# Basics for algebraic numbers and a proof of Liouville's theorem in C-CoRN

Valentin Blot

July 18, 2009

**Abstract**

This report is that of the training period that ended my first year of Master at the Ecole Normale Supérieure de Lyon. This training period took place at the Radboud Universiteit Nijmegen (Netherlands), and was supervised by Freek Wiedijk. The subject studied is the formalization of algebraic numbers in Coq. In a first part we will show how we formalized Euclidean polynomial division and Cayley-Hamilton theorem (by using SSReflect), which are key lemmas for the first results on algebraic numbers, and in a second part we will present a constructive proof of Liouville's theorem, which states that if $a$ is an algebraic irrational number, there exists $C \in \mathbb{R}$ and $n \in \mathbb{N}$ such that :

$$\forall \frac{p}{q} \in \mathbb{Q}, \left| a - \frac{p}{q} \right| > \frac{C}{q^n}$$

## 1 Introduction

The formalization of algebraic numbers is a challenging task, since it involves the manipulation of multiple algebraic structures. For example, if $A$ is a subring of $B$ and if $\alpha \in B$, then $A[\alpha]$ is a ring between $A$ and $B$. This manipulation of multiple algebraic structures in Coq must be done very carefully, and the best way to manage it, as explained at the end of the first part, is to use the analogy between type theory and category theory. The training period was too short to build an entire library of algebraic numbers in Coq, so we formalized two fundamental theorems useful for basic results on algebraic numbers, and explained the analogy between type theory and category theory that should be used for a future formalization. The two theorems are the following :

- Euclidean polynomial division on a ring : if $f, g$ are polynomials, if $g$ is monic (its highest coefficient is 1), then there exists a unique pair of polynomials $(q, r)$ such that $\begin{cases} f = q * g + r \\ dr < dg \end{cases}$
- Cayley-Hamilton theorem : if $A$ is a matrix over a ring, if $\chi_A(X) = \det(A - XI_n)$ is its characteristic polynomial, then $\chi_A(A) = 0$

In a second part, we describe a certified algorithm which, from a rational polynomial $P$, builds a polynomial which roots are exactly the irrational roots of $P$. More precisely, if

$$\frac{p_1}{q_1}, \ldots, \frac{p_m}{q_m}$$

are the rational roots of $P$ (with multiplicity), then we build the polynomial :

$$\frac{P}{\prod_{i=1}^{m} \left(X - \frac{p_i}{q_i}\right)}$$

The hard task in such an algorithm is to find the rational roots, and to find every rational root. In order to achieve this, we use decidability of equality on $\mathbb{Q}$ and arithmetic properties of $\mathbb{Z}$. Once we have this algorithm, the proof of Liouville's theorem is quite straight-forward.

# 2 Combining SSReflect, C-CoRN and Type Classes

## 2.1 What are SSReflect, C-Corn and Type Classes

### 2.1.1 C-CoRN

C-CoRN (constructive Coq repository at Nijmegen) is a big constructive library of mathematics (see [5], [6], [3] and [4]), which was born with the project of a constructive formalization of the fundamental theorem of algebra. C-CoRN constitutes the main environment in which I worked, and in which my work (has been/will be) integrated. The main parts I used in C-CoRN were :

- The CRing structure of commutative rings
- The cpoly structure of polynomials (encoded as an inductive type : 0 is a polynomial and if P is a polynomial, c+X*P is a polynomial)
- The formalization of the ring homorphisms theorem

The first lemma we needed to prove was the euclidean division of polynomials, in order to prove that for every $\alpha$ algebraic on $R$, $R[\alpha]$ is finitely generated as an $R$-module. Here is the statement of that lemma :

```
Variable CR:CRing
Lemma cpoly_div:
 forall (f g:cpoly_cring CR) (n:nat), monic n g ->
  ex_unq (fun (qr:ProdCSetoid (cpoly_cring CR) (cpoly_cring CR))=>
   f[=](fst qr)[*]g[+](snd qr) and degree_lt_pair (snd qr) g).
```

Where `monic n g` means that g is of degree n and g's nth coefficient is 1, and `degree_lt_pair r g` means that for every n, if g is of degree less than (n+1), then r is of degree less than n, and if g is of degree 0, then r = 0.

### 2.1.2 SSReflect

SSReflect is an extension to the Coq toplevel, providing new tactics, adapted for small scale reflection. This extension was written by Georges Gonthier in order to formalize the 4-colour theorem (see [7]), and then updated by Georges Gonthier and Assia Mahboubi in [8], in order to build an entire group theory library on it in [9], called math components.

SSReflect only talks about decidable, by the use of the "reflect" inductive type :

```
Inductive reflect (P:Prop) :  bool -> Prop :=
          | ReflectT : P -> reflect P true
          | ReflectF : ~P -> reflect P false.
```

After that, we only work with elements of the type `eqType` which are types `T` for which we have a binary function `eqb : T -> T -> bool` together with a proof of :

```
forall x y : T, reflect (x = y) (eqb x y).
```

which means that the Leibniz equality is decidable (reflected by the boolean function `eqb`). We can remark that for example the set of real numbers cannot be implemented as an `eqType`, since we cannot decide equality of arbitrary real numbers (however, apartness is semi-decidable).

The main reason why I needed to use SSReflect is that the Cayley-Hamilton theorem (which is useful to prove that if $a$ is algebraic then $R[a]$ is an algebraic extension) had recently been formalized in math components in [2], and so I planned to use their results in my formalization of algebraic numbers. However, the approaches of C-CoRN and SSReflect are very different, since C-CoRN is entirely constructive, and SSReflect only talks about decidable. Therefore I couldn't use the SSReflect version of Cayley-Hamilton as-is in C-CoRN. More details are given in section 2.2.

### 2.1.3 Type classes

Type classes are part of Coq standard library. Type classes were proposed by Matthieu Sozeau in [10], and his work has recently been integrated into Coq 8.2. The Coq type classes are inspired by Haskell's type classes and use the databases of the auto tactic and implicit arguments in their implementation in Coq. Type classes are mainly an improvement of structures which uses existing features of Coq to emulate Haskell's type classes, and indeed nothing had to be added to the Coq kernel for the implementation. Here is the basic use of type classes :

First, one declares a class `example` with parameters `T`, `param`, ... and members `member1`, ..., just as a regular structure :

```
Class example (T:Type) (param:T)...:={ member1:...; ...}.
```

Then one can declare "instances" of the class `example` by giving the parameters, and then eventually using tactics to build the members (usefull when members are properties on parameters for example) :

```
Instance example_inst:example T p.  Proof.  ...  Qed.
```

This is the first improvement : the definition of an instance can be done using tactics.

A fundamental class is the class of equivalence relations :

```
Class Equivalence {A:Type} (R:relation A):Prop:=
          {Equivalence_Reflexive:Reflexive R;
          Equivalence_Symmetric:Symmetric R;
          Equivalence_Transitive:Transitive R}.
```

where `Reflexive`, `Symmetric` and `Transitive` are themselves type classes with one member, for example, `Reflexive` is defined as :

```
Class Reflexive {A:Type} (R:relation A):Prop:=
          {reflexivity:forall x, R x x}.
```

Another improvement is the command `Context`. If one wants to prove lemmas on a structure which has an equivalence relation, the common beginning of the section will look like this :

```
Variable R:Type.
Variable req:relation R.
Variable r_st:Equivalence req.
```

But type classes introduce a new command, `Context`, which makes heavy use of implicit arguments, and which allows the three lines above to be written equivalently as :

```
Context `{r_st:Equivalence R req}.
```

(Note the ` after "`Context`").

Here is a very simple example of the use of type classes :

```
Context `{r_st:Equivalence R req}.
Class associative (op:binop R):=
          assoc:forall x y z,op x (op y z) === op (op x y) z.
Instance addn_assoc:associative addn.  Proof.  ...  Qed.
Goal forall x y z:nat, (x + y) + z = x + (y + z)
Proof.  intros x y z;rewrite assoc;reflexivity.  Qed.
```

We see that we can use the `assoc` member of the class `associative` for rewritting without the need for additional parameters. Coq will find the necessary instance `addn_assoc` automatically using a database of instances with the `auto` tactic.

Let see now a less trivial example :

We first declare three classes for binary operations :

```
Class commutative (op:binop R):=
          commut:forall x y, op x y === op y x.
Class left_unit (op:binop R) (idm:R):=
          left_id:forall x, op idm x === x.
Class right_unit (op:binop R) (idm:R):=
          right_id:forall x, op x idm === x.
```

Then we suppose we have a setoid equivalence and a commutative operation on it (we can note that Coq infers automatically the parameters `R` and `req` of `commutative`:

```
Context `{r_st:Equivalence R req}.
Context {mul:binop R} {mulC:commutative mul}.
```

Then, for any `idm:R` we build an instance of `right_id mul idm` from an

```
instance of left_id mul idm :
Instance mulC_id_l {idm:R} {H:left_unit mul idm}:right_unit mul idm.
Proof.  ...  Qed.
```
Let now suppose we have built instances of `commutative` and `left_id` for the addition on `nat` :
```
Instance addnC:commutative addn.  Proof.  ...  Qed.
Instance addnLU:left_unit addn 0.  Proof.  ...  Qed.
Then :
Goal forall x:nat, x + 0 = x.
Proof.  rewrite right_id; reflexivity.  Qed.
```
Coq automatically finds the instance of `right_unit addn 0` with the instances `mulC_id_l`, `addnC` and `addnLU`.

## 2.2 The combination

The idea of using SSReflect came when we realized that the Cayley-Hamilton theorem was useful to show that for any $\alpha$ algebraic on $R$, $R[\alpha]$ is an algebraic extension of $R$. For a matrix $A \in M_n(R)$, we define its characteristic polynomial as :

$$\chi_A(X) = \det(A - XI_n)$$

The Cayley-Hamilton theorem says :

$$\forall A \in M_n, \chi_A(A) = 0$$

Therefore, in order to prove this theorem, we need to formalize the determinant of a matrix :

$$\det(A) = \sum_{s \in S_n} \epsilon(s) \prod_{i=1}^{n} A_{i,s(i)}$$

We need to formalize a sum on every permutation and the signature of a permutation. We will also need for the Cayley-Hamilton theorem to perform variable changes on this sum, and therefore we need to formalize the fact that the signature is a group homomorphism. All this is quite a big work and it would have been impossible to do it within the training period. The only solution was to find a way to use SSReflect's results, but in C-CoRN, with a general setoid equivalence, and with no need for decidability on it. The idea is to use the results of SSReflect for the set of indices of the sums, products, and matrices, because these sets are finite, and therefore decidable, while using a general setoid equivalence for the coefficients. This leaded to a rewriting of two libraries of SSReflect : the bigops library, which formalizes the use of big operations like $\sum, \prod, \bigcup, \bigcap$, and the matrix library, which formalizes cut-paste operations, addition, multiplication on matrices and determinant.

### 2.2.1 The SSReflect version

The SSReflect version of bigops was made in a way that allows its use within general types (see [1]), not necessarily decidable, but unfortunately, the prop-

erties of operations like associativity, commutativity or distributivity that are needed to use the lemmas of the library must be proven for the Leibniz equality, and not a general setoid equivalence. The purpose of the rewritting is therefore to be able to use it with a general equivalence.

The SSReflect version of the bigops library uses canonical structures. For example the following structure is defined :

```
Structure law:Type:=Law{
          operator:>T -> T -> T;
                  _:associative operator;
                  _:left_id idm operator;
                  _:right_id idm operator}.
```

With for example `associative` defined as :

```
Definition associative op:=forall x y, op x y = op y x
```

(note the use of Leibniz equality).
Then, if one wants to use the lemmas of the library that use associativity and identity element, one first has to define an element of the structure `law`, and then declare it as a canonical structure.
For example, with the addition on `nat` :

```
Definition addn_law :=
        Law (T:=nat) addn_assoc addn_lid addn_rid.
Canonical Structure addn_law.
```

Concerning the matrix library, the set of coefficients in the SSReflect version live in a type of type `ringType`, which is the decidable type for rings in SSReflect. Therefore we needed to rewrite it in the case of rings with a general setoid equivalence.

### 2.2.2 The rewritten version

The rewriten version replaces the canonical structures which represent the properties of operators with Leibniz equality with type classes which represent the properties of operators with a general setoid equivalence. We define classes of operations as in 2.1.3, so if one has to use the lemmas of the rewritten library, one only has to declare the appropriate instances for the operator. For example, if we have the following in the library :

```
Context `{Equivalence R req} {op:binop R} {idm:R}.
Context {op_morph:Morphism(equiv==>equiv==>equiv)op}.
Context {op_assoc:associative op}.
Context {op_left_id:left_unit op idm}.
Context {op_right_id:right_unit op idm}.
Lemma bla_bla:bla_bla_bla.  Proof.  ...  Qed.
```

Then one only has to declare the appropriate instances (`Equivalence`, `Morphism`, `associative`, `left_id` and `right_id`) to be able to use the lemma `bla_bla` without any argument. And for example if there exists an instance of `commutative` for the law, then one doesn't have to declare the instance of `right_id`, since it will be automatically found by using `mulC_id_l`, as explained in 2.1.3.

Concerning the matrix library, we defined a class `Ring` which takes an equiv-

alence as parameter and which two members are elements of the structures ring_theory and ring_eq_ext of the standard Coq ring library. This use of the structures of standard rings allows the use of the ring tactic with the instances of the class Ring. Therefore, the lemmas in the rewritten version of the matrix library are written in the following context :

```
Context `{r_st:Equivalence R req}.
Context `{r_ring:Ring}.
```

We then use the rewritten version of the bigops library to define the determinant and prove its properties.
The determinant is defined as follows :

```
Definition determinant n (A:'M_n) :=
    \sum_(s:'S_n) (-1)^+s*\prod_(i:'I_n) A i (s i).
```

thanks to the coercion 'S_n>->bool>->nat of SSReflect. In this definition we use the rewritten version of the bigops library for which indices live in SSReflect's finType, therefore we can use all the theoretical results about permutations that exist in SSReflect.

The main lemma for the Cayley-Hamilton proof is the following :

```
Lemma mulmx_adjr :  forall n (A:'M_n),
          A *m adjugate A === scalar_mx (\det A).
```

where adjugate is the transposed matrix of cofactors of $A$, and scalar_mx a is the matrix $a \times I_n$.

### 2.2.3 Cayley-Hamilton in C-CoRN

Using the rewritten library, and the fact that every CRing (the rings of C-CoRN) is a Ring (the class defined above), we are now able to prove the following theorem :

```
Lemma Cayley_Hamilton:
    forall (A:M(CR)) (a:CR) (X:matrix CR n 1),
        A *m X === (scalar_mx a) *m X ->
            (scalar_mx (char_poly !  a)) *m X === '0m.
```

Indeed, this is not exactly the Cayley-Hamilton theorem stated above, but the general version would require the evaluation of a polynomial on a matrix, which is not possible in C-CoRN, since the polynomials are defined on CRing structure, which is commutative, and the matrices do not form a commutative ring. The exact version of Cayley-Hamilton would therefore require a new structure for non-commutative rings in C-CoRN, which would have required too much time. But this version is almost equivalent, and the standard version of Cayley-Hamilton would be easily proved from this lemma if we had the appropriate structures.

## 2.3 Algebraic numbers

We give here the mainlines for a future formalization of algebraic numbers in Coq, since the training period was too short to write an entire library on algebraic numbers. We present the first implementation (using subsets) in which we

got lost, and then we present a more category-related implementation, which we unfortunately did not implement, because of the lack of time.

### 2.3.1 The bad way of seeing $\mathbf{R}[\alpha]$

The first idea in order to formalize ring/field extensions is to consider a ring $B$ and a predicate $P$ on $B$ which preserves operations, so the set :

$$A = \{x : B | Px\}$$

is a subring of $B$. Then for any $\alpha : B$, we can define the predicate $P'$ on $B$ :

$$\forall x : B, P'x \Leftrightarrow \exists p \in A[X], p(\alpha) = x$$

and then we have

$$A[\alpha] = \{x : B | P'x\}$$

but we also need to see $A$ as a subring of $A[\alpha]$, so we need a predicate $P''$ on $A[\alpha]$ defined as :

$$\forall x : A[\alpha], P''x \Leftrightarrow P(x : B)$$

so we have two representations for $A$ :

$$\{x : B | Px\} \text{ and } \{x : A[\alpha] | P''x\}$$

so it becomes messy, and even more when considering $A[\alpha][\beta]$. In order to have a clean formalization, we need to have a categorical vision of ring/field extensions.

### 2.3.2 The good way of seeing $\mathbf{R}[\alpha]$

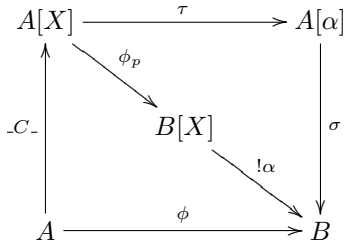
Instead of using predicates and subring, we generalize this notion by considering two rings $A$ and $B$, together with a ring homomorphism $\phi : A \to B$. Therefore, $\phi(A)$ can be seen as a subring of $B$.
We also have the following ring homomorphisms :

- the canonical injection $\_C\_ : A \to A[X]$
- $\phi_p : A[X] \to B[X]$ which applies $\phi$ to each coefficient of a polynomial
- $!\alpha : B[X] \to B$ which applies a polynomial in $\alpha$

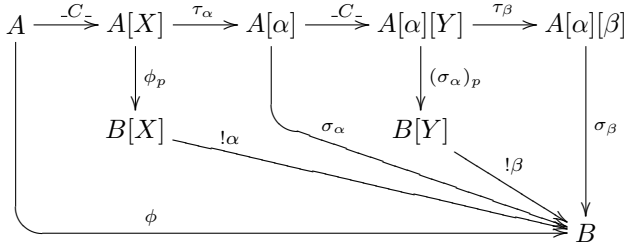
So $!\alpha \circ \phi_p : A[X] \to B$ is a ring homomorphism, and we can build $A[\alpha]$ as the quotient of the ring $A[X]$ by the kernel of $!\alpha \circ \phi_p$ and the ring homomorphism theorem gives us two ring homomorphisms :

- $\tau : A[X] \to A[\alpha]$ surjective
- $\sigma : A[\alpha] \to B$ injective

This is summarized in the following commutative diagram :




After that, it is quite straightforward to define multiple extensions :




# 3 Effective constructive proof of Liouville's theorem

## 3.1 Liouville's theorem

Let $a$ be an algebraic non-rational number, then there exists $C \in \mathbb{R}$ and $n \in \mathbb{N}$ such that :

$$\forall p \in \mathbb{Z}, \forall q \in \mathbb{N}^*, \left| a - \frac{p}{q} \right| > \frac{C}{q^n}$$

Proof :
Let $P_a$ be the minimal polynomial of $a$, let $n = \deg P_a$, then $\forall p, q, P_a\left(\frac{p}{q}\right) \neq 0$ (otherwise $\frac{P_a}{X - \frac{p}{q}}$ is a nonzero polynomial satisfying $a$, and of degree $< n$).

$$\Rightarrow \left| q^n \left( P_a(a) - P_a\left(\frac{p}{q}\right) \right) \right| = \left| q^n P_a\left(\frac{p}{q}\right) \right| \in \mathbb{N}^*$$

$$\begin{cases} \left| a - \frac{p}{q} \right| \leq 1 \Rightarrow q^{-n} \leq \left| P_a(a) - P_a\left(\frac{p}{q}\right) \right| \leq \sup_{[a-1;a+1]} |P_a'| \left| a - \frac{p}{q} \right| \\ \left| a - \frac{p}{q} \right| \geq 1 \Rightarrow \left| a - \frac{p}{q} \right| \geq \frac{1}{q^n} \end{cases}$$

$$\text{Let } C = \min\left(1, \frac{1}{\sup_{[a-1;a+1]} |P_a'|}\right)$$

$$\text{Then } \forall \frac{p}{q} \in \mathbb{Q}, \left| a - \frac{p}{q} \right| \geq \frac{C}{q^n}$$

The issue in this proof is that in general, the existence of a minimal polynomial is non-constructive. But we can go through this when considering numbers that are algebraic on $\mathbb{Q}$, because of the decidability of $\mathbb{Q}$ , and thanks to the arithmetic properties of $\mathbb{Z}$. In fact we do not exactly need the notion of a minimal polynomial of an algebraic number, we can use a weaker result :

$$\forall a \text{ non-rational}, \forall P \in \mathbb{Q}[X],$$

$$P(a) = 0 \Rightarrow \exists Q \in \mathbb{Q}[X], \begin{cases} Q(a) = 0 \\ \forall x \in \mathbb{Q}, Q(x) \neq 0 \end{cases}$$

Since Rolle's theorem has already been formalized in C-CoRN, the major part is to build a certified algorithm which builds $Q$ from $P$.

## 3.2 Introduction to the algorithm

Let us suppose $P \in \mathbb{Z}[X]$, and $\frac{p}{q} \in \mathbb{Q}$ such that $P\left(\frac{p}{q}\right) = 0$

$$\text{Then : } 0 = q^n P\left(\frac{p}{q}\right) = \sum_{i=0}^{n} P_i p^i q^{n-i} \in \mathbb{Z}$$

- $P_0 q^n = -\sum_{i=1}^{n} P_i p^i q^{n-i} = -p \sum_{i=1}^{n} P_i p^{i-1} q^{n-i}$
  so $p | P_0 q^n$, and since we can consider $p \wedge q = 1$, we have $p | P_0$
- $P_n p^n = -\sum_{i=0}^{n-1} P_i p^i q^{n-i} = -q \sum_{i=0}^{n-1} P_i p^i q^{n-i-1}$
  so $q | P_n p^n$, and since we can consider $p \wedge q = 1$, we have $q | P_n$

$\Rightarrow$ there is a finite amount of possible rational roots !

The principle of the algorithm is the following :
For any $P \in \mathbb{Q}[X]$,
either

$$\forall \frac{p}{q} \in \mathbb{Q}, P\left(\frac{p}{q}\right) \neq 0$$

in which case the algorithm is over, or

$$\exists \frac{p}{q} \in \mathbb{Q}, P\left(\frac{p}{q}\right) = 0$$

in which case we iterate the algorithm on

$$\frac{P}{X - \frac{p}{q}}$$

which is of degree less than $P$.

## 3.3 canonical rationals

The sentence "we can consider $p \wedge q = 1$" implies the need for a formalization of $\mathbb{Q}$'s canonical representation :

```
Definition Q_can_num(q:Q_as_CRing):Z_as_CRing:=
          (Qnum q)/(Zgcd(Qnum q)(Qden q)).
Definition Q_can_den(q:Q_as_CRing):Z_as_CRing:=
          (Qden q)/(Zgcd(Qnum q)(Qden q)).
Definition Q_can(q:Q_as_CRing):=
          (Q_can_num q)#(Q_can_den_pos_val q).
Lemma Q_can_spec:forall q:Q_as_CRing,
          q[=]Q_can q.
Lemma Q_can_spec2:forall q:Q_as_CRing,
          Zrelprime(Qnum (Q_can q))(Qden (Q_can q)).
```

## 3.4 from Q[X]to Z[X]

We build a polynomial in $\mathbb{Z}[X]$ from a polynomial in $\mathbb{Q}[X]$ by multiplying each coefficient by the lcm of the denominators of the coefficients

We first define the lcm and generalized lcm over a list :

```
Definition Zlcm(a b:Z_as_CRing):Z_as_CRing:=
                    (a[*]b)/(Zgcd a b).
Fixpoint Zlcm_gen(l:list Z_as_CRing):Z_as_CRing
Lemma Zlcm_gen_spec:forall l x,
          In x l -> Zdivides x (Zlcm_gen l).
Lemma Zlcm_gen_spec2:forall l x,
          (forall y, In y l -> Zdivides y x)
                    -> Zdivides (Zlcm_gen l) x.
```

We then build the list of canonical denominators of $\mathbb{Q}[X]$ polynomials and the lcm over it :

```
Fixpoint den_list(P:QX):list Z_as_CRing
Definition Zlcm_den_poly(P:QX):=Zlcm_gen(den_list P).
```

We define the polynomial of $\mathbb{Z}[X]$ obtained by taking the canonical numerators of the coefficients :

```
Fixpoint Q_can_num_poly (P:QX):ZX
```

qx2zx takes a polynomial of $\mathbb{Q}[X]$, multiplies every coefficient by the lcm of the canonical denominators, and then takes the polynomial of the canonical numerators, which is in $\mathbb{Z}[X]$ :

```
Definition qx2zx(P:QX):ZX:=
          Q_can_num_poly(_C_ Zlcm_den_poly P [*] P).
```

zx2qx is the canonical injection from $\mathbb{Z}[X]$ to $\mathbb{Q}[X]$ :

```
Definition zx2qx:=cpoly_map injZ_rh.
```

The predicate in_ZX is true if each of the coefficients of the polynomial has a canonical denominator equal to 1.

```
Lemma zx2qx_spec:forall P:QX,
          in_ZX P -> P [=] zx2qx (Q_can_num_poly P).
```

We can prove that a polynomial of $\mathbb{Q}[X]$ multiplied by the lcm of the canonical denominators of its coefficients is indeed in $\mathbb{Z}[X]$ :

```
Lemma Zlcm_den_poly_spec:
         forall P, in_ZX (_C_ Zlcm_den_poly P [*] P).
```

And finally the following lemma :

```
Lemma qx2zx_spec:forall P,
        zx2qx (qx2zx P) [=] _C_ Zlcm_den_poly P [*] P.
```

### 3.5 q^n P(p/q) is in Z

We also need the following fact for polynomials in $\mathbb{Z}[X]$ :

$$q^n P\left(\frac{p}{q}\right) = \sum_{i=0}^{n} P_i p^i q^{n-i} \in \mathbb{Z}$$

```
Lemma Q_Z_poly_apply:forall (P:ZX) (p:Z_as_CRing) (q:positive),
     let n:=ZX_deg P in
          (injZ_rh q)[^]n [*] (zx2qx P) ! (p # q)
               [=]injZ_rh (Sum 0 n (fun i => (nth_coeff i P)[*]p[^]i[*]q[^](n-i))).
```

We now define the coefficient of highest degree of `qx2zx P` for $P \in \mathbb{Q}[X]$ :

```
Let Pn(P:QX):=nth_coeff (QX_deg P) (qx2zx P).
```

And we prove that the canonical denominator of any rational root of $P$ divides this coefficient :

```
Lemma den_div_Pn:forall (P:QX) (a:Q_as_CRing),
        P!a[=]Zero -> Zdivides(Q_can_den a)(Pn P).
```

We also define the coefficient of lowest degree :

```
Let P0(P:QX):=nth_coeff 0 (qx2zx P).
```

And prove that the canonical numerator of any rational root of $P$ divides this coefficient :

```
Lemma den_div_P0:forall (P:QX) (a:Q_as_CRing),
        P!a[=]Zero -> Zdivides(Q_can_num a)(P0 P).
```

### 3.6 The roots are localized

`(list_Q a b)` is the list of all rational numbers $\frac{p}{q}$ such that $\begin{cases} |p| \leq |a| \\ |q| \leq |b| \end{cases}$ :

```
Definition list_Q (a b:Z_as_CRing):list Q_as_CRing
```

In particular, `(list_Q a b)` contains all rational numbers $\frac{p}{q}$ such that $\begin{cases} p|a \\ q|b \end{cases}$ :

```
Lemma list_Q_spec:forall (a b:Z_as_CRing) q,
        a [#] Zero -> b [#] Zero ->
        Zdivides (Q_can_num q) a ->
        Zdivides (Q_can_den q) b ->
               In (Q_can q) (list_Q a b).
```

We prove that if the lowest degree coefficient is nonzero, then any rational root

is in (list_Q (P0 P) (Pn P)) :

```
Lemma QX_root_loc:forall (P:QX) (a:Q_as_CRing),
       P!Zero[#]Zero -> P!a [=] Zero ->
              In (Q_can a) (list_Q (P0 P) (Pn P)).
```

## 3.7 The extraction algorithm

The function `QX_find_root` tries 0 and every element of (list_Q (P0 P) (Pn P)) to find a root of the polynomial :

```
Definition QX_find_root(P:QX):option Q_as_CRing
Lemma QX_find_root_spec_none:forall P,
         QX_find_root P=None ->
              forall q:Q_as_CRing, P!q[#]Zero.
Lemma QX_find_root_spec_some:forall P x,
         QX_find_root P=Some x -> P!x[=]Zero.
```

We now build a recursive function which tries to find $n$ roots (`RX_div P x` is the result of the euclidean division of $P$ by $(X - x)$ :

```
Fixpoint QX_extract_roots_rec (n:nat) (P:QX):=
  match n with | O => P | S n =>
    match QX_find_root P with | None => P | Some x =>
      QX_extract_roots_rec n (RX_div Q P x) end end.
```

We finally define the function `QX_extract_roots` by calling `QX_extract_roots_rec` with $n = degP$ :

```
Definition QX_extract_roots (P:QX):=
                    QX_extract_roots_rec (QX_deg P) P.
```

And prove the specifications :

```
Lemma QX_extract_roots_spec_rat:forall P a,
         P[#]Zero -> (QX_extract_roots P)!a[#]Zero.
Lemma QX_extract_roots_spec_nrat:forall (P:QX) (x:IR),
         (forall y:Q_as_CRing, x[~=](inj_Q_rh y)) ->
              (inj_QX_rh P)!x[=]Zero ->
         (inj_QX_rh (QX_extract_roots P))!x[=]Zero.
```

## 3.8 Liouville's theorem

We now have all the tools necessary to Liouville's theorem for which we give two versions :

Let $a$ be an irrational real number :

```
Variable a:IR.
Hypothesis a_irrat:forall x:Q, a[~=]inj_Q IR x.
```

Let $P$ be a nonzero rational polynomial which satisfies $a$ :

```
Variable P:cpoly_cring Q_as_CRing.
Hypothesis P_nz:P[#]Zero.
Hypothesis a_alg:(inj_QX_rh P)!a[=]Zero.
```

We define `Liouville_constant` and `Liouville_degree` such that :

```
Theorem Liouville_theorem:forall (x:Q),
```

```
    (Liouville_constant[*]inj_Q IR (1#Qden x)[^]Liouville_degree)
                    [<=] AbsIR (inj_Q IR x [-] a).
```

And here is the second version :

```
Theorem Liouville_theorem2:
          {n:nat|{C:IR|Zero[<]C|forall (x:Q),
               (C[*]inj_Q IR (1#Qden x)[^]n)
                    [<=]AbsIR(inj_Q IR x[-]a)}}.
```

# 4 Conclusion and future work

Concerning the first part, using the two theorems (Euclidean division on polynomials and the Cayley-Hamilton) that we proved in C-CoRN, and using a point of view of categories (as seen in 2.3.2), we have now the basics in order to build an algebraic numbers library in C-CoRN.
There are also probably interesting theorems that may be ported from SSReflect to C-CoRN, as we have done with Cayley-Hamilton.

Concerning the second part, we have an algorithm of rational roots extraction from rational polynomials which leads to the proof of Liouville's theorem. Now, a nice thing would be to prove that Liouville's number :

$$\sum_{i=0}^{+\infty} 10^{-i!}$$

is transcendental, by the contraposite of Liouville's theorem. Indeed, this number was the first explicit example of a transcendental number (as we already knew there were some, by the denombrability of algebraic numbers, but we did not have any explicit).
However, with the formalization presented here, we could probably only show that its value on any nonzero rational polynomial on this number is [˜=] (not equal) to zero. A stronger result would be that it is [#] (apart) from zero.

# References


[1] Y. Bertot, G. Gonthier, S.O. Biha, and I. Pasca. Canonical big operators. *Lecture Notes in Computer Science*, 5170:86–101, 2008.

[2] S.O. Biha et al. Formalisation des mathématiques: une preuve du théorème de Cayley-Hamilton. *Journees Francophones des Langages Applicatifs*, pages 1–14, 2008.

[3] L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. *Lecture Notes in Computer Science*, 3119:88–103, 2004.

[4] L. Cruz-Filipe and B. Spitters. Program extraction from large proof developments. *Lecture notes in computer science*, pages 205–220, 2003.

[5] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation*, 34(4):271–286, 2002.

[6] H. Geuvers, F. Wiedijk, and J. Zwanenburg. A constructive proof of the Fundamental Theorem of Algebra without using the rationals. *Lecture notes in computer science*, pages 96–111, 2002.

[7] G. Gonthier. A computer-checked proof of the four colour theorem. *Available at research.microsoft.com/˜gonthier/4colproof. pdf.*

[8] G. Gonthier and A. Mahboubi. A small scale reflection extension for the Coq system. 2008.

[9] G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A modular formalisation of finite group theory. *Lecture Notes in Computer Science*, 4732:86, 2007.

[10] M. Sozeau. *Un environnement pour la programmation avec types dependants*. PhD thesis, Ph. D. thesis, Universite Paris 11, Orsay, France, 2008.