Final Report for Deliverable Nr. 1.5

Sub-resultants, Multivariable GCD Computations

Responsible: Assia Mahboubi, Assia.Mahboubi@inria.fr
Site: INRIA, France

Deliverable Date: February 2012
Actual Date: July 2013

1 Content of the deliverable

This deliverable reports on our work on the certification of symbolic computations of greatest common divisors (GCD) for multivariate polynomials. Our contribution is organized along two directions. First, we consider the problem of computing common divisors of multivariate polynomials using the Coq system both as a programming language and as a proof assistant. The second direction deals with the formal study of algorithms devoted to the efficient computation of pseudo-remainder sequences: these algorithms generalize the Euclidean algorithm to the case of polynomials with coefficients in an integral domain (as opposed to a field for the case of the Euclidean algorithm). These two perspectives raise distinct issues that we discuss in this deliverable. In section 2, we describe our experiments using certified computations of common divisors of multivariate polynomials. We deal with rather large polynomials, in the context of a development which combines hand-written scripts with algebraic data generated by a computer algebra system. In section 3, we describe our experiments about the formalization of subresultant coefficients, and the properties of the related pseudo-remainder sequences.

2 Multivariate GCD computations inside Coq

Collaborations and dissemination This work is a joint work of Assia Mahboubi with collaborators outside the Formath project, namely: Frédéric Chyzak (Inria, France), Thomas Sibut-Pinote (Éns de Lyon, France) and Enrico Tassi (Inria, France). It started as the undergraduate research internship Thomas Sibut-Pinote did under the joint supervision of Frédéric Chyzak and Assia Mahboubi. Assia Mahboubi has presented early versions of this work as a seminar at the Computer Science Laboratory of the École Normale Supérieure de Lyon, at the CACOS'12 international workshop satellite of the ISSAC'12 conference (Grenoble, France) and at the Formath day organized as a satellite workshop of the MAP 2012 conference. Thomas Sibut-Pinote's report (in English) is available at the following URL:

http://perso.ens-lyon.fr/thomas.sibutpinote/rapport_zeta_3.pdf

Since this work was only very recently completed, we do not have yet tried to publish it but we plan to submit a description of this work for the next edition of the relevant international conferences.

Context The Coq system implements a type theory that embed a real (purely functional) programming language. The developers of the Coq system have devoted a continuous effort to the improvement of the efficiency of this programming language [6, 2]. Coq can hence be used as a programming language to define Coq terms that are used as programs and evaluated inside the logic. These functions can also be certified by formal proofs of correctness properties which produce correctness theorems for *every* value obtained by a run of the function. This is now a standard way to implement in Coq formal proof producing automated decision procedures, based on Coq implementations of decision algorithms. Here we consider the computations that are the core of automated decision procedures comparing expressions based on the signature of a field structure and deciding their equality modulo the axioms of field. This decision procedure, called field, is available in the standard distribution of the Coq system [12]. Based on an original work by Delahaye and Mayero [3], it has been later re-implemented and improved by Grégoire and Théry [12]. This tactic is based on a so-called *reflexion scheme* and on an other proof producing decision procedure for the (simpler) signature and theory of rings [7].

Application Symbolic operations on rational fractions like putting terms over a common denominator or canceling common factors are basic and important components of computer algebra systems (CAS). We have evaluated the performance and usability of the field tactic used as a formal counterpart of these CAS tools in a work which combines hand-written proof scripts with data generated by a CAS (here Maple) used as an oracle. More precisely, our aim is to prove formally the irrationality of the $\zeta(3)$ constant. This result is originally due to Apéry [10] and the original proof crucially relies on the observation that some sequences expressed as indefinite sums of hypergeometric sequences satisfy the same linear recurrence of order two. Validating this observation was left as an exercise by Apéry in its original presentation and proved a difficult task that conferred a controversial status to Apéry's claim for a while [9]. For the two past decades, impressive progress have been made in the field of symbolic algorithms for *D-finite* functions, that is the functions (sequences or complex) that obey linear functional equations (recurrences or differential equations) with polynomial coefficients. In particular, the effective proofs that this class of objects is closed under various operations (like sum, product, indefinite sum...) have been turned into efficient algorithms and implemented as libraries available for mainstream CAS like Maple or Mathematica [8]. For instance, if enough recurrence relations are known on two sequences $(u_n)_{n\in\mathbb{N}}$ and $(v_n)_{n\in\mathbb{N}}$, it is possible to guess the recurrence relations of which $(w_n)_{n\in\mathbb{N}}$ is solution when $w_n := u_n+v_n$. Similarly, one can compute the recurrences canceling $(U_n)_{n\in\mathbb{N}}$ with $U_n := \sum_{k=0}^{n} u_{n,k}$ from the ones canceling the summand $(u_{n,k})_{n,k\in\mathbb{N}}$.

It is hence possible to replace the hard work of the referees of Apéry's original proof by symbolic computations run on a computer, that validate the fact that the two sequences of interest are actually canceled by the same recurrence [11]. Our motivation here is to devise a

complete formal proof of the irrationality of the $\zeta(3)$ constant based on the certification of the computer-algebra version of the proof. We would like to insist on the fact that formalizing this proof does not only boil down to a routine check of symbolic computations. Actually, computer algebra algorithms rely on algebraic idealizations of objects, replacing for instance sequences by germs of sequences, and that a precise hand-written formal proof of correctness of the symbolic computations described for instance in the Maple worksheet "proving" the irrationality of the $\zeta(3)$ is not available and requires filling some gaps in the proof that are nowhere described in the literature. However these topics are out of the scope of the present deliverable. We focus here on the fact that an important part of the cost of verifying the steps obtained algorithmically consists in normalizing rational fractions with coefficients in the field of rational numbers.

Contributions We have completed a formal proof of the recurrence relations for Apéry's sequences with the Coq system.

The formal development is standalone and some of the data it features, namely intermediate recurrence relations for auxiliary sequences have been obtained by a script copying the results computed by Maple and the Algolib package. We have hence used the CAS as an untrusted external oracle to explore and guess auxiliary results, that are later formally checked by formal proof producing normalization of expressions, and in particular of rational fractions.

The rational fractions we manipulate are large objects (the largest instances are stored in a 3GB file) but this is only due to a large number of monomials: degrees are rather small (about 10) and so is the size of the constant coefficients (less that 10^6) as well as the number of variables (less than 20). On these instances, a CAS like Maple answers in less than one or two seconds. On the largest instances, the field tactic takes 200 seconds to answer[1], but on the many much smaller instances it is instantaneous.

Comparing the timings observed when using a CAS on one hand and when calling the field tactic on the other hand does not make much sense without being more precise on the ingredients that contribute to the time measured on a run of the tactic. There are actually three main possibly expensive steps in such a run:

- the cost of a change of representation. When the field tactic is called, the Coq current goal is translated into an element of a data structure adapted to computations. After the normalization process is completed, a syntactic check is performed to ensure that the translation was correct, and hence led to a correct normalization of the original goal.

- the cost of arithmetic computation. The cost of the normalization phase of a polynomial or rational fraction depends on the efficiency of the arithmetic operations on constants (integers or rational numbers in our case).

- the efficiency of the normalization algorithms. The field tactic actually depends on two normalization algorithms used independently: one for normalizing polynomial expres-

[1] on a Linux 64 bits machine, with an Intel Core i7 processor with 9Gio RAM

sions, and one for normalizing rational fractions. The second one actually essentially puts fractions over a common denominator, simplifying obvious common factors.

Changing the representation of the Coq terms involved in the equality to be checked, in order to make them amenable to computation, and checking the correctness of this translation is a process linear in the size of the problem. Hence we were expecting to be exposed to limitations due to either the cost of the arithmetic operations or the efficiency of the normalization algorithms. However, it turned out that the step requiring most efforts was actually the change of representation step. Indeed the libraries about integer and rational arithmetic shipped with the standard distribution of Coq proved sufficiently efficient for our needs. The efficiency of the normalization procedures were also no bottleneck, but we certainly benefited from the fact that we are working with algebraic objects generated by a computer algebra system oracle. In our case it was possible to generate the polynomial expressions in their most convenient form for our needs. For instance, we required the rational fractions generated to be collected under a common denominator and to be as factored as possible, which significantly eased the task of the field tactic.

In our case the change of representation triggered by calls to the field tactic involve both a syntaxification step and a change in the representation of arithmetic constants. Obtaining a truly linear behavior has been an issue for several reasons, mostly because of a combination of the lack of modularity of the user interfaces proposed by the Coq normalization tactics on one hand, and the efficiency/control issues of the reduction mechanisms implemented by the Coq system for its programming language. We also had to correct some bugs in the ring and field tactics, that were minor issues which become blocking when dealing with such large instances. One of the specifics of the instances on which call the ring and field tactics is the versatility of (convertible) forms that can take the operations and constants of a ring or field instance. This versatility is due to the use of the generic notations and theories proposed by the hierarchy of algebraic structures of the Mathematical Components library [5]. In order to obtain the time results mentioned above, we preprocessing code to ease the syntaxification of the expressions we manipulate and that are composed with these generic operators. This code is of general interest for users of the Mathematical Components library.

Conclusion and perspectives This experiment on the formal proof of identities and recurrences discovered by computer algebra algorithmic means is very promising. We plan to investigate how the approach scales on other emblematic success of this kind.

On the other hand our experiments on embedding efficient computations with multivariate rational polynomials took a different direction than the one expected at the time we described the objectives of this task. We think that the approach based on refinements developed in the context of the project by Dénès, Mörtberg and Silès [4] and mentioned in deliverable 4.3 should apply to standard algorithms for multivariate GCD computations. However our experiments indicate that other issues have to be addressed for these computations to be really useful inside decision procedures. Hence improving the normalization algorithms implemented by the field tactic seemed to us a lower priority than investigating the issues raised by the control of the reduction strategies available to the user of the Coq system.

These issues have also been raised by Magron when working with the certification of

polynomial approximations for the certification of inequalities involving transcendental functions, see deliverable 4.4. In his case, adopting an alternative reduction mechanism recently integrated to the Coq system [2], based on native compilation was necessary.

We also think that the specific preprocessing code that we have developed should be replaced by a more global work on the interfaces proposed by the Coq normalization tactics like ring, field, omega... These tools actually currently suffer a lack of genericity and are too much tight to a specific representation of arithmetic.

3 Subresultants

Collaborations and dissemination This work is a joint work of Cyril Cohen (University of Gothenburg, Sweden), Assia Mahboubi, Marie-Françoise Roy (University of Rennes, France) and Henri Lombardi (University of Besançon, France). Several working session between theses collaborators have been organized in the context of the Formath project. The Coq code has been developed by Cyril Cohen. This work is still unpublished.

Context We propose a formal study of algorithms devoted to the efficient computation of pseudo-remainder sequences: these algorithms generalize the Euclidean algorithm to the case of polynomials with coefficients in an integral domain (as opposed to a field for the case of the Euclidean algorithm). This time our motivation is to understand how to represent these objects and the proofs of their main properties in our Coq libraries. But, as opposed to the situation described in the first directions, this time we do not aim at implementing these efficient algorithms using Coq as a programming language: executing the programs we formally prove would require first extracting them. Our main objective here is to prove formally the properties of the subresultant algorithm, which is one of the most efficient ways of computing pseudo-remainder sequences.

Application This efficiency makes subresultant polynomials a crucial ingredient of efficient quantifier elimination algorithms for real closed field structures. This work on formal proofs about subresultant polynomials is actually part of our work on cylindrical algebraic decomposition, described in deliverable 1.6. A pseudo-remainder sequence computed from two polynomials P and Q carries an important amount of information on the polynomials P and Q. In particular, controlling the complexity of pseudo-remainder sequences plays an important role in controlling the complexity of the elimination phase.

Contributions The contribution consists here in a formal development which provides formal proofs of the properties of subresultant coefficients as presented in chapter 4 of the book [1]. We start by defining (a variant of) Sylverster-Habicht matrices, and subresultant coefficients are obtained as (minor) determinants of sub-matrices of a Sylverster-Habicht matrix. We then prove the correspondence between subresultant coefficients and GCD. Finally we establish the recurrence relation between subresultant coefficients that allows their efficient computation.

The main issue in this formalization is to deal with block matrices and lines and row operations in order to minimize the burden of dependent type bureaucracy and spurious sign alternations. These issues lead to minor divergences from the textbook presentation that are explained in the comments of the formal library. We reproduce here after the statements of the lemmas we have formally proved and the related comments. We include the several small extensions to the Mathematical libraries that were needed for this work for instance on binomial coefficients or permutations.

```
(****************************************************************************)
(* The statements and proofs in this file are largely inpired by BPR *)
(* (http://perso.univ-rennes1.fr/marie-francoise.roy/bpr-ed2-posted2.html) *)
(* *)
(* SylvesterHabitch_mx p q j == a variant of the Sylvester-Habitch matrix *)
(* (cf comment below) *)
(* subresultant p q j == a variant of the subresultant *)
(****************************************************************************)

Notation pred_class_of T := (pred_sort (predPredType T)).
Section extra_perm.

(* Remark: generalize (D : {set aT}) to (D : pred_class_of aT) in finset.v *)
(* Meanwhile, we reprove it in this more general case *)

Lemma eq_imset (aT rT : finType) (f g : aT -> rT) (D : pred_class_of aT) :
  f =1 g -> [set f x | x in D] = [set g x | x in D].

Definition perm_rev {n} : 'S_n := perm rev_ord_inj.

Lemma perm_rev0 : @perm_rev 0 = 1%g.

Lemma perm_rev1 : @perm_rev 1 = 1%g.

Lemma perm_cycle_proof n :
  injective ((fun k : 'I_n => insubd k (if k.+1 < n then k.+1 else 0%N))).

Definition perm_cycle n : 'S_n := perm (@perm_cycle_proof n).

Lemma pcycle_perm_cycle n x : pcycle (perm_cycle n) x = setT.

Definition imset_const (aT rT : finType) (A : pred_class_of aT)
  (a : aT) (x : rT) : (a \in A) -> [set x | _ in A] = [set x].

Lemma pcycles_perm_cycle n : pcycles (perm_cycle n.+1) = [set setT].

Lemma perm_cycle0 : perm_cycle 0%N = 1%g.

Lemma odd_perm_cycle n : perm_cycle n.+1 = odd n :> bool.

Lemma odd_perm_rev n : @perm_rev n = odd 'C(n, 2) :> bool.

Lemma perm_rev2 n : (perm_rev * perm_rev)%g = 1%g :> 'S_n.
```

```
End extra_perm.

Section bin2.

Lemma bin2D m n : 'C(m + n, 2) = 'C(m, 2) + 'C(n, 2) + m * n.

Lemma odd_bin2M2 n : odd 'C(n.*2, 2) = odd n.

Lemma odd_bin2D2n k n : odd 'C(k.*2 + n, 2) = (odd k (+) odd 'C(n, 2)).

Lemma bin2DB m n : odd 'C(m + n, 2) = odd (minn m n) (+) odd 'C(`|m - n|, 2).

Lemma odd_bin2_dist (m n : nat) :
  odd 'C(`|m - n|, 2) = odd (minn m n) (+) odd 'C(m + n, 2).

Lemma odd_bin2B m n : n <= m -> odd 'C(m - n, 2) = odd n (+) odd 'C(m + n, 2).

Lemma bin2B_trans n p m : p <= n <= m ->
   'C(m - p, 2) = 'C(m - n, 2) + 'C(n - p, 2) + (m - n) * (n - p).

End bin2.

Section extra_polydiv.

Variable R : idomainType.

Lemma Poly_take_drop n (p : {poly R}) :
   (Poly (take n p) = p %% 'X^n) * (Poly (drop n p) = p %/ 'X^n).

Lemma Poly_take (p : {poly R}) n : Poly (take n p) = p %% 'X^n.

Lemma Poly_drop (p : {poly R}) n : Poly (drop n p) = p %/ 'X^n.

Definition lcmp (p q : {poly R}) := (p * q) %/ gcdp p q.

Lemma mulp_gcd_lcm (p q : {poly R}) :
  gcdp p q * lcmp p q =
  lead_coef (gcdp p q) ^+ scalp (p * q) (gcdp p q) *: (p * q).

Lemma lcm0p (p : {poly R}) : lcmp 0 p = 0.

Lemma lcmp0 (p : {poly R}) : lcmp p 0 = 0.

Lemma lcmp_eq0 (p q : {poly R}) : (lcmp p q == 0) = (p == 0) || (q == 0).

Lemma dvdp_lcml (p q : {poly R}) : p %| lcmp p q.

Lemma dvdp_lcmr (p q : {poly R}) : q %| lcmp p q.

Lemma dvdp_lcm (p q r : {poly R}) : (lcmp p q %| r) = (p %| r) && (q %| r).

Lemma size_lcm (p q : {poly R}) : p != 0 -> q != 0 ->
  size (lcmp p q) = ((size (p * q)%R).+1 - size (gcdp p q))%N.
```

```
Lemma size_gcd (p q : {poly R}) : p != 0 -> q != 0 ->
  size (gcdp p q) = ((size (p * q)%R).+1 - size (lcmp p q))%N.

End extra_polydiv.

Notation perm_rev_mx := (perm_mx perm_rev).
Section extra_matrix.

Lemma mulmxP (K : ringType) (m n : nat) (A B : 'M[K]_(m, n)) :
  reflect (forall u : 'rV__, u *m A = u *m B) (A == B).

Lemma rsubmx_col (R : Type) (m n p k: nat)
  (A : 'M[R]_(m, p + k)) (B : 'M[R]_(n, p + k)) :
  rsubmx (col_mx A B) = col_mx (rsubmx A) (rsubmx B).

Lemma lsubmx_col (R : Type) (m n p k: nat)
  (A : 'M[R]_(m, p + k)) (B : 'M[R]_(n, p + k)) :
  lsubmx (col_mx A B) = col_mx (lsubmx A) (lsubmx B).

Lemma mulmx_rsub (R : ringType) (m n p k: nat)
  (A : 'M[R]_(m, n)) (B : 'M[R]_(n, p + k)) :
  A *m rsubmx B = (rsubmx (A *m B)).

Lemma mulmx_perm_rev_col (R : ringType) (m n p k: nat)
  (A : 'M[R]_(m, p + k)) (B : 'M[R]_(n, p + k)) :
  perm_rev_mx *m col_mx A B =
  castmx (addnC _ _,erefl) (col_mx (perm_rev_mx *m B) (perm_rev_mx *m A)).

End extra_matrix.

Section extra_mxpoly.

Lemma rsubmx_poly_rV (R : idomainType) (m n: nat) (p : {poly R}):
  rsubmx (poly_rV p : 'rV_(m + n)) = poly_rV (p %/ 'X^m).

Lemma poly_rV_eq0 (R : ringType) n (p : {poly R}) :
  size p <= n -> (poly_rV p == 0 :> 'rV[R]_n) = (p == 0).

End extra_mxpoly.

(*************************************************************************)
(* About Permanence Minus Variations *)
(* *)
(* Remark, what is called /variations/ in BPR, we call it /changes/. *)
(* Instead, what we call variation witnesses a strict sign change and was *)
(* defined in qe_rcf_th. *)
(*************************************************************************)

Module Import PMV.
Section PMV.
(* part of it should be backported to polyrcf/qe_rcf_th *)
(* and generalized to numDomainType *)
```

```
Lemma variationrr (R : rcfType) (x : R) : variation x x = 0.

Lemma variationE (R : rcfType) (y x : R) :
  variation x y != 0 -> variation x y = sgz y.

Lemma variation_eq0 (R : rcfType) (x y : R) :
  (variation x y == 0) = (x * y == 0) || (sgz x == sgz y).

Lemma variation_neq0 (R : rcfType) (x y : R) :
  (variation x y != 0) = (x != 0) && (y != 0) && (sgz x == - sgz y).

Lemma novariation_sgz (R : rcfType) (y x : R) :
  x != 0 -> y != 0 -> variation x y = 0 -> sgz x = sgz y.

Lemma variationNr (R : rcfType) (x y : R) :
  variation (- x) y = sgz y * (x != 0) * (variation x y == 0).

Lemma variationrN (R : rcfType) (x y : R) :
  variation x (- y) = - sgz x * (y != 0) * (variation x y == 0).

Lemma variation_sgzLR (R : rcfType) (y x : R) :
  variation x y != 0 -> sgz x = - sgz y.

Lemma variation_neq0l (R : rcfType) (y x : R) :
  variation x y != 0 -> x != 0.

Lemma variation_neq0r (R : rcfType) (x y : R) :
  variation x y != 0 -> y != 0.

Lemma variationNrr (R : rcfType) (x : R) : variation (- x) x = sgz x.

Lemma variationrNr (R : rcfType) (x : R) : variation x (- x) = - sgz x.

Lemma crossRE (R : rcfType) (p : {poly R}) :
  crossR p = sgz (lead_coef p) *+ (odd (size p).-1).

(* Notation 4.31. from BPR *)
Definition pmv (R : numDomainType) (s : seq R) : int :=
  let fix aux a n s :=
      if s is b :: s then
        if b == 0 then aux a n.+1 s
        else aux b 0%N s + (-1) ^+ 'C(n, 2) * sgz (a * b)
      else 0 in
  if s is a :: s then aux a 0%N s else 0.

Notation nonzero := (forall x, x != 0).

Fixpoint permanences (R : numDomainType) (s : seq R) : nat :=
  (if s is a :: q then (a * (head 0 q) > 0)%R + permanences q else 0)%N.

(* First remark about Notation 4.31 in BPR *)
Lemma nonzero_pmvE (R : rcfType) (s : seq R) :
  {in s, nonzero} -> pmv s = (permanences s)%:Z - (changes s)%:Z.
```

```
Lemma changes_minftyE (R : rcfType) (sp : seq {poly R}) :
  (forall i, (size sp`_i.+1) = (size sp`_i).-1) ->
  changes_minfty sp = permanences (map lead_coef sp).

(* Second remark about Notation 4.31 in BPR *)
Lemma pmv_changes_poly (R : rcfType) (sp : seq {poly R}) :
  {in sp, nonzero} -> (forall i, (size sp`_i.+1) = (size sp`_i).-1) ->
  pmv (map lead_coef sp) = changes_poly sp.

End PMV.
End PMV.

(************************************************)
(* Here begins the development on subresultants. *)
(************************************************)
Local Notation band r := (lin1_mx (poly_rV \o r \o* rVpoly)).
Section SubResultant.

Variables (R : ringType) (np nq k : nat) (p q : {poly R}).

(*************************************************************************)
(* We define the SylvesterHabitch_mx in this way, in order to be able to *)
(* reuse the poly_rV and rVpoly mappings rather than redefining custom *)
(* ones in big endian encoding and reversing some rows of the matrix when *)
(* reasonning about the nullity of its determinant. Instead, we apply *)
(* the row reversal of the lower part of the matrix in the definition of *)
(* the subresultant, and we also correct the sign. *)
(* *)
(* Note also that the matrix is generalized to arbitrary sizes. Although *)
(* it makes sense only with the instanciations given in the definition of *)
(* the subresultant, this helps stating various lemmas without any casts *)
(* (cf SylvesterHabicht_mod and SylvesterHabicht_scaler) or only one *)
(* (cf SylvesterHabicht_reduce) *)
(*************************************************************************)

Definition SylvesterHabicht_mx : 'M[R]_(np + nq, k) :=
  col_mx (band p) (band q).

Lemma SylvesterHabicht_mxE (m : 'I_(np + nq)) (n : 'I_k):
  let S_ r k := r`_(n - k) *+ (k <= n) in
  SylvesterHabicht_mx m n =
  match split m with inl k => S_ p k | inr k => S_ q k end.

End SubResultant.

(* Note: it is unclear yet whether the appropriate formulation is *)
(* ((size q).-1 - j) or (size q - j.+1) -- Cyril *)
Definition subresultant (R : ringType) j (p q : {poly R}) :=
  let np := ((size p).-1 - j)%N in let nq := ((size q).-1 - j)%N in
 (- 1) ^+ 'C(nq + np, 2) *
 \det (rsubmx ((block_mx perm_rev_mx 0 0 1%:M) *m
                SylvesterHabicht_mx nq np (j + (nq + np)) p q)).

Lemma subresultantE (R : comRingType) j (p q : {poly R}) :
```

```
  let np := ((size p).-1 - j)%N in let nq := ((size q).-1 - j)%N in
  subresultant j p q =
  (-1) ^+ ('C(nq + np, 2) + 'C(nq, 2)) *
  \det (rsubmx (SylvesterHabicht_mx nq np (j + (nq + np)) p q)).

Remark subresultant0 (R : comRingType) (p q : {poly R}) :
  subresultant 0 p q =
  (-1) ^+ ('C((size q).-1 + (size p).-1, 2) + 'C((size q).-1, 2)) *
  resultant p q.

Lemma subresultant_eq0 (R : comRingType) j (p q : {poly R}) :
  let np := ((size p).-1 - j)%N in let nq := ((size q).-1 - j)%N in
  (subresultant j p q == 0) =
  (\det (rsubmx (SylvesterHabicht_mx nq np (j + (nq + np)) p q)) == 0).

(* Remark 4.23. from BPR *)
Lemma SylvesterHabicht_mxP (R : ringType) (np nq k : nat)
  (p q u v : {poly R}) : size u <= np -> size v <= nq ->
  row_mx (poly_rV u) (poly_rV v) *m SylvesterHabicht_mx np nq k p q
                                    = poly_rV (u * p + v * q).

Lemma lsubmx_SylvesterHabicht (R : ringType)
      (np nq k1 k2 : nat) (p q : {poly R}) :
  lsubmx (SylvesterHabicht_mx nq np (k1 + k2) p q) =
  SylvesterHabicht_mx nq np k1 p q.

(* Lemma 4.24. from BPR, not so trivial because *)
(* row_mx u v != 0 is not equivalent to u != 0 and v != 0) *)
Lemma subresultantP (R : idomainType) j (p q : {poly R}) :
  p != 0 -> q != 0 -> j <= (size p).-1 -> j <= (size q).-1 ->
  reflect (exists2 u : {poly R} * {poly R},
                        (0%N < size u.1 <= (size q).-1 - j)
                      && (0%N < size u.2 <= (size p).-1 - j)
                       & size (u.1 * p + u.2 * q)%R <= j)%N
          (subresultant j p q == 0).

Fact gt_size_gcd (R : idomainType) (p q u v : {poly R}) j :
  p != 0 -> q != 0 -> u != 0 ->
  j < size (gcdp p q) -> j <= (size q).-1 -> size u <= (size q).-1 - j ->
  size (u * p + v * q) <= j -> j < (size (gcdp p q)).-1.

(* Proposition 4.25. from BPR *)
Lemma geq_gcdp_subresultant (R : idomainType) j (p q : {poly R}) :
  j <= (size p).-1 -> j <= (size q).-1 ->
  (size (gcdp p q)).-1 >= j = [forall i : 'I_j, subresultant i p q == 0].

(* Proposition 4.26. from BPR *)
Lemma gcdp_subresultant (R : idomainType) j (p q : {poly R}) :
  p != 0 -> q != 0 ->
  j <= (size p).-1 -> j <= (size q).-1 ->
  (size (gcdp p q)).-1 == j =
  [forall i : 'I_j, subresultant i p q == 0] && (subresultant j p q != 0).

Lemma subresultantC (R : idomainType) j (p q : {poly R}) :
```

```
  subresultant j p q =
  (−1) ^+ ('C((size p).−1 − j + ((size q).−1 − j), 2)) ∗ subresultant j q p.

Lemma SylvesterHabicht_mod (R : idomainType) np nq k (p q : {poly R}) :
  (size (p %/ q)%R + np <= nq.+1)%N ->
  (SylvesterHabicht_mx np nq k (p %% q) q) =
  ((block_mx 1%:M (band (− (p %/ q))) 0 1%:M) ∗m
  (SylvesterHabicht_mx np nq k (lead_coef q ^+ scalp p q ∗: p) q)).

Lemma SylvesterHabicht_reduce (R : idomainType)
  (m' k' n m k : nat) (p q : {poly R})
  (pP : (n + m' + (k − k') = n + m)%N) (qP : ((k' + (k − k') = k)%N)) :
  n <= k' -> m' <= k' ->
  ((size p).−1 <= k' − n)%N -> ((size q).−1 <= k' − m')%N ->
  SylvesterHabicht_mx n m k p q =
  castmx (pP, qP) (col_mx (row_mx (SylvesterHabicht_mx n m' k' p q) 0)
                          (band ('X^m' ∗ q))).

Lemma SylvesterHabicht_scaler (R : idomainType) np nq k
  (p q : {poly R}) (c : R) : c != 0 ->
  SylvesterHabicht_mx np nq k (c ∗: p) q =
  block_mx c%:M 0 0 1%:M ∗m SylvesterHabicht_mx np nq k p q.

Lemma subresultant_scaler (R : idomainType) j (p q : {poly R}) (c : R) :
  c != 0 ->
  subresultant j (c ∗: p) q = c ^+ ((size q).−1 − j) ∗ subresultant j p q.

Lemma subresultant_scalel (R : idomainType) j (p q : {poly R}) (c : R) :
  c != 0 ->
  subresultant j p (c ∗: q) = c ^+ ((size p).−1 − j) ∗ subresultant j p q.

Lemma det_rsub_band (R : idomainType) m n (p : {poly R}) :
  (size p).−1 = n ->
  \det (rsubmx (band p : 'M_(m, n + m))) = lead_coef p ^+ m.

(* Something like Proposition 4.37 from BPR *)
Lemma subresultant_mod (R : idomainType) j (p q : {poly R})
      (c := (−1) ^+ 'C((size p).−1 − (size q).−1, 2) ∗
              lead_coef q ^+ ((size p).−1 − (size (p %% q)).−1)) :
  p != 0 -> q != 0 -> (size q).−1 <= (size p).−1 -> j <= (size (p %% q)).−1 ->
  subresultant j (lead_coef q ^+ (scalp p q) ∗: p) q =
  c ∗ subresultant j q (− (p %% q)).

(* Lemma 4.35 from BPR is cindexR_rec from qe_rcf_th, except it uses rmodp *)
```

Conclusion and perspectives Writing this library was significantly eased by the content already available in the Mathematical Component libraries, in particular the support for definition of and operation on block and "band" matrices and the divisibility theory for polynomials. We plan to integrate this work in our previous version of a quantifier elimination algorithm for real closed fields. We will thus obtain a better elimination operator than the one we previously formally described, based on naive pseudo-remainder sequences.

References

[1] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[2] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In *Certified Programs and Proofs*. Springer, Springer, 2011.

[3] David Delahaye and Micaela Mayero. Dealing with Algebraic Expressions over a Field in Coq using Maple. *Journal of Symbolic Computation: special issue on the integration of automated reasoning and computer algebra systems*, 39(5):569–592, 2005.

[4] Maxime Dénès, Anders Mörtberg, and Vincent Siles. A refinement-based approach to computational algebra in coq. In Lennart Beringer and Amy Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, volume 7406 of *Lecture Notes In Computer Science*, pages 83–98, Princeton, États-Unis, 2012. Springer, Springer.

[5] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Tobias Nipkow and Christian Urban, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Germany, 2009. Springer.

[6] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.

[7] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in coq. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005.

[8] M. Petkovšek, H.S. Wilf, and D. Zeilberger. *A + B*. Ak Peters Series. Peters, 1996.

[9] Alfred Poorten and Roger Apry. A proof that euler missed ... *The Mathematical Intelligencer*, 1(4):195–203, 1979.

[10] Roger Apéry. Irrationalité de $\zeta(2)$ et $\zeta(3)$. *Astérisque*, 61:11–13, 1979.

[11] Bruno Salvy. An Algolib-aided Version of Apery's Proof of the Irrationality of $\zeta(3)$. http://algo.inria.fr/libraries/autocomb/Apery2-html/apery.html.

[12] The Coq Proof Assistant. The coq system. Technical report, Inria, 2013. http://coq.inria.fr.