[Version 2.1 = Introduction + Abstract machine + Various sorts of effective $\mathbb{Z}$-modules + The homological problem for a differential module + Solving extension problems + Exact couples + Effective exact couples + The fundamental theorem]

1 Introduction.

We present in these notes a totally new method to make *effective* standard homological algebra. Among other methods to attack the same problem, it is certainly the closest of "traditional" homological algebra. It is well known (?) "standard" homological algebra is *not* effective. Except in *specific* situations, the exact and spectral sequences which are proposed to solve homological problems *are not* algorithms computing the desired homology groups: they can be used *only* in "favourable" circumstances. This restriction "only" is rather unfair, if we consider the accumulation of wonderful results already obtained since Serre's thesis from these *non-algorithmic* methods. But anyway, the fact that the "standard" exact and spectral sequences *are not* algorithms is an *essential gap* in our domain.

1.1 Previous works.

Several methods have already been proposed to fill in this gap. Rolf Schön published a nice memoir [12] explaining how his organization based on inductive limits allows him to solve the algorithmic problem of homological algebra. It is really a pity this method has never been concretely used, except in a non-published work by Alain Clément [6].

Maybe the most promising method would be based on *operads*. Many sorts of operads are available, each one adapted to some subdomain, therefore which could be used for some particular application without being troubled by irrelevant details. The most striking theoretical result was obtained by Michael Mandell [8], proving every homotopy type of finite type can be defined by a chain complex of finite type provided with an E_∞-module structure. Who will be able to attack the problem of a concrete implementation of these operadic methods?

A more elementary method based on the so-called *basic perturbation lemma* was previously developed, see [13, 9, 10, 11]. It was proved most common exact and spectral sequences can then be transformed into algorithms. The method has been also concretely programmed, see [7], allowing us to reach some homology and homotopy groups so far unknown, raising also interesting new problems in theoretical and concrete computer science, because of the high level of functional programming which is necessary.

1.2 The constructive philosophy.

We present here a totally new method, with a larger scope, based on two ideas:

- The notion of *solution of the homological problem* (SHP) for a chain complex.

- The notion of *effective* exact couple.

Most mathematicians think solving a homological problem consists in computing some homology groups. It is interesting, but it is also a *partial* result. Let C_* be some chain complex and let us consider the possible result: $H_6(C_*) = \mathbb{Z}/12$. What is the exact meaning? This "equality" in fact means *there exists* an isomorphism $H_6(C_*) \cong \mathbb{Z}/12$. But the constructivists know it is always better in such a situation to *explicitly construct* such an isomorphism. This is rarely done in homological algebra, it is a lacuna and the unavoidable penalty is soon pending: for the *next* calculations using this homology group, not *explicitly* knowing this isomorphism often prevents from solving the frequent ambiguities raised by exact and spectral sequences coming from example from extension problems or from mysterious differentials.

In principle, extension problems could be solved by the Bockstein-Browder spectral sequence, but the usual presentation of it is not an algorithm either, so that the extension problem often remains in front of us without any available method.

Consider the differential equation $y'' + y' + y = 0$ for $y : \mathbb{R} \to \mathbb{R}$. Please solve this equation! A possible answer could be: the "solution is $\mathbb{R}^2$". It is true the set of solutions is *isomorphic* to $\mathbb{R}^2$, but most users of differential equations would not be very happy with such a "solution": they would prefer to know an *explicit* isomorphism between $\mathbb{R}^2$ and the set of solutions.

1.3 Homological problem.

Using the very spirit of constructive logic, we precisely define what the *solution of the homological problem* (SHP) for some chain complex is. It is a collection of algorithms ready to answer the various questions of homological nature which can be asked for this chain complex *and* its elements, in particular the cycles. For example: "Please what is the homology class of this cycle z"; "It is null!"; "Ah! Then could you give me a preimage for the boundary operator?"...

We will see that the extension problems can be easily solved for the usual exact sequences when *complete* SHPs of the relevant chain complexes are known.

The previous work using the basic perturbation lemma shows such SHPs are often reachable, theoretically and concretely as well. The results of the present paper significantly extend the scope of this observation.

1.4 Effective exact couples.

The same technique can be applied to many domains, in particular to the exact couples. An exact couple is a specific triangle diagram where some exactness properties must be satisfied. If $A \overset{g}{\leftarrow} B \overset{f}{\leftarrow} C$ is a sequence of two morphisms, exactness in B is usually translated by $\ker g = \operatorname{im} f$. But this is not enough: if $x \in B$ satisfies $gx = 0$, this means *there exists* a $y \in C$ satisfying $fy = x$.

Most often in homological algebra, when such a property is stated, this existence property is not constructive. Making it constructive consists in producing an algorithm $\ker g \xrightarrow{\sigma} C$ satisfying $f\sigma = \mathrm{id}_{\ker g}$. It is then not so easy to organize the environment, for such a section σ cannot be in general a module morphism.

Working with this idea in mind, we design the notion of *effective exact couple*. This notion has a striking advantage: a general algorithm can take as input an effective exact couple and return as output the derived exact couple, also organized as an effective exact couple. This algorithm can therefore be trivially iterated, and if a spectral sequence is associated to this exact couple, this spectral sequence is calculable, in particular *all* its differentials $d^r_{p,q}$ in the bigraded case.

This requires subtle methods of coding for the modules of the exact couple called usually D, occuring at the upper side of the triangle diagram. This coding is very *weak*, in particular the isomorphism class is unreachable, the nullity of an element of this coding is in general undecidable: such a module is not even an *effective* setoid! Yet this coding is essential when computing the derived exact couple!

Finally, combining effective exact couples with the method explained in Section 5, the often difficult extension problem to reconstruct at abutment the limit homology group H_{p+q} from the $E^\infty_{p,q}$ is solved when this makes sense.

1.5 Concrete Implementation.

The concrete implementation work of our method is just starting and will certainly need several years.

When we started to design the Kenzo program [7] to implement the ideas of our previous version of *effective homology* resting on the basic perturbation lemma, a general scepticism was observed in our community. We remember a high level colleague, yet having a good concrete experience in computer science, telling us: “You do not imagine you can implement simplicial groups in Lisp?” At this time, almost all our papers proposed for publication were rejected. The simplicial groups *are* today simply and smartly implemented in the Kenzo program [7], and experience shows the difficulty is not at all in the simplicial groups, it is in the terrible *effective* version of the *simplest* Eilenberg-Zilber theorem, which should raise more interest than observed today.

You could think our Fundamental Theorem (see Section 8) cannot be reasonably implemented? Please cautiously wait for the continuation of the story.

2 Abstract machine.

2.1 What is a machine?

As explained in the introduction, a mathematical machine model is indispensable, modelling the ordinary computers in a mathematical framework. According to the standard computability theory, see for example [1], all the usual machine models are equivalent and the technical details are irrelevant. The quick presentation given in this section is elementary and sufficient for our purpose. We choose a simplified version of the Lisp machine, see [2] for a detailed description if wished.

A universe $\mathcal{U}$, some countable set, is defined, the set of all the *machine objects*, such as integers, symbols, character strings, lists or arrays of such objects, and so on. The *evaluator* ε of our machine is a function $\varepsilon : \mathcal{U} \to \mathcal{U} \amalg \{\infty\}$; think an arbitrary object $\omega \in \mathcal{U}$ can after all be considered as a *program*; when launching such a program, most often you wait for some result (output), described in our model by the value $\rho = \varepsilon(\omega) \in \mathcal{U}$ of the execution of the program ω; if so, the program ω *terminates* and *returns* the result ρ.

Think of this machine as a Lisp *interpreter*. Such an interpreter displays on the screen a character string, the *prompt*, meaning the interpreter is awaiting an input ω. The input is entered by the user from the keyboard according to some conventions; for example typing `(+ 3 2)` describes the input ω as the *list* made of three components, the *symbol* `+` and the integers `3` and `2`. The Lisp machine is so designed that in this case the program terminates and returns $\varepsilon($`(+ 3 2)`$) =$ `5` probably in less than a microsecond. On the author's Lisp machine, this is displayed:

```
> (+ 3 2) ✠
5
```

The initial `>` is the Lisp prompt. The input `(+ 3 2)` is entered by the user. The Maltese cross ✠ is in fact *not* displayed on the screen; it is added here just to help the reader to identify the end of the input, automatically detected by the Lisp machine from the input syntax. Then Lisp computes and displays the output.

2.2 About equality in $\mathcal{U}$.

A technical point is important. The readers experienced with programming know the *equality problem* is a hard one, rarely appropriately processed by the programming languages, except Lisp and its derived languages. Typically, it is not obvious to decide whether two *copies* of the float number 4.56 stored at different addresses are *equal or not*. Experience shows the answer must be let to the programmer, who has the responsability to take the good decision, not so easy. Lisp example:

```
> (eq 4.56 4.56) ✠
NIL
> (eql 4.56 4.56) ✠
T
```

The `eq` function requires for equality the objects are installed at the same address, while `eql` does a *numerical* comparison between the "abstract" mathematical objects *represented* by the objects to be compared. It is not sensible to decide once for all whether two true twins are equal or not, it depends on the point of view.

Fortunately, this difficult matter can be ignored in this text, a fact which is summarized in the next statement. The readers interested by this problem must study the notion of *setoid*, see for example [15] for a starting point.

Restriction 1 — *We consider in this text the universe $\mathcal{U}$ as a "set" unambiguously defined: two objects are equal if and only if they are "the same", without any further explanation.*

2.3 Programs which do not terminate.

A program is most often designed to *terminate*, the runtime can be a microsecond, a few seconds, a few days, a few centuries... But sometimes the program never terminates, which is *mathematically* modelled as the relation $\varepsilon(\omega) = \infty$. In general the user is not *informed* of this fact, and he could hope that waiting a little more will finally give some output. For example the Lisp input:

```
> (progn
    (setf i 0)
    (loop (setf i (+ i 1)))) ✠
? ? ? ? ?
```

is a program initializing the variable `i` to `0` and then indefinitely increasing this value by `1`; such a program does not terminate. It is easy to *prove* this simple program does not terminate.

But Post's theorem, an avatar of Gödel's famous incompleteness theorem, proves (!) there exists a non-empty subset $\mathcal{T} \subset \mathcal{U}$ satisfying the amazing following property: for every $\omega \in \mathcal{T}$, the evaluation $\varepsilon(\omega)$ *does not* terminate but unfortunately there *does not* exist any proof of this fact: such programs will remain definitively "mysterious": experience seems to show such a program does not terminate, but maybe, waiting a little longer, who knows... Of course the *existence* of this subset $\mathcal{T}$ is not at all constructive: any element of it will *never* be identified as such. A program ζ running the roots z_n of Riemann's zeta function satisfying $0 < \mathrm{Re} < 1$ and $\mathrm{Im} > 0$, looking for a counter-example to Riemann's conjecture, *maybe* is an element of $\mathcal{T}$; which would then "imply" Riemann's conjecture is "true" but unprovable...

2.4 Algorithms.

An *algorithm* is a triple $\phi = (S, T, f)$ satisfying the properties:

1. The *source* component S is some subset $S \subset \mathcal{U}$, the set of allowed *inputs* for our algorithm ϕ.

2. The *target* component T is some subset $T \subset \mathcal{U}$: an object (output) returned by our algorithm ϕ is guaranteed being an element of T.
3. The *functional* component f is a *functional object.* This object can be combined with some allowed input $\omega \in S$, this process producing a pair $[f, \omega] \in \mathcal{U}$, and the evaluator working on this pair will terminate and will return an output $\varepsilon([f, \omega]) = \rho \in T \subset \mathcal{U}$.

For example the symbol '`+`' is a functional object for a Lisp machine, and the input in the addition example of Section 2.1 is the list $\omega =$ `(3 2)`; it happens in the Lisp organization that the *pair* [`+`, `(3 2)`] is in fact the same object as the *list* `(+ 3 2)`. The corresponding output is then the integer `5`. The corresponding algorithm is the triple $(\mathbb{Z} \times \mathbb{Z}, \mathbb{Z}, \texttt{+})$ where $\mathbb{Z} \times \mathbb{Z}$ denotes here the set of lists of two integers.

We most often simply write $\varepsilon([f, \omega]) =: f(\omega)$ and also $f : S \to T$; also, by abuse of language, the functional object f itself is often named as an algorithm, the source set S and the target set T being maybe implied by the context.

The relation $\varepsilon([f, \omega]) =: f(\omega)$ hides one of the deepest theorems of mathematics. The organization of our machine through a *unique* evaluator ε is an avatar of the Gödel-Turing theorem about the *constructive* existence of a universal machine, the very source of the notion of *universal* computer, so amazing. The laptop used to prepare this text, putting every black pixel at the right position, is as well able to decode a mp3 version of a Beethoven piano sonata, to compute thousands of digits of $\zeta(-1/3)$; and it can also implement effective versions of spectral sequences; all these sophisticated activities with the same small microprocessor are possible thanks to the Gödel-Turing theorem.

The source and the target of an algorithm are not elements but subsets of $\mathcal{U}$: the relation $S \in \mathcal{U}$ does not make sense; on the contrary, the relation $S \subset \mathcal{U}$ makes sense and is required. Yet it is often interesting to consider some *subsets* of $\mathcal{U}$ also as machine objects, at least when it is possible, through the notion of *data type*, in short *type*.

2.5 Types[1].

Two *logical objects* $\bot, \top \in \mathcal{U}$ are defined, modelling the usual logigal values *false* and *true*, and the set $\mathbb{B} = \{\bot, \top\} \subset \mathcal{U}$ is the (sub)set of the Boolean objects.

A *type* is described thanks to an algorithm $\tau : \mathcal{U} \to \mathbb{B}$; one says such a τ is a *universal predicate.* The associated type is the set of objects $S_\tau = \{\omega \in \mathcal{U} \ \underline{\textbf{st}}\ \tau(\omega) = \top\}$. In other words the subset S_τ is described through a characteristic function which is an *algorithm*, the source being the whole universe $\mathcal{U}$ and the target the set $\mathbb{B}$ of booleans. The abusive identification $S_\tau \sim \tau$ allows a user to think of the *subset* $S_\tau \subset \mathcal{U}$ as the *element* $\tau \in \mathcal{U}$.

[1]Please note our organization of types through characteristic functions everywhere defined *does not* match the usual notion of type of the computer scientists; we could as well use the latter but this would lead to technicalities out of scope here.

For example the Lisp expression:

```
(lambda (object)
  (and (integerp object)
       (evenp object)))
```

is an algorithm $\mathcal{U} \rightarrow \mathbb{B}$ describing the type of even integers. Illustration:

```
> ((lambda (object)
      (and (integerp object)
           (evenp object)))
   4) ✠
T
> ((lambda (object)
      (and (integerp object)
           (evenp object)))
   5) ✠
NIL
> ((lambda (object)
      (and (integerp object)
           (evenp object)))
   "an arbitrary character string") ✠
NIL
```

Read `T` $= \top$ and `NIL` $= \bot$. Our algorithm first examines whether the argument object is an integer, and *then*, *if* the answer is positive, if it is even. Any object $\omega \in \mathcal{U}$ is a legal argument, for example a character string. Or even the type itself[2]:

```
> ((lambda (object)
      (and (integerp object)
           (evenp object)))
     (lambda (object)
        (and (integerp object)
             (evenp object)))) ✠
NIL
```

Do not forget the universe $\mathcal{U}$ is countable; the set of functional objects, a subset of $\mathcal{U}$, is countable too, while the set of $\mathcal{U}$-subsets is not countable. In other words, most of the $\mathcal{U}$-subsets *are not* types.

This notion of *type* naturally came from the notion of *algorithm*. If an algorithm $\phi = (S, T, f)$ is defined, it is advised, in particular for debugging, before launching the calculation of $f(\omega)$, to *verify* whether the argument ω really is an element of S. Such a verification can be done only if S is a type. But this is not always required. In particular, when functional programming is used, that is, when the input ω and the output $f(\omega)$ are also functional objects, then it is in general not possible to do such a verification. Even in such a case, frequent, we nevertheless continue to call $\phi = (S, T, f)$ an algorithm.

[2] The functions capable of working on themselves are important in logic, they lead to the Gödel-Church-Turing-Post series of negative theorems.

3 Various sorts of effective $\mathbb{Z}$-modules.

The organization allowing us to make *constructive* the notions of *exact couple* and *spectral sequence* rests on *very different* implementations for the $\mathbb{Z}$-modules we must work with. The case of modules of finite type is standard, nothing new; the case of modules *not of finite type* is on the contrary subtle and amazing. Which is presented here is also deeply different from previous solutions for effective homology due to the same author and other collaborators, described for example at [9, 10].

Before processing $\mathbb{Z}$-modules, let us consider the case of the *sets*, where the difference between *effective* and *locally effective* sets is already there.

3.1 Effective sets.

Definition 2 — An *effective* set is a list of *different* machine objects.

A *list* in a computational environment is always a *finite* list. To be distinguished from an (infinite) *sequence* defined by an algorithm $\sigma : \mathbb{N} \to \mathcal{U}$ modelling the mathematical sequence $(\sigma(n))_{n\in\mathbb{N}} = (\sigma(0), \sigma(1), \ldots)$.

An exemple of effective set is $A =$ `(1 2 3 4)` $= \{1, 2, 3, 4\}$. It will be convenient not to hesitate to mix traditional notations for "mathematical" objects with the *representations* of these objects as machine objects, when such a coding is defined. We say the mathematical object $\{1, 2, 3, 4\}$ is *represented* by the machine object `(1 2 3 4)`; it is also represented by the *different* machine object `(2 1 3 4)`.

Conversely, we say the machine object `(1 2 3 4)` *codes* the mathematical object $\{1, 2, 3, 4\}$; the logicians know it would be better to speak of the mathematical object *defined by* the term $\{1, 2, 3, 4\}$, a term *different* from the term $\{2, 1, 3, 4\}$, but both terms define the same object; more precisely both terms are *equal*, which is *formally* written $\{1, 2, 3, 4\} = \{2, 1, 3, 4\}$.

An effective set is defined by a *finite* enumeration. In the traditional terminology, it is a *finite extensional set.* We can reasonably consider we *globally* know such a set.

The equality relation being defined unambiguously between machine objects, it is legitimate to require the elements of a set are different from each other.

3.2 Locally effective sets.

Definition 3 — A *locally effective set* S is a *type* $S \in \mathcal{U}$. A functional object $\sigma : \mathcal{U} \to \mathbb{B}$ which is a characteristic function for S is called a *membership function* for S.

No difference between a type and a locally effective set. Speaking of a locally effective set only means the environment is rather mathematically oriented, nothing more.

A locally effective set is an *intensionally* defined subset of $\mathcal{U}$, but intensionally defined by a *type*, that is, an algorithm $\tau : \mathcal{U} \to \mathbb{B}$. The standard cardinality argument shows "most" subsets of $\mathcal{U}$ cannot be defined as locally effective sets.

Many different membership functions are valid for the same locally effective set, that is, for the same type. For many *different* programs have the *same* behaviour with respect to the evaluator ε.

3.3 About the notion of *locally* defined object.

Why the qualifier "locally" assigned to the adjective *effective* when we speak of a locally effective set? It is a subtle but important point. The information available through a type can be applied to any machine object, deciding if this object is an element of the type or not. But in general *no global information* can be deduced from a type. For example it is not hard to define as a machine object the type G (Goldbach) of pairs of twin prime numbers: a simple program can examine if some object is a list, if the length of this list is 2, if both elements are natural numbers, if they are odd prime numbers a and b, and if their difference $b - a = 2$. You may be interested by the cardinality of G, that is, by the Goldbach conjecture, but no known process is able when this text is written to determine this cardinality, a *global* information.

In the same spirit, consider the sequence (z_n) made of the zeroes of the Riemann zeta function in the strip $0 < \mathrm{Re} < 1$ and $\mathrm{Im} > 0$ of the complex plane. Then the set of the integers n corresponding to counter-examples z_n of Riemann's conjecture *is* a type, for an algorithm can determine, given $n \in \mathbb{N}$, whether $\mathrm{Re}\, z_n = 1/2$ or not. The "*local*" information "is $\mathrm{Re}\, z_n = 1/2$?" is reachable, decidable, but the global information "is this type void" is (currently) not.

You see the right interpretation of our qualifier "locally" must be done by opposition to the opposite qualifier "globally". No topology at all in this matter. A more precise terminology would consist in speaking of *element-wise* effective sets, but it is a little heavy and we prefer the qualifier "locally".

3.4 $\mathbb{Z}$-modules of finite type.

Definition 4 — An *effective $\mathbb{Z}$-module* is a finite list $(d_1, \ldots, d_n)$ of integers $d_i \geq 0$ satisfying $d_i \neq 1$ and the divisibility condition d_{i-1} divides d_i for $1 < i \leq n$.

For example the machine object `(2 6 0 0)` (that is, (2,6,0,0) in mathematical notation) is an effective $\mathbb{Z}$-module, commonly denoted by $\mathbb{Z}/2 \oplus \mathbb{Z}/6 \oplus \mathbb{Z} \oplus \mathbb{Z}$. The usual structure theorem for the $\mathbb{Z}$-modules of finite type implies this notion of effective $\mathbb{Z}$-module exactly corresponds to the notion of *isomorphism class* of $\mathbb{Z}$-module of finite type.

Definition 5 — An element of an effective module $(d_1, \ldots, d_n)$ is an arbitrary integer list of the same length.

For example `(3 4 5 6)` $\in$ `(2 6 0 0)` and it is the same element as `(1 4 5 6)`. Think that, if (g_1, g_2, g_3, g_4) is the list of the canonical generators of `(2 6 0 0)`, the element represented by the list `(3 4 5 6)` is $3g_1 + 4g_2 + 5g_5 + 6g_4 = g_1 + 4g_2 + 5g_5 + 6g_4$. You could systematically prefer the *canonical* representation of an element of $(d_1, \ldots, d_n)$, the representation $(a_1, \ldots, a_n)$ which satisfies the extra condition $0 \leq a_i < d_i$ when $d_i > 0$, but this is just a matter of taste, and sometime of program efficiency.

Morphisms between effective $\mathbb{Z}$-modules are represented by integer matrices, the notion of canonical representation making sense again. Elementary algorithms compute the kernel and the image of such a morphism.

Definition 6 — An *effective differential* $\mathbb{Z}$*-module* is a pair (C, d) where C is an effective $\mathbb{Z}$-module and $d : C \to C$ is a morphism satisfying the differential condition $d^2 = 0$.

As usual, $Z(C, d) := \ker d$, $B(C, d) := \operatorname{im} d$, the condition $d^2 = 0$ is equivalent to the relation $B(C, d) \subset Z(C, d)$, and the corresponding homology group is $H(Z, d) := Z(C, d)/B(C, d)$.

Proposition 7 — *A general algorithm* $H_{\underline{\text{ecc}}}$ *computes:*

- **Input:** *An effective differential* $\mathbb{Z}$*-module* (C, d).
- **Output:** *The homology group* $H(C, d)$. ♣

3.5 Locally effective $\mathbb{Z}$-modules.

It is sensible to say the *effective* $\mathbb{Z}$-modules are *globally* known, in particular they are defined through their isomorphism class, the most important global information. Such a module is of finite type, and the numerous modules of homological algebra which *do not* satisfy this finiteness condition cannot be processed in this way. These modules can most often be *locally* implemented in *two very different ways*, depending on the status of the *equality relation.*

Definition 8 — *A locally effective* $\mathbb{Z}$*-module* M *is a 4-tuple* $M = (\tau, +, 0, -)$ *where:*

- *The first component* τ *is a type* $\tau : \mathcal{U} \to \mathbb{B}$*, the type of the* M*-elements: a machine object* $\omega \in \mathcal{U}$ *is an element of* M *if and only if* $\tau(\omega) = \top$*. In other words,* $M = S_\tau$*.*
- *The second component* $+$ *is an algorithm* $+ : M \times M \to M$ *implementing the addition of* M*-elements.*
- *The third component 0 is the neutral element of* M*.*
- *The fourth component* $-$ *is an algorithm* $- : M \to M$ *implementing the opposite operator.*

It is clearly a "local" definition: any calculation starting from particular elements of M can be processed, but in general no global information can be deduced from this implementation alone.

Let us recall we have assumed, cf. Restriction 1, the equality relation between arbitrary machine objects is well defined. In particular the comparison of an M-element with the neutral element 0 is decidable.

No global information is in general available for such a module but this does not prevent from defining many operators taking these modules as input. A typical example follows.

Proposition 9 — *A general algorithm* <u>sum</u> *can be written down:*

- **Input:** *A pair (M_1, M_2) of locally effective $\mathbb{Z}$-modules.*
- **Output:** *An implementation $(\tau_M, +_M, 0_M, -_M)$ of the sum (or product) module $M = M_1 \oplus M_2$.*

♣ The input is made of the respective implementations $(\tau_{M_1}, +_{M_1}, 0_{M_1}, -_{M_1})$ and $(\tau_{M_2}, +_{M_2}, 0_{M_2}, -_{M_2})$. We may decide an object $\omega \in M$ is a *list* of two elements (ω_1, ω_2), with $\omega_1 \in M_1$ and $\omega_2 \in M_2$. A type τ for the M-elements is therefore the algorithm which examines whether the argument object ω is a list of two elements, and then, *using the provided types* τ_1 and τ_2, examines also if the components of ω really are elements of M_1 and M_2.

Functional programming, a non-trivial computing technique, allows the user to design a *general* algorithm $(\tau_1, \tau_2) \mapsto \tau$, using as inputs two functional objects, returning an object which is also a functional object.

In the same way the desired algorithm $+_M$ can be computationnally deduced from $+_{M_1}$ and $+_{M_2}$ by a general algorithm and the same for the opposite operator $-_M$. ♣

Such general algorithms constructing a package of algorithms from some input mainly made also of algorithms are more easily implemented with the programming languages explicitly oriented toward *functional programming*, typically Lisp and its derived languages, Haskell, and many others. Alonzo Church invented in the thirties the first functional programming language, the λ-calculus, when the computers did not yet exist, except the Turing "machine" designed also in the thirties.

3.6 Locally effective subquotients.

The process described in the previous section looks so simple, at least when functional programming is understood, that you could think all the standard constructions for the modules used in textbooks of homological algebra can be implemented as easily. This is a strong error: the *quotient* operator cannot in general be implemented in this framework.

The problem is the following. Let us assume two locally effective modules M_1 and M_2 are implemented, and also a morphism $f : M_1 \to M_2$. Being able to construct $M = M_2/f(M_1)$ is important, but it is in general impossible to present M also as a locally effective module.

We require a *type* for the elements of M. To define such a type, we must be able to choose a canonical representant for any equivalence class modulo $f(M_1)$. In fact we are even in general unable to decide whether two elements $x, y \in M_2$ are equivalent modulo $f(M_1)$, because this is a *global* information about M_1. Typically, if M_1 is not of finite type, an infinite set of possible relations is to be studied and a computer cannot do such a work.

Still worse, sometimes the isomorphism class of M_1 could be unknown, the user being unable to know if M_1 is null or not, has a finite type or not, so that the equivalence relation defined in M_2 by M_1 and f is rather difficult to use! And this is not at all "exotic" circumstances which "of course" do not really happen in "ordinary" life: we will see on the contrary it is the *ordinary situation* when handling exact couples computing spectral sequences.

Definition 10 — A *locally effective subquotient* is a triple $S = (M_1, M_2, f)$ where the components M_1 and M_2 are locally effective $\mathbb{Z}$-modules and $f : M_1 \to M_2$ is a module morphism. The mathematical object so represented is the quotient module $M_2/f(M_1)$.

The best example for such a subquotient requires the obvious next definition.

Definition 11 — A *locally effective chain complex* is an algorithm $\underline{\text{lecc}} : \mathbb{Z} \to \mathcal{U}$ where, for every $n \in \mathbb{Z}$, the value $\underline{\text{lecc}}(n) = (C_n, d_n)$ is a pair made of a locally effective $\mathbb{Z}$-module C_n and a morphism $d_n : C_n \to C_{n-1}$ satisfying the relation $d_{n-1}d_n = 0$.

Proposition 12 — *Let (C_*, d_*) be a locally effective chain complex. Then the homology groups $H_n(C_*)$ are locally effective subquotients.*

♣ A homology group is a quotient $\ker/\operatorname{im} = Z_n/B_n = \{\text{cycles}\}/\{\text{boundaries}\}$ where the roles of the numerator and the denominator are *not* symmetric. For $\ker d_n$ *is* a locally effective module when $\operatorname{im} d_{n+1}$ in general *is not*.

Defining $\ker d_n$ as a locally effective module works as follows. The underlying type is defined as the elements of $x \in C_n$, already defined as a type, satisfying the extra relation $d_n x = 0$, a relation which is decidable, see the comments of Definition 8. So that the *type* $\ker d_n$ is defined, and adding the $+$, 0 and $-$ components of C_n achieves to define $\ker d_n$ as a locally effective module.

On the contrary, we have already explained in this section the relation of membership to $\operatorname{im} d_{n+1}$ is in general not decidable, so that it is not possible to present $\operatorname{im} d_{n+1}$ as a locally effective module.

But the triple $(C_{n+1}, \ker d_n, d_{n+1})$ is defined and *is* a locally effective subquotient representing the homology group $H_n(C_*) = Z_nC_*/B_nC_*$. ♣

This object is obviously quite misleading. It is installed as a machine object in your machine and this object really *codes* the homology group you are probably interested in. But you are probably still more interested by the isomorphism class of this homology group, which isomorphism class is in general non-deducible from the machine object.

So that you could wonder what the real usefulness of such an object could be? We will see this object will be the central object allowing us to make constructive exact couples and spectral sequences.

3.7 Vanishing certificates.

If (M_1, M_2, f) is a locally effective subquotient representing a $\mathbb{Z}$-module M, a situation denoted by $M \leftarrowtail (M_1, M_2, f)$, and if $\mathfrak{x} \in M$ is represented by an element $x \in M_2$, a situation denoted by $\mathfrak{x} \leftarrowtail x$, then the nullity of $\mathfrak{x}$, that is, the relation $x \in f(M_1)$ is in general undecidable.

Constructive logic is hidden here. Saying $\mathfrak{x} = 0$ is claiming *there exists* $v \in M_1$ satisfying $f(v) = x$. But a constructive existence is always better, often crucial.

Definition 13 — *A vanishing certificate for* $\mathfrak{x} \in M$ *represented by* $x \in M_2$ *is an element* $v \in M_1$ *satisfying* $f(v) = x$.

St Thomas can use this certificate and compare the objects $f(v)$ and x in M_2 to verify the claim, for M_2 is a locally effective module, where equality is decidable. But if he unfortunately loses this certificate, he is in general unable to find it again.

Relations such as $M \leftarrowtail (M_1, M_2, f)$ and $\mathfrak{x} \leftarrowtail x$ when $\mathfrak{x} \in M$ and $x \in M_2$ will have to be frequently considered in this paper. They mean $M = M_2/f(M_1)$ and $\mathfrak{x}$ is the equivalence class of $x \in M_2$ modulo $f(M_1)$. Such a situation will be quickly described by $x \mathrel{\overline{\in}} M$. And if v is a vanishing certificate for x, in fact for its equivalence class modulo $f(M_1)$, we write $v \xrightarrow{0} x$.

4 The homological problem for a differential module.

4.1 Introduction.

A central subject in most books about Homological Algebra, in particular the numerous books about Algebraic Topology, is the hunt for homology groups. For example for some specific space X, it could be proved that $H_6(X) = \mathbb{Z}/12$. The result $H_6(X)$ "=" $\mathbb{Z}/12$ is in fact a shorthand for the more precise statement: there *exists* an isomorphism between the group $H_6(X)$ and the well known group $\mathbb{Z}/12$, to be considered as a preferred representant of an isomorphism class. But it is exceptional this proof is *constructive*, we mean such an isomorphism is rarely made explicit. The situation is complex, because it is most often clear that, with a little patience (!?), such a proof could be made constructive. Conversely, at first sight, it is not obvious to find really useful such a constructive proof. In fact this appreciation is incorrect.

The basic "axiom" of Constructivism is the following: "Be sure a constructive existence proof is certainly better than a non-constructive one". This is now sufficiently well understood to need further explanations. In Homological Algebra, the justification of this axiom is simple: assume for example we intend to determine some unknown homology group H after calculations involving intermediate groups H_1, H_2 and $H_3 = H$; if the result about the computed H_2 is not constructive, then often the alleged "algorithm" $H_2 \mapsto H_3$ in fact fails: maybe for example some necessary differential in a spectral sequence is in fact out of scope, or some extension problem is unsolvable with the available information.

The present section is intended to very precisely define what a *constructive* solution for a homology group is.

4.2 Isomorphism class of $\mathbb{Z}$-module.

We need in particular a precise definition for the notion of *isomorphism class* of $\mathbb{Z}$-modules. In short, such an isomorphism class must be a machine object *and* the "elements" of such an isomorphism class must be clearly defined.

We firstly consider an example, sufficient for the rest of the paper, in fact definitively sufficient if you consider the situations where the homology groups are $\mathbb{Z}$-modules of finite type give you enough work.

We decided in Definition 4 to code an *effective module* as a list of natural numbers satisfying the divisor condition. For example the machine list `(2 6 0 0)`, usually mathematically written $(2, 6, 0, 0)$, so codes the $\mathbb{Z}$-module $\mathbb{Z}/2 \oplus \mathbb{Z}/6 \oplus \mathbb{Z} \oplus \mathbb{Z}$. The standard structure theorem for the $\mathbb{Z}$-modules of finite type explains every isomorphism class of such a module is coded by exactly one integer list $(d_1, \ldots, d_n)$ satisfying the conditions stated in Definition 4. The first step of our definition is done: we have decided to represent a large set $\mathcal{M}$ of isomorphism classes of $\mathbb{Z}$-modules, those of finite type, by some appropriate machine objects.

Important: representing the isomorphism class *is not enough*: we must also be able to represent the "elements" of such a class. More precisely, the isomorphism class must be represented by a *specific* $\mathbb{Z}$-module, the elements of which being also represented by some *non-ambiguous* process. A typical example, in fact again sufficient for this paper, is given in Definition 5. An element of `(2 6 0 0)`, that is, $\mathbb{Z}/2 \oplus \mathbb{Z}/6 \oplus \mathbb{Z} \oplus \mathbb{Z}$, is unambiguously represented by a list $(a_1, \ldots, a_4)$ of four integers; this representation is *sound*, for an algorithm can decide if $(a_1, \ldots, a_4) = (b_1, \ldots, b_4)$: the algorithm has just to determine whether $2|(b_1 - a_1)$ and $6|(b_2 - a_2)$, and also if $a_3 = b_3$ and $a_4 = b_4$. Another method could consist in choosing the *canonical* representation of an element, satisfying the conditions $0 \leq a_1 < 2$ and $0 \leq a_2 < 6$; as you prefer.

The $\mathbb{Z}$-module $M_0 = \mathbb{Z}^{(\mathbb{N})}$, the direct *sum* of a countable number of copies of $\mathbb{Z}$, is also an acceptable isomorphism class: an element of $a \in M_0$ can be coded as a list $(n, a_0, \ldots, a_{n-1})$ where the first component n is an upper bound for the indices i such that $a_i \neq 0$; and it is easy to decide whether two such elements are equal or not.

On the contrary, the module $M_1 = \mathbb{Z}^{\mathbb{N}}$, the direct *product* of a countable number of copies of $\mathbb{Z}$, is *not* an acceptable isomorphism class, for the number of available machine objects to code its elements is countable, while the cardinality of M_1 is not. You can also consider a variant $M_2 = \mathbb{Z}^{\overline{\mathbb{N}}}$, where the component sequence of an element $a = (a_i)_{i \in \mathbb{N}}$ must be *recursive*, that is, produced by some *algorithm* $a : \mathbb{N} \to \mathbb{Z}$; we let the reader determine if such an isomorphism class is acceptable or not.

In short, an acceptable isomorphism class of $\mathbb{Z}$-modules must be soundly represented by a machine object, coding a preferred representant of the isomorphism class, and all the elements *of this representant* must also be soundly represented by appropriate machine objects.

Restriction 14 — *From now on, when considering* homological problems*, the collection $\mathcal{M}$ of acceptable isomorphism classes used for the homology groups is restricted to the collection of $\mathbb{Z}$-modules of finite type as defined in Section 3.4.*

This class of acceptable isomorphism classes could be easily and significantly extended to more general classes, but it is not the subject of the present paper.

Because of this restriction, every common problem concerning modules elements of $\mathcal{M}$, such as determining kernels, images, cokernels and so on has a *constructive* solution.

4.3 The homological problem.

Definition 15 — A *solution for the homological problem* (SHP) posed by the *locally effective* differential module (M, d) is a 4-tuple $(H, \sigma_2, \sigma_3, \sigma_4)$ where:

- The component H is an isomorphism class of $\mathbb{Z}$-module $H \in \mathcal{M}$. This group is isomorphic to the genuine homology group $H(M,d) := Z(M,d)/B(M,d)$ through the isomorphism defined by σ_2 and σ_3.
- The component σ_2 is an algorithm $\sigma_2 : H \to Z(M,d)$ giving for every "abstract" homology class $h \in H$ a cycle $z = \sigma_2(h) \in Z(M,d)$ representing this homology class. In general, the map σ_2 cannot be a module morphism.
- The component σ_3 is an algorithm $\sigma_3 : Z(M,d) \to H$ computing for every cycle $z \in Z(M,d)$ "its" homology class $h = \sigma_3(z) \in H$. The composition $\sigma_3\sigma_2$ must be the identity of H. The map σ_3 is necessarily a module morphism.
- The component σ_4 is an algorithm $\sigma_4 : \ker \sigma_3 \to M$ satisfying $d\sigma_4 = \mathrm{id}_{\ker \sigma_3}$.

The component σ_4 is nothing but an algorithm producing a *certificate* for a cycle $z \in M$ claimed having a null homology class by the algorithm σ_3

Didactical example. We assume $\mathcal{M}$ is the set of $\mathbb{Z}$-modules of finite type as defined in Definition 4. Let us take in particular $M =$ `(0 0)` $= \mathbb{Z}^2$ provided with the differential $d = \begin{bmatrix} 0 & 2 \\ 0 & 0 \end{bmatrix}$.

Then $H =$ `(2)` $= \mathbb{Z}/2$ has two elements `(0)` and `(1)`. We can choose $\sigma_2($`(0)`$) =$ `(0 0)` and $\sigma_2($`(1)`$) =$ `(1 0)`. Note it is impossible to define a σ_2 which is a *module morphism*, a constant and unavoidable difficulty in these constructive questions. Here $Z(M, d) = \{$`(a 0)`$\}$ and no choice, $\sigma_3($`(a 0)`$) =$ `(a')` with $a' = a \bmod 2$. The σ_3 component is necessarily a module morphism, its kernel is defined and in this case it is $\ker \sigma_3 = \{$`(2a 0)`$\}$ and we can choose $\sigma_4($`(2a 0)`$) =$ `(0 a)`.

This example is so elementary the reader may wonder what this notion of SHP adds to the usual style. The present paper is devoted to this matter.

Non-pedagocical examples. Our previous solution [10] to make constructive Homological Algebra was based on the Basic Perturbation Lemma; its scope is more restricted, mainly we did not succeed with this strategy in processing the case of the spectral sequences not produced by filtrated complexes, but "only" by exact couples, typically the Bockstein-Browder and Bousfield-Kan spectral sequences. Nevertheless, when it can be applied, our previous method already gave striking results.

If interested, the reader can find a detailed exposition of our previous solution at [11]. In particular, [11, Section 4.7] displays a detailed particular case of sophisticated complex with a SHP computed: this is applied in this case in standard Commutative Algebra, when computing *effective* homology of Koszul complexes coming from ideals of polynomials. It is proved also in the same paper the *effective* homology of such a Koszul complex produces as a side effect a *resolution* of the considered ideal. While the standard organization does the contrary: computing the *ordinary* homology of a Koszul complex from a resolution!

Another striking example can be found at [3]. The problem of computing the homology of *iterated* loop spaces that are not suspensions is known to be a hard one. It is explained in [3] the constructive methods give an easy *and natural* solution, so easy that it is not difficult to concretely implement it on a computer and produce boundary matrices and homology groups so far unreachable.

5 Solving extension problems.

5.1 Standard theory.

Let A, B and C be three *commutative* groups and a short exact sequence:

$$0 \to A \xrightarrow{i} B \xrightarrow{j} C \to 0$$

The central group B is said to be an *extension* of C by A. In various contexts, the groups A and C are known, and you must "guess" the group B, that is, you have to determine its isomorphism class.

The standard method works as follows, see for example [5, Chapter IV]. The map j is surjective and there *exists* a section $\sigma : C \to B$ satisfying $j\sigma = \mathrm{id}_C$; in general this section cannot be a module morphism, it is sometimes said only "set-theoretic". We always assume the so-called normalization property is satisfied: $\sigma(0) = 0$. Most often, nothing is said about the *constructive* existence of such a section σ.

Also, because of the exactness property, a canonical map $\rho : \ker j \to A$ is defined, which is a module morphism. This quasi-retraction ρ is canonical, but its *constructive* existence is also problematic. No problem if A has a finite type; in the general case, the known existence of a unique i-preimage for an element of $\ker j$ *does not* give an algorithm computing such a preimage. Anyway, in "classical" mathematics, you can use the section σ and the quasi-retraction ρ.

The quasi-retraction ρ can be completed into a genuine retraction again denoted by $\rho : B \to A$ defined by $\rho(b) = \rho(b - \sigma j b)$. This retraction is "complete" but in general is nomore a morphism. The three classical formulas:

$$\begin{aligned} \rho i &= \mathrm{id}_A \\ i\rho + \sigma j &= \mathrm{id}_B \\ j\sigma &= \mathrm{id}_C \end{aligned}$$

are then satisfied.

A cohomology class $\chi \in H^2(C, A)$ is then obtained from the 2-cocycle also denoted by χ:

$$\chi(a, b) := \rho(\sigma(a) + \sigma(b) - \sigma(a + b))$$

This cocycle measures the failure of σ to be a morphism. Then a group law is defined on $A \times C$ by:

$$(a, c) + (a', c') = (a + a' + \chi(c, c'), c + c')$$

and tho inverse isomorphisms $\alpha : B \overset{\cong}{\to} A \times_\chi C$ and $\alpha^{-1} : A \times_\chi C \overset{\cong}{\to} B$ are defined by $\alpha(b) = (\rho b, jb)$ and $\alpha^{-1}(a, c) = ia + \sigma c$.

We so obtain, up to a known isomorphism, a complete description of the group B as the "twisted" product $A \times_\tau C$.

5.2 Constructive short exact sequence.

Definition 16 — A *constructive short exact sequence* is a diagram:

$$\begin{tikzcd} 0 \arrow[r] & A \arrow[r, shift right, "i"'] & B \arrow[l, shift right, "\rho"'] \arrow[r, shift right, "j"'] & C \arrow[l, shift right, "\sigma"'] \arrow[r] & 0 \end{tikzcd}$$

where A, B and C are locally effective $\mathbb{Z}$-modules, i and j are module morphisms, and the following relations are satisfied:

$$\begin{aligned} \rho i &= \mathrm{id}_A \\ i\rho + \sigma j &= \mathrm{id}_B \\ j\sigma &= \mathrm{id}_C \end{aligned}$$

The maps ρ and σ are not necessarily module morphisms. The relations $ji = 0$ and $\rho\sigma = 0$ are satisfied, and our sequence is ordinarily exact in the direction west-east. The maps ρ and σ are available to allow you to *constructively* use this property.

In particular, if such a constructive exact sequence is given, you can construct the cocyle χ classifying it and also the corresponding isomorphism $B \cong A \times_\chi C$.

Definition 17 — A *constructive short exact sequence of chain complexes* is a diagram:

$$\begin{tikzcd}
 & {} \arrow[d, dashed] & {} \arrow[d, dashed] & {} \arrow[d, dashed] & \\
0 \arrow[r] & A_n \arrow[r, shift right, "i"'] \arrow[d, "d"'] & B_n \arrow[l, shift right, "\rho"'] \arrow[r, shift right, "j"'] \arrow[d, "d"'] & C_n \arrow[l, shift right, "\sigma"'] \arrow[r] \arrow[d, "d"'] & 0 \\
0 \arrow[r] & A_{n-1} \arrow[r, shift right, "i"'] \arrow[d, dashed] & B_{n-1} \arrow[l, shift right, "\rho"'] \arrow[r, shift right, "j"'] \arrow[d, dashed] & C_{n-1} \arrow[l, shift right, "\sigma"'] \arrow[r] \arrow[d, dashed] & 0 \\
 & {} & {} & {} &
\end{tikzcd}$$

where:

- Every column is a chain complex of locally effective $\mathbb{Z}$-modules.
- Every horizontal row is a constructive short exact sequence.
- The horizontal morphisms i and j are compatible with the differentials.

5.3 A constructive long exact sequence.

In classical homological algebra, when a short exact sequence of chain complexes $0 \to A_* \xrightarrow{i} B_* \xrightarrow{j} C_* \to 0$ is given, a long exact sequence is deduced:

$$\cdots \to H_{n+1}C \xrightarrow{\partial} H_nA \xrightarrow{i} H_nB \xrightarrow{j} H_nC \xrightarrow{\partial} H_{n-1}A \to \cdots$$

The morphism ∂ is called the *connection* morphism, and the morphisms again denoted by i and j come from the given short exact sequence.

If you know for example the groups H_*A and H_*C, this long exact sequence is supposed to allow you to determine the unknown groups H_*B. Most books of homological algebra, mainly those principally devoted to algebraic topology, state such a claim in a rather fuzzy way, but "typical" striking examples are soon presented to convince the reader this "implicit" statement is correct.

Examining more carefully this matter, the long exact sequence produces a short exact sequence:

$$0 \to \operatorname{coker}(H_{n+1}C \xrightarrow{\partial} H_nA) \xrightarrow{i} H_nB \xrightarrow{j} \ker(H_nC \xrightarrow{\partial} H_{n-1}A) \to 0$$

and we are in front of an *extension problem.* Before *solving* this extension problem, you must compute a cokernel and a kernel with respect to the connection morphisms that are mathematically defined, but rarely in a constructive way!

If you have overcome this first obstacle, there remains to determine the cohomology class classifying the extension. And you are in a strange situation: you have to determine the corresponding section σ and retraction ρ, having as target or source the group H_nB that you do not yet know!

In this simple situation, the solution is well known, instead of homology classes inside an *unknown* group, you must use cycles in Z_nB *representing* these homology classes; so the cohomology class χ classifying the extension can be determined. Making systematic such a solution is easy when solutions for the homological problem SHP_A and SHP_C are given for the chain complexes A_* and C_*. Working in this way, we easily obtain again a SHP_B for the central chain complex B_*, so that this SHP_B can be used later for another analogous work. And so on.

The appropriate presentation of this method consists in considering the unknown homology group H_nB as coded as a *locally effective subquotient* as explained in Proposition 12: this coding is summarized by the formula: $H_nB \leftarrowtail (B_{n+1}, Z_nB, d)$: we *know* the *unknown* (!) group H_nB is the quotient $H_nB = Z_nB/d(B_{n+1})$. So every cycle $a \in Z_nB$ codes *some* homology class $\mathfrak{a} \in H_nB$, this being usable even if we do not know the isomorphism class of H_nB.

This point of view will be also the key point to prove our main theorem about the *effective exact couples* in Section 8, and we think the detailed proof below solving our extension problem is also a good introduction to this technique.

Theorem 18 — *An algorithm can be written down:*

- **Input:** *A constructive short exact sequence of locally effective chain complexes:*

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

Respective SHP_A and SHP_C for the chain complexes A_* and C_*.

- **Output:** *A SHP_B for the chain complex B_*.*

In the main applications, the involved chain complexes are *not* of finite type, so that no algorithm can directly compute H_*B from B_* alone.

♣ We must (very) carefully study the short exact sequence:

$$\begin{tikzcd} 0 \arrow[r] & \mathrm{Cok} \arrow[r, dashed, shift right, "i"'] & H_nB \arrow[l, dashed, shift right, "\rho"'] \arrow[r, dashed, shift right, "j"'] & \mathrm{Ker} \arrow[l, dashed, shift right, "\sigma"'] \arrow[r] & 0 \end{tikzcd}$$

where in short we denote $\mathrm{Cok} := \mathrm{coker}(H_{n+1}C \xrightarrow{\partial} H_nA)$ and $\mathrm{Ker} = \ker(H_nC \xrightarrow{\partial} H_{n-1}A)$.

We have printed in gray the central group H_nB to recall this group is currently unknown and elements of this group can be used only through *cycles* representing these homology classes. Deciding once for all that chains in general, cycles in particular, are denoted by latin characters a, b, $\ldots$, and homology classes are denoted by gothic characters $\mathfrak{a}$, $\mathfrak{b}$, $\ldots$, should help reading.

The map i of the diagram is obtained by following the diagram chasing path:

$$\mathrm{Cok} \to H_nA \overset{\sigma_2}{\to} Z_nA \overset{i}{\to} Z_nB$$

where the map σ_2 is produced by SHP_A, see Definition 15. Also the map $\mathrm{Cok} \to H_nA$ consists in choosing a representant of an element of the cokernel; because of Restriction 14, this can be elementary done. The morphism $i : \mathrm{Cok} \to H_nB$ is so implemented as a map $i : \mathrm{Cok} \to Z_nB$ which in general is neither canonical nor a morphism.

The map j of the diagram is canonical and is *implemented* as a genuine morphism $j : Z_nB \to \mathrm{Ker}$, it is simply obtained from the path:

$$Z_nB \overset{j}{\to} Z_nC \overset{\sigma_3}{\to} \mathrm{Ker} \subset H_nC$$

where the "map" $Z_nC \to \mathrm{Ker}$ is in fact incorrect; we mean only that if an element of Z_nC is in the image of j, then its homology class is necessarily in Ker; which homology class is determined through the σ_3 component of SHP_C.

The map $\sigma : \mathrm{Ker} \to H_nB$ implemented as a map $\sigma : \mathrm{Ker} \to Z_nB : \mathfrak{a} \mapsto a_6$ is obtained from the diagram:

$$\begin{tikzcd}
 & & a_6 \arrow[dl, dashed, bend right] \arrow[dr, dashed, bend left] & & & \\
a_4 \arrow[r, mapsto] \arrow[d, mapsto] & a_5 \arrow[dr, mapsto] & & a_1 \arrow[r, mapsto] \arrow[dl, mapsto] & a_0 \arrow[d, mapsto] & \mathrm{row}_n \\
a_3 \arrow[rr, mapsto] & & a_2 \arrow[rr, mapsto] & & 0 & \mathrm{row}_{n-1} \\
A_* & & B_* & & C_* &
\end{tikzcd}$$

to be understood as follows. In these episodes of diagram chasing, the chains are denoted a_k, the index k showing the construction chronology. The arrows illustrate the "main" maps i, j and d. For example a_1 above is obtained *after* a_0, in fact from a_0 thanks to the section $\sigma : C_n \to B_n$, but we prefer to display the map $j : a_1 \mapsto a_0$ than the map $\sigma : a_0 \mapsto a_1$. The dashed arrows $\dashleftarrow\dashrightarrow$ are used to illustrate differences.

If $\mathfrak{a} \in \mathrm{Ker} \subset H_nC$ is a homology class, then the elements of the above diagram are obtained as follows:

- $a_0 \in Z_nC$ is a representant of $\mathfrak{a}$ produced by σ_2 in SHP_C.
- $a_1 \in B_n$ is a j-preimage of a_0 produced by $\sigma : C_n \to B_n$.
- $a_2 = da_1 \in B_{n-1}(B)$ necessarily satisfies $ja_2 = 0$, for a_0 is a cycle.
- $a_3 \in Z_{n-1}A$ is an i-preimage of a_2 produced by $\rho : B_{n-1} \to A_{n-1}$.
- Because $\mathfrak{a} \in \mathrm{Ker}$, the homology class of a_3 is null and the component σ_4 of SHP_A produces a d-preimage $a_4 \in A_n$.

- $a_5 = ia_4 \in B_n$ and necessarily $da_5 = a_2$.
- Finally the difference $a_6 = (a_1 - a_5) \in Z_nB$.

Then the map $\sigma : \mathrm{Ker} \to H_nB$ implemented as $\sigma : \mathrm{Ker} \to Z_nB$ is defined by $\sigma\mathfrak{c} = a_6$. You observe nothing is new with respect to the standard proof of exactness of the long exact sequence at H_nC; the only important point is that the various *constructive* data allowed us to *construct* in turn the map σ. It is easy for a machine program, using the now standard methods of functional programming, to produce the functional object $\sigma : \mathrm{Ker} \to Z_nB$ from all the functional objects which are available in our environment. In much less time than it was necessary to write down these explanations.

The retraction $\rho : H_nB \to \mathrm{Cok}$ implemented as $\rho : Z_nB \to \mathrm{Cok}$ is firstly defined for the elements of $\ker(j : Z_nB \to \mathrm{Ker})$. Let a_0 be such an element.

$$\begin{tikzcd}
 & & & a_3 \arrow[r, mapsto] \arrow[d, mapsto] & a_2 \arrow[dd, mapsto] & \mathrm{row}_{n+1} \\
 & & & a_4 \arrow[dr, mapsto] & & \\
\mathfrak{a}_6 & \mathfrak{a}_5 \arrow[l, mapsto] & a_5 \arrow[l, mapsto] \arrow[ur, dashed, mapsto] \arrow[dr, dashed, mapsto] & & a_1 & \mathrm{row}_n \\
 & & & a_0 \arrow[ur, mapsto] & & \\
 & & A_* & B_* & C_* &
\end{tikzcd}$$

- The cycle $a_1 = ja_0 \in Z_nC$ must be a boundary.
- The d-preimage $a_2 \in C_{n+1}$ of a_1 is produced by the component σ_4 of SHP_C.
- The map $\sigma : C_{n+1} \to B_{n+1}$ produces a j-preimage $a_3 \in B_{n+1}$.
- $a_4 = da_3 \in B_n(B)$ has the same image in C_n as a_0.
- An i-preimage $a_5 \in Z_nA$ of $(a_0 - a_4)$ is produced by $\rho : B_n \to A_n$.
- $\mathfrak{a}_5 \in H_nA$ is the homology class of a_5, computed by the σ_3 component of SHP_A.
- $\mathfrak{a}_6 \in \mathrm{Cok}$ is the equivalence class of $\mathfrak{a}_5$ modulo the image of $\partial : H_{n+1}C \to H_nA$, computable because of Restriction 14.

Then the desired map $\rho' : \ker(j : H_nB \to \mathrm{Ker}) \to \mathrm{Cok}$, implemented as a map $\rho' : \ker(j : Z_nB \to \mathrm{Ker}) \to \mathrm{Cok}$ is defined by $\rho'(a_0) = \mathfrak{a}_6$. This map is a morphism and is canonical, but do not forget in general "canonical" *does not* imply "computable". Then ρ can be completed into a map $\rho : Z_nB \to \mathrm{Cok}$ by $\rho(a) := \rho'(a - \sigma ja)$, but this map in general is nomore a morphism.

$$\begin{tikzcd}
0 \arrow[r] & \mathrm{Cok} \arrow[r, shift right, "i"'] & H_nB \arrow[l, dashed, shift right, "\rho"'] \arrow[r, shift right, "j"'] & \mathrm{Ker} \arrow[l, dashed, shift right, "\sigma"'] \arrow[r] & 0
\end{tikzcd}$$

The relations $\rho i = \mathrm{id}_{\mathrm{Cok}}$ and $j\sigma = \mathrm{id}_{\mathrm{Ker}}$ are satisfied. If $a \in Z_nB$, then $i\rho a + \sigma ja$ is only homologous to a, but we will see in a moment we can compute a d-preimage of the difference.

If now $\mathfrak{c}, \mathfrak{c}' \in \mathrm{Ker}$ are two arguments for the looked-for cocycle χ defining our extension, in general the difference $\sigma(\mathfrak{c}) + \sigma(\mathfrak{c}') - \sigma(\mathfrak{c} + \mathfrak{c}')$ is not null, but it is certainly in the kernel of $j : Z_n B \to \mathrm{Ker}$. The cocyle $\chi \in H^2(\mathrm{Ker}, \mathrm{Coker})$ is therefore defined as:

$$\chi(\mathfrak{c}, \mathfrak{c}') = \rho(\sigma(\mathfrak{c}) + \sigma(\mathfrak{c}') - \sigma(\mathfrak{c} + \mathfrak{c}')).$$

The standard extension theory proves $H_n B \cong \mathrm{Coker} \times_\chi \mathrm{Ker}$ and an elementary calculation can produce the unique possible isomorphism class $H_B \in \mathcal{M}$ of this group and also some explicit isomorphism $H_n B \leftrightarrow \mathrm{Coker} \times_\chi \mathrm{Ker}$.

We need also the three components σ_2, σ_3 and σ_4 to achieve the construction of SHP_B. We define σ_2 and σ_3 firstly with respect to the model $\mathrm{Coker} \times_\chi \mathrm{Ker}$ of $H_n B$. It is easy to justify $\sigma_2(\mathfrak{a}, \mathfrak{c}) = i\mathfrak{a} + \sigma\mathfrak{c} \in Z_n B$ if $\mathfrak{a} \in \mathrm{Cok}$ and $\mathfrak{c} \in \mathrm{Ker}$. In the same way, $\sigma_3(b) = (\rho b, jb)$ is the unique possible definition of σ_3. These definitions can then be converted into correspondances with H_B thanks to an arbitrary isomorphism $\mathrm{Coker} \times_\chi \mathrm{Ker} \cong H_B$.

Constructing the map σ_4 is a little more complicated, but no choice. We start with an $a_0 \in Z_n B$ satisfying $ja_0 = 0 \in \mathrm{Ker}$ and $\rho a_0 = 0 \in \mathrm{Cok}$. The diagram previously drawned to construct $\rho : Z_n B \to \mathrm{Cok}$ can then be completed as below:

$$\begin{tikzcd}
 & & a_7 \arrow[rrr, mapsto] \arrow[dd, mapsto] & & & a_6 \arrow[dddd, mapsto] & \\
a_{10} \arrow[r, mapsto] \arrow[ddd, mapsto] & a_{11} \arrow[ddd, mapsto] & & a_3 \arrow[r, mapsto] \arrow[dd, mapsto] & a_2 \arrow[ddd, mapsto] & & \mathrm{row}_{n+1} \\
a_9 \arrow[rr, mapsto] & & a_8 & & & & \\
 & & & a_4 \arrow[r, mapsto] & & & \\
a_5 \arrow[rrr, mapsto] \arrow[u, dashed] & a_{11}' \arrow[ur, dashed] & & & a_1 & 0 & \mathrm{row}_n \\
 & & & a_0 \arrow[ur, mapsto] & & & \\
A_* & & B_* & & C_* & &
\end{tikzcd}$$

The added ingredients are:

- The homology class of the cycle a_5 is now in the image of the connection morphism $\partial : H_{n+1} C \to H_n A$. This produces as usual $a_6 \in Z_{n+1} C$, $a_7 \in B_{n+1}$, $a_8 \in B_n(B)$ and $a_9 \in Z_n A$ such that the difference $a_5 - a_9$ is a boundary.
- The σ_4 component of SHP_A produces a d-preimage $a_{10} \in A_{n+1}$ of $a_5 - a_9$.
- $a_{11} = ia_{10} \in B_{n+1}$ therefore satisfies $da_{11} = i(a_5 - a_9) = a_0 - a_4 - a_8$.
- This implies $d(a_{11} + a_7 + a_3) = a_0$ and we can define $\sigma_4(a_0) = a_{11} + a_7 + a_3$.

Again this is nothing but the standard proof of the short exact sequence connecting Coker, $H_n B$ and Ker, verifying along the proof that the existence of every necessary object is *constructive*. A routine exercise. ♣

This proof is a routine exercise, but constructively (!) writing down the corresponding machine program maybe is a little less routine.

To finish with this matter, let us mention the same method can also be used when SHPs are provided for A_* and B_*, or for B_* and C_* to *compute* the missing SHP_C or SHP_A.

6 Exact couples.

6.1 Definition.

Definition 19 — An *exact couple* is a diagram

$$\begin{tikzcd} D \arrow[rr, "i"] & & D \arrow[dl, "j"] \\ & E \arrow[ul, "k"] & \end{tikzcd}$$

where the following conditions are satisfied:

- The components D and E are $\mathbb{Z}$-modules.
- The components i, j and k are $\mathbb{Z}$-module morphisms.
- The circular sequence is exact, that is, $\ker j = \operatorname{im} i$, $\ker k = \operatorname{im} j$ and $\ker i = \operatorname{im} k$.

Such an exact couple is denoted by the 5-tuple (D, E, i, j, k).

The components of an exact couple are often graded or bigraded with various compatibility conditions between module gradings and morphism gradings, but the general theory of exact couples as newly organized here is independent of such gradings.

The composition jk is an *endo*morphism of E and it is a *differential*: $jkjk = 0$. This produces a $\mathbb{Z}$-module of cycles $Z(E, jk) := \ker jk$, a module of boundaries $B(E, jk) := \operatorname{im} jk$ and a homology group $H(E, jk) := Z(E, jk)/B(E, jk)$.

6.2 Derived exact couple.

Definition 20 — Let (D, E, i, j, k) be an exact couple. Then the *derived exact couple*:

$$\begin{tikzcd} D' \arrow[rr, "i'"] & & D' \arrow[dl, "j'"] \\ & E' \arrow[ul, "k'"] & \end{tikzcd}$$

is defined as follows.

- The component D' is the i-image of D: $D' := i(D) = \ker j \subset D$.
- The component E' is the homology group $E' := H(E, jk) := Z(E, jk)/B(E, jk)$.
- The component i' is the restriction of i to the submodule D'.

- Let $a \in D'$. There exists $b \in D$ satisfying $ib = a$; then $j'a := \overline{jb} \in H(E, jk) = E'$ is the homology class of the cycle $jb \in Z(E, jk)$.
- Let $\mathfrak{a} \in E'$. The homology class $\mathfrak{a} \in E' = H(E, jk)$ is represented by a cycle $a \in Z(E, jk) \subset E$; then $k'\mathfrak{a} := ka$.

Proposition 21 — *Definition 20 is coherent.*

♣ D' and E' are $\mathbb{Z}$-modules. The image of i' is in the image of i, that is, in D'.

The map j' is well defined. On the one hand, $jk(jb) = 0$, for $kj = 0$, and $jb \in Z(E, jk)$ is satisfied. On the other hand, if b' is another i-preimage of a, then $i(b - b') = 0$, and the exactness of the initial exact couple produces $c \in E$ with $kc = b - b'$; so that $jb - jb' = jkc \in B(E, jk)$ and the cycles jb and jb' are homologous: $j'a := \overline{jb} = \overline{jb'}$ does not depend on the choice of the preimage b.

The map k' is well defined. On the one hand, $a \in Z(E, jk)$, so that $j(ka) = 0$ and $k'\mathfrak{a} := ka \in \ker j = D'$. If a' is another representant for the homology class $\mathfrak{a}$, this implies $a - a' = jkb$ for some $b \in E$, and $ka - ka' = kjkb = 0$: the definition of $k'\mathfrak{a}$ does not depend on the choice of the representant a.

The relation $j'i' = 0$ is satisfied. Let $a \in D'$; then $j'i'a := \overline{ja} = \overline{0}$, for $D' = \ker j$.

The relation $k'j' = 0$ is satisfied. If $a \in D'$ and $ib = a$, then $j'a := \overline{jb} =: \mathfrak{c}$; a cycle representing $\mathfrak{c}$ is jb, so that $k'j'a = k'\mathfrak{c} := kjb = 0$.

The relation $i'k' = 0$ is satisfied. If $\mathfrak{a} \in E'$ is represented by $a \in Z(E, jk)$, then $i'k'\mathfrak{a} = ika = 0$.

Let $a \in \ker j' \subset D'$. Let $b \in D$ be an i-preimage of a, that is, $ib = a$. Then $j'a := \overline{jb}$ and $j'a = 0$ means $jb \in B(E, jk)$; this produces $c \in E$ satisfying $jb = jkc$; therefore $b - kc \in \ker j = \operatorname{im} i$ which produces in turn $d \in D$ satisfying $id = b - kc$. Applying i to the last relation gives $iid = ib - ikc = ib = a$, in other words $a \in ii(D) = i(D') = i'(D')$. The relation $\ker j' = \operatorname{im} i'$ is proved.

Let $\mathfrak{a} \in \ker k' \subset E' = H(E, jk)$. If $a \in Z(E, jk)$ represents the homology class $\mathfrak{a}$, the relation $ka = 0$ is satisfied. The exactness of the initial exact couple produces $b \in D$ with $jb = a$; then $c = ib \in D'$ and $j'c := \overline{jb} = \overline{a} = \mathfrak{a}$. The relation $\ker k' = \operatorname{im} j'$ is proved.

Let $a \in \ker i' \subset D'$. On the one hand $a = ib$ for some $b \in D$ and on the other hand $ia = 0$. The exactness of the initial exact couple produces $c \in E$ with $kc = a$. But $c \in Z(E, jk)$, for $jkc = ja = jib = 0$, so that the homology class $\overline{c}$ is defined and $k'\overline{c} = a$. The relation $\ker i' = \operatorname{im} j'$ is proved. ♣

7 Effective exact couples.

Definition 22 An *effective exact couple* is a 5-tuple (D, E, i, j, k):

$$\begin{tikzcd}
D \arrow[rr, "i"] & & D \arrow[dl, "j"] \\
& E \arrow[ul, "k"] &
\end{tikzcd}$$

where:

- *The component D is a* locally effective subquotient*.*
- *The component E is an* effective $\mathbb{Z}$-module*.*
- *The components i, j and k are morphism modules, satisfying the exactness properties usually required for exact couples under an* effective *form detailed below.*

Observe the very different natures of the D and E components. The E component is the easiest: E is an "ordinary" effective module, in particular of finite type, and "everything" about it can be easily deduced from the machine object implementing it. On the contrary, the D component is only a locally effective subquotient, the fuzziest sort of machine object in our environment. A fuzzy object but which will be the key object allowing us to compute the related spectral sequence.

The module D is a locally effective subquotient, represented by a triple (D_1, D_2, f). The module morphism i is implemented as a f-coherent pair of morphisms $i_1 : D_1 \to D_1$ and $i_2 : D_2 \to D_2$. In particular, if $v \overset{0}{\to} x$, then $i_1 v \overset{0}{\to} i_2 x$.

The morphism $j : D \to E$ is implemented as a morphism $j_2 : D_2 \to E$; this morphism must satisfy the coherence condition $j_2 f = 0$. The morphism k is implemented as a *map* $k_2 : E \to D_2$ which in general *cannot be* a module morphism: the genuine morphism $k : E \to D$ in general cannot be lifted in a module *morphism* $E \to D_2$.

We will see the fact k_2 is not necessarily a module morphism is a real obstacle when computing the derived exact couple of an effective exact couple. We can decide once for all $k_2 0 = 0$. But what about the compatibility of k_2 with addition? The solution is the following: we *must* assume an algorithm $\underline{\mathrm{lk}} : E \times E \to D_1$ ($\underline{\mathrm{lk}} =$ legal k) is *provided*, satisfying the relation $k_2(a+b) - k_2(a) - k_2(b) = f(\underline{\mathrm{lk}}(a,b))$ for every $a, b \in E$. In other words, the algorithm $\underline{\mathrm{lk}}$ is a *certificate* the map $k : E \to D$ *represented by* the map $k_2 : E \to D_2$ really is a module morphism.

The module E is of finite type and can be presented by a $\mathbb{Z}$-resolution $0 \leftarrow E \leftarrow E_2 \leftarrow E_1 \leftarrow 0$ implicitly given by Definition 4. The morphism k can therefore be organized as a pair of coherent morphisms $E_2 \to D_2$ and $E_1 \to D_1$ from which k_2 and $\underline{\mathrm{lk}}$ are easily deduced. It is tempting to organize the whole work using also analogous resolutions for D, but the standard result about a resolution of length 2 for every $\mathbb{Z}$-module is *not constructive* in infinite dimension; this is the reason why it seems impossible to avoid these technicalities.

The relation $ji = 0$ means that for every $a \mathrel{\overline{\in}} D$, the relation $j_2 i_2 a = 0$ is satisfied. Because the image $j_2 i_2 a \in E$ which is an effective *module*, no certificate is necessary.

The converse situation is quite different. If $a \mathrel{\overline{\in}} D$ satisfies $j_2 a = 0$, then a *provided* algorithm $\underline{\mathrm{ee}}'_{ji} : \ker j_2 \to D_2 \times D_1 : a \mapsto (b, c)$ produces a "preimage" $b \in D_2$ (or $\mathrel{\overline{\in}} D$) and a *certificate* $f(c) = a - i_2 b$. Read $\underline{\mathrm{ee}}'_{ji}$ as "effective exactness for $\ker j \subset \operatorname{im} i$". It is not in general possible to directly require $i_2 b = a$.

The relation $kj = 0$ must be validated by a provided algorithm $\underline{\mathrm{ee}}_{kj} : D_2 \to D_1 : a \mapsto b$ satisfying $fb = k_2 j_2 a$: no reason the choice made for $k_2(ja) \mathrel{\overline{\in}} D$ when defining k_2 sends ja to 0; this would imply the relation "membership to $\operatorname{im} j$" is decidable, which is false in general. Read $\underline{\mathrm{ee}}_{kj}$ as "effective exactness for $\ker k \supset \operatorname{im} j$".

In the same way, an algorithm $\underline{\mathrm{ee}}'_{kj} : \ker(k_2 \ominus f) \to D_2$ is provided. The source is made of all the pairs $(a, b) \in E \times D_1$ satisfying $k_2 a = fb$. Then $\underline{\mathrm{ee}}'_{kj}(a, b) = c$ satisfies $j_2 c = a$. This means that if "someone" provides a proof that $ka = 0$, then, probably using this proof, an algorithm computes a j-preimage or more precisely a representant for this preimage.

The reader has already guessed two other effectiveness algorithms are required. On the one hand a map is provided $\underline{\mathrm{ee}}_{ik} : E \to D_1 : a \mapsto b$, which satisfies $fb = ika$, proving the relation $0 = ik : E \to D$. On the other hand, more complicated, another map is provided $\underline{\mathrm{ee}}'_{ik} : \ker(i \ominus f) \to E \times D_1 : (a, b) \mapsto (c, d)$; the source of this map is the type of pairs $(a, b) \in D_2 \times D_1$ satisfying $i_2 a = fb$; that is, the element of D represented by $i_2 a$ is proved null thanks to the *provided* certificate b; then, probably using in particular this certificate, our algorithm knows how to compute a k-preimage c for the element of D represented by a and also a certificate d validating $a = k_2 c$ modulo $f(D_1)$.

7.1 The graded and bigraded cases.

The most interesting exact couples are defined in a graded or bigraded context. All the definitions and results must then be adapted in a simple way: consider the situation for every degree (or bidegree) separately. For example the Bockstein exact couple is graded, and $D = \oplus D_n$, $E = \oplus E_n$, the morphisms i and j have degree 0 and the morphism k has degree -1.

Then you must consider E is an algorithm $E : \mathbb{Z} \to \mathcal{U}$ such that for every $n \in \mathbb{Z}$, the image $E(n) =: E_n$ is an effective module, and the same for the decomposition of D in an infinite sum of locally effective subquotients. The definition of exact couple is then easily adapted, and also the proof of the fundamental theorem of the next section. Note in this case *every* E_n is required effective when $E = \oplus E_n$ in general is not.

The same in the bigraded case.

8 The fundamental theorem.

Theorem 23 — *A general algorithm* $\underline{\mathrm{dec}}$ *(derive exact couple) can be written down:*

- **Input:** *An* effective *exact couple (D,E,i,j,k).*
- **Output:** *The derived exact couple* (D', E', i', j', k') *also organized as an* effective *exact couple.*

The output exact couple satisfies the same properties as the input exact couple, and this algorithm can therefore be trivially iterated; a repetitive work which will provide all the terms of the possibly associated spectral sequence. In particular all the differentials of the spectral sequence can be deduced from the initial exact couple.

♣ The component E' is the homology group of the differential module (E, jk) with $jk = j_2k_2$ computable, a genuine module morphism, even if k_2 is not. The module E is effective, which allows to elementarily compute E' also as an effective module, see Proposition 7.

The component D' is defined as $D' := \operatorname{im} i = \ker j$. The relation $0 = j_2 f : D_1 \to E$ is satisfied, which implies $\operatorname{im} f \subset \ker j_2$. Which implies in turn $D' := \ker j$ is also a locally effective subquotient defined by the triple $D' \longleftarrow\!\!\!< (D_1, \ker j_2, f)$. As already observed, the module E is effective and $\ker j_2$ is a locally effective module, in particular a type.

The restriction i'_2 of $i_2 : D_2 \to D_2$ to $D'_2 = \ker j_2$ is sufficient to define i', for $j_2 i_2 = 0$, without any certificate. We must keep the same $i'_1 := i_1$. We observe the "fuzzy" module D is in fact rather convenient.

Let us construct now the morphism j', that is a morphism $j'_2 : \ker j_2 \to E'$. If $a \in D'_2 = \ker j_2$ represents $\mathfrak{a} \in D'$, we know $a \in \ker j_2$ and the algorithm $\underline{\mathrm{ee}}'_{ji}$ computes $(b, c) \in D_2 \times D_1$ satisfying $fc = a - i_2 b$. In particular the equivalence class $\mathfrak{b}$ of b satisfies $i\mathfrak{b} = \mathfrak{a}$. The j_2-image of b is the j-image of $\mathfrak{b}$, and it is necessarily a cycle of the differential module (E, kj), the homology class $\overline{j_2 b} \in E'$ of this cycle being elementarily computable, for the module E is effective. We obtain in this way an algorithm $a \mapsto \overline{j_2 b}$ which is the desired j'_2 implementing the genuine $j' : D' \to E'$.

Constructing $k' : E' \to D'$, that is, $k'_2 : E' \to D'_2$ is more direct. Given a homology class $\mathfrak{a} \in E' := H(E, jk)$, because the differential module (E, jk) is effective, an elementary algorithm can produce a *cycle* $a \in (E, jk)$ representing this homology class. In particular $j_2 k_2 a = 0$ and $k_2 a \in \ker j_2 = D'_2$. Then $k'_2 \mathfrak{a} := k_2 a$ is appropriately defined.

The main ingredients of the derived couple are constructed, and there remains to verify this derived couple is also effective.

Constructing $\underline{\mathrm{ee}}'_{j'i'}$. Let $a \in D'_2$ satisfying $j'_2 a = 0$. Necessarily, $a \in \ker j_2$. Using the effectiveness algorithm $\underline{\mathrm{ee}}'_{ji}$, we obtain a pair $(b, c) \in D_2 \times D_1$ satisfying $i_2 b = a - fc$ and $j'_2 a$ is the homology class of the cycle $j_2 b$ which therefore is a boundary. The differential module (E, jk) is effective, and we can elementarily compute a preimage d of $j_2 b$ for the boundary operator jk; in other words, $j_2 k_2 d = j_2 b$ and $b - k_2 d \in \ker j_2 = D'_2$ is a good candidate. Let us examine $i'_2(b - k_2 d) = i_2 b - i_2 k_2 d = a - fc - i_2 k_2 d$. The algorithm $\underline{\mathrm{ee}}_{ik}$ applied to d produces $e \in D_1 = D'_1$

satisfying $fe = i_2k_2d$, so that finally $i'_2(b - k_2d) = a - f(c + e)$. The algorithm $\ker j'_2 \to D'_2 \times D'_1 : a \mapsto (b - k_2d, c + e)$ is a solution for $\underline{\mathrm{ee}}'_{j'i'}$.

CONSTRUCTING $\underline{\mathrm{ee}}_{k'j'}$. Let $a \in D'_2 = \ker j_2$. We must be able to certify $k'_2j'_2a \in f(D'_1) = f(D_1)$. The construction of j'_2a uses $(b, c) \in D_2 \times D_1$ satisfying $i_2b = a - fc$ and then j'_2a is the homology class of j_2b in (E, jk). Applying k'_2 to this class has been *previously* done by choosing a cycle d representing this homology class, *not necessarily* the cycle j_2b, and $k'_2j'_2a = k_2d$ which is to be proved in $f(D_1)$. The cycles j_2b and d are homologous and we can elementarily compute $e \in E$ satisfying $j_2k_2e = d - j_2b$. So that $k_2d = k_2(j_2b + (d - j_2b)) = k_2(j_2b + j_2k_2e) = k_2j_2(b + k_2e)$ and the vanishing certificate for the last element provided by the initial exact couple is a valid certificate for the derived one: we may choose $\underline{\mathrm{ee}}_{k'j'}(a) := \underline{\mathrm{ee}}_{kj}(b + k_2e)$. The fact k_2 is not a morphism was not here an obstacle.

CONSTRUCTING $\underline{\mathrm{ee}}'_{k'j'}$. Let $\mathfrak{a} \in H(E, kj)$ represented by $a \in Z(E, jk)$ and $b \in D_1$ a vanishing certificate for $k_2a \mathrel{\overline{\in}} D'$ representing $0 = k'\mathfrak{a} \in D'$. Me must find a j'-preimage for $\mathfrak{a}$. The effectiveness algorithm $\underline{\mathrm{ee}}_{kj}$ produces $c \in D_2$ satisfying $j_2c = a$. Then the element $d = i_2c$ is in $\mathrm{im}\, i_2 \subset \ker j_2 = D'_2$ and it is clear that $j'_2d = \mathfrak{a}$.

CONSTRUCTING $\underline{\mathrm{ee}}_{i'k'}$. If $\mathfrak{a} \in H(E, jk)$ is represented by the cycle $a \in Z(E, jk)$, then $i'_2k'_2\mathfrak{a} := i_2k_2a$. The vanishing certificate for the last element provided by the initial exact couple can be used as well for the derived exact couple.

CONSTRUCTING $\underline{\mathrm{ee}}'_{i'k'}$. Let $a \in D'_2 = \ker j_2$ and $b \in D_1$ a certificate satisfying $i'_2a := i_2a = fb$. We must find a k'_2-pseudo-preimage $\mathfrak{c} \in E'$ for a and a certificate in D_1 proving $k'_2\mathfrak{c} = a$ modulo this certificate. The initial effectiveness algorithm $\underline{\mathrm{ee}}'_{ik}$ produces $c \in E$ and $d \in D_1$ satisfying $k_2c = a + fd$. Then c is a cycle in (E, jk), for $j_2k_2c = j_2(a + fd) = 0$. The homology class $\mathfrak{c}$ of the cycle c is a good candidate for the wished homology class. But computing $k'_2\mathfrak{c}$ consists in choosing a cycle c', often different from c, and $k'_2\mathfrak{c} := k_2c'$. The cycles c and c' are homologous and an element $e \in E$ can be produced satisfying $c' - c = j_2k_2e$. An obvious calculation gives $a - k'_2\mathfrak{c} = a - k_2c' = a - k_2(c + (c' - c)) = a - (k_2c + k_2(c' - c) + fe') = (a - k_2c) + k_2j_2k_2e + fe'$ and the three summands of the last term can be certified as elements of $f(D_1)$. In particular e' has been produced by the algorithm $\underline{\mathrm{lk}}$ of the initial exact couple to correct the fact that k_2 is not necessarily compatible with addition.

CONSTRUCTING $\underline{\mathrm{lk}}'$. Let $\mathfrak{a} \in E'$ (resp. $\mathfrak{a}' \in E'$). Computing $k'_2(\mathfrak{a} + \mathfrak{a}')$ (resp. $k'_2\mathfrak{a}$, $k'_2\mathfrak{a}'$) is made through *choices* of cycles of (E, jk), resp. a'', a and a'. The relation $a'' = a + a'$ is in general *not* satisfied, let us call a''' the error $a''' = a'' - a - a'$. The homology class of a''' in the differential module (E, jk) is null and $a''' = j_2k_2b$ for a computable $b \in E$. Then, using $\underline{\mathrm{ee}}_{kj}$, we obtain $c \in D_1$ satisfying $fc = k_2a'''$. Using now the $\underline{\mathrm{lk}}$ component of the initial exact couple, we successively find out the elements $c', c'' \in D_1$ justifying the calculation: $k'_2(\mathfrak{a} + \mathfrak{a}') - k'_2(\mathfrak{a}) - k'_2(\mathfrak{a}') = k_2(a'') - k_2(a) - k_2(a') = k_2(a''' + a + a')) - k_2(a) - k_2(a') = k_2(a''') + k_2(a + a') + f(c') - k_2(a) - k_2(a') = f(c) + f(c') + f(c'') = f(c + c' + c'')$. We can therefore define $\underline{\mathrm{lk}}'(\mathfrak{a}, \mathfrak{a}') = c + c' + c''$. ♣

9 The didactic example of bicomplexes.

[See [14, pp. 19-21].]

10 The Bockstein spectral sequence.

11 Solving extension problems at abutment.

References

[1] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[2] John Allen. *Anatomy of Lisp.* McGraw-Hill, 1978.

[3] Ainhoa Berciano, Julio Rubio and Francis Sergeraert. *A case study of A_∞-structure.* To appear.
`http://www-fourier.ujf-grenoble.fr/ sergerar/Papers/Tornike.pdf`

[4] Edgar H. Brown Jr.. *Finite computability of Postnikov complexes.* Annals of Mathematics, 1957, vol. 65, pp. 1-20.

[5] Kenneth S. Brown. *Cohomology of groups.* Springer, 1982.

[6] Alain Clément. `http://www.xing.com/profile/Alain_Clement`

[7] Xavier Dousson, Julio Rubio, Francis Sergeraert and Yvon Siret. *The Kenzo program.* `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`

[8] Michael A. Mandell. *Cochains and homotopy types.* Publications Mathématiques de l'IHES, vol 103, pp 213-246.

[9] Julio Rubio, Francis Sergeraert. *Constructive Algebraic Topology.* Bulletin des Sciences Mathématiques, 2002, vol. 126, pp. 389-412.

[10] Julio Rubio, Francis Sergeraert. *Algebraic Models for Homotopy Types.* Homology, Homotopy and Applications, 2005, vol.7, pp.139160.

[11] Julio Rubio, Francis Sergeraert. *Genova Lecture Notes.*
`http://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Genova-Lecture-Notes.pdf`

[12] Rolf Schön. *Effective algebraic topology.* Memoirs of the American Mathematical Society, 1991, vol. 451.

[13] Francis Sergeraert. *The computability problem in algebraic topology.* Advances in Mathematics, 1994, vol. 104, pp. 1-29.

[14] Francis Sergeraert.
`http://www-fourier.ujf-grenoble.fr/ sergerar/Talks/09-01-Logrono.pdf`

[15] Wikipedia. *Setoid.* `http://en.wikipedia.org/wiki/Setoid`