Report Jónathan Heras's stay in Göteborg*

Jónathan Heras

June 5, 2012

Abstract

From 29th April to 20th May, J. Heras visited Göteborg to work with T. Coquand, A. Mörtberg and V. Siles. In this short document, we report the problems that have been undertaken during such a stay. In summary these were:

1. computation of the inverse of triangular matrices,
2. isomorphic vector spaces,
3. elementary collapses, and
4. persistent homology.

1 Inverse of a triangular matrix

The first problem that we undertook was the formalization of an executable version of the inverse of lower triangular matrices.

1.1 Motivation

In [5], we presented the formalization of the computation of discrete vector fields. This technique allows one to reduce the size of topological objects but preserving their homology properties. In particular, given a chain complex (C_q, d_q) where every C_q is finitely generated, we can represent every d_q as a matrix. Discrete vector fields are computed from those matrices and allow us to construct a new chain complex $(\widehat{C}_q, \widehat{d}_q)$ which is smaller than the original one.

Given d_q represented as a matrix M and a discrete vector field on it, we can reorder the lines and columns of the matrix as follows:

$$\left(\begin{array}{c|c} \epsilon & \phi \\ \hline \psi & \beta \end{array}\right)$$

where ϵ is a lower triangular matrix with ones in the diagonal. We can construct $\widehat{d}_q$ as $\beta - \psi\epsilon^{-1}\phi$. A more detailed description of this process can be seen in [7].

*Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

The SSReflect library contains all the necessary definitions to construct $\widehat{d_q}$ in an abstract way, that is to say using the matrix representation of SSReflect. However using such a representation we cannot perform computations. To handle this issue, we can use the CoqEAL library [1] where executable counterparts of some of the algorithms for matrices have been defined. However, this library does not contain an algorithm to compute the inverse of matrices.

The first attempt to formalize an executable counterpart for the inverse algorithm algorithm was quite naive. In particular, using the different tools available in the CoqEAL library we defined a direct translation of the inverse algorithm (which is based on the computation of adjugates). This implementation is quite inefficient and we are only able to compute the inverse of matrices up to size 7×7.

Then, we undertook the task of developing an efficient inverse algorithm for the case of lower triangular matrices with ones in the diagonal.

1.2 Formalization in Coq/SSReflect

In order to formalize the inverse of lower triangular matrices with 1's in the diagonal, we have followed the methodology presented in [1].

Firstly, we have defined an algorithm for SSReflect matrices; since, we know that we are going to work with triangular matrices with 1's in the diagonal, we can define a quite efficient algorithm. Such an algorithm is defined recursively, in particular, we have a matrix of the form:

$$M = \left(\begin{array}{c|ccc} 1 & 0 & \dots & 0 \\ \hline c & & N & \end{array}\right)$$

if we know the inverse for N, then we can construct the inverse of M as follows:

$$M = \left(\begin{array}{c|ccc} 1 & 0 & \dots & 0 \\ \hline -N^{-1} \times c & & N & \end{array}\right)$$

The base case is just the matrix (1). Such an algorithm is defined in SSReflect as follows:

```
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
  | O => fun _ => 1%:M
  end.
```

Subsequently, we prove that this algorithm is equivalent to the inverse algorithm implemented in SSReflect. To this aim, we firstly define the notion of lower triangular matrix with ones in the diagonal.

```
Definition lower1 m (M : 'M[R]_m) :=
  forall (i j : 'I_m), i <= j -> M i j = (i == j)%:R.
```

Afterwards, we prove that given a lower triangular matrix with ones in the diagonal M, then $M \times \widehat{M} = 1$ where $\widehat{M}$ is the result obtained with our algorithm, and 1 is the identity matrix.

Then, we prove that the inverse of a matrix is unique:

```
Lemma invmx_uniq m (M M' : 'M[R]_m) :
  M *m M' = 1%:M -> M' = invmx M.
```

Finally, we formalize that our algorithm is equivalent to the inverse of SSReflect for lower triangular matrix with ones in the diagonal.

```
Lemma fast_invmxP m (M : 'M[R]_m) (H : lower1 M) :
  fast_invmx M = invmx M.
```

Once that this task is fulfilled, we can define the computation of the inverse of matrices encoded as sequences of sequences using the CoqEAL library [1].

```
Fixpoint cfast_invmx (m : nat) (M : seqmatrix CR) :=
  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.
```

The correctness of this algorithm is expressed through the following morphism lemma, stating that the concrete representation of the inverse of a matrix is the concrete inverse of its concrete representation:

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

Now, we can use this algorithm to effectively compute the inverse of lower triangular matrices with ones in the diagonal.

With this algorithm we can handle matrices up to size 20×20.

1.3 Haskell algorithm

Using the extraction mechanism of Coq, we can obtain the Haskell code for our algorithm to compute the inverse of matrices. Some modification are necessary in the code extracted, but can be considered quite direct.

Using the Haskell algorithm we can compute the inverse of matrices of approximately 10000×10000 in a few seconds.

2 Isomorphic Vector Spaces

In this section, we present the formalization in Coq/SSReflect of the mathematical result which says that two vector spaces of the same dimension are isomorphic.

2.1 Motivation and previous work

The notion of reduction is an instrumental notion in the Effective Homology Theory [8].

Definition 1 A *reduction* ρ between two chain complexes C_* and D_*, denoted in this report by $\rho : C_* \Rrightarrow D_*$, is a triple $\rho = (f, g, h)$

$$\begin{tikzcd} C_* \arrow[loop left, "h"] \arrow[r, bend left, "f"] & D_* \arrow[l, bend left, "g"] \end{tikzcd}$$

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

1) $f \circ g = Id_{D_*}$;

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

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

The importance of reductions lies in the following fact.

Theorem 2 Let $C_* \Rrightarrow D_*$ 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.

In the work [2], we presented a formalization of homology groups working with vector spaces in Coq/SSReflect.

In addition, we have defined the notion of reduction in such a theorem prover.

```
(* Chain Complex definition *)

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
  }.

(* Chain Complex Morphism definition *)

Notation
```

```
 "'d1' C" := (d1 C) (at level 0, format "'[' 'd1' C ']'").
Notation
 "'d2' C" := (d2 C) (at level 0, format "'[' 'd2' C ']'").

Definition is_ChainComplexMorphism_VS (K : fieldType)
   (C D : ChainComplex_VS K) (f0 : 'Hom((V0 C),(V0 D)))
   (f1 : 'Hom((V1 C),(V1 D))) (f2 : 'Hom((V2 C),(V2 D))) :=
   ((f0 \o d1(C)) = (d1(D) \o f1))%VS /\ ((f1 \o d2(C)) =
   (d2(D) \o f2))%VS.

Record ChainComplexMorphism_VS (K : fieldType)
 (C D : ChainComplex_VS K) :=
 { f0_VS : 'Hom((V0 C),(V0 D));
   f1_VS : 'Hom((V1 C),(V1 D));
   f2_VS : 'Hom((V2 C),(V2 D));
   CCM_VS_proof: is_ChainComplexMorphism_VS f0_VS f1_VS f2_VS
  }.

(* Homotopy operator definition *)

Record HomotopyOperator_VS (K : fieldType)
 (C : ChainComplex_VS K) :=
 { h0_VS : 'Hom((V0 C),(V1 C));
   h1_VS : 'Hom((V1 C),(V2 C))
  }.

(* Reduction definition *)

Notation
 "'d1' C " := (d1 C) (at level 0, format "'[' 'd1' C ']'").
Notation
 "'d2' C" := (d2 C) (at level 0, format "'[' 'd2' C ']'").
Notation
 "f '0'" := (f0_VS f) (at level 0, format "'[' f '0' ']'").
Notation
 "f '1'" := (f1_VS f) (at level 0, format "'[' f '1' ']'").
Notation
 "f '2'" := (f2_VS f) (at level 0, format "'[' f '2' ']'").
Notation
 "h 'o0'" := (h0_VS h) (at level 0, format "'[' h 'o0' ']'").
Notation
 "h 'o1'" := (h1_VS h) (at level 0, format "'[' h 'o1' ']'").

Definition is_Reduction_VS (K : fieldType)
  (C D : ChainComplex_VS K) (f : ChainComplexMorphism_VS C D)
```

```
  (g : ChainComplexMorphism_VS D C) (h : HomotopyOperator_VS C)
     :=
   ((f)0 \o (g)0 = \1)%VS /\
   ((f)1 \o (g)1 = \1)%VS /\
   ((f)2 \o (g)2 = \1)%VS /\
   (d2(C) \o (h)o1) \+ ((h)o0 \o d1(C)) \+ ((g)1 \o (f)1) = \1 /\
   ((h)o1 \o (h)o0 = \0)%VS /\
   ((f)1 \o (h)o0 = \0)%VS /\
   ((f)2 \o (h)o1 = \0)%VS /\
   ((h)o0 \o (g)0 = \0)%VS /\
   ((h)o1 \o (g)1 = \0)%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
  }.
```

It is worth noting that we do not work with a complete chain complex, but just with the minimal part to compute a homology group; that is three vector spaces and two morphism between them. The rest of definitions (chain complex morphisms, homotopy operators and reductions) are consistent with this representation.

Moreover, we have proved in Coq/SSReflect that given a reduction $\rho : C_* \Rrightarrow D_*$ the dimension of $H_*(C_*) = H_*(D_*)$. This result is stated in the following way.

```
Variables (K : fieldType) (C D : ChainComplex_VS K)
  (rho : Reduction_VS C D).

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

where `Betti C` computes the dimension of the homology group associated with `C`.

Therefore, if we are able to prove that two vector spaces with the same dimension are isomorphic, we have the result stated in Theorem 2.

2.2 Formalization

Vector spaces are internally encoded as matrices; and the dimension of a vector space is the rank of the matrix associated with it.

Therefore, to prove that two vector spaces with the same dimension are isomorphic, we firstly prove a result about matrices, which says that given two matrices `M` and `M'` with the same rank; then `M *m invmx (row_ebase M)*m row_ebase M' = M'` where `row_ebase` is the extended row base of a matrix.

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

Definition base_change m (M M' : 'M[K]_m) :=
  invmx (row_ebase M) *m row_ebase M'.

Lemma base_changeP m (M M' : 'M[K]_m) (hM : \rank M = \rank M') :
 (M *m base_change M M' == M')%MS.

(* Any two vector spaces of the same dimension are isomorphic *)
Lemma iso (V1 V2 : {vspace V}) (hdim : \dim V1 = \dim V2) :
  exists (f : 'End(V)), bijective f /\ (f @: V1 == V2)%VS.
```

3 Elementary collapses

In this section, we introduce the formalization of elementary collapses in Coq/SSReflect.

3.1 Motivation and previous work

The notion of elementary collapses is presented in Section 2.4 of the book [6]. Elementary collapses are a method which allows one to reduce the number of simplexes (cubes) of a simplicial (cubical) complex but preserving the homology of the object.

We have noticed that an elementary collapse is just a particular case of an admissible discrete vector field. Therefore, we do not need to develop a new whole Coq/SSReflect theory about elementary collapses but we can reuse the previous work done for admissible discrete vector fields.

Given a simplicial complex, an elementary collapse is a pair (σ, τ) such that σ and τ are simplexes of dimension n and $n+1$, σ is a face of τ and σ is not face of any other simplex of the simplicial complex.

Elementary collapses can be obtained from the incidence matrices. Namely, the rows of an incidence matrix which consists of one 1 and the rest elements 0s comes from elementary collapses.

As we have explained in Subsection 1.1, when we have a matrix M and a discrete vector field on it (in this case coming from an elementary collapse), we can reorder the lines and columns of the matrix as follows:

$$\left(\begin{array}{c|c} \epsilon & \phi \\ \hline \psi & \beta \end{array}\right)$$

and construct the reduced matrix $\widehat{d_q}$ as $\beta - \psi\epsilon^{-1}\phi$. In the case of elementary collapses, we have that ϕ is a null line, therefore, the reduce matrix is just β.

3.2 Formalization

First of all, we have defined the function in charge of looking for a collapse in a matrix (represented as a sequence of sequences). Such a function is called `find_collapse_rows` and search a line of the matrix with just one 1 and the rest of the elements null.

```
Fixpoint search (T : eqType) (a : pred (seq T)) (s : seq (seq T))
     :=
match s with
| nil => nil
| x :: s' => if a x then x else (search a s')
end.

Definition find_collapse_rows (s : seqmatrix Z2) :=
   search (fun i => (count (@pred1 Z2 1) i) == 1%N) s.
```

Subsequently, we define a function to find the position of such a line (`position_collapse_aux`), and another one to generate the relations which are necessary to construct the admissible discrete vector field (`generate_relations`).

```
Definition position_collapse_aux (s : seqmatrix Z2) (l : seq Z2)
    :=
  (pair (index l s) (index 1 l)).

Definition generate_relations i j M :=
   map (fun s => i::s::nil) (filter (fun m => ((compij m j M) !=
       0) && (i != m)) (iota 0 (size M))).
```

Afterwards, we prove that elementary collapses really produce admissible discrete vector fields.

```
Lemma Vecfieldadm_collapse M m n: (is_matrix m n M) -> (M ==
    [::]) = false -> (find_collapse_rows M == [::]) = false ->
   let pos_collapse := (position_collapse_aux M (
       find_collapse_rows M)) in
       Vecfieldadm M [::pos_collapse] (generate_orders (fst
           pos_collapse) (snd pos_collapse) M).
```

As a result of that, we can reuse the previous development about discrete vector fields, in particular, the function `getMatrixReduced` which produces the reduced matrix as explained at the beginning of this section and `reorderM_dvf` which reorders a matrix given an admissible discrete vector field.

```
Definition reduce_with_collapse_step1 (s : seqmatrix Z2) collapse
     :=
  getMatrixReduced 1 (reorderM_dvf [::(position_collapse_aux s
      collapse)] s).
```

The method of elementary collapses can be applied several consecutive times, to this aim, we have defined the following function.

```
Function reduce_with_collapses (M : seqmatrix Z2) {measure (fun M
     => (size M))} : (seqmatrix Z2) :=
if M == nil then
   M
else
 let collapse := find_collapse_rows M in
     if (collapse == nil)
     then M
     else reduce_with_collapses (reduce_with_collapse_step1 M
         collapse).
```

The main particularity of such recursive function is that it has been defined using the command `Function` instead of `Fixpoint`. This is necessary because we need to provide a measure to ensure that such a function finishes. That measure is the size of the input matrix which is reduced in each recursive step since every time that we have an elementary collapse, the size of the matrix produced by `reduce_with_collapse_step1` is the size of the input matrix minus one.

The combination of this method and the one presented in [5] can considerably reduce the size of the matrices making the computation of homology groups faster. It is worth noting that the combination of the methods is better that the application of just one of them. On the one hand, the theory of elementary collapses just removes "geometric" collapses but using the method introduced in [5] we can also removes "algebraic" ones. On the other hand, the main problem of the method of [5] is the computation of big inverse matrices; then, if we can firstly reduce the matrices using the method of elementary collapses, we will deal with the inverse of smaller matrices.

4 Persistent Homology

4.1 Motivation

The motivation of this part of the work was given in the talk [3] which was presented in the Programming Logic Seminar during the stay of J. Heras in Göteborg.

4.2 Basis algorithm

Definition of an algorithm to compute a basis of a given matrix.

```
Fixpoint comp (n : nat) : 'rV[R]_n -> 'rV[R]_n -> bool * (R * R)
    :=
  match n return 'rV[R]_n -> 'rV[R]_n -> bool * (R * R) with
  | S _ => fun (C D : 'rV[R]_(1 + _)%N) =>
     if lsubmx C == 0
        then if lsubmx D == 0 then comp (rsubmx C) (rsubmx D) else
             (false,(C 0 0,D 0 0))
        else (true,(C 0 0,D 0 0))
```

```
  | 0 => fun _ _ => (true,(0,0))
  end.

End Comp.

Notation "A >= B" := (comp A B).1 : comp_scope.
Notation "A == B" := ((A >= B) && (B >= A))%CS : comp_scope.
Notation "A > B" := ((A >= B) && ~~ (B >= A))%CS : comp_scope.

Section basis.

Variable K : fieldType.

Fixpoint insert (m n : nat) (r1 : 'rV[K]_n) : 'M[K]_(m,n) -> 'M[K
    ]_(m.+1,n) :=
  match m return 'M[K]_(m,n) -> 'M[K]_(1 + m,n) with
  | S _ => fun (M : 'M[K]_((1 + _)%N,n)) =>
     let r2 := row 0 M in
     if (r1 > r2)%CS
        then col_mx r1 M
        else if (r1 == r2)%CS
             then let: (a,b) := (comp r1 r2).2 in
                  col_mx r2 (insert (r1 - a *: (b^-1 *: r2)) (
                      dsubmx M))
             else col_mx r2 (insert r1 (dsubmx M))
  | 0 => fun _ => r1
  end.

Fixpoint basis (m n : nat) : 'M[K]_(m,n) -> 'M[K]_(m,n) :=
  match m return 'M[K]_(m,n) -> 'M[K]_(m,n) with
  | S _ => fun (M : 'M[K]_((1 + _)%N,n)) => insert (row 0 M) (
      basis (dsubmx M))
  | 0 => fun _ => 0
  end.
```

In order to prove the correctness of basis algorithm, we firstly verify that given a matrix M it generates the same row space that M.

```
Lemma eq_basis : forall m n (M : 'M[K]_(m,n)), (basis M :=: M)%MS
    .

Lemma eq_basis_row_base m n (M : 'M[K]_(m, n)) : (basis M :=:
    row_base M)%MS.
```

4.3 Kernel algorithm

Abstract version of the kernel algorithm:

```
Fixpoint ker_step (m n : nat) {struct n} :=
  match n return
     'M[K]_(1+m,1+n) -> ('M[K]_(1+n) * 'M[K]_(m,n) * K * 'cV[K]_m)
         +
                       ('M[K]_(m, 1 + n)) with
  | S n' => fun (M: 'M[K]_(1 + _, 1 + _)) =>
     let: a := M 0 0 in
       if a == 0 then
         match ker_step _ _ (rsubmx M) with
           | inl (((P,R),a),c) =>
               (* let: P' := tperm_mx 0 1 *m block_mx (1%:M: 'M_1)
                   0 0 P *)
               let: P' := xcol 1 0 (block_mx (1%:M: 'M_1) 0 0 P)
               in inl (((P',row_mx (dlsubmx M) R),a),c)
           | inr R => inr (row_mx (dlsubmx M) R)
         end
       else
         let R := drsubmx M - (a^-1) *: (dlsubmx M *m ursubmx M)
             in
         let D := block_mx 1%:M (-(a^-1)*: (ursubmx M)) 0 1%:M in
           inl (((D,R),a),dlsubmx M)
  | _ => fun (M: 'M[K]_(1 + m,1 + 0)) =>
           let: a := M 0 0 in
             if a == 0 then
               inr (dsubmx M)
             else
               inl (((1%:M,0),a),dsubmx M)
   end.

Fixpoint ker_dep (m n:nat) {struct m} :=
  match n,m return 'M[K]_(m,n) -> {p: nat & 'M[K]_(p,m)} with
      | S n', S m' => fun (M: 'M[K]_(1 + _, 1 + _)) =>
         match ker_step M with
           | inl (((P,R),a),c) =>
               let (q,Y) := ker_dep _ _ R in
                 existT (fun p => 'M[K]_(p,1+m'))
                        q (row_mx (- (a^-1) *: (Y *m c)) Y)
           | inr R => let (q,Y) := ker_dep _ _ R in
                 existT (fun p => 'M[K]_(p,1+m'))
                        (q.+1) (block_mx (1%:M: 'M_1) 0 0 Y)
         end
      | O,_ => fun _ => existT (fun p => 'M[K]_(p,m)) m (1%:M)
      | _,_ => fun _ => existT (fun p => 'M[K]_(p,0)) 0 0
  end.
```

```
Definition ker (m n : nat) (M : 'M[K]_(m,n)) := projT2 (ker_dep M
    ).
```

The main properties of the `ker` algorithm are the following ones.

```
Lemma eqmx_ker m n (M : 'M[K]_(m,n)) : (ker M :=: kermx M)%MS.

(* A <= B <-> exists Y, A = Y *m B *)

Lemma sub_kerP m n p (M : 'M[K]_(m,n)) (X : 'M[K]_(p,m)) :
  reflect (X *m M = 0) (X <= ker M)%MS.

Lemma ker_row_free : forall m n (M: 'M[K]_(m,n)), row_free (ker M
    ).
```

Executable version of the kernel, which is straightforward from the abstract one.

```
Fixpoint ker_seqmx_step (m n : nat) (M : seqmatrix CK)
  : (seqmatrix CK * CK * seqmatrix CK) + (seqmatrix CK) :=
  match n with
  | S n' =>
      let a := M 0 0 in
      if a == zero CK then
         match ker_seqmx_step m n' (rsubseqmx 1 M) with
           | inl ((R,a),c) => inl ((row_seqmx (dlsubseqmx 1 1 M) R
               ,a),c)
           | inr R => inr (row_seqmx (dlsubseqmx 1 1 M) R)
         end
      else
         let R := subseqmx (drsubseqmx 1 1 M) (scaleseqmx (cinv a)
                      (mulseqmx (dlsubseqmx 1 1 M) (ursubseqmx 1 1
                          M))) in
           inl ((R,a),dlsubseqmx 1 1 M)
  | _ =>
      let a := M 0 0 in
      if a == zero CK then inr (dsubseqmx 1 M) else inl ((seqmx0
          CK m n,a),dsubseqmx 1 M)
  end.

Fixpoint ker_seqmx_dep (m n:nat) (M : seqmatrix CK) {struct m} :
    seqmatrix CK :=
  match n,m with
    | S n', S m' =>
         match ker_seqmx_step m' n' M with
           | inl ((R,a),c) =>
               let Y := ker_seqmx_dep m' n' R in
               row_seqmx (scaleseqmx (opp (cinv a))
```

```
                    (* A trick to avoid the problem with zero
                       size in mulseqmxE... *)
                    (if m' == 0 then seqmx0 _ (size Y) 1 else
                       mulseqmx Y c)) Y
          | inr R => let Y := ker_seqmx_dep m' n R in
                    block_seqmx (seqmx1 _ 1) (seqmx0 _ 1 m') (
                       seqmx0 _ (size Y) 1) Y
        end
      | 0,_ => seqmx1 _ m
      | _,_ => seqmx0 _ 0 0
  end.
```

We prove that the abstract and concrete versions of the kernel are equivalent module a change of domain.

```
Lemma ker_seqmx_depE : forall m n (M : 'M[K]_(m,n)),
  seqmx_of_mx _ (ker M) = ker_seqmx_dep m n (seqmx_of_mx _ M).
```

4.4 Formalization of persistent homology

Definition of a Persistent Homology groups.

```
Variable (K : fieldType).
Variables (V1 V2 V3 V4 : vectType K)
          (f : linearApp V1 V2) (g : linearApp V3 V4)
          (i : linearApp V1 V4).

Hypothesis (i_inj : injective i).

Definition PHomology := ((i @: (lker f)) :\: ((limg g) :&: (i @:
    (lker f))))%VS.
```

A explicit formula to compute the persistent betti numbers.

```
Definition PBetti := \dim PHomology.

Lemma PBettiE_2 :
  PBetti = vdim V1 - \dim (limg f) - (\dim (limg g) +
           (vdim V1 - \dim (limg f)) - \dim ((limg g) + (i @: (
               lker f)))).
```

Vector spaces as SSReflectmatrices, so to compute the dimension we just compute the rank.

```
Variable K : fieldType.
Variable (v1 v2 v3 v4 : nat).

Definition V1 := (matrixVectType K 1 v1).
Definition V2 := (matrixVectType K 1 v2).
```

```
Definition V3 := (matrixVectType K 1 v3).
Definition V4 := (matrixVectType K 1 v4).

Variable (mxf: 'M[K]_(vdim V1,vdim V2)) (mxg : 'M[K]_(vdim V3,
    vdim V4))
         (mxi : 'M[K]_(vdim V1,vdim V4)).

Definition PBetti_rank :=
   (v1 - \rank mxf - (\rank mxg + (v1 - \rank mxf) -
   \rank (col_mx mxg (projT2 (ker_dep mxf) *m mxi))))%N.
```

Correctness of `PBetti_rank`.

```
Lemma dimHomologyrankE : injective (LinearApp mxi) ->
  PBetti_rank = PBetti (LinearApp mxf) (LinearApp mxg) (LinearApp
       mxi).
```

Computable version of persistent betti numbers.

```
Variable (K : fieldType).
Variable (CK : cunitRingType K).
Variable (v1 v2 v3 v4 : nat).

Definition W1 := (matrixVectType K 1 v1).
Definition W2 := (matrixVectType K 1 v2).
Definition W3 := (matrixVectType K 1 v3).
Definition W4 := (matrixVectType K 1 v4).

Definition ex_PBetti (mxf mxg mxi : seqmatrix CK) :=
  let rf := rank v1 v2 mxf in
  let rg := rank v3 v4 mxg in
  v1 - rf - (rg + (v1 - rf) -
  rank (v3 + size (ker_seqmx_dep v1 v2 mxf)) v4
       (col_seqmx mxg (mulseqmx (ker_seqmx_dep v1 v2 mxf) mxi))).
```

Correctness of `ex_PBetti`.

```
 Lemma ex_PBetti_rank_PBetti_rank_E :
  forall (mxf: 'M[K]_(vdim W1, vdim W2)) (mxg : 'M[K]_(vdim W3,
      vdim W4))
         (mxi : 'M[K]_(vdim W1, vdim W4)),
   ex_PBetti (seqmx_of_mx CK mxf) (seqmx_of_mx CK mxg) (
       seqmx_of_mx CK mxi) =
   PBetti_rank mxf mxg mxi.
```

We reuse previous developments: incidence matrices [4], homology [2], CoqEAL library [1].

We define the notions of filtration of simplicial complexes:

```
Variable V : finType.

(* Filtration definifition *)
Definition filtration (f : seq {set simplex V}) :=
    (forall x, x \in f -> simplicial_complex x) /\
    (forall i j, i <= j -> i < size f -> j < size f -> (nth set0 f
         i) \subset (nth set0 f j)).
```

inclusion matrices:

```
Variables Left Top : seq (simplex V).

Definition inclusionMatrix :=
  \matrix_(i < m, j < n)
    if (nth set0 Left i == nth set0 Top j) then 1 else 0:bool.

Variable (V:finType) (p k i:nat) (f: (seq {set (simplex V)})).
Hypothesis f_is_filtration : filtration f.
Hypothesis i_is_in_filtration : i < size f.
Hypothesis i_add_p_is_in_filtration : i+p < size f.

Definition inclusion_mx :=
  inclusionMatrix (vdim (incidencematrices.V1 (nth set0 f i) k
      .+1))
               (vdim (incidencematrices.V2 (nth set0 f (i + p))
                   k))
               (enum (n_k_simplices i))
               (enum (n_k_simplices (i+p))).
```

We prove that the linear application associated with the inclusion matrix is injective.

```
Lemma injective_LinearApp_inclusion_mx : injective (LinearApp (
    inclusion_mx)).
```

Subsequently, we define persistent betti numbers associated with a filtration and prove the correctness of such a definition.

```
Section persistent_incidence_mx.

Open Local Scope ring_scope.

Variable (V:finType) (p k i:nat) (f: (seq {set (simplex V)})).
Hypothesis f_is_filtration : filtration f.
Hypothesis i_is_in_filtration : i < size f.
Hypothesis i_add_p_is_in_filtration : i+p < size f.

Definition p_persistent_k_betti_K_i :=
```

```
 PBetti_rank (incidence_mx_n (nth set0 f i) k.+1)
               (incidence_mx_n (nth set0 f (i+p)) k)
               (inclusion_mx p k i f).

Lemma p_persistent_k_betti_K_iE :
  p_persistent_k_betti_K_i =
  PBetti (LinearApp (incidence_mx_n (nth set0 f i) k.+1))
              (LinearApp (incidence_mx_n (nth set0 f (i+p)) k))
              (LinearApp (inclusion_mx p k i f)).
Proof.
rewrite /p_persistent_k_betti_K_i dimHomologyrankE //.
by apply : injective_LinearApp_inclusion_mx.
Qed.
```

We define the executable counterpart of inclusion matrices.

```
Variable V : finType.
Variable leT : rel V.

Hypothesis tr_leT : transitive leT.
Hypothesis irr_leT : irreflexive leT.
Hypothesis t_leT : total leT.

Variable filtration_faces : seq (seq (seq V)).
Variable n i p: nat.

Hypothesis Hisize : i < size filtration_faces.
Hypothesis Hipsize : i+p < size filtration_faces.

Hypothesis Hfaces : all (fun s => all (sorted leT) s)
    filtration_faces.
Hypothesis Hfaces_uniq : all (fun s => all uniq s)
    filtration_faces.


Definition ex_inclusion_mx :=
  let ex_boundaries := (ex_n_simplices (nth nil filtration_faces
      (i+p)) n) in
  (map (fun x => map (fun y => if x == y then 1 else 0:bool)
      ex_boundaries) (ex_n_simplices (nth nil filtration_faces i)
      n)).


Definition InM := inclusionMatrix
                    (vdim (incidencematrices.V2 (sc_set (nth nil
                        filtration_faces i)) n))
```

```
                    (vdim (incidencematrices.V2 (sc_set (nth nil
                        filtration_faces (i+p))) n))
                    (n_simplices_seq (nth nil filtration_faces i)
                        n)
                    (n_simplices_seq (nth nil filtration_faces (i+
                        p)) n).
```

Correctness of the executable version of inclusion matrices:

```
Lemma InME : ex_inclusion_mx = seqmx_of_mx _ InM.
```

Executable persistent betti numbers of a filtration.

```
Definition ex_p_persistent_k_betti_K_i :=
 ex_PBetti
   (size (ex_incidence_mx (nth nil f i) k.+1))
   (size (rowseqmx (ex_incidence_mx (nth nil f i) k.+1) 0))
   (size (ex_incidence_mx (nth nil f (i+p)) k))
   (size (rowseqmx (ex_incidence_mx (nth nil f (i+p)) k) 0))
   (ex_incidence_mx (nth nil f i) k.+1)
   (ex_incidence_mx (nth nil f (i+p)) k)
   (ex_inclusion_mx f k i p).
```

References

[1] M. Dénès, A. Mörtberg, and V. Siles. A refinement-based approach to computational algebra in Coq. In *Proceedings Interactive Theorem Proving 2012 (ITP'2012)*, Lectures Notes in Computer Science, 2012.

[2] 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'2012)*, volume 7309 of *Lectures Notes in Computer Science*, pages 49–57, 2012.

[3] J. Heras, G. Mata, and J. Rubio. Neuronal structure detection using persistent homology, 2012. `https://esus.unirioja.es/psycotrip/archivos_eventos/stacks_persistence.pdf`.

[4] 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'2011)*, volume 6824 of *Lectures Notes in Computer Science*, pages 30–44, 2011.

[5] J. Heras, M. Poza, and J. Rubio. Verifying an algorithm computing Discrete Vector Fields for digital imaging. In *Proceedings Calculemus 2012*, volume 7362 of *Lectures Notes in Computer Science*, pages 215–229, 2012.

[6] T. Kaczynski, K. Mischaikow, and M. Mrozek. *Computational Homology*, volume 157 of *Applied Mathematical Sciences*. Springer, 2004.

[7] A. Romero and F. Sergeraert. Discrete Vector Fields and Fundamental Algebraic Topology, 2010. `http://arxiv.org/abs/1005.5685v1`.

[8] J. Rubio and F. Sergeraert. Constructive Algebraic Topology. *Bulletin des Sciences Mathématiques*, 126(5):389–412, 2002.