for `forall x y, f x = f y -> x = y`.

Now, before giving the definition of the n-th incidence matrix of a simplicial complex, we can define the more generic notion of incidence matrix of two sequences of simplices.

Representing an incidence matrix requires an indexing of the simplices in `Left` (for the rows) and `Top` (for the columns). Then `Left` and `Top` are defined as sequences of simplices. Moreover, a coefficient a_{ij} of the incidence matrix will be 1 if the i-th simplex of `Left` is a face (subset) of the j-th simplex of `Top` and 0 otherwise.

Thus we can define the incidence matrix of two finite sets of simplices using the incidence function `boundary` as follows:

```
Definition incidenceMatrix :=
 \matrix_(i < m, j < n)
   if (nth set0 Left i) \in (boundary (nth set0 Top j))
   then 1 else 0:bool.
```

The type annotation `0:bool` indicates that the 0 and 1 appearing as coefficients of the matrix are two booleans elements. Let us note that the booleans are a representation of the field $\mathbb{Z}_2$. Moreover, `set0` is the empty set.

We now define the `n`-th incidence matrix of a simplicial complex `c` instantiating `Left` as the set of (`n-1`)-simplices of `c` and `Top` as the set of `n`-simplices. The function `n_1_simplices` returns a set which consists of the simplices of `c` whose cardinality is `n` (that is, the (`n-1`)-simplices of `c`) and analogously for the function `n_simplices`. As `Top` and `Left` are sequences of simplices we enumerate the `n_simplices` and `n_1_simplices` using `enum` in the call to the `incidenceMatrix` definition.

```
Variable c: {set simplex}.
Variable n:nat.
Definition n_1_simplices := [set x \in c | #|x| == n].
Definition n_simplices := [set x \in c | #|x| == n+1].

Definition incidence_mx_n :=
  incidenceMatrix (enum n_1_simplices)(enum n_simplices).
```

Then we have all the ingredients to state the boundary theorem (see Theorem 1):

```
Theorem incidence_matrices_sc_product:
  forall (V:finType) (n:nat) (sc: {set (simplex V)}),
  simplicial_complex sc ->
  (incidence_mx_n sc n) *m (incidence_mx_n sc (n.+1)) = 0.
```

The proof of this theorem uses two important SSReflect libraries which are `matrix` and `bigop`. In order to show what are the kinds of results which can be proved with these libraries, let us present a sketch of the proof and some comments about it.

Sketch of the proof. Let S_{n-1}, S_n, S_{n+1} be the set of $(n-1)$-simplices of $\mathcal{K}$, the set of n-simplices of $\mathcal{K}$ and the set of $(n+1)$-simplices of $\mathcal{K}$ respectively. Then, the incidence matrices are:

$$M_n(\mathcal{K}) = \begin{array}{c} \\ S_{n-1}[1] \\ \vdots \\ S_{n-1}[r2] \end{array} \begin{array}{c} \begin{array}{ccc} S_n[1] & \cdots & S_n[r1] \end{array} \\ \begin{pmatrix} a_{1,1} & \cdots & a_{1,r1} \\ \vdots & \ddots & \vdots \\ a_{r2,1} & \cdots & a_{r2,r1} \end{pmatrix} \end{array}, M_{n+1}(\mathcal{K}) = \begin{array}{c} \\ S_n[1] \\ \vdots \\ S_n[r1] \end{array} \begin{array}{c} \begin{array}{ccc} S_{n+1}[1] & \cdots & S_{n+1}[r3] \end{array} \\ \begin{pmatrix} b_{1,1} & \cdots & b_{1,r1} \\ \vdots & \ddots & \vdots \\ b_{r1,1} & \cdots & b_{r1,r3} \end{pmatrix} \end{array}$$

where $r1 = \sharp|S_n|$, $r2 = \sharp|S_{n-1}|$ and $r3 = \sharp|S_{n+1}|$. Thus,

$$M_n(\mathcal{K}) \times M_{n+1}(\mathcal{K}) = \begin{pmatrix} c_{1,1} & \cdots & c_{1,r3} \\ \vdots & \ddots & \vdots \\ c_{r2,1} & \cdots & c_{r2,r3} \end{pmatrix} \quad \text{where } c_{i,j} = \sum_{1 \leq k \leq r1} a_{i,k} \times b_{k,j}$$

To prove that $M_n \times M_{n+1}$ is equal to the null matrix, it is enough to prove that $\forall i, j$ such that $1 \leq i \leq \sharp|S_{n-1}|$ and $1 \leq j \leq \sharp|S_{n+1}|$, then $c_{i,j} = 0$. Each of these coefficients is written:

$$c_{i,j} = \sum_{1 \leq k \leq r1} a_{i,k} \times b_{k,j}$$

Since k enumerates the indices of elements of S_n, we may write:

$$c_{i,j} = \sum_{X \in S_n} F(S_{n-1}[i], X) \times F(X, S_{n+1}[j]) \quad \text{with } F(Y, Z) = \begin{cases} 1 & \text{if } Y \in dZ \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

dZ is the analogous in our context of the differential operator defined by 1.24 and is equal to:

$$dZ = \{Z \setminus \{x\} \mid x \in Z\}$$

This summation can be split depending on whether $X \in \partial S_{n+1}[j]$ or $X \notin \partial S_{n+1}[j]$.

$$\begin{aligned} c_{i,j} &= \sum_{X \in S_n | X \in \partial S_{n+1}[j]} F(S_{n-1}[i], X) \times 1 \qquad (5.2) \\ &\quad + \sum_{X \in S_n | X \notin \partial S_{n+1}[j]} F(S_{n-1}[i], X) \times 0 \\ &= \sum_{X \in S_n | X \in \partial S_{n+1}[j]} F(S_{n-1}[i], X) \qquad (5.3) \end{aligned}$$

The last summation is expressed over the image of the face operator $x \mapsto S_{n+1}[j] \setminus \{x\}$ which, restricted to $S_{n+1}[j]$, is injective. Thus, we can reindex:

$$c_{i,j} = \sum_{x \in S_{n+1}[j]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) \qquad (5.4)$$

Subsequently, this summation can also be split depending on whether $x \in S_{n-1}[i]$ or $x \notin S_{n-1}[i]$.

$$\begin{aligned} c_{i,j} &= \sum_{x \in S_{n+1}[j] | x \in S_{n-1}[i]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) + \\ &\quad \sum_{x \in S_{n+1}[j] | x \notin S_{n-1}[i]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) \qquad (5.5) \end{aligned}$$

Let us note that if $x \in S_{n-1}[i]$ then $S_{n-1}[i] \not\subset S_{n+1}[j] \setminus \{x\}$, hence the first sum above is 0.

$$c_{i,j} = \sum_{x \in S_{n+1}[j] | x \notin S_{n-1}[i]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) \qquad (5.6)$$

Here, we can split our proof considering two cases: $S_{n-1}[i] \not\subset S_{n+1}[j]$ and $S_{n-1}[i] \subset S_{n+1}[j]$.

In the first case, we have: $\forall x \in S_{n-1}[i], F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) = 0$, hence the result holds.

In the second case, $S_{n-1}[i] \subset S_{n+1}[j]$ implies that if $x \notin S_{n-1}[i]$ then $S_{n-1}[i] \in \partial S_{n+1}[j] \setminus \{x\}$, so the terms are all 1.

$$\begin{aligned} c_{i,j} &= \sum_{x \in S_{n+1}[j] | x \notin S_{n-1}[i]} 1 \\ &= \sharp |S_{n+1}[j] \setminus S_{n-1}[i]| \\ &= n + 2 - n = 2 = 0 \bmod 2 \end{aligned} \tag{5.7}$$

■

As can be seen in the sketch of the proof, a large part of it is devoted to the work with summations, for which the `bigop` library has played a key role in its formalization.

For instance, the first summation splitting (equation (5.2)) is performed by:

```
rewrite (bigID (mem (boundary (enum_val j)))).
```

where j belongs to S_{n+1}.

The lemma `bigID` states that an iterated operation using a commutative monoidal operator can be split:

$$\sum_{i \in r | P_i} F_i = \sum_{i \in r | P_i \wedge a_i} F_i + \sum_{i \in r | P_i \wedge \sim a_i} F_i$$

It is also possible to split a summation (equation (5.5)) and at the same time rewrite the first resulting sum to 0:

```
rewrite (bigID (mem (enum_val i))) big1.
```

The lemma `big1` states that a monoidal operator iterated over elements that are all equal to the neutral element produced as result the neutral element:

$$\sum_{i \in r | P_i} 0 = 0$$

Therefore, after the last tactic, the system will require a proof that all the terms of the first resulting summation are zero. The lemma `big1` is applied to obtain equations 5.3 and 5.6 of the sketch of the proof.

Our proof relies on two main reindexations: from ordinals to n-simplices (5.1) and later on from simplices to vertices (5.4). To perform the first reindexation, the script has the following shape:

```
rewrite (reindex_onto (enum_rank_in Hx0) enum_val); last first.
 by move=> x _; exact:enum_valK_in.
```

where:

- `Hx0` is a proof that there exists at least one n-simplex,
- `enum_rank_in` enumerates the n-simplices since `Hx0` ensures there is at least one,
- `enum_val` enumerates the ordinals over which the sum is expressed,
- `reindex_onto` reindexes from ordinals to n-simplices, given a bijection between both sets. Indeed, the second line proves that `enum_val` $\circ$ `enum_rank_in` $=$ id.

The second reindexation is based on the injectivity of the face operator:

```
rewrite big_imset; last exact:face_op_inj2.
```

Rewriting with the lemma `big_imset` triggers a check that the summation is expressed over the image of a set by a function. In our case, the system automatically infers that this function is the face operator `face_op`, and will then ask for a proof of its injectivity.

The lemma `eq_big` and its variants `eq_bigl` and `eq_bigr` allow us to rewrite the predicate or the operand of an iterated operation. It is applied in particular to obtain equation 5.7.

```
rewrite (eq_bigr (fun _ => 1)).
```

The system will, of course, require a proof that the operand is equal to 1. Then it will rewrite the expression to a constant summation, allowing the use of the

lemma `big_const` to replace it with a product (cardinal of the iterated set by the constant value).

Simple arithmetic arguments on cardinals will then complete the proof.

Finally, we can instantiate Lemma `incidence_matrices_sc_product` for any simplicial complex. In particular, for the ones generated by the function `create_sc` as we can see in the following lemma.

```
Lemma incidence_matrices_sc_product_facets (V:finType) (n:nat)
  (s: seq (simplex V)) : (n >= 1)%N ->
  (incidence_mx_n (create_sc s) n) *m
  (incidence_mx_n (create_sc s) (n.+1)) = 0.
Proof.
by rewrite incidence_matrices_sc_product //;
  apply: create_sc_correct.
Qed.
```

5.2.3 Abstract formalization of homology

The SSReflect libraries include enough ingredients to undertake the task of defining and computing homology from matrices.

First of all, we define the notion of homology for vector spaces. Let K be a field, $V1$,$V2$,$V3$ vector spaces on K, and $f : V1 \to V2, g : V2 \to V3$ linear applications; then, the *homology of* f and g is the quotient between the *kernel* of g and the *image* of f (taking into account that the condition $g \circ f = 0$ is verified). This is translated into SSReflect in the following way.

```
Variable (K : fieldType) (V1 V2 V3 : vectType K)
         (f : linearApp V1 V2) (g : linearApp V2 V3).
Definition Homology := ((lker g) :\: (limg f)).
```

Let us note that the kernel of a linear function, as for instance `f`, in SSReflect is defined with the function `lker`, `lker f`, and its image with `limg`, `limg f`, and the quotient between vector spaces `A` and `B` is denoted as `A::B`.

With these abstract definitions, we can prove the following results.

- $g \circ f = 0 \rightarrow dim(\ker g \cap \text{im}\, f) = dim(\text{im}\, f)$
- $dim(\ker g / \text{im}\, f) = dim(\ker g) - dim(\ker g \cap \text{im}\, f)$
- $dim(\ker g) = dim\ V2 - dim(\text{im}\, g)$
- $g \circ f = 0 \rightarrow dim\ Homology = dim\ V2 - dim(\text{im}\, g) - dim(\text{im}\, f)$ (5.8)

Let us show the statement in SSReflect of the first property.

```
Lemma dim_ker_intersection_im:
  (g \o f = \0)%VS-> \dim((lker g) :&: (limg f)) = \dim(limg f).
```

We can see that the notation to represent the composition of two linear functions is `\o`. Moreover, the intersection between two subspaces is denoted by `:&:`. Let us highlight that this abstract definition makes the proof of the above properties easier since SSReflect libraries include a lot of lemmas about vector spaces.

In particular, we can easily prove that $g \circ f = 0 \rightarrow \text{im}\, f \subseteq \ker g$. Then, $g \circ f = 0 \rightarrow \ker g \cap \text{im}\, f = \text{im}\, f$ (the `capvkr` lemma provides this result) and, therefore, the previous lemma is proved.

In this way, we define homology for vector spaces and provide a formula to compute its dimension (see Formula 5.8). Nevertheless, we do not usually work with linear applications when trying to compute homology but with matrices which represent those linear applications. In particular, given two matrices with coefficients in a field K *mxf* and *mxg*, of sizes $v1 \times v2$ and $v2 \times v3$ respectively, and such that their product is the null matrix, the dimension of the corresponding homology vector space is given by the formula: $v2 - (rank\ mxg) - (rank\ mxf)$. This definition is introduced as follows.

```
Definition dim_homology (mxf:'M[K]_(v1,v2)) (mxg:'M[K]_(v2,v3))
  := v2 - \rank mxg - \rank mxf.
```

Now, the correctness of `dim_homology` can be shown by proving that given two matrices `mxf` and `mxg` whose product is the null matrix (`mxf *m mxg = 0`), then the result obtained using `dim_homology` is equal to the dimension of the homology group associated with the linear applications defined from `mxf` and `mxg`. Since

given a matrix `M`, `(LinearApp M)` builds the linear application associated with `M`, we can state such a result as follows.

```
Lemma dimHomologyrankE:
 mxf *m mxg = 0 -> \dim Homology (LinearApp mxf) (LinearApp mxg)
                  = dim_homology mxf mxg.
```

Finally, we can connect homology with simplicial complexes. Namely, in order to define the homology in degree n of a simplicial complex we instantiate the `Homology` definition using the linear applications associated with the incidence matrices in degree $n+1$ and n as f and g respectively.

```
Definition Homology_sc_n (sc : {set simplex}) (n : nat) :=
  Homology (LinearApp (incidence_mx_n c n.+1))
           (LinearApp (incidence_mx_n c n)).
```

It is worth noting that this development take up about 1300 lines. Namely, it involves 38 definitions and 77 auxiliary lemmas. However, the definitions presented up to now cannot be used to compute homology since some notions, like matrices or bigops, are locked in a way that do not allow us direct computations. This is because the use of SSReflect libraries may trigger heavy computations during deduction steps, that would not terminate within a reasonable amount of time. To overcome this pitfall and following the idea which we have used in Section 3.5 in relation to the notion of reduction, we proceed to introduce the same development but using structures which can be used to compute.

5.3 An effective formal development

Let us introduce the notions presented in previous section but in a way that they can be used to compute. Let us recall that a vertex was represented having a finite type `V`. Moreover, a simplex was defined as a set of `V` endowed with a partial order between its elements was associated. A consequence of putting an order on vertexes is that simplices can be represented in a unique way as ordered lists. Taking this into account, now we define a simplex as a sequence of `V` because we can compute with this data type. Then we have to provide a relation between the components of the simplex. This relation, called `leT`, is a binary boolean relation over `V`. Apart from that, this relation has to be transitive and irreflexive.

Moreover, if `faces` is a sequence of simplices (sequences of `V`), every element of each one of these simplices also have to be reordered according to the same relation. In addition, every simplex has not repeated vertices.

```
Variable V : finType.
Variable leT : rel V.
Variable simplex : seq V

Hypothesis tr_leT : transitive leT.
Hypothesis irr_leT : irreflexive leT.

Variable faces : seq (simplex).

Hypothesis Hfaces : all (sorted leT) faces.
Hypothesis Hfaces_uniq : all uniq faces.
```

Furthermore, we define the equivalent notions to `powerset` and `create_sc` on sequences which are respectively `powerset_seq` and `sc`.

```
Fixpoint powerset_seq (x : seq V) :=
  if x is h::t then
    let X := powerset_seq t in foldl (fun xs x => (h::x)::xs) X X
  else [::[::]].
Definition bigcup_seq s (f : seq V -> seq (seq V)) :=
  foldl (fun xs x => (f x) ++ xs) [::] s.
Definition sc := undup (bigcup_seq faces powerset_seq).
```

Some remarks about the above functions are required. The statement `foldl f a s` where `f: seq V -> seq (seq V)`, `a:seq V`, `s:seq V` behaves as `f (f ... (f a x_1) ... x_n-1) x_n` where `x_1`...`x_n` are the elements of `s`. Given a function `f` and a sequence, the `bigcup_seq` function produces the concatenation of all the subsequences and finally, the `undup` function removes the duplicate elements.

In this way, we have defined the executable counterpart of simplicial complexes. Let us focus now on the incidence matrices. To define the concrete face operator `ex_face_op` we use the function `rem`. This function takes as argument `x: V` and `s: seq V`, and returns the sequence obtained of removing `x` from `s`. In

our case, it removes a vertex of a simplex. Then the function `ex_boundary` is defined using `ex_face_op`.

```
Definition ex_boundary (S : seq V) :=
  foldl (fun xs x => ex_face_op S x :: xs) [::] S.
```

Let us recall that the definition of the incidence matrices involves an indexing of the simplices for the rows and for the columns. Consequently, we define these sets as sequences of sequences of `V` instead of sequences of sets of `V` to be able to compute with them. Namely, to define the simplices of degree `n` we use the command `filter f sc` which lets us select the elements of `sc` which satisfy the boolean function `f`. In this case, `f` selects the simplices whose size is `n`. Finally, we can define the executable version of the incidence matrix.

```
Definition ex_n_1_simplices := filter (fun s => size s == n) sc.
Definition ex_n_simplices := filter (fun s => size s == n.+1) sc.

Definition ex_incidence_mx :=
  let ex_boundaries := map ex_boundary ex_n_simplices in
  map (fun x => map (fun y => if x \in y then 1 else 0:bool)
      ex_boundaries) ex_n_1_simplices.
```

The heart of this definition consists in checking for every `x`, a `(n-1)`-simplex, and for every `y`, a `n`-simplex of a simplicial complex, if `x` belongs to `y`. Let us highlight that to build incidence matrices is not necessary the complete simplicial complex but only its facets.

Finally, it leaves the definition of the executable version to compute the homology dimension. We use the rank algorithm developed in [CDMS] to define `ex_homology` which takes as argument two executable matrices (represented by means of sequences of sequences) `mxf` and `mxg` whose dimensions are `v1`×`v2` and `v2`×`v3` respectively, and computes the homology dimension.

```
Definition ex_homology (v1 v2 v3:nat) (mxf mxg : matZ2):=
  v2 - (rank v2 v3 mxg) - (rank v1 v2 mxf).
```

This definition can be instantiated for the case of our executable incidence matrices. Up to now, we have two different definitions of both incidence matrices and homology dimension. For the first development (the abstract one), we have

proved a series of properties. However, these results have not been proved for the second representation. We can face this problem in two different ways. The former would be proving directly the same results with this new representation. In this case, the difficulty of the proofs will be higher than in the abstract case since the big power of SSReflect was used to work with the abstract representation. The latter is proving the equivalence between both definitions up to a change of representation. We have chosen this second option.

5.4 The bridges between both representations

This section is split into two parts. Firstly, we focus on proving the equivalence between the different representations of incidence matrices presented in Section 5.2 and Section 5.3. Secondly, we link the different definitions given to define the homology.

5.4.1 Incidence matrices bridge

We focus on creating the bridge between both definitions of incidence matrices. To this aim, the simplices will be represented as a sequence of sets of a finite type instead of as sequence of sequences of a finite type (used in the effective formal development). Then, we will define again a sequence of `n`-simplices (`n_simplices_seq`) and the (`n-1`)-simplices (`n_1_simplices_seq`), any set of faces (`faces_set`) and finally the simplicial complex (`sc_set`) from the definitions given in the concrete version. For instance, to define `faces_set` we use `faces`. Let us show two of these definitions.

```
Definition faces_set := [seq [set t \in x] | x <- faces].
Definition sc_set := create_sc faces_set.
```

Let us note that in the definition of `sc_set` we can use the abstract function named `create_sc` since `faces_set` has the proper type `seq {set V}`.

Two definitions in SSReflect for incidence matrices have been introduced: `incidenceMatrix`, the abstract one, and `ex_incidence_mx`, the effective one. The main difference is the data type of these matrices. In particular, if we define an abstract matrix and then we want to obtain the effective one, we will only use

the function `seqmx_of_mx` which transforms a matrix into a sequence of sequences as we explained in Section 3.3. To prove this equality between both definitions, we are going to follow the idea presented in [DMS12a] which we have already used in our developments (see Subsection 3.4.3.4). Then, we refine the function to transform the incidence matrix into an efficient one using SSReflect structures. In this case, we can define a new incidence matrix `im_set` using `incidenceMatrix` with the new definitions about the `n` or `(n-1)`-simplices.

```
Definition im_set :=
  incidenceMatrix n_1_simplices_seq n_simplices_seq.
```

Let us note that in this case the aim of proving that this definition behaves like the abstract version is extremely easy, due to the use of `incidenceMatrix` to define it.

The second step is proving that the concrete definition `ex_incidence_mx` is equal to the change of representation (from abstract matrices to sequence of sequences) applied over `im_set`. Then, we need to prove similar lemmas which link both developments. For instance, lemmas related to boundaries or to `n`-simplices of a chain complex. Let us show the statements of two examples.

```
Lemma ex_boundaryP s :
 uniq s ->
 boundary[set t \in s]=i [seq [set t \in x]| x <- ex_boundary s].
Lemma n_simplices_seqE :
  n_simplices_seq =i n_simplices sc_set n.
```

Both statements have similar structures and relate the abstract development to the efficient one. For instance, the second one states that the efficient function `n_simplices_seq` builds the same n-simplices than the abstract one `n_simplices`. Concretely, the statement `P =i Q` means that the `P` and `Q` sets are extensionally equivalent i.e., $\forall$`x`, `x` belongs to `P` if and only if `x` is also in `Q`.

Finally, let us show the statement which relates both definitions.

```
Lemma im_setE : ex_incidence_mx = seqmx_of_mx im_set.
```

5.4.2 Homology bridge

Finally, we prove the correctness of `ex_homology` by showing its equivalence to `\dim Homology` up to a change of representation (this domain transformation is given by `seqmx_of_mx`).

```
Lemma ex_homology_rankE:
  forall(mxf:'M[K]_(w1,w2))(mxg:'M[K]_(w2,w3)),
  ex_homology (seqmx_of_mx mxf) (seqmx_of_mx mxg)
  = \dim Homology mxf mxg.
```

This lemma will be proved following the schema of Figure 5.2.

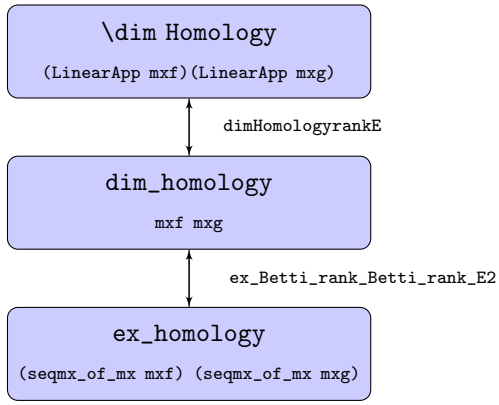


Figure 5.2: Bridges between the different homology concepts

Let us note that we use the definition `dim_homology` as a bridge between the other definitions. Moreover, `ex_homology_rankE` will be proved applying the lemmas which can be seen in Figure 5.2: `dimHomologyrankE` (see Subsection 5.2.3) and `ex_Betti_rank_Betti_rank_E2`. The last one is an equivalence between both definitions, in fact the unique difference is the type of the matrices.

Thus, we have an executable program to compute homology from a simplicial complex whose correctness has been verified in Coq; therefore, we can claim that its results will always be correct. Now let us present how we can use this development to compute homology from digital images.

5.5 From digital images to homology

It is worth noting that there are several methods to construct a simplicial complex from a digital image [ADFQ03]. We are going to explain one of these methods. Roughly speaking, the chosen method consists of obtaining a sequence of facets from a digital image. Then, as we have explained in the previous section, we can obtain the simplicial complex associated with the facets. So, we only need to explain how to get the facets from a digital image.

We are going to work with 2D monochromatic images, that are images with only black and white pixels. This kind of images can be encoded as sequences of sequences of booleans where the values `true` and `false` represent respectively black and white pixels.

Let $\mathcal{I}$ be an image encoded as a sequence of sequences of booleans. Let $V = (\mathbb{N}, \mathbb{N})$ be the vertex set, each vertex is a pair of natural numbers. Let $p = (a, b)$ be the position of a black pixel in $\mathcal{I}$. For each p we can obtain two 2-simplices which are two facets of the simplicial complex associated with $\mathcal{I}$ (which we name `set1ij` and `set2ij`). Namely, for each $p = (a, b)$ we obtain the following facets: $((a, b), (a + 1, b), (a + 1, b + 1))$ and $((a, b), (a, b + 1), (a + 1, b + 1))$. If we repeat the process for the positions which correspond with all the black pixels in $\mathcal{I}$, we obtain the facets of a simplicial complex associated with $\mathcal{I}$, let us called it $\mathcal{K}_{\mathcal{I}}$.

Example 5.1. Consider the image depicted in Figure 5.3. This image, $\mathcal{I}$, can be codified by means of the 2-dimensional array: `((true,false),(false,true))`.

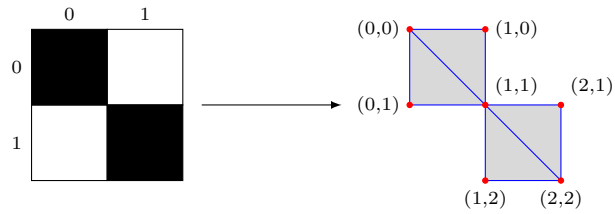


Figure 5.3: A digital image and its simplicial complex representation

Then, with the previously explained process we obtain the facets of $\mathcal{K}_{\mathcal{I}}$. The positions of the black pixels of the image $\mathcal{I}$ are $(0, 0)$ and $(1, 1)$, so the facets that

we obtain are:

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

This method to obtain the facets of a simplicial complex associated with a 2D image can be generalized to higher-dimensional images [OS03].

It is worth noting that even the bigger digital images have always a finite number of components, hence a finite number of vertices and then our vertex set V consists of a finite number of vertices. Therefore, simplicial complexes coming from digital images are always of finite type.

Let us note that every pixel of the image is located knowing its position. Then the function `setijList` receives two naturals which represent the position of a black pixel and creates their two corresponding facets `set1ij` and `set2ij`.

```
Definition setijList (i j:nat):= (set1ij i j)::(set2ij i j)::nil.
```

At this moment our aim is building all the facets of a digital image. The procedure to obtain the facets associated with an image consists of running the successive elements of the image gathering the facets associated with each black pixel. This task is carried out with the `createfacets` function. This function receives a sequence of sequences of booleans and returns a sequence of sequences of sequences of natural numbers.

Up to now we have only defined a way to build the facets from a digital image. Then we introduce some properties which the `createfacets` function has to satisfy to make sure that the results of this function are the expected ones. The three first properties are in charge of the completeness properties of `createfacets` and the last one of its correctness. Let i and j be natural numbers and M be a sequence of sequences of booleans:

- If `set1ij` $\in$ `createfacets M` then `set2ij` $\in$ `createfacets M`.
- If `set2ij` $\in$ `createfacets M` then `set1ij` $\in$ `createfacets M`.
- If $M[i,j] = true$ then `set1ij` $\in$ `createfacets M`.
- If `set1ij` $\in$ `createfacets M` then $M[i,j] = true$.

Let us recall that our functions about simplicial complexes receive a sequence of simplices of type `seq (seq V)` with `V` a finite type. However, the function which builds the facets of an image returns the type `seq (seq (seq nat))`. In particular, a vertex is defined as a sequence of natural numbers. In fact, this sequence always consists of two natural numbers where the first number is bounded by the number of rows, `m`, and the second one by the number of columns, `n`. Then a vertex can be encoded as a pair of elements of finite type. Namely, the first element of the pair will belong to the ordinal `'I_m` where `m` is the number of rows of the image, and the second one is in the ordinal `'I_n` with `n` the number of columns of the image. Using this idea, we can transform the output produced by the `createfacets` function into a suitable representation for our simplicial complexes programs. The `listSimplex_to_seqSimplex` function is in charge of this task.

Taking everything into account, we define the corresponding ordinals. Let us define the ordinals `Zm` and `Zn` with the constructor `ordinal_finType`. Then a vertex is defined as follows.

```
Definition Vertex := (prod_finType Zm Zn).
```

Then the type `Simplex` which represents any face is a sequence of vertices (`seq Vertex`). Finally, we have to define the conversion functions, for instance, the function which obtains an item of type `Vertex` from a sequence of natural numbers. In particular, the `listSimplex_to_seqSimplex` function allow us to define a sequence of `Simplex` from a sequence of sequences of sequences of natural numbers.

Therefore, we can link this construction about digital images with our programs to compute homology from simplicial complexes. To sum up, given an image, we compute the facets of the simplicial complex, then we construct the incidence matrices and finally, compute the homology. In order to clarify how we can use those programs, let us present an example in the following section.

5.6 Computing homology within Coq

First, we show the way of computing the homology from an image step by step starting with the computation of the facets of the image and finishing with its

homology dimension. Let us compute the homology of the image of Figure 5.4.

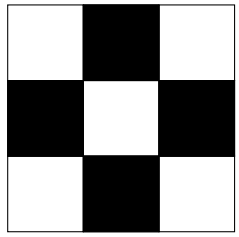


Figure 5.4: An image with four black pixels and a hole

To work with this image, we define it in the following way.

```
Definition image := ((false::true::false::nil)::
                     (true::false::true::nil)::
                     (false::true::false::nil)::nil).
```

The following step consists in defining the facets and later on, the simplicial complex associated with these facets.

```
Definition facets := (createfacets image).
Definition c := (listSimplex_to_seqSimplex (ltn0Sn (size image))
  (ltn0Sn (size (head nil image))) facets).
```

We can show the result of these definitions using the command `Eval vm_compute`. For instance, to compute the facets we use:

```
Eval vm_compute in facets.
= [:: [:: [:: 0; 1]; [:: 1; 1]; [:: 1; 2]];
      [:: [:: 0; 1]; [:: 0; 2]; [:: 1; 2]];
      [:: [:: 1; 0]; [:: 2; 0]; [:: 2; 1]];
      [:: [:: 1; 0]; [:: 1; 1]; [:: 2; 1]];
      [:: [:: 1; 2]; [:: 2; 2]; [:: 2; 3]];
      [:: [:: 1; 2]; [:: 1; 3]; [:: 2; 3]];
      [:: [:: 2; 1]; [:: 3; 1]; [:: 3; 2]];
      [:: [:: 2; 1]; [:: 2; 2]; [:: 3; 2]]]: seq (seq (seq nat))
```

Then we can define the incidence matrices which are related to this image, concretely to its 0-simplices (vertices), 1-simplices (edges) and 2-simplices (trian-

gles) of the image. These elements appear in the incidence matrix in degree 0, 1, and 2.

```
Definition im0 := ex_incidence_mx (sc c) 0.
Definition im1 := ex_incidence_mx (sc c) 1.
Definition im2 := ex_incidence_mx (sc c) 2.
```

Finally, using the function `ex_homology` we can define the homology in degrees 0 and 1. This function receives three natural numbers (m, n, and p) and the necessary incidence matrices (M with size $m \times n$ and $M1$ with size $n \times p$) to define the homology in the corresponding degree.

```
Definition H0:= ex_homology 0 (size (ex_n_simplices sc 0))
 (size (ex_n_simplices sc 1)) im0 im1.
Definition H1:= ex_homology (size (ex_n_simplices sc 0))
 (size(ex_n_simplices sc 1)) (size(ex_n_simplices sc 2)) im1 im2.
```

Evaluating these two last definitions, we obtain that the (dimension of the) homology in degree 0 is 1 and the one in degree 1 is also 1. This result has an interpretation on the initial image: the image has one connected component and one hole.

The way of computing the homology is not the most convenient since we have to define the previous objects to be able to define the homology and finally, to compute it. Then, we are going to define and prove some results to compute the homology of degree n from the image directly.

Let us recall that any set of faces was represented as a sequence of sequences `s`, of a finite type `V`. Moreover, we defined a relation between the elements of `V` which is transitive and irreflexive. In these conditions, we can wrapper the details or the parameters which are necessary to use the function `ex_homology` in the definition `exHomologySCFacets`.

```
Definition exHomologySCFacets := ex_homology
  (size (ex_n_1_simplices (sc s) n))
  (size (ex_n_simplices (sc s) n))
  (size (ex_n_simplices (sc s) n.+1))
  (ex_incidence_mx (sc s) n) (ex_incidence_mx (sc s) (n.+1)).
```

Let us note that the statement `(sc s)` returns the simplicial complex associated with `s`. Moreover, the following lemma verifies the correctness of this new definition since the output of `exHomologySCFacets` is equal to the computation of the dimension of the homology computed with the abstract definition `Homology`.

```
Lemma exHomologySCFacetsE :
  exHomologySCFacets =
  \dim Homology
  (LinearApp (mx_of_seqmx
   (vdim (W1 bool_fieldType (size(ex_n_1_simplices (sc ss) n))))
   (vdim (W2 bool_fieldType (size(ex_n_simplices (sc ss) n))))
   (ex_incidence_mx (sc ss) n)))
  (LinearApp (mx_of_seqmx
   (vdim (W2 bool_fieldType (size(ex_n_simplices (sc ss) n))))
   (vdim (W3 bool_fieldType (size(ex_n_simplices (sc ss) n.+1))))
   (ex_incidence_mx (sc ss) n.+1))).
```

We are going to apply this new definition to compute the homology in degree n given an image. We start working with a variable named `imag` which is a sequence of sequences of booleans which represents the image. Then, we define `m` and `n` as the dimensions of `imag` enforcing the conditions of that these values are higher than 0. Let us highlight that we have to define a boolean relation between the vertices since in the concrete version we only have a general relation between them. This relation is defined as follows. Let $v1 = (v1.1, v1.2)$ and $v2 = (v2.1, v2.2)$ be two vertices, $v1$ is lower than $v2$ if $v1.1 < v2.1$ or if $v1.1 = v2.1 \wedge v1.2 < v2.2$. Such a relation is transitive and irreflexive. Let us show this definition in SSReflect.

```
Definition ltord (i j : Vertex m n) :=
 ((fst i) < (fst j))
 || (((fst i) == (fst j)) && ((snd i) < (snd j))).
```

Moreover, we can define the homology in degree `n` given an image as follows.

```
Definition HomologyImage (n:nat) := exHomologySCFacets n
  (listSimplex_to_seqSimplex m0 n0 (createfacets image)).
```

Finally, we are going to compute the groups of homology in degree 0 and 1 of Figure 5.4 using the last definition. Let us recall that the definition `image`

defined as a sequence of sequences of booleans represents the image. Before computing the groups of homology, we have to prove that the dimensions of `image` are higher than 0, i.e., it has at least a row and a column. These results, which correspond respectively with `mimage` and `nimage` lemmas, are proved just by computing. Let us focus on the first lemma, we define `m` as the number of rows of a sequence of sequences of booleans, then (`m image`) computes the number of rows of the sequence `image`. The `mimage` lemma is proved simply using the instruction `by compute`.

Finally, we compute the homology in degree 0 and 1 in the following way.

```
Eval vm_compute in (HomologyImage mimage nimage 0).
Eval vm_compute in (HomologyImage mimage nimage 1).
```

We need as input parameters the two previous lemmas and the degree of the homology. Let us note, that Coq is able to infer the image from the arguments `mimage` and `nimage`.

Up to now, we have introduced a method to compute the dimension of the homology groups from an image. This task is carried out by means of the computation of the incidence matrices of the chain complex associated with an image. Let us recall that we could reduce this chain complex according the explanation in Chapter 3 based on admissible discrete vector fields. In the following section, we focus on computing homology using this tool.

5.7 Computing homology using discrete vector fields within Coq

We focus on defining the homology of an image using the method described in the previous sections of this chapter but using the reduction tool based on admissible discrete vector field which we have introduced in Chapter 3.

The image of Figure 5.4 is defined in the same way than in the previous section just as its facets and its simplicial complex `sc` associated. Then let us define the incidence matrices but in this case, these matrices are defined as sequences of sequences of $\mathbb{Z}_2$ instead of booleans, because we use the function `dvford` which computes the ordered and admissible discrete vector field from a matrix represen-

ted as sequence of sequences of $\mathbb{Z}_2$. We use the function `matboolseq_to_matZ2` to change the type of these matrices. Let us show the definition of the incidence matrix in degree 0 `im0_Z2`. The other matrices will be defined in an analogous way, `im1_Z2` and `im2_Z2`. Then, we define the ordered and admissible discrete vector field `dvfo`.

```
Definition im0_Z2 :=
  (matboolseq_to_matZ2 (ex_incidence_mx (sc c) 0).
Definition dvfo := dvford (im1_Z2).
```

Thanks to this vector field, we can reduce the incidence matrices as we explained in Section 3.1. Let us recall that the incidence matrix in degree 0 is empty when we work with 2D images. Consequently, we cannot reduce it. Then, we focus on reducing the other matrices. But, previously, we define the sequences which allows us to reorder the matrices and finally, to reduce them. These sequences come from the vector field computed.

```
Definition s1:= fill (getfirstElemseq dvfo) (size im1_Z2).
Definition s2:=
  fill (getsndElemseq dvfo) (size (head [::] im1_Z2)).

Definition im1_red:=
  (getMatrixReduced (size dvfo)(reorderM_dvf dvfo im1_Z2)).
Definition im2_red:=
  (dsubseqmx (size dvfo)(t_take_rows_s s2 im2_Z2)).
```

Then, we can define the homology in degree 0 and 1 using the function `ex_homology` defined in Section 5.3.

```
Definition H0_dvf:= ex_homology 0 (size im1_red)
  (size (head [::] im1_red)) im0_Z2 im1_red.
Definition H1_dvf:= ex_homology (size im1_red)
  (size im2_red) (size (head [::] im2_red)) im1_red im2_red.
```

Finally, the homology groups of Figure 5.4 are computed in the following way.

```
Eval vm_compute in (H0_dvf image1).
Eval vm_compute in (H1_dvf image1).
```

The obtained results are the same that we got in the last section. The size of the image is small and the difference of time required in both cases is not appreciable. In the next chapter we will include more interesting examples.

Chapter 6

Experimental aspects

This chapter is devoted to show some experiments carried out throughout the whole work. First, we will explain the testing performed on our implementation in Haskell. It is important to remove the possible bugs to reduce the time involved in the verification process. Second, we focus on profiling and analyzing the behavior of our algorithm paying attention to the possible improvements in our developments. Third, we show some time comparisons to compute the dimensions of homology groups of different images, depending on the system (Haskell or SSReflect), or whether the reduction process has been applied. Finally, we briefly include some further experiments with alternative algorithms to the RS algorithm.

6.1 Testing

In this section, we detail the testing process which has been applied to increase the reliability of our code. Although this is not enough to ensure that a program is correct, it worked properly for a considerable amount of examples. First, we start testing our implementation of the RS algorithm with the incidence matrices associated with several images. We want to compute the homology groups in degree 0 and in degree 1 of the corresponding images. If the image is small we can manually check that the final result is equal to the connected components or

to the holes, respectively, which can be simply observed in the image. It could also happen that the final result was correct but the method was not properly implemented. Then, we can split the process and check every computation step by step. There are five main steps: the computation of an admissible discrete vector field, the reordering of this vector field, the reordering of the incidence matrices, the computation of the reduced matrices, and the computation of the homology using the reduced matrices.

Let us show as a first example the image of Figure 6.1 (1×3 pixels). It consists of two black pixels. The incidence matrices associated with the image are:

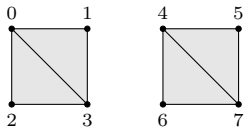


Figure 6.1: 2 black pixels

$$d_1 = \begin{array}{c} \\ (0) \\ (1) \\ (2) \\ (3) \\ (4) \\ (5) \\ (6) \\ (7) \end{array} \begin{array}{c} \begin{array}{cccccccccc} (0,1) & (0,2) & (0,3) & (1,3) & (2,3) & (4,5) & (4,6) & (4,7) & (5,7) & (6,7) \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

$$d_2 = \begin{array}{c} \\ (0,1) \\ (0,2) \\ (0,3) \\ (1,3) \\ (2,3) \\ (4,5) \\ (4,6) \\ (4,7) \\ (5,7) \\ (6,7) \end{array} \begin{array}{c} \begin{array}{cccc} (0,1,3) & (0,2,3) & (4,5,7) & (4,6,7) \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

From these matrices we can compute every step and test the properties ob-

tained with the expected results.

Let us note that the incidence matrices are big regarding to the size of the image. These incidence matrices are built from the black pixels of the image. Each one of these pixels can originate 4 rows and 5 columns in d_1 and 5 rows and 2 columns in d_2. In this example, the incidence matrices are of the following size: 8×10 and 10×4. For the image of Figure 6.2 (5×5 pixels), they are of size: 26×45 and 45×24.

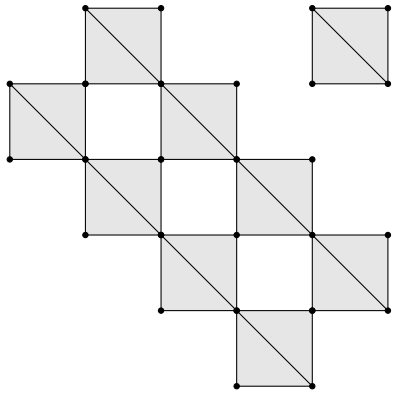


Figure 6.2: Image 5×5 pixels

6.1.1 Automated testing

The manual testing is very tedious and requires much time. So, it is not a feasible option if we want to check that the reduction preserves the homology for a big amount of examples. To this aim, we use *fKenzo* [Her09] which is a friendly interface of the *Kenzo* system [DRSS98]. fKenzo includes a module for digital images which can compute the required homology groups. To this aim, we need the incidence matrices which are obtained from the chain complex associated with an image.

Let us create a file with a battery of pairs of matrices (the corresponding incidence matrices) which come from digital images and their computed homology groups. The examples of 2D digital images are randomly generated as matrices whose entries are 0's and 1's. Then, Kenzo computes the homology from the image

as we explained in Section 1.1. Afterwards, we have implemented an algorithm in Haskell which allows us to read this file and for every pair of matrices we compute the homology groups. Then, it is checked that the dimensions of the homology groups are the same that the ones computed by Kenzo. We also implemented in Haskell the RS algorithm which allows us to obtain the reduced matrices. This also allows us to calculate the percentage of reduction of the incidence matrices. Moreover, the homology groups are computed using the reduction obtained by applying the RS algorithm. Finally, it is also checked that we obtain the same groups that the ones obtained using the previous methods.

This study has been performed with 500 matrices which corresponds with 250 digital images. From this process, we can highlight the following comments. First, let us focus on the percentage of reduction. In our work, we compute the admissible discrete vector field of the incidence matrix in degree 1, d_1 and then we reorder both d_1 and d_2. Finally, we obtain the reduced matrices $\hat{d}_1$ and $\hat{d}_2$. However, this process can be repeated computing now an admissible discrete vector field over the differential $\hat{d}_2$. The average percentages of reduction of the initial matrices applying once or twice the RS algorithm are shown in Table 6.1. Let us distinguish between the differential in degree 1 and the one in degree 2 since the results are different. Let us note that if the algorithm is applied twice all the matrices d_2 are completely reduced.

	Algorithm applied once		Algorithm applied twice	
	d1	d2	d1	d2
% of reduction	98	49	95	100

Table 6.1: Percentages of reduction of matrices

As we said, apart from the percentages of reduction we can check if the applied reduction keeps the homological properties. In particular, it allows us to test if the results obtained when computing the homology groups in dimension 0 and 1 using the reduction algorithm, coincide with the ones computed in the file. This is another way of checking that our implementation in these test cases is correct.

6.1.2 Testing with QuickCheck

As we said in Subsection 1.3.2, QuickCheck allows us to randomly test properties of programs. The process to test that a program verifies some properties consists of two steps:

- Providing a specification of the program.
- Testing that the properties included in the specification are satisfied by a number of cases which have been randomly generated.

QuickCheck is a tool which can be used in different systems. In our case, we use it into Haskell. The basic instructions to define a property are the following ones:

```
<name_property><parameters> = <prop>
<name_property><parameters> = <conditions> ==> <prop>
```

The difference between these two expressions is that the second one includes some preconditions. Then the command `QuickCheck` tests that the property is true in generated cases. Some examples were introduced in Subsection 1.3.2.

The system returns whether the testing has been successful. In the failed cases, the system shows the concrete example which has caused that the property was false. The problem can appear due to two reasons. One reason is the input parameters do not satisfy the preconditions of the property. The other one is that the program is not correct. This is a good point to sort out some of the mistakes which could appear in an implementation.

In order to test the RS algorithm, we state the properties which should be verified. In our case, we want to check that the output given by `gen_admdvf_ord` (function which uses `genDvfOrders` and builds an admissible discrete vector field) consists of an admissible discrete vector field and the set of relations built in the RS algorithm. Then, M is a matrix over $\mathbb{Z}_2$ with m rows and n columns, $V = (a_i, b_i)_i$ is a discrete vector field from M and $ords$ is the transitive closure of the relations associated with V, the properties to test are the ones coming from Definition 1.42 and the admissibility property adapted to the $\mathbb{Z}_2$ case.

1. $1 \leq a_i \leq m$ and $1 \leq b_i \leq n$.
2. $\forall i$, $M(a_i, b_i) = 1$.
3. $(a_i)_i$ (resp. $(b_i)_i$) are pairwise different.
4. *ords* does not have any loop (admissibility property).

These four properties have been encoded in Haskell by means of a function called `isAdmVecfield`. To test in QuickCheck that our implementation of the RS algorithm fulfills the specification given by `isAdmVecfield`, the following *property definition*, using QuickCheck terminology, is defined.

```
condAdmVecfield M = let advf = (gen_admdvf_ord M) in
  isAdmVecfield M (advf.1) (advf.2)
```

The definition of `condAdmVecfield` states that given a matrix `M`, the output of `genDvfOrders`, both the discrete vector field (first component) and the relations (second component) from `M`, fulfill the specification of the property called `isAdmVecfield`. Now, we can test whether `condAdmVecfield` satisfies such a property.

```
> quickCheck condAdmVecfield
+ + + OK, passed 100 tests.
```

The result produced by QuickCheck means that QuickCheck has generated 100 random values for `M`, and the property was true for all these cases.

We could also check other properties such as the top-left submatrix of the reordered matrix using the admissible discrete vector field is a triangular matrix. Moreover, this matrix is invertible and therefore, its determinant is 1. Initially, we define this property in the following way.

```
prop_conjuntaZ M =
 isAdmVecfield M vf ord && ((triangular 1 (epsilon M)) == True)
 && (det 1 (epsilon M) == 1)
 && ((mulseqmx (inverse (length (epsilonseq (length vf) mord))
 (epsilonseq (length vf) (mord)))(epsilonseq (length vf) (mord)))
  == (matrixIdentity (length (epsilonseq (length vf) (mord)))))
 where vf = genDvf M
       ord = genOrders M}
```

But, if this property is executed it fails. As we said, QuickCheck returns cases where this property is false. Some of them can be empty lists, null lists, lists with elements different from 0 and 1 or a list which has not the shape of a matrix. The preconditions about the input parameter `M` (list of lists) have to delete these cases. However, in the testing only around 2 matrices of every 500 matrices randomly generated verify the preconditions so, the condition is tested only for this pair of cases.

As our main aim is testing this property in a considerable number of cases, this would not be very useful. Consequently, we have modified the sequence randomly generated in order to verify the preconditions. First, we have transformed the sequence with `matrixM01` to get that the elements of the sequence are only 0's and 1's and so, reducing a precondition. This is necessary since QuickCheck generates integer matrices and we work with $\mathbb{Z}_2$. In particular, the examples randomly generated will be modified changing the even numbers by 0 and the odd ones by 1. Then, the testing will be applied around 20 matrices of every 500 because the rest of preconditions are not verified. Second, we define a function whose name is `conversion` which is in charge of converting a null or empty sequence of integers into a matrix whose elements are only 0's and 1's. Moreover, this matrix has to contain at least an element whose value is 1 in every row. Finally, it is important to note that every row of the list (matrix) has the same size. The necessary elements will be added to get this.

```
prop_conjuntaZ M =
  let m = (conversion (matrixM01 M)) in
  let vf = (dvford m) in
  let mord = ((matrixM01 (reorderM_dvf vf m))) in
  isAdmVecfield vf ord &&
  ((triangular 1 (epsilonseq (length vf) mord)) == True) &&
  (det 1 (epsilonseq (length vf) mord)  == 1) &&
  ((mulseqmx (inverse (length (epsilonseq (length vf) mord))
   (epsilonseq (length vf)(mord)))(epsilonseq (length vf)(mord)))
   == (matrixIdentity (length (epsilonseq (length vf) (mord)))))
  where ord = genOrders m}
```

Let us note that all the sequences of sequences which are randomly generated in QuickCheck are converted into a matrix with some particular features and, in

this way, we can test the properties for all those sequences.

6.2 Profiling in Haskell

Haskell provides us a time and space profiling system [Hut07]. This allows us to understand the execution behavior of our program, so we can improve it. Profiling a program is a three-step process. First, we have to re-compile our program to profile with the `-prof` and `-auto-all` option. Second, we have to run our program with one of the profiling options, in our case, we will use `+RTS -p`. This generates a file of profiling information called <`prog`>`.prof` with `prog` the name of the program. Finally, we examine the generated profiling information, using one of the GHC's profiling tools. GHC's profiling system assigns costs (the time invested in a computation) to cost centers. In other words, each function shows the division of costs corresponding to other functions which are used from it. Furthermore, GHC gives us the stack of enclosing cost centers showing a call-graph of cost contributions. This tool has been useful to realize where the system wastes time, what parts of the implementations should be improved, and if the data types used are suitable.

The first part of the profiling file gives us the program name and the options used in the compiling, and the total time and total memory allocation consumed during the running of the program. The second part is a decomposition by cost center of the most costly functions in the program.

In Figure 6.3 we show an extract of a profiling file where it is detailed the costs in the computation of the reduced matrix of a small matrix of size 100×200. We can see in the top part of this file (Figure 6.3) that the significant functions in the program are `getfirstE` (responsible for 27.6% of the total time) and `getcol` and `firstElem1` (both are responsible for 13.8% of the total time). In the bottom part of the file, we obtain a profile break-down by cost-center stack of the significant functions used in the computation of the reduced matrix. In this case, we can see that there are two main methods where the time cost is higher: `dvford` nearly 76% and `getMatrixReduced` with almost 14%. Let us highlight that `dvford`, which computes the reordered admissible discrete vector field, uses the most significant functions: `getfirstE`, `getcol` and `firstElem1`.

The biggest problem is the computation of the admissible discrete vector field

```
        Dvf_CalculoHom2.exe +RTS -p -RTS

      total time  =        0.58 secs   (29 ticks @ 20 ms)
      total alloc = 168,421,952 bytes  (excludes profiling overheads)

COST CENTRE                    MODULE                %time %alloc

getfirstE                      Dvf_efficiency_viejo  27.6   15.8
getcol                         Dvf_efficiency_viejo  13.8    3.6
firstElem1                     Dvf_efficiency_viejo  13.8   35.8
prodE                          Dvf_efficiency_viejo   6.9    5.4
canAddOrders                   Dvf_efficiency_viejo   6.9    7.1
prodR                          Dvf_efficiency_viejo   3.4    0.7
oprowM                         Dvf_efficiency_viejo   3.4    5.7
isinSeq                        Dvf_efficiency_viejo   3.4    0.3
canAddOrder                    Dvf_efficiency_viejo   3.4    0.5
addOrder_orders                Dvf_efficiency_viejo   3.4    1.8
addOrder_order                 Dvf_efficiency_viejo   3.4    0.9
genfich2                       Main                   3.4    0.9
genFich                        Main                   3.4    0.0
CAF                            Main                   3.4    0.0
nocycles_order                 Dvf_efficiency_viejo   0.0    2.2
getsndElemseq                  Dvf_efficiency_viejo   0.0    2.1
getfirstElemseq                Dvf_efficiency_viejo   0.0    5.0
createRowIdent                 Dvf_efficiency_viejo   0.0    5.6
stringV                        Main                   0.0    1.7
split                          Main                   0.0    1.1


                                                          individual     inherited
COST CENTRE              MODULE                  no.    entries  %time %alloc   %time %alloc

MAIN                     MAIN                     1          0   0.0    0.0   100.0  100.0
 CAF                     Main                   404          8   3.4    0.0   100.0  100.0
  genFich                Main                   411          1   3.4    0.0    96.6  100.0
   genfich2              Main                   412          1   3.4    0.9    93.1  100.0
    getMatrixReduced     Dvf_efficiency_viejo   454          1   0.0    0.0    13.8   17.7
     inverse             Dvf_efficiency_viejo   466          1   0.0    0.0     3.4   11.4
      glueIdent          Dvf_efficiency_viejo   469          1   0.0    0.1     0.0    5.7
       createRowIdent    Dvf_efficiency_viejo   471         84   0.0    5.6     0.0    5.6
      opM                Dvf_efficiency_viejo   468          1   0.0    0.0     3.4    5.7
       oprowM            Dvf_efficiency_viejo   470         84   3.4    5.7     3.4    5.7
        addR             Dvf_efficiency_viejo   475          2   0.0    0.0     0.0    0.0
      rsubseqmx          Dvf_efficiency_viejo   467          1   0.0    0.0     0.0    0.0
     epsilonseq          Dvf_efficiency_viejo   464          0   0.0    0.0     0.0    0.1
      ulsubseqmx         Dvf_efficiency_viejo   465          1   0.0    0.1     0.0    0.1
     psiseq              Dvf_efficiency_viejo   462          0   0.0    0.0     0.0    0.0
      dlsubseqmx         Dvf_efficiency_viejo   463          1   0.0    0.0     0.0    0.0
     mulseqmx            Dvf_efficiency_viejo   461         14   0.0    0.0    10.3    6.1
      prodR              Dvf_efficiency_viejo   473       1002   3.4    0.7    10.3    6.1
       prodE             Dvf_efficiency_viejo   474      85170   6.9    5.4     6.9    5.4
     ...
     ...
    dvford               Dvf_efficiency_viejo   417          1   0.0    0.0    75.9   78.5
     genDvfOrders        Dvf_efficiency_viejo   420       3544   0.0    0.4    51.7   62.1
      getcol             Dvf_efficiency_viejo   433          0  13.8    3.6    13.8    3.6
```

Figure 6.3: Profiling file

of a matrix. This computation depends on the representation chosen for matrices. It is known that the most efficient algorithm is usually the one which has the most complicated proof ([DMS12a]). However, we cannot forget that the final aim is the proof of the correctness of the algorithm. So, in this case it is complicated to choose the best representation. We have to take into account that the algorithm needs to access to any position of the matrix and to extract a column to compute the admissible discrete vector field. We also want to take blocks of matrices and to reorder the matrix to obtain the reduced matrix. There exist several representations of a matrix that can be chosen in our implementations. Then, a representation can ease some computations but can grow worse the others. Since we work with 2D monochromatic images and the incidence matrices are only composed by 0's and 1's where most of these elements are 0, we think that a sparse representation for matrices could be useful. Although this representation is efficient for some operations, it is not suitable to access to a column or to extract blocks. Another possible representation is included in [CMS12] where a matrix is represented as a "fan". In other words, a matrix can be divided into four parts: an element (the component $(1,1)$ of the matrix), the rest of the first row, the rest of the first column and the block composed by the rest of the matrix. This representation is not convenient to reorder a matrix but is useful to compute the rank of a matrix. The matrix could also be represented by a sequence of rows or by a sequence of columns. The sequence of columns representation is convenient for generating the partial orders whenever a vector is added to the vector field. But, it is not suitable to run over the elements of the matrix looking for possible vectors. Finally, we have not found a better matrix representation that the one initially used for this algorithm. It consisted in using sequences of rows where every row is a sequence.

In neuron images, whose size are at least of 1000×1500 pixels, a main problem is the computation of the vector field. However, the computational cost to obtain the reduced matrix from the matrix already sorted is also very high. This is due to the implementation of the operations with general matrices. To this aim, we try to improve the implementations given to reorder matrices, the product of matrices and the inverse of upper triangular matrices. In fact, an improvement of this inverse in SSReflect was introduced in [HK13]. An "automatic" way exists to extract SSReflect code (Coq code in general) (see [tdt12]). For instance, let us show the extraction of the method `addseqmx` in charge of the addition of sequences which represent matrices. Its definition in SSReflect is as follows.

```
Definition addseqmx (M N : seq (seq nat)) : (seq (seq nat)) :=
  map2seqmx M N (fun x y => x + y).
```

The syntax to extract this method is the following: `Extraction "name_file" addseqmx`. The extracted Haskell code is stored in the file `name_file`. Let us show a part of the generated file.

```
module name_file where
import qualified Prelude

data Nat =
   O
 | S Nat
...
...
addseqmx ::
  (List (List Nat)) -> (List (List Nat)) -> List (List Nat)
addseqmx m n =
  map2seqmx m n (\x y -> addn x y)
```

The problem of this process is that basic definitions are also defined without taking into account the definitions already implemented in Haskell. In fact, data types are redefined, in this case, the natural numbers. Then, it is necessary to adapt this output renaming manually the methods which belong to Haskell.

For instance, let us note that a `(List (List Nat))` should be translated into `[[Int]]` and the function which adds two natural numbers `addn` should not be used because Haskell has already implemented the addition of integers. However, the other function `map2seqmx` has been directly extracted from the SSReflect code.

```
addseqmx :: [[Int]] -> [[Int]] -> [[Int]]
addseqmx m n =
  map2seqmx m n (\x y -> x + y)
```

Using these new implementations the time cost devoted to the process of reducing the matrix from the reordered matrix decreases. Namely, we can see in Figure 6.4 that it spends 11.1% of the total time cost instead of 13.8% shown in Figure 6.3. Moreover, if we pay attention to the product of matrices `mulseqmx`,

the time cost of this computation has also decreased from 10.3% to 3.7%. This difference increases when the size of the image is larger. For instance, in the example of images of neurons, this process decreased 14%.

```
        Fri Jan 04 09:53 2013 Time and Allocation Profiling Report  (Final)

           Dvf_CalculoHom2.exe +RTS -p -RTS

        total time  =        0.54 secs   (27 ticks @ 20 ms)
        total alloc = 168,259,856 bytes  (excludes profiling overheads)

...
...
                                                                 individual    inherited
COST CENTRE              MODULE              no.    entries  %time %alloc   %time %alloc

MAIN                     MAIN                  1          0   3.7    0.0   100.0  100.0
 CAF                     Main                404          8   0.0    0.0    92.6  100.0
  genFich                Main                411          1   0.0    0.0    92.6  100.0
   genfich2              Main                412          1   0.0    0.9    92.6  100.0
    getMatrixReduced     Dvf_efficiency      454          1   0.0    0.0    11.1   17.6
     ...
     psiseq              Dvf_efficiency      463          0   0.0    0.0     0.0    0.0
      dlsubseqmx         Dvf_efficiency      464          1   0.0    0.0     0.0    0.0
     mulseqmx            Dvf_efficiency      462          2   0.0    1.8     3.7    3.6
      fold12             Dvf_efficiency      467      85170   3.7    1.2     3.7    1.2
      trseqmx            Dvf_efficiency      465          2   0.0    0.5     0.0    0.5
       rowseqmx          Dvf_efficiency      466          2   0.0    0.0     0.0    0.0
     phiseq              Dvf_efficiency      460          0   0.0    0.0     0.0    0.0
```

Figure 6.4: Profiling file with improvements

6.3 Computational results

Using the methodology presented throughout Section 1.3, we have verified the correctness of the RS algorithm. This let us obtain a reduced matrix preserving the homological properties of the original one. The formalization process was detailed in Chapters 3 and 4. Previously, the Haskell implementation was tested in Chapter 2. Moreover, we have a Coq implementation which can compute the homology presented in Chapter 5. Namely, we have integrated the programs presented in this work with the ones devoted to the computation of homology groups of digital images introduced in [HDM+12].

In this subsection, we compare the computational time required to obtain the H_0 and H_1 in Haskell and SSReflect. We also distinguish two ways of computing the homology: directly from the incidence matrices of the image or

after applying the reduction process using the RS algorithm.

Let us begin considering some small images. The first image (corresponding to Figure 6.5) is composed by 3×3 pixels. The sizes of its associated incidence matrices are: 12×20 and 20×8.

Figure 6.5: An image 3×3 pixels

	Haskell	*SSReflect*
Without advf	**0.06**	**0.46**
With advf	**0.046875**	**1.016**

Table 6.2: Computational times in secs. of H_0 and H_1 of Figure 6.5

Knowing the main functionalities of both programs, we notice that the computation in Haskell is much quicker than the one in SSReflect. The former is focused on programming methods instead of verification which is the heart of SSReflect. However, we are completely sure that our programs work properly using the SSReflect code. The timing cost devoted to this computation is shown in Table 6.2. Let us note that the computational times of H_0 and H_1 using admissible discrete vector fields is higher due to the intermediary steps such as the computation of the vector field. There are not big differences between the two ways of computing the homology because the image is small. However, the larger the image is the bigger are the differences between both systems.

Let us see that the differences increase with Figure 6.6. This image is composed by 68 (4×17) pixels. The sizes of the associated incidence matrices are 68×124 and 124×58.

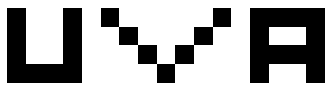


Figure 6.6: UVA image

The results of the computations are shown in Table 6.3.

		Haskell	*SSReflect*
Without advf		**0.9**	**9**
With advf	Computation advf	2.484	101
	Ordered matrices	0	14
	Reduced matrix d_1	0	1
	Reduced matrix d_2	0.0624	5
	H_0 and H_1	1.252	3
	Total	**3.9**	**139**

Table 6.3: Computational times in secs. of H_0 and H_1 of Figure 6.6

The computation in Haskell is much quicker, but we are interested in the verification of the programs. In other words, we want to be able to compute but being completely sure of the correct behavior of our programs. Therefore, Haskell cannot be used to this aim. In SSReflect, we see that the computation of the 0 and 1-dimensional homology groups of the reduced matrices is faster than the computed from the initial matrices (3 seconds instead of 9 seconds). We are computing homology groups of the reduced matrices whose dimensions are 4×60 and 60×58 (the size of admissible discrete vector field which is 64).

The time devoted to reduce those matrices is considerable. The time which takes the reduction of the incidence matrix in degree 1 in SSReflect is not exactly what appears in the fourth row of the SSReflect column in Table 6.3. Indeed, SSReflect cannot compute the inverse of a matrix 64×64 of the top-left block of the ordered matrix. We denote this matrix as `ulM`. This inverse is necessary to obtain the reduced matrix as we said in Section 3.1. Let us recall that this matrix is an square matrix whose size is given by the discrete vector field. Then, we have used Haskell as an oracle to obtain that inverse. The process carried out is the extraction of `ulM` to Haskell code, computing the inverse in Haskell, and returning the inverse to SSReflect. Finally, in SSReflect, we prove that the new matrix is really the inverse of `ulM`. In this way, the whole process is completely verified.

Let us detail this verification. We have proved in Coq that the inverse of a matrix is unique and that $\forall M, MM^{-1} = \mathrm{id} \Rightarrow M^{-1}M = \mathrm{id}$. Then, we introduce the following lemma which states that the product of `ulM` and its inverse is the

identity matrix.

```
Lemma m_invmxm : (mulseqmx ulM invmx_ulM)
  == (scalar_seqmx 64 (Ordinal (ltn_pmod 1 (ltn0Sn 1)))).
```

This is proved evaluating this expression using the `vm_compute` tactic.

This way of verifying is not the standard one but it is sound. We can delegate to Haskell heavy computations, obtaining in Haskell the required objects, and proving in SSReflect the properties of the computed objects.

Anyway, in the case of the image in Figure 6.6, the RS algorithm is not necessary because we can compute the dimension of the homology groups with the direct method, without using discrete vector fields.

In conclusion, the reduction method is not very useful when dealing with small images because we can directly compute the homology groups. However, if the images grow this method cannot be executed in SSReflect. Then, the RS algorithm will have more sense. In the following subsection, we deal with larger images which come from a real biomedical problem.

6.3.1 Biomedical images

Biomedical images are a suitable benchmark for testing our programs, the reason is twofold. First, the amount of information included in this kind of images is usually quite big; then, a process able to reduce those images but keeping the homological properties can be really useful. In addition, software systems dealing with biomedical images must be trustworthy; this is our case since we have formally verified the correctness of our programs.

As an example, we can consider the problem of counting the number of *synapses* in a neuron. Synapses [BCP06] are the points of connection between neurons and are related to the computational capabilities of the brain. Therefore, the treatment of neurological diseases, such as Alzheimer, may take advantage of procedures modifying the number of synapses [C+11].

Up to now, the study of the *synaptic density evolution* of neurons was a time-consuming task since it was performed, mainly, manually. To overcome this issue, an automatic method was presented in [HMPR11]. Briefly speaking, such a pro-

cess can be split into two parts. First, from three images of a neuron (the neuron with two antibody markers and the structure of the neuron) a monochromatic image is obtained, see Figure 6.7[1]. In such an image, each connected component represents a synapse. Then, the problem of measuring the number of synapses is translated into a question of counting the connected components of a monochromatic image.

In the context of Algebraic Digital Topology, this issue can be tackled by means of the computation of the homology group H_0 of the monochromatic image. This task can be performed in Coq through the formally verified programs which were presented in [HDM+12]. Nevertheless, such programs are not able to handle images like the one of the right side of Figure 6.7 due to its size. In spite of the fact that the aim of this type of tools is not the efficiency, there is an effort towards the efficient implementations of mathematical algorithms running inside Coq, as shown by recent works on efficient real numbers [KS11], machine integers and arrays [AGST10] or an approach to compiled execution of internal computations [GL02]. In order to overcome this drawback we have integrated our reduction programs with the ones presented in [HDM+12].

We focus on reducing the incidence matrix in degree 1 which is enough to compute the H_0 of the chain complex associated with the image. The problem is that SSReflect cannot compute the reduced matrix. The image is composed by 871×560 pixels and the corresponding incidence matrix in degree 1 consists of 743 rows and 1424 columns.

Therefore, the process which we have carried out is the following one.

1. Computing in Haskell the reduced matrix $Mred$ and the components $f0$, $f1$, $g0$, $g1$ and $h0$ which define a 2-truncated reduction.
2. Transform the matrices to Coq/SSReflect.
3. Prove in Coq/SSReflect that these matrices establish a reduction.
4. Compute H_0 from the reduced matrix $Mred$ in Coq/SSReflect.

Using this approach, we can successfully compute the homology dimension of the biomedical image using the reduced matrix in just 5 seconds, a remarkable

[1]The same images with higher resolution can be seen in `http://www.unirioja.es/cu/joheras/synapses/`

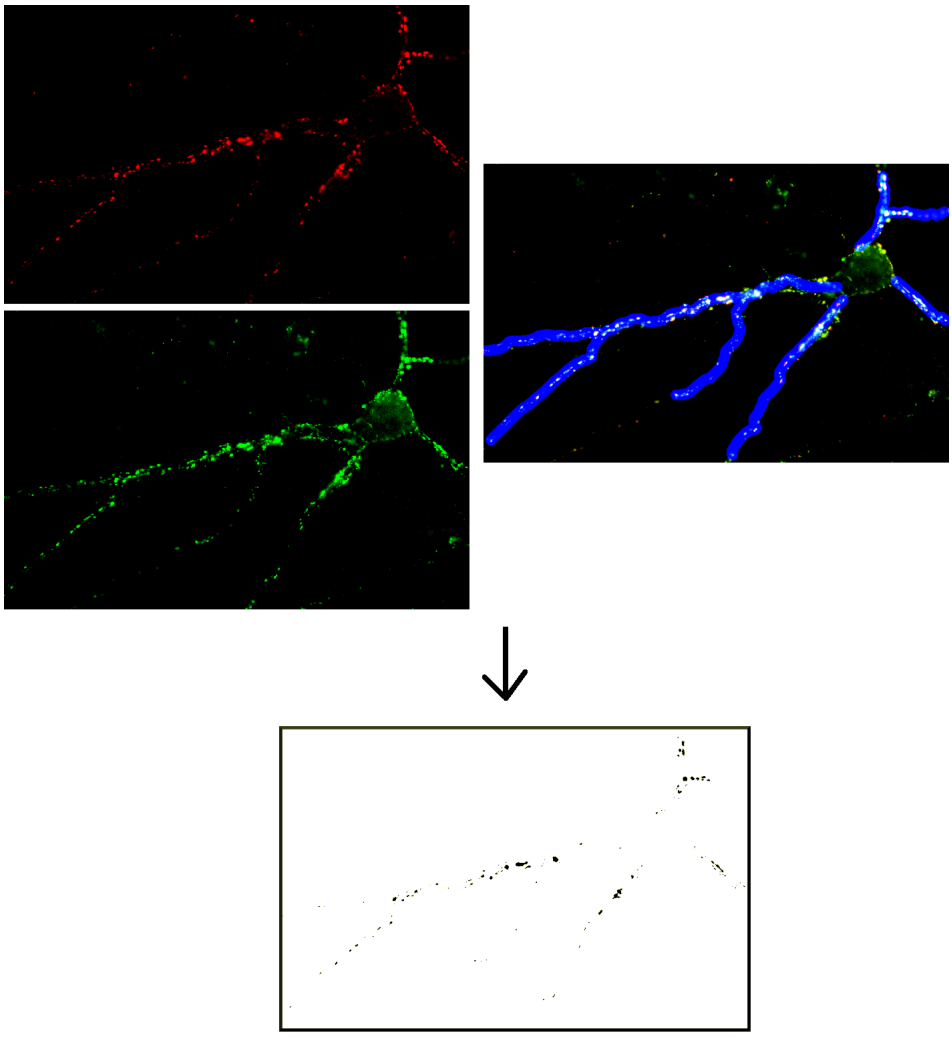


Figure 6.7: Synapses extraction from three images of a neuron

time for an execution inside Coq. This is due to the incidence matrix is reduced to a matrix 59×740 thanks to the length of the admissible discrete vector field computed which is 684. Finally, it is obtained the computation of H_0 which is 59. This means that the image is composed by 59 connected components, and there are 59 synapses in the original image.

Then, we compute the proofs which establish that there is a reduction between the initial matrix and the reduced one. The time that Coq devotes to define the big matrices is 10 minutes and 12 seconds, and to obtain the proofs, approximately 12 hours. These times are shown in Table 6.4.

		Haskell (sec)	*SSReflect*
Without advf		**0.68**	**Not available**
With advf	Computation advf	108	Proofs
	Reduced matrices	26	12h 4min 55sec
	H_0	0.012	5sec
	Total	**130**	**12h 5min**

Table 6.4: Computational times of H_0 of Figure 6.7

In contrast, let us show a part of an image of a neuron in Figure 6.8 where we need to apply the RS algorithm to reduce the incidence matrix in degree 1. As the image is large and the information is located into the left part of the image, we enlarge this zone (see Figure 6.9) to be able to see the number of connected components which is exactly 32. Anyway, we will deal with the whole piece.

Figure 6.8: A part of a neuron

Applying the RS algorithm, we reduce the incidence matrix in degree 1 (480×826) to a matrix whose number of rows is 32 and the number of columns is 378. The time spent on every step is detailed in Table 6.5.

The different ways of computing the homology presented in Table 6.5 give the same result, which is that the 0-dimensional homology group is 32. In that case, the direct computation of homology cannot be obtained. However, the RS algorithm can directly be applied. In this case, it is not necessary to extract every

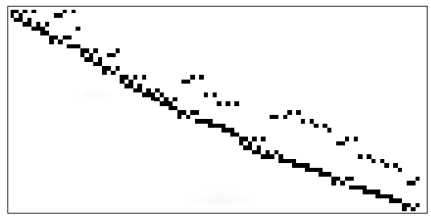


Figure 6.9: Zoom of Figure 6.8

		Haskell (min)	*SSReflect*
Without advf		**0.26**	**Not available**
With advf	Computation advf	15.19	97h 7min 50sec
	Reduced matrix	5.20	29min 10sec
	H_0	0.002	27 sec
	Total	**20.402**	**97h 37min 27sec**

Table 6.5: Computational times of H_0 of Figure 6.8

component of the 2-truncated reduction to Haskell but it is enough to delegate the computation of the inverse to Haskell.

6.4 Other algorithms

If our purpose only consists in computing the 0-dimensional homology group of an image, as we have said this is enough in the case of synapses, there is another algorithm to compute an admissible discrete vector field which takes profit of the very particular structures involved in that computation. Concretely, the chain complex has only one non-null incidence matrix. In this case, it is possible to introduce an efficient algorithm to obtain an admissible discrete vector field for this matrix. This matrix relates edges to vertices. The matrix associated with an image in dimension 1 has only two non-null elements in every column. In this case, if a vector is selected only one relation will be added. Therefore, it is easier to obtain the transitive closure than in the algorithm implemented in Section 2.2. Moreover, the features of this incidence matrix allow us to develop some heuristic techniques and efficient specific methods. This algorithm is implemented in the

system *Kenzo*. However, in this system, you can only compute the vector field and the size of the reduced matrix, but the reduced matrix is not provided. Both the algorithm to build an admissible discrete vector field from a matrix of this concrete case and its reduction has been implemented in Haskell. Table 6.6 contains the timing devoted to the computation of H_0 of Figure 6.7.

		Haskell (sec)
With advf (Efficient version)	Computation advf	2.24
	Reduced matrices	27.42
	H_0	0.02
	Total	**29.79**

Table 6.6: Efficient computation of H_0 from Figure 6.7

Using this algorithm, the size of the admissible discrete vector field is 646. This size is slightly below than the computed using the RS algorithm (whose length was 684). Then, the reduced matrix is a 97×878 matrix instead of 59×740 with the RS algorithm. Although the reduced matrix is slightly bigger than the one obtained using the RS algorithm, it is good enough taking into account that the size of the initial incidence matrix was 743×1424. Of course, the result of the computation coincides with the output of the other methods.

However, it would be a challenging task the verification of this algorithm due to the heuristic techniques used in the implementation.

In Section 3.6, we introduced and formally verified another reduction method based on collapses. This algorithm is quicker than the RS algorithm but the reduced matrices that we obtain are bigger. The comparison of the computational times of the 0 and 1-dimensional homology groups of the images shown in Coq in Figure 6.5 and Figure 6.6 of both methods are shown in Table 6.7.

	Figure 6.5	Figure 6.6
RS algorithm	1.016	139
Collapses algorithm	0	29

Table 6.7: Computational times of H_0 and H_1 in SSReflect using reduction methods

Anyway, this method cannot be either applied over huge images like biomedical digital images, as the one of Figure 6.7.

Conclusions and further work

In this memoir, we have reported on a research which provides a certified computation of homology groups associated with some digital images coming from a biomedical problem. The main contributions allowing us to reach this challenging goal have been the following ones.

- The implementation in Coq/SSReflect of the Romero-Sergeraert algorithm [RS10] computing an admissible discrete vector field for a digital image.
- The complete formalization in Coq/SSReflect of the theorem known as Basic Perturbation Lemma (BPL).
- Two formalizations of the Vector-Field Reduction Theorem for matrices. One is proved using the BPL and the other one applying the Hexagonal Lemma [RS10].
- A discussion on different methods to overcome the efficiency problems appearing when executing programs inside proof assistants. In particular, the Haskell programming language has been used in two different ways: first, to model algorithms which are subsequently implemented in Coq and, second, as an oracle to produce results whose properties are verified in the proof assistant.
- A verified program to compute homology groups of a simplicial complex obtained from a digital image.

- As a by-product of all the other contributions, an application of Algebraic Topology to study biomedical images in a reliable manner. Our methodology ensures that the final homological calculations are correct.

By observing the memoir as a whole, it is clear that, even if our initial emphasis was in biomedical applications, much of the work has been devoted to the formalization of mathematics and program verification. The reason is that, even if we have built over very solid foundations (effective homology [RS10] from the algorithmic side and SSReflect [GM10] as theorem proving basis), we needed to reproduce inside Coq/SSReflect a part of Computational Algebraic Topology. As a consequence, we focused on obtaining a complete verified path from biomedical digital images to homological computing, but without getting a good performance in the final implementation. Thus, our research should be considered as a proof of concept: homological image processing can be implemented in a verified manner by using interactive theorem provers. Our results are therefore rather a starting point, instead of a closed problem.

As for future work, several problems remain open. The most evident one, after our previous discussion, is obtaining a better performance in the execution process. This can be undertaken at three different levels. First, by using other algorithms to compute the main objects in our approach (discrete vector fields, inverses of matrices, and so on).

Second, and not independent from the previous one, by implementing more efficient data structures and representations. A first idea is to work with *cubical complexes* [ZA02] instead of with simplicial ones. Also more specific data refinements [DMS12a] could be considered, trying to automatically translate proofs from one (abstract) representation to other (efficient) ones.

As a third more technological aspect about performance, the improvement of the running environments in proof assistants could be approached.

With respect to applications, in the area of Computational Algebraic Topology, our results could be extended to homology with integer coefficients. This generalization could make it possible to undertake the verification of other Kenzo results, as those reported in [RR12].

Another line of research is to apply our methodology and techniques to other problems related to the homological processing of biomedical images. The best

candidate is *persistent homology*, which has been already applied and formalized (see [HCMS]). Namely, it could be applied in stacks of neurons to remove the noise of the images and help to the detection of the dendrites (the branches of the neuron). The project would be to study whether our reduction strategy can be also profitable in this new homological context. Furthermore, we can focus on recognizing the structure of a neuron; a problem which seems to involve the computation of homology groups in dimension 1, see [M$^+$12], a question which could be tackled with our tools.

Bibliography

[ABR08] J. Aransay, C. Ballarin, and J. Rubio. A mechanized proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning*, 40(4):271–292, 2008.

[ABR10] J. Aransay, C. Ballarin, and J. Rubio. Generating certified code from formal proofs: a case study in homological algebra. *Formal Aspects of Computing*, 2(22):193–213, 2010.

[AD09] J. Aransay and C. Domínguez. Modelling Differential Structures in Proof Assistants: The Graded Case. In *Proceedings 12th International Conference on Computer Aided Systems Theory (EUROCAST'2009)*, volume 5717 of *Lecture Notes in Computer Science*, pages 203–210, 2009.

[AD10] J. Aransay and C. Domínguez. Formalizing simplicial topology in Isabelle/HOL and Coq. In L. Lambán, A. Romero, and J. Rubio, editors, *Contribuciones científicas en honor de Mirian Andrés Gómez*, pages 21–42. Universidad de La Rioja, 2010.

[ADFQ03] R. Ayala, E. Domínguez, A.R. Francés, and A. Quintero. Homotopy in digital spaces. *Discrete Applied Mathematics*, 125:3–24, 2003.

[AGST10] M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with imperative features and its application to SAT verification. In *Proceedings 1st Interactive Theorem Proving (ITP'10)*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98, 2010.

[BC02] G. Barthe and P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. In *Proceedings 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'02)*, volume 2410 of *Lectures Notes in Computer Science*, pages 31–46, 2002.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[BCP06] M. Bear, B. Connors, and M. Paradiso. *Neuroscience: Exploring the Brain*. Lippincott Williams & Wilkins, 2006.

[Ben06] N. Benton. *Machine Obstructed Proof: How many months can it take to verify 30 assembly instructions?* Microsoft Research, 2006. `http://research.microsoft.com/en-us/um/people/nick/mop.pdf`.

[BGBP08] Y. Bertot, G. Gonthier, S.O. Biha, and I. Pasca. Canonical big operators. In *Proceedings 21st International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 86–101, 2008.

[BKM96] B. Brock, M. Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'1996)*, volume 1166 of *Lecture Notes in Computer Science*, pages 275–293, 1996.

[BL91] D. Barnes and L. Lambe. Fixed point approach to Homological Perturbation Theory. *Proceedings of the American Mathematical Society*, 112(3):881–892, 1991.

[Bro67] R. Brown. The twisted Eilenberg-Zilber theorem. *Celebrazioni Archimedi de Secolo XX, Simposio di Topologia*, pages 34–37, 1967.

[C+11] G. Cuesto et al. Phosphoinositide-3-Kinase Activation Controls Synaptogenesis and Spinogenesis in Hippocampal Neurons. *The Journal of Neuroscience*, 31(8):2721–2733, 2011.

[CDMS] C. Cohen, M. Dénès, A. Mörtberg, and V. Siles. Smith Normal form and executable rank for matrices. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ProofExamples`.

[CH88] T. Coquand and G. P. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[CH00] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, 2000.

[CM12] C. Cohen and A. Mahboubi. Formal proofs in real algebric geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science*, 8:1–40, 2012.

[CMS12] T. Coquand, A. Mörtberg, and V. Siles. A formal proof of Sasaki-Murao algorithm. *Journal of Formal Reasoning*, 5(1):27–36, 2012.

[DLR07] C. Domínguez, L. Lambán, and J. Rubio. Object oriented institutions to specify symbolic computation systems. *Rairo - Theoretical Informatics and Applications*, 41:191–214, 2007.

[DMS] M. Dénès, A. Mörtberg, and V. Siles. Module seqmatrix of CoqEAL. `http://www-sop.inria.fr/members/Maxime.Denes/coqeal/seqmatrix.html`.

[DMS12a] M. Dénès, A. Mörtberg, and V. Siles. A Refinement Based Approach to Computational Algebra in Coq. In *Proceedings 3rd International Conference on Interactive Theorem Proving 2012 (ITP'12)*, volume 7406 of *Lecture Notes in Computer Science*, pages 83–98, 2012.

[DMS12b] M. Dénès, A. Mörtberg, and V. Siles. CoqEAL, the Coq Effective Algebra Library, 2012. `http://www-sop.inria.fr/members/Maxime.Denes/coqeal`.

[DR10] C. Domínguez and J. Rubio. Computing in Coq with Infinite Algebraic Data Structures. In *Proceedings 17th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning*

(Calculemus'10), volume 6167 of *Lectures Notes in Artificial Intelligence*, pages 204–218, 2010.

[DR11] C. Domínguez and J. Rubio. Effective Homology of Bicomplexes, formalized in Coq. *Theoretical Computer Science*, 412:962–970, 2011.

[DRSS98] X. Dousson, J. Rubio, F. Sergeraert, and Y. Siret. The Kenzo program. Institut Fourier, Grenoble, 1998. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`.

[EH10] H. Edelsbrunner and J. Harer. *Computational topology: An introduction*. American Mathematical Society, 2010.

[For] Formath: Formalisation of Mathematics. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath`.

[For98] R. Forman. Morse theory for cell complexes. *Advances in Mathematics*, 134:90–145, 1998.

[GDMRSP05] R. González-Díaz, B. Medrano, P. Real, and J. Sánchez-Peláez. Algebraic Topological Analysis of Time-Sequence of Digital Images. In *Proceedings 8th International Conference on Computer Algebra in Scientific Computing (CASC'05)*, volume 3718 of *Lecture Notes in Computer Science*, pages 208–219, 2005.

[GDR05] R. González-Díaz and P. Real. On the Cohomology of 3D Digital Images. *Discrete Applied Mathematics*, 147(2-3):245–263, 2005.

[GGMR09] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Proceedings 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342, 2009.

[GL02] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, volume 37, pages 235–246, 2002.

[GM10] G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[GMR+07] G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A Modular Formalisation of Finite Group Theory. Rapport de recherche RR-6156, INRIA, 2007.

[Gon08] G. Gonthier. Formal proof - The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[Gon11] G. Gonthier. Point-free, set-free concrete linear algebra. In *Proceedings 2nd Interactive Theorem Proving (ITP'11)*, volume 6898 of *Lectures Notes in Computer Science*, pages 103–118, 2011.

[Gug72] V. K. A. M. Gugenheim. On the chain complex of a fibration. *Illinois*, 16(3):398–414, 1972.

[GWZ02] H. Geuvers, F. Wiedijk, and J. Zwanenburg. A Constructive Proof of the Fundamental Theorem of Algebra without using the Rationals. In *Selected papers from the International Workshop on Types for Proofs and Programs*, TYPES '00, pages 96–111, 2002.

[Hal05a] T. Hales. A proof of the Kepler conjecture. *Annals of Mathematics*, 162:1065–1185, 2005.

[Hal05b] T. Hales. The Flyspeck Project fact sheet. Project description available at `http://code.google.com/p/flyspeck/`, 2005.

[Har09] J. Harrison. The HOL Light System Reference - Version 2.20. 2009. `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[HCMS] J. Heras, T. Coquand, A. Mörtberg, and V. Siles. Computing Persistent Homology within Coq/SSReflect. To appear in ACM Transactions on Computational Logic.

[HDM+12] J. Heras, M. Dénès, G. Mata, A. Mörtberg, M. Poza, and V. Siles. Towards a certified computation of homology groups for digital images. In *Proceedings 4th International Workshoph on Computational Topology in Image Context (CTIC'12)*, volume 7309 of *Lectures Notes in Computer Science*, pages 49–57, 2012.

[Her] D. Herington. The HUnit package. Technical report. `http://hackage.haskell.org/package/HUnit-1.2.4.2`.

[Her09] J. Heras. The *fKenzo* program. University of La Rioja, 2009. `http://www.unirioja.es/cu/joheras/fKenzo`.

[HK13] J. Heras and E. Komendantskaya. Statistical Proof-Patterns in Coq/SSReflect. 2013. `http://arxiv.org/abs/1301.6039`.

[HMPR11] J. Heras, G. Mata, M. Poza, and J. Rubio. Homological processing of biomedical digital images: automation and certification. In *17th International Conferences on Applications of Computer Algebra. Computer Algebra in Algebraic Topology and its applications session*, 2011.

[HPDR11] J. Heras, M. Poza, M. Dénès, and L. Rideau. Incidence simplicial matrices formalized in Coq/SSReflect. In *Proceedings 18th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning (Calculemus'11)*, volume 6824 of *Lectures Notes in Computer Science*, pages 30–44, 2011.

[HPR12] J. Heras, M. Poza, and J. Rubio. Verifying an algorithm computing discrete vector fields for digital imaging. In *Proceedings Conferences on Intelligence Computer Mathematics (CICM'12)*, volume 7362 of *Lectures Notes in Computer Science*, pages 215–229, 2012.

[Hut07] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[J+03] S. P. Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, 2003. `http://www.haskell.org`.

[KM] M. Kaufmann and J S. Moore. ACL2. `http://www.cs.utexas.edu/users/moore/acl2/`.

[Koz07] D. Kozlov. *Combinatorial Algebraic Topology*. Springer, 2007.

[KS11] R. Krebbers and B. Spitters. Computer certified efficient exact reals in Coq. In *Proceedings Conferences on Intelligent Computer Mathematics (CICM'11)*, volume 6824 of *Lecture Notes in Computer Science*, pages 90–106, 2011.

[Ler09] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[LLT04] T. Lewiner, H. Lopes, and G. Tavares. Applications of Forman's discrete Morse theory to topology visualization and mesh compression. *Transactions on Visualization and Computer Graphics*, 10(5):499–508, 2004.

[MÏ0] A. Mörtberg. Constructive algebra in functional programming and type theory. Mathematics, Algorithms and Proofs 2010, 2010. `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/PapersAndSlides`.

[M+12] M. Mrozek et al. Homological methods for extraction and analysis of linear features in multidimensional images. *Pattern Recognition*, 45(1):285–298, 2012.

[Mac63] S. MacLane. *Homology*. Springer, 1963.

[Mat12] Mathematical components team. Formalization of the Odd Order theorem. Technical report, 2012. `http://www.msr-inria.inria.fr/Projects/math-components`.

[Mau96] C.R.F. Maunder. *Algebraic Topology*. Dover, 1996.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[OS03] D. Orden and F. Santos. Asymptotically efficient triangulations of the d-cube. *Discrete and Computational Geometry*, 30(4):509–528, 2003.

[PDHR13] M. Poza, C. Domínguez, J. Heras, and J. Rubio. A certified reduction strategy for homological image processing. *Preprint*, 2013. `http://www.unirioja.es/cu/cedomin/crship/`.

[RR12] A. Romero and J. Rubio. Homotopy groups of suspended classifying spaces: an experimental approach. *To be published in Mathematics of Computation*, 2012. `http://www.ams.org/journals/mcom/0000-000-00/S0025-5718-2013-02680-4/home.html`.

[RS97] J. Rubio and F. Sergeraert. Constructive Algebraic Topology, Lecture Notes Summer School on Fundamental Algebraic Topology. Institut Fourier, Grenoble, 1997. http://www-fourier.ujf-grenoble.fr/~sergerar/Summer-School/.

[RS02] J. Rubio and F. Sergeraert. Constructive Algebraic Topology. *Bulletin des Sciences Mathématiques*, 126(5):389–412, 2002.

[RS10] A. Romero and F. Sergeraert. Discrete Vector Fields and Fundamental Algebraic Topology, 2010. http://arxiv.org/abs/1005.5685.

[RS12] A. Romero and F. Sergeraert. Homological Perturbation Theorem and Eilenberg-Zilber vector field. 2012. http://www-fourier.ujf-grenoble.fr/~sergerar/Talks/12-06-Zurich-1.pdf.

[Ser87] F. Sergeraert. Homologie effective. *Comptes Rendus des Séances de l'Academie des Sciences de Paris*, 304(11 and 12):279–282 and 319–321, 1987.

[Ser92] F. Sergeraert. Effective homology, a survey. Technical report, Institut Fourier, 1992. http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Survey.pdf.

[Ser94] F. Sergeraert. The computability problem in Algebraic Topology. *Advances in Mathematics*, 104(1):1–29, 1994.

[Shi62] W. Shih. Homologie des espaces fibrés. *Publications mathématiques de l'Institut des Hautes Études Scientifiques*, 13:1–88, 1962.

[tdt10] *Coq* development team. The Coq Proof Assistant, version 8.3, 2010.

[tdt12] *Coq* development team. The Coq Proof Assistant, version 8.4, Chapter 22 Extraction of programs in Objective Caml and Haskell, 2012. http://coq.inria.fr/refman/Reference-Manual027.html#toc149.

[Veb31] O. Veblen. *Analysis Situs*. AMS Coll. Publ., 1931.

[ZA02] D. Ziou and M. Allili. Generating Cubical Complexes from Image Data and Computation of the Euler number. *Pattern Recognition*, 35:2833–2839, 2002.

Verificación de algoritmos homológicos para estudiar imágenes biomédicas

María Poza López de Echazarreta

Memoria presentada para la
obtención del grado de Doctor

Directores: Dr. D. César Domínguez Pérez
Dr. D. Julio Rubio García

Universidad de La Rioja
Departamento de Matemáticas y Computación

Logroño, abril 2013

Este trabajo ha sido parcialmente subvencionado por el proyecto MTM2009-13842-C02-01 del Ministerio de Educación y Ciencia, y por el proyecto Europeo Formath nº 243847 del programa FET del 7º Programa Marco de la Comisión Europea.

Agradecimientos

A Álvaro
A mis padres y mi hermano

En primer lugar, me gustaría agradecer a mis directores Julio Rubio y César Domínguez el apoyo y dedicación que han depositado en mi en esta aventura. Gracias por dedicarme vuestro tiempo, estoy segura que sin vuestras sugerencias y correcciones esta tesis no hubiese sido posible. No puedo olvidarme de Jónathan Heras. Gracias, por tu ayuda y por tu apoyo incondicional tanto a nivel científico como personal. Sobre todo me gustaría remarcar tu entendimiento y paciencia conmigo. Tu tesis ha sido un punto de referencia para mi y él un ejemplo a seguir.

Quiero agradecer también al resto de compañeros del grupo por el ambiente tan fantástico que ha habido en este tiempo. Desde el principio hicieron lo posible porque me integrara y así comencé a jugar al pádel. Además, a mi compañera de despacho Clara Jiménez que nos hemos hecho compañía mutuamente sobre todo a lo largo del último año. Especialmente quiero nombrar a Gadea Mata por sus palabras de ánimo y por nuestras comidas y cafés al sol, e incluso cuando no había sol.

Por último, pero no por eso menos importante, me gustaría agradecer el interés y la paciencia de mis padres y hermano en la evolución de este trabajo porque no he sido la mejor compañía en muchas ocasiones a lo largo de este tiempo. A Álvaro, por estar siempre cuando le he necesitado, por su apoyo y por su gran

confianza en mí. A mis amigos que aunque no sabían muy bien a que dedicaba mi tiempo me preguntaban por la evolución de esta tesis.

Gracias a todos.

Índice General

Introducción

El cálculo científico es una herramienta excelente para ayudar a los investigadores en ciencias experimentales. Cuando se aplica a problemas biomédicos, la precisión y la fiabilidad de los cálculos son particularmente importantes. En consecuencia, la posibilidad de aumentar la confianza en el software científico por medio de herramientas para el razonamiento mecanizado de teoremas se convierte en un área interesante de investigación.

Los asistentes interactivos para la demostración de teoremas son herramientas diseñadas para ayudar a los investigadores en el desarrollo de pruebas formales. Estos sistemas requieren la cooperación de los seres humanos y las máquinas. Concretamente, el usuario se encarga de diseñar las pruebas, dar grandes pasos como es habitual en matemáticas y la máquina, con la ayuda del ser humano, y rellenar los huecos. Hay varios ejemplos de asistente de pruebas, por ejemplo, HolLight [Har09], Isabelle/HOL [NPW02], ACL2 [KM], y Coq [BC04, BGBP08]. Los asistentes para la demostración están lo suficientemente maduros para enfrentarse a problemas interesantes en el contexto de pruebas matemáticas y también para verificar la corrección tanto de software como de hardware. Algunos ejemplos sorprendentes son las formalizaciones del Teorema de los Cuatro Colores [Gon08], el Teorema Fundamental del Algebra [GWZ02], la conjetura de Kepler [Hal05a], la verificación de un compilador C [Ler09] o la formalización de la corrección de un microprocesador AMD [BKM96]. Algunos proyectos son realizados con el único propósito de probar algunos resultados relevantes como el caso del proyecto Flyspeck [Hal05b] para formalizar la prueba de Hales sobre la conjetura de Kepler [Hal05a].

En nuestro caso, nosotros usamos el asistente de demostración de teoremas

Coq que se basa en el Cálculo de Construcciones Inductivas [CH88]. Este sistema tiene una característica interesante que nos permite extraer programas desde pruebas constructivas. Además, hemos usado SSReflect [GM10], una extensión de Coq. El lenguaje de tácticas y la librería de SSReflect fueron inicialmente diseñadas para probar el Teorema de los Cuatro Colores. Después, SSReflect ha sido mejorado para atacar el Teorema de Feit-Thompson (conocido como el Teorema del Orden Impar) [Mat12].

En este trabajo usamos Coq con el objetivo de certificar algunos procedimientos para el procesamiento de imágenes. Las imágenes se toman al principio de nuestra investigación sobre cultivos de neuronas usando microscopios [C$^+$11]. La técnica que nosotros usamos para procesar estas imágenes está basada en cálculos de Topología Algebraica.

El cálculo en Topología Algebraica es un campo que está emergiendo y que atrae el interés de investigadores, tanto en aspectos teóricos como industriales (ver por ejemplo [EH10]). Aunque el interés está creciendo en los últimos años, la historia del cálculo en Topología Algebraica es larga (al menos, con respecto a los estándares en álgebra computacional y cálculo científico). Uno de las primeras teorías en este campo se llama *homología efectiva* [RS02], creada por F. Sergeraert, y que dio lugar al sistema *Kenzo* [DRSS98]. Kenzo se dedica a calcular los grupos de homología y de homotopía de espacios topológicos, y algunos de sus cálculos producen resultados desconocidos [Ser92] o incluso corrigen teoremas previamente publicados [RR12]. En consecuencia, los matemáticos deberían poder confiar en los resultados de Kenzo. Para reforzar esta confianza, varias contribuciones se han hecho aplicando métodos formales al estudio de Kenzo y sus algoritmos subyacentes (ver, entre otros, [ABR08, ABR10, AD09, AD10, DLR07, DR10, DR11]).

Este es el marco donde se sitúa la investigación presentada en esta memoria. Algunos algoritmos de Romero y Sergeraert [RS10] fueron implementados en Kenzo para ser aplicados a imágenes digitales. Aquí, empezamos la verificación de estos algoritmos en Coq, emulando los procesos Kenzo.

Las bases para el procesamiento homológico de imágenes digitales están basadas en la disciplina llamada *Topología Digital* [ADFQ03]. Las imágenes digitales deben interpretarse como espacios topológicos de una manera combinatoria. El método más elemental para establecer una conexión entre Topología General y Topología Combinatoria está basado en el uso de complejos simpliciales. La noción

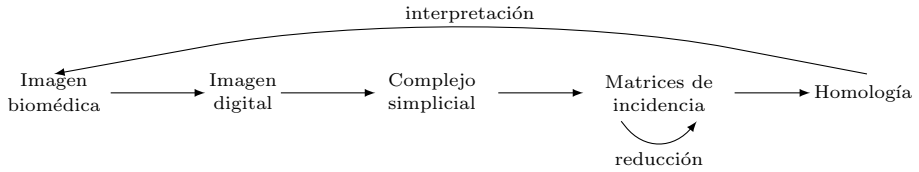


Figura 6.10: Cálculo homológico de una imagen digital

de espacio topológico es demasiada "abstracta" para transferirla a una máquina. Los complejos simpliciales proporcionan una descripción puramente combinatoria de espacios topológicos que admiten una triangulación. El cálculo de invariantes, tales como los grupos de homología, desde un complejo simplicial finito asociado con un espacio topológico es conocido y, por ejemplo en el caso de grupos de homología el algoritmo usa álgebra lineal [Veb31]. Luego, un investigador en topología algebraica puede identificar un espacio topológico compacto y triangulable (como una interpretación "continua" de una imagen digital) con un complejo simplicial finito, haciendo posible los cálculos. El papel de la Topología Algebraica en imágenes digitales es conocida (ver por ejemplo la serie de conferencias llamadas *Computational Topology in Image Context*)

El proceso estudiado en esta memoria está descrito en la Figura 6.10. Poniéndolo con palabras, después de un preprocesamiento de una imagen biomédica, se consigue una imagen monocromática; luego, desde los pixeles negros de dicha imagen se obtiene un complejo simplicial (por medio de un proceso de triangulación); seguidamente, desde el complejo simplicial, sus *matrices borde (o de incidencia)* son construidas, y finalmente, la *homología* puede ser calculada. Si trabajamos con coeficientes sobre un cuerpo (se conoce que es suficiente con considerar el cuerpo $\mathbb{Z}_2$ cuando trabajamos con imágenes digitales 2D y 3D) y si sólo estamos interesados en las *dimensiones* de los grupos de homología (como espacios vectoriales), es suficiente con tener un programa que calcule el rango de una matriz para llevar a cabo la tarea completa.

Esta arquitectura es particularizada en esta tesis con un problema real que apareció en una aplicación industrial y con el asistente de teoremas Coq como herramienta de programación y verificación. El problema biológico (concretamente, el conteo del número de sinapsis de una imagen de una neurona) puede ser identificado con el cálculo de un invariante topológico (el rango de un grupo de

homología). Luego, todos nuestros esfuerzos se concentran en calcular, de una manera certificada, tal invariante.

Ya que el tamaño de las imágenes biomédicas de la vida real es demasiado grande para tratarlas de una manera directa, proponemos una *estrategia de reducción* (ver el paso de reducción en la Figura 6.10), que nos permite trabajar con estructuras de datos más pequeñas, pero preservando todas sus propiedades homológicas. Para este objetivo, usamos las noción de *campo de vectores discreto* [For98], siguiendo la aproximación presentada por Romero y Sergeraert en [RS10].

Para verificar la corrección de estos procedimientos, ha sido necesario formalizar una gran cantidad de contenidos matemáticos. La pieza más significativa formalizada en este trabajo es el llamado *Lema Básico de Perturbación* (o abreviando, BPL). La prueba de este teorema ya ha sido implementada en el demostrador de teoremas Isabelle/HOL [ABR08]. La formalización del BPL presentada en esta memoria es mucho más corta y compacta que la presentada en [ABR08]. Hay dos razones para esta mejora de la prueba formal. La primera es que en este trabajo hemos seguido una prueba nueva y más corta del BPL (debido otra vez a Romero y Sergeraert [RS12]). La segunda razón es que hemos construido nuestra prueba con la potente librería SSReflect de Coq [GM10] (por el contrario, una gran parte de la infraestructura requerida tuvo que ser definida desde cero en [ABR08]).

A parte de la eficiencia en la escritura de pruebas, usar SSReflect también tiene otras consecuencias. Ya que SSReflect se diseñó para tratar sólo con *estructuras finitas*, la prueba del BPL presentada aquí es aplicada sólo sobre *grupos finitamente generados* (la prueba formalizada en [ABR08] no tiene esta restricción). Además, tratar con estructuras finitas, y dentro de la lógica constructiva de Coq, facilita la ejecutabilidad de las pruebas, y en consecuencia la generación de programas certificados (las mismas tareas en Isabelle/HOL plantean más dificultades; ver [ABR10]).

Para probar la corrección de los programas generados, debemos establecer y mantener una unión entre la imagen biomédica inicial y la estructura de datos final, que es más pequeña, donde los cálculos homológicos son llevados a cabo. Esto implica una gran cantidad de procesamiento, y no nos permite ejecutar todos los pasos *dentro de* Coq (el camino completo ha sido recorrido, pero sólo

para ejemplos de *juguete*). Luego hemos utilizado un lenguaje de programación, Haskell [Hut07] en nuestro caso, para integrar cálculo y deducción.

Haskell interviene en dos pasos distintos de nuestra metodología. En las etapas iniciales del desarrollo, los prototipos de los algoritmos Haskell son sistemáticamente testeados usando la herramienta QuickCheck [CH00]. Esto nos permite eliminar muchos errores pequeños y comunes, que podrían entorpecer el proceso de la prueba en Coq. En el paso final de cálculo, Haskell se usa como *oráculo* para Coq. Las partes más costosas del cálculo (en nuestro caso, un importante cuello de botella es calcular la matriz inversa) son delegados a programas Haskell; la corrección de los *resultados* obtenidos por Haskell es *probada* dentro de Coq.

Con esta técnica híbrida, logramos el objetivo de calcular, de una manera certificada, la homología de imágenes biomédicas provenientes de experimentos neurológicos.

La estructura de la memoria es la siguiente.

En el Capítulo 1 se presentan los preliminares de nuestra investigación. Incluye tanto aspectos matemáticos como de demostradores de teoremas. Además, se detallan las matemáticas que se formalizarán en el resto de la memoria.

El Capítulo 2 está dedicado a la implementación y la verificación del algoritmo de Romero-Sergeraert para calcular un campo de vectores discreto asociado con una imagen digital. Los resultados presentados en este capítulo han sido parcialmente publicados en el artículo [HPR12].

En los Capítulos 3 y 4 describimos respectivamente la construcción de la reducción algebraica definida por un campo de vectores discreto, y una prueba formal del BPL. Los resultados de estos dos capítulos son el tema del artículo [PDHR13].

El Capítulo 5 se trata la aplicación de los resultados previos al camino representado en la Figura 6.10. Concretamente, implementamos complejos de cadenas y matrices de incidencia (parcialmente publicados en [HPDR11]) y calculamos la homología dentro de Coq (ver nuestro artículo [HDM+12]). Luego, la estrategia de reducción certificada es integrada en este esquema general para mejorar la eficiencia.

En el Capítulo 6 se realizan algunos experimentos de cálculo (con Kenzo, con Haskell o en Coq), mostrando las fortalezas y debilidades de nuestro alcance.

La memoria acaba con una sección dedicada a conclusiones y trabajo futuro, y la bibliografía.

El lector interesado puede consultar el código presentado a lo largo de esta memoria en [For].

Resumen de los capítulos

Presentamos a continuación un breve resumen de cada uno de los capítulos de esta memoria.

1 Preliminares

En este capítulo presentamos el contexto y las herramientas que usaremos en el resto de la memoria. La primera sección está dedicada a introducir las nociones matemáticas empleadas en este trabajo. A parte de las definiciones básicas sobre Álgebra Homológica y Topología Simplicial, presentamos las relaciones tanto entre complejos simpliciales y complejos de cadenas como entre imágenes digitales y complejos simpliciales. Además se presentan resultados sobre el concepto de reducción y un marco algebraico de la Teoría Discreta de Morse que nos permite reducir información preservando las propiedades homológicas [For98].

Seguidamente, presentamos una breve introducción al sistema Coq y su librería SSReflect. Este sistema será usado para la formalización de los algoritmos y la verificación de los resultados matemáticos. Además, se muestra la metodología utilizada en nuestro trabajo. Dicha metodología puede resumirse en:

1) Implementar en Haskell.

2) Testear con QuickCheck.

3) Verificar con Coq/SSReflect.

2 Formalización de un algoritmo para calcular campos de vectores discretos

En este capítulo se presenta el algoritmo de Romero-Sergeraert para calcular un campo de vectores discreto admisible de una matriz, así como una implementación del mismo en Haskell. Para asegurarnos de que nuestros programas son correctos, aplicamos la metodología detallada en la Section 1.3, con el objetivo de comprobar que la salida de este algoritmo satisface las propiedades dadas en la definición de un campo de vectores discreto admisible. A continuación, testeamos nuestros programas con QuickCheck [CH00] y formalizamos el algoritmo con Coq [tdt10] y SSReflect [GM10]. Además presentamos un algoritmo no determinista en SSReflect para construir un campo de vectores discreto de una matriz. En consecuencia, este algoritmo no puede ser ejecutado pero su verificación es sencilla.

3 Reducción asociada a un campo de vectores discreto admisible y ordenado

Presentamos un método de reducción utilizando campos de vectores discretos, así como su implementación en Haskell y su formalización en Coq/SSReflect. Esto nos permite reducir el complejo de cadenas asociado a una imagen. La prueba de esta reducción está basada en el Lema Hexagonal.

Además, se prueba que una reducción entre dos complejos de cadenas preserva los números de Betti. Esto implica que podemos calcular los números de Betti de uno de los complejos de cadenas a partir del otro.

Finalmente, presentamos otro algoritmo para reducir complejos de cadenas basado en la noción de colapsos. Esta reducción podría ser usada como un preprocesamiento de la imagen con el objetivo de aplicar luego la reducción anterior sobre el complejo de cadenas reducido.

4 Formalización del Lema Básico de Perturbación

Nos centramos en presentar la prueba matemática del Lema Básico de Perturbación y su formalización en SSReflect. Este lema es esencial en la teoría de Álgebra Homológica. La prueba presentada está basada en dos resultados: el teorema de Descomposición, que construye una descomposición de un complejo de cadenas dada una reducción de dicho complejo de cadenas y la Generalización del Lema Hexagonal.

Además, usamos este lema para un caso particular que viene de imágenes digitales dos dimensionales. El Lema Básico de Perturbación nos servirá para reducir el complejo de cadenas asociado a una imagen digital.

5 Procesamiento homológico de imágenes digitales

Nos centramos en formalizar el proceso de cálculo de la homología de una imagen digital que detallamos a continuación. Por medio de un procedimiento de triangulación, construimos el complejo simplicial de una imagen digital monocromática. Después, construimos las matrices de incidencia asociadas y finalmente, las dimensiones de los grupos de homología del complejo de cadenas definido por estas matrices pueden ser calculadas. Para dicho cálculo es suficiente con obtener el rango de las matrices de incidencia asociadas con el complejo de cadenas.

Se presenta una versión abstracta de esta formalización usando las librerías de SSReflect. Pero esta formalización no es útil para nuestro objetivo: poder calcular la dimensión de los grupos de homología. Para resolver este problema, se presenta una implementación ejecutable de este proceso. Para verificar este desarrollo nos hemos centrado en probar que las definiciones dadas en ambas versiones son equivalentes. Finalmente, podemos calcular la dimensión de los grupos de homología de una imagen dentro de Coq. Esto también nos permite calcular la dimensión de la homología del complejo de cadenas reducido usando un campo de vectores discreto de la imagen.

6 Experimentación

Este capítulo está dividido en tres secciones. La primera, habla sobre el testeo que se ha desarrollado sobre nuestras implementaciones en Haskell, con el objetivo de limpiar y eliminar los posibles errores y, de este modo, reducir el tiempo invertido en la verificación. Explicamos un testeo manual, un testeo "automatizado" usando el sistema Kenzo, y finalmente, un testeo con la herramienta QuickCheck. La segunda sección trata sobre el análisis de rendimiento en Haskell para entender el comportamiento de nuestros programas y ver qué funciones son las que consumen más tiempo y de este modo, realizar mejoras en la implementación. Finalmente, presentamos unos resultados sobre el tiempo dedicado al cálculo de la homología de una imagen distinguiendo entre los sistemas Haskell y SSReflect, y aplicando o no la reducción obtenida gracias a los campos de vectores discretos.

Además tratamos una aplicación concreta que es el conteo de sinapsis de una neurona que puede ser resuelto calculando H_0. El tamaño de estas imágenes es considerable, luego el uso de los campos de vectores discretos es necesario. Finalmente, implementamos otros algoritmos más eficientes para tratar este problema concreto.

Conclusiones y trabajo futuro

En esta memoria hemos presentado una investigación que estudia el uso de cálculo certificado para analizar los grupos de homología asociados con imágenes digitales provenientes de un problema biomédico. Las principales contribuciones que nos han permitido alcanzar este objetivo han sido las siguientes.

- La implementación en Coq/SSReflect del algoritmo de Romero-Sergeraert [RS10] para el cálculo de un campo de vectores discreto y admisible de una imagen digital.

- La formalización completa en Coq/SSReflect del teorema conocido como el Lema Básico de Perturbación (BPL).

- Dos formalizaciones del Teorema de Reducción de Campos de Vectores para matrices. Una de ellas es probada usando el BPL y la otra aplicando el Lema Hexagonal [RS10].

- Una discusión de los diferentes métodos para superar los problemas de eficiencia que aparecen al ejecutar programas dentro de asistentes para la demostración. En particular, el lenguaje de programación Haskell ha sido utilizado de dos formas distintas: primero, para modelar algoritmos que son luego implementados en Coq y, segundo, como un oráculo para producir resultados cuyas propiedades son verificadas en el asistente para la demostración.

- Un programa verificado para calcular los grupos de homología de un complejo simplicial obtenido a partir de una imagen digital.
- Como una consecuencia de todas las otras contribuciones, una aplicación de la Topología Algebraica para estudiar imágenes biomédicas de una manera fiable. Nuestra metodología asegura que los cálculos homológicos finales son correctos.

Observando la memoria al completo, está claro que, incluso si nuestro énfasis fue en aplicaciones biomédicas, gran parte del trabajo ha sido dedicado a la formalización de matemáticas y a la verificación de programas. La razón es que, aunque lo hemos construido sobre fundamentos muy sólidos (homología efectiva [RS10] desde la parte algorítmica y SSReflect [GM10] como base para la prueba de teoremas), necesitabamos reproducir dentro de Coq/SSReflect una parte de Topología Algebraica Computacional. Como consecuencia, nos centramos en los aspectos de formalización en lugar de en los de eficiencia en el cálculo homológico de imágenes digitales. Por ello, nuestra investigación debería ser considerada como una prueba de conceptos: el procesamiento homológico de una imagen puede ser implementado y verificado usando demostradores de teoremas. Nuestros resultados son más bien un punto de partida, en vez de un problema cerrado.

Como trabajo futuro, varios problemas quedan abiertos. El más evidente, después de nuestra discusión previa, es mejorar la eficiencia de nuestros programas. Esto puede ser llevado a cabo desde tres niveles distintos. El primero consistiría en utilizar mejores algoritmos para el cálculo de, por ejemplo, campos de vectores discretos, inversas de matrices, etc.

El segundo, aunque no es independiente del anterior, sería la implementación de estructuras de datos y representaciones más eficientes. Una primera idea es trabajar con *complejos cúbicos* [ZA02] en vez de complejos simpliciales. También se podría considerar el uso de refinamientos de estructuras de datos [DMS12a], intentando traducir automáticamente pruebas desde una representación (abstracta) a otras (eficientes).

Como tercer aspecto podría ser interesante mejorar los entornos de ejecución en asistentes para la demostración de teoremas.

Con respecto a las aplicaciones, en el área de la Topología Algebraica Computacional, nuestros resultados podrían ser extendidos a la homología con coeficien-

tes en los enteros. Esta generalización podría hacer posible la verificación de otros resultados de Kenzo, como los presentados en [RR12].

Otra línea de investigación es aplicar nuestra metodología y técnicas a otros problemas relaciones con el procesamiento homológico de imágenes biomédicas. El mejor candidato es la *homología persistente*, que ya ha sido formalizada (ver [HCMS]). Concretamente, podría ser aplicado en stacks de neuronas para eliminar el ruido de las imágenes y ayudar a la detección de las dendritas (las ramas de la neurona). El proyecto sería estudiar si nuestra estrategia de reducción puede ser también beneficiosa en este nuevo contexto homológico. Además, podemos centrarnos en reconocer la estructura de una neurona; un problema que parece involucrar el cálculo de los grupos de homología en dimensión 1, como puede verse en [M$^+$12], una cuestión que podría ser abordada con nuestras herramientas.