Relaxed Hensel lifting of triangular sets

Romain Lebreton

University of Waterloo
Waterloo, ON, Canada

Abstract

In this paper, we present a new lifting algorithm for triangular sets over general p-adic rings. Our contribution is to give, for any p-adic triangular set, a shifted algorithm of which the triangular set is a fixed point. Then we can apply the relaxed recursive p-adic framework and deduce a relaxed lifting algorithm for this triangular set.

We compare our algorithm to the existing technique and report on implementations inside the `C++` library Geomsolvex of Mathemagix van der Hoeven et al. (2002). Our new relaxed algorithm is competitive and compare favorably on some examples.

Key words: polynomial system solving, online algorithm, relaxed algorithm, triangular set, univariate representation, p-adic integer, power series

1. Introduction

The introduction is made of five subsections. We present the setting of triangular sets with p-adic coefficients in Section 1.1, together with the statement of our lifting problem. We present in Section 1.2 our model of computation for algorithms on p-adics. Section 1.3 introduces the framework in which online algorithms are used to lift triangular sets. Finally, our results and contributions are stated in Section 1.4, followed by an outline of the paper in Section 1.5.

1.1. Statement of the problem

Our goal in this paper is to extend a growing body of work on *relaxed algorithms* to the context of lifting techniques for *univariate representations* and *triangular sets*.

It is well-known that, under some regularity conditions, techniques such as Newton iteration can be used to compute a power series root of an equation such as $f(T, x(T)) = 0$,

⋆ This work has been partly supported by the ANR grant HPAC (ANR-11-BS02-013).
Email address: `rlebreton@uwaterloo.ca` (Romain Lebreton).

with f in $\Bbbk[T, X]$, or a p-adic integer root of an equation of the form $f(x) = 0$ with f in $\mathbb{Z}[X]$.

Relaxed methods, introduced by van der Hoeven (van der Hoeven, 2002), offer an alternative to Newton iteration. The case of computing one power series root, or one p-adic root, of a *system* of polynomial equations was worked out in (van der Hoeven, 2011; Berthomieu and Lebreton, 2012); for this problem, the relaxed algorithm was seen to behave better than Newton iteration in some cases, for instance for multivariate systems with a large number of equations.

In this paper, we go beyond the case of lifting a single root of a multivariate system: we deal with all roots at once, introducing relaxed algorithms that deal with objects such as univariate and triangular representations. This paper is based on the Ph.D. thesis (Lebreton, 2012).

Example 1. We consider the polynomial system $\boldsymbol{f} = (f_1, f_2)$ in $\mathbb{Z}[X_1, X_2]$ with

$$\begin{aligned} f_1 &:= 33X_2^3 + 14699X_2^2 + 6761112X_2 + 276148X_1 - 11842820 \\ f_2 &:= X_2^2 + 66X_1X_2 - 75X_2 - 94X_1 - 22. \end{aligned}$$

Let $\boldsymbol{t}_0$ be the triangular set of $(\mathbb{Z}/7\mathbb{Z})[X_1, X_2]$, that is a lexicographical Gröbner basis for $X_1 < X_2$, given by

$$\boldsymbol{t}_0 := \left(X_1^2 + 5X_1,\ X_2^2 + 3X_1X_2 + 2X_2 + 4X_1 + 6\right).$$

We lift the triangular set $\boldsymbol{t}_0$ defined modulo 7 to triangular sets $\boldsymbol{t}$ defined modulo $7^2, 7^3$ and so on. At the first step, we have

$$\begin{aligned} \boldsymbol{t}_1 = (&X_1^2 + (5 + 5 \cdot 7)X_1 + 7, \\ &X_2^2 + (3 + 2 \cdot 7)X_1X_2 + (2 + 3 \cdot 7)X_2 + 4X_1 + (6 + 3 \cdot 7)) \end{aligned}$$

in $(\mathbb{Z}/7^2\mathbb{Z})[X_1, X_2]$. We iterate again and find

$$\begin{aligned} \boldsymbol{t}_2 = (&X_1^2 + (5 + 5 \cdot 7 + 6 \cdot 7^2)X_1 + (7 + 7^2),\ X_2^2 + (3 + 2 \cdot 7 + 7^2)X_1X_2 + \\ &(2 + 3 \cdot 7 + 5 \cdot 7^2)X_2 + (4 + 5 \cdot 7^2)X_1 + (6 + 3 \cdot 7 + 6 \cdot 7^2)) \end{aligned}$$

in $(\mathbb{Z}/7^3\mathbb{Z})[X_1, X_2]$. The precision is enough to recover the integer triangular set

$$\boldsymbol{t} := \left(X_1^2 - 9X_1 + 56,\ X_2^2 + 66X_1X_2 - 75X_2 - 94X_1 - 22\right) \in \mathbb{Z}[X_1, X_2].$$

Our techniques of p-adic lifting applies to general p-adic rings. Let R be a commutative domain with unit. We consider an element $p \in R - \{0\}$, and we write R_p for the completion of the ring R for the p-adic valuation. We will assume that $R/(p)$ is a field (equivalently, that p generates a maximal ideal). This is not compulsory but will be useful later on when we deal with linear algebra modulo (p). We also assume that $\cap_{i \in \mathbb{N}}(p^i) = \{0\}$, so that R can be seen as a subset of R_p. Note that the set of natural numbers $\mathbb{N}$ contains 0 in this paper.

Two classical examples of p-adic rings are the formal power series ring $\Bbbk[[T]]$, which is the completion of the ring of polynomials $\Bbbk[T]$ for the ideal (T), and the ring of p-adic

integers $\mathbb{Z}_p$, which is the completion of the ring of integers $\mathbb{Z}$ for the ideal (p), with p a prime number.

Consider a system of polynomial equations $\boldsymbol{f} = (f_1, \ldots, f_s) \in R[X_1, \ldots, X_s]$. Denote by $\mathcal{I}$ the ideal generated by $(f_1, \ldots, f_s)$ in $Q[X_1, \ldots, X_s]$, where Q is the total field of fractions of R. In what follows, we make the following assumptions, denoted (H):

(1) the algebraic set $V = V(\mathcal{I}) \subset \overline{Q}^s$ has dimension zero in an algebraic closure $\overline{Q}$;

(2) the Jacobian determinant of $(f_1, \ldots, f_s)$ vanishes nowhere on V.

Let d be cardinality of V; due to our non-vanishing assumption on the Jacobian of $\boldsymbol{f}$, the extension $Q \to A := Q[X_1, \ldots, X_s]/\mathcal{I}$ is a product of separable field extensions. As a result, A has dimension d over Q.

In order to describe V, we will consider two data structures, which we briefly describe now. These data structures encode $\Bbbk$-algebras A of finite dimension as vector spaces, $\Bbbk$ being any field.

An element g of $A = \Bbbk[X_1, \ldots, X_s]/\mathcal{I}$ will be called *primitive* if the $\Bbbk$-algebra $\Bbbk[g] \subset A$ spanned by g in A is equal to A itself. If Λ is a primitive linear form in A, a *univariate representation* of A consists of polynomials $\mathcal{U} = (q, r_1, \ldots, r_s)$ in $\Bbbk[Z]$, where Z is a new variable, with $\deg(r_i) < \deg(q)$ for all i and such that we have a $\Bbbk$-algebra isomorphism

$$\begin{aligned} A = \Bbbk[X_1, \ldots, X_s]/\mathcal{I} &\to \Bbbk[Z]/(q) \\ X_1, \ldots, X_s &\mapsto r_1, \ldots, r_s \\ \Lambda &\mapsto Z. \end{aligned}$$

In particular, q has degree d in Z; we will say that $\mathcal{U}$ has degree d.

Such representations, or slight modifications thereof (using for instance a rational form for the r_i's) have been used for decades in computer algebra, under the names of Shape Lemma, Rational Univariate Representation, Geometric Resolution, etc (Gianni and Mora, 1989; Giusti and Heintz, 1991; Giusti et al., 1997a; Rouillier, 1999; Giusti et al., 2001; Heintz et al., 2001). The oldest trace of this representation is to be found in (Kronecker, 1882) and a few years later in (König, 1903). Different algorithms compute this representation, from a geometric resolution (Giusti et al., 1997b,a, 2001; Heintz et al., 2001) or using Gröbner bases (Rouillier, 1999).

On the other side, one finds triangular representations. A *triangular set* is a set of s polynomials $\boldsymbol{t} = (t_1, \ldots, t_s) \subseteq \Bbbk[X_1, \ldots, X_s]$ such that for all i, t_i is in $\Bbbk[X_1, \ldots, X_i]$, monic and reduced with respect to $(t_1, \ldots, t_{i-1})$. This is thus a reduced Gröbner basis for the lexicographic order $X_1 \ll \cdots \ll X_s$. The notion of triangular set comes from (Ritt, 1966) in the context of differential algebra. Many similar notions were introduced afterwards (Wu, 1984; Lazard, 1991; Kalkbrener, 1993; Aubry et al., 1999); although all these notions do not coincide in general, they are the same for zero-dimensional ideals.

Assume that $\mathcal{I}$ admits a triangular family of generators $\boldsymbol{t}$, and write $d_i := \deg_{X_i}(t_i)$ for all i. Then, we have the equality $d = d_1 \cdots d_s$; we will say that $\boldsymbol{t}$ has multi-degree $\boldsymbol{d} = (d_1, \ldots, d_s)$.

As it turns out, univariate representations can be seen as a special case of triangular sets. Indeed, with the notations above, the family $(q(T), X_1 - r_1(T), \ldots, X_s -$

$r_s(T)$) is a triangular set of generators of $(T - \Lambda(X_1, \ldots, X_s), f_1, \ldots, f_s)$ in the algebra $R[T, X_1, \ldots, X_s]$.

Both kinds of data structures have found numerous applications, and it is not our purpose here to compare their respective merits. Univariate representations always exist, provided the base field is large enough (Gianni and Mora, 1989). On the other hand, the ideal $\mathcal{I}$ may not admit a triangular family of generators: such issues are handled using *triangular decompositions* of $\mathcal{I}$ (Dahan et al., 2005). We will not enter this discussion here, and we will simply suppose when needed that $\mathcal{I}$ admits one or the other such representation.

Let us for instance suppose that $\mathcal{I}$ admits a univariate representation $\mathcal{U}$. By hypothesis (p) is a maximal ideal in R_p, with residual field $R/(p)$. We make the following assumptions, denoted by $(\mathsf{H}')_{\mathcal{U},p}$:

(1) none of the denominators appearing in $\mathcal{U}$ vanishes modulo p;
(2) the polynomials $\boldsymbol{f} \bmod p$ still satisfy (H);
(3) $\mathcal{U}_0 = \mathcal{U} \bmod p$ is a univariate representation of $R/(p)[X_1, \ldots, X_s]/(\boldsymbol{f} \bmod p)$.

Then, given $\mathcal{U}_0 = \mathcal{U} \bmod p$ and $\boldsymbol{f}$, our objective is to compute objects of the form $\mathcal{U}_{0\ldots n} = \mathcal{U} \bmod p^n$, for higher powers of n.

Similar questions can be asked for triangular representations: suppose that $\mathcal{I}$ admits a triangular representation $\boldsymbol{t}$, and that the natural analogue $(\mathsf{H}')_{\boldsymbol{t},p}$ of $(\mathsf{H}')_{\mathcal{U},p}$ holds; we will show how to compute $\boldsymbol{t}_{0\ldots n} = \boldsymbol{t} \bmod p^n$, for high powers of n, starting from $\boldsymbol{t}_0 = \boldsymbol{t} \bmod p$ and $\boldsymbol{f}$.

We remark that for both the univariate and the triangular case, assumption $(\mathsf{H}')_{\mathcal{U},p}$, resp. $(\mathsf{H}')_{\boldsymbol{t},p}$, holds for all values of p except finitely many, at least when $R_p = \Bbbk[[T]]$ or $\mathbb{Z}_p$: this is analyzed in detail in (Schost, 2003a,b; Dahan et al., 2005); thus, these are very mild assumptions.

Let us say a few words about the applications of these kinds of techniques. One direct application is naturally to solve polynomial systems with rational (resp. rational function) coefficients: the techniques discussed here allow one to compute a description of V over $\mathbb{Q}$ (resp. $\Bbbk(T)$) by computing it modulo a prime p (resp. $(T - t_0)$) and lifting it to a sufficient precision (and possibly applying rational reconstruction to recover coefficients from their p-adic expansion).

Such lifting techniques are also at the core of many further algorithms: for instance, it is used in the geometric resolution algorithm (Giusti et al., 2001; Heintz et al., 2001) that computes univariate representations. This algorithm relies on an iterative lifting / intersection process: the lifting part involves lifting univariate representations, as explained in this paper, while the intersection part uses resultant computations. On the triangular side, a similar lifting / intersection process is used in the change of order algorithm of (Dahan et al., 2008).

1.2. Model of computation

We begin with a description of the representation of elements $a \in R_p$, which are called *p-adics*. Any p-adic $a \in R_p$ can be written (in a non unique way) $a = \sum_{i \in \mathbb{N}} a_i p^i$ with coefficients $a_i \in R$. To get a unique representation of elements in R_p, we will fix a subset

M of R such that the projection $\pi : M \to R/(p)$ is a bijection. Then, any element $a \in R_p$ can be uniquely written $a = \sum_{i\in\mathbb{N}} a_i p^i$ with coefficients $a_i \in M$.

Regarding the two classical examples of p-adic rings, we naturally take $M = \Bbbk$ for $R = \Bbbk[T]$; for $R = \mathbb{Z}$, we choose $M = \left\{-\frac{p-1}{2}, \ldots, \frac{p-1}{2}\right\}$ if p is odd and $M = \{0, 1\}$ for $p = 2$.

Once M has been fixed, we have a well-defined notion of *length* of a (non-zero) p-adic: if $a = \sum_{i\in\mathbb{N}} a_i p^i$, then we define $\lambda(a) := 1 + \sup(i \in \mathbb{N} | a_i \neq 0)$, so that $\lambda(a)$ is in $\mathbb{N}_{>0} \cup \{\infty\}$; for $a = 0$, we take $\lambda(a) = 0$. We say that we have computed a p-adic at precision n if the result holds modulo p^n.

Throughout this paper, we choose to denote elements by small letters, e.g. $a \in R_p$, vectors by bold fonts, e.g. $\boldsymbol{a} \in (R_p)^s$, and matrices by bold capital letters, e.g. $\boldsymbol{A} \in \mathcal{M}_s(R_p)$.

Roughly speaking, we measure the cost of an algorithm by the number of arithmetic operations with operands in M it performs. More precisely, we assume that we can do the following at unit cost:

(1) given a_0, b_0 in M, compute the coefficients c_0, c_1 of $c = a_0 b_0$ at unit cost, and similarly for the coefficients of $a_0 \pm b_0$
(2) given a_0 in $M - \{0\}$, compute b_0 in $M - \{0\}$ such that $a_0 b_0 = 1 \bmod p$.

We remark that when $R = \Bbbk[T]$, we are simply counting arithmetic operations in $\Bbbk$.

The main operations we will need on p-adics are sum, difference and multiplication (of course, these algorithms only operate on truncated p-adics). For the moment, we will simply define the problems, and introduce notation for their complexity.

Addition of two p-adics takes linear time in the precision. For the multiplication of a and b of length at most n, we will let $\mathsf{I} : \mathbb{N} \to \mathbb{N}$ be such that all coefficients of ab can be computed in $\mathsf{I}(n)$ operations. We will assume that $\mathsf{I}(n)$ satisfies the property that $\mathsf{I}(n)/n$ is non-decreasing. The cost function $\mathsf{I}(n)$ depends on the p-adic ring R_p but we can always take $\mathsf{I}(n)$ *quasi-linear*, that is, linear up to logarithmic factors (Berthomieu et al., 2011, Proposition 4).

As customary, let us denote by $\mathsf{M}(n)$ a function such that over any ring, polynomials of degree at most $n-1$ can be multiplied in $\mathsf{M}(n)$ base operations, and such that $\mathsf{M}(n)/n$ is non-decreasing (super-linearity hypothesis, see (von zur Gathen and Gerhard, 2013, Section 8.3)). It is classical (Schönhage and Strassen, 1971; Cantor and Kaltofen, 1991) that $\mathsf{M}(n) = \mathcal{O}(n \log(n) \log(\log(n)))$ and thus $\mathsf{I}(n) = \mathcal{O}(n \log(n) \log(\log(n)))$ when $R = \Bbbk[T]$ and $p = T$.

The presence of carries when computing with integers complicates the situation for all operations. Surprisingly, though, it is possible to obtain a slightly faster plain multiplication than in the polynomial case; two integers with n digits in base p can be multiplied in bit-complexity $\mathcal{O}(n \log(n) 2^{\log^*(n)})$ where $\log^*$ denotes the iterated logarithm (Fürer, 2007; De et al., 2008). Note that this result involves a different complexity model than ours. It seems that the ideas of (De et al., 2008) could be adapted to give the same result in our complexity model; we would then have $\mathsf{I}(n) = \mathcal{O}(n \log(n) 2^{\log^*(n)})$ when $R_p = \mathbb{Z}_p$.

The bulk of the computations will rely on *modular* arithmetic: we will need to perform arithmetic operations, mostly additions and multiplications, modulo either a univariate polynomial (in the case of univariate representations) or a whole triangular set.

In the former case, it is known that if $q \in R[Z]$ has degree d (for some ring R), additions and multiplications modulo q can be done using $\mathcal{O}(\mathsf{M}(d))$ operations in R. When R is a field, inversion modulo q, when possible, can be done in time $\mathcal{O}(\mathsf{M}(d)\log(d))$ (von zur Gathen and Gerhard, 2013).

If $\boldsymbol{t} = (t_1, \ldots, t_s)$ is a triangular set in $R[X_1, \ldots, X_s]$ with degrees $d_1, \ldots, d_s$, where R is a ring, we will let $\mathsf{MT}(\boldsymbol{d})$ denote an upper bound on the number of operations in R needed to perform multiplication modulo $\boldsymbol{t}$. If R is a field, we let $\mathsf{IT}(\boldsymbol{d})$ denote an upper bound for the cost of inversion modulo $\boldsymbol{t}$ (when possible). These operations are less straightforward than in the univariate case. Known upper bounds are $\mathsf{MT}(\boldsymbol{d}) = \mathcal{O}(4^s d\log(d)\log(\log(d)))$ where $d = d_1 \cdots d_s$ and $\mathsf{IT}(\boldsymbol{d}) = \mathcal{O}(c^s d\log(d)^3)$, for some (large) constant c (Dahan et al., 2006, 2008; Li et al., 2009).

We seize the opportunity to state our first result. By modifying slightly the algorithm of (Li et al., 2009), we are able to improve its complexity (see Section 3.2).

Proposition 2. *The multiplication modulo a triangular set $\boldsymbol{t}$ of multi-degree $\boldsymbol{d}$ can be done in $\mathsf{MT}(\boldsymbol{d}) = \mathcal{O}(3^s d\log(d)\log(\log(d)))$ operations in R.*

We will further need to solve linear systems over p-adic modular multivariate polynomials. We denote by ω the exponent of linear algebra on any field $\Bbbk$, so that we can multiply and invert matrices in $\mathcal{M}_m(\Bbbk)$ in $\mathcal{O}(m^\omega)$ arithmetic operations. Using the algorithms of (Coppersmith and Winograd, 1990; Stothers, 2010; Vassilevska Williams, 2011) for the multiplication, one can take $\omega < 2.38$. We will also need to invert matrices over rings that are not fields, e.g. in quotients of polynomial rings $R[T]/(q)$. We denote by $\mathcal{O}(m^\Omega)$ the arithmetic complexity of the elementary operations on $m \times m$ matrices over any commutative ring: addition, multiplication, determinant and adjoint matrix. In fact, Ω can be taken less than 2.70 (Berkowitz, 1984; Kaltofen, 1992; Kaltofen and Villard, 2004).

For the special case of matrix inversion in $(R/(p))[X_1, \ldots, X_n]/(\boldsymbol{t})$ for $\boldsymbol{t}$ a triangular set of multi-degree $\boldsymbol{d}$, we add the costs of determinant and adjoint matrix $\left(\mathcal{O}\left(m^\Omega \mathsf{MT}(\boldsymbol{d})\right)\right)$, of the inversion of the determinant $(\mathsf{IT}(\boldsymbol{d}))$ and multiplication of the adjoint by this inverse $\left(\mathcal{O}\left(m^2 \mathsf{MT}(\boldsymbol{d})\right)\right)$, to obtain a total cost of $\mathcal{O}\left(\mathsf{IT}(\boldsymbol{d}) + m^\Omega \mathsf{MT}(\boldsymbol{d})\right)$.

1.3. Online algorithms and recursive p-adics

The property of being *online* (or equivalently, *relaxed*) for an algorithm with p-adic input and output controls the access to the p-adic coefficients of the input during the computation. The notion of on-line algorithm was introduced by (Hennie, 1966). Informally, consider an algorithm with at least one p-adic input $a = \sum_{i \geqslant 0} a_i p^i$ and whose output is another p-adic $c = \sum_{i \geqslant 0} c_i p^i$. We say that this algorithm is *online* with respect to a if, during the computation of the ith p-adic coefficient c_i of the output, the algorithm reads at most the coefficients $a_0, \ldots, a_i$ of the input a. We say that an algorithm is online if it is online with respect to all its p-adic inputs.

The major advantage of online algorithms is that they enable the lifting of *recursive* p-adics. A recursive p-adic y is a p-adic that satisfies $y = \Phi(y)$ for an operator Φ such that the nth coefficient of the p-adic $\Phi(y)$ does not depend on the coefficients of order greater or equal to n of y. As a consequence, y can be computed recursively from its first coefficient y_0 and Φ.

One can find in (Watt, 1989; van der Hoeven, 2002) an algorithm that computes y from its fixed point equation $y = \Phi(y)$ in the context of power series. A key aspect of this algorithm is that *its cost is the one of evaluating* Φ by an online algorithm. Using the fast online multiplication algorithm of (van der Hoeven, 1997), it leads to an efficient framework for computing with recursive p-adics. An issue about the correctness of relaxed recursive p-adics was raised and solved in (Berthomieu and Lebreton, 2012), leading to the notion of *shifted algorithm*. We will review in Section 2 the online framework for computing recursive p-adics and generalize it to vectors of p-adics.

As van der Hoeven was apparently not aware of previous work on online algorithms, he called his algorithm "relaxed multiplication algorithm" and the subsequent algorithms for recursive p-adics were called relaxed algorithms (van der Hoeven, 2002). Therefore we can retrospectively define a *relaxed algorithm* to be a fast online algorithm. These definitions, and their adaptation to general p-adic rings (Berthomieu et al., 2011), will be used from now on. Apart from Section 2, we will use the terminology "relaxed" algorithms since our goal is to have asymptotically fast algorithms.

The literature contains applications of the relaxed recursive p-adic framework to standard and important systems of equations: linear (van der Hoeven, 2002; Berthomieu and Lebreton, 2012), algebraic (van der Hoeven, 2011; Berthomieu et al., 2011; Berthomieu and Lebreton, 2012) and differential (van der Hoeven, 2002, 2010, 2011; Bostan et al., 2012).

This paper is the natural extension of (Berthomieu and Lebreton, 2012) and the relaxed recursive p-adic framework is the cornerstone of our result. Our contribution consists in finding a recursive equation of which the triangular set we seek to lift is a fixed point. More precisely, the challenge was to find a shifted algorithm that computes an operator Φ satisfying $\boldsymbol{t} = \boldsymbol{\Phi}(\boldsymbol{t})$, which will be done in Section 4.

Let us know turn to the complexity of relaxed algorithm. We denote by $\mathsf{R} : \mathbb{N} \to \mathbb{N}$ a function such that *relaxed* multiplication of p-adics can be done at precision n in time $\mathsf{R}(n)$. Even though the algorithm of (Fischer and Stockmeyer, 1974; Schröder, 1997; van der Hoeven, 1997; Berthomieu et al., 2011) performs multiplication over any p-adic ring, its complexity bound $\mathsf{R}(n) = \mathcal{O}(\mathsf{M}(n)\log(n))$ is proven only in the power series and p-adic integers cases. The issue with general p-adic rings is the management of carries. Although we do not prove it here, we believe that this complexity result carries forward to any p-adic ring.

Recent progress has been made on relaxed multiplication: the multiplication of two p-adics at precision n can be done in time $\mathsf{R}(n) = \mathsf{M}(n)\log(n)^{o(1)}$ (van der Hoeven, 2007, 2012). Once again, this complexity statement has been proven only for power series and p-adic integers rings. We also believe that this result should carry forward to any p-adic ring.

1.4. Our contribution

We will make the following assumption on the representation of the input system: we assume that $\boldsymbol{f}$ is given by a straight-line program with inputs $T, X_1, \ldots, X_s$ and s outputs corresponding to $f_1, \ldots, f_s$, using operations in $\{+, -, *\}$. We let L be the size of this straight-line program.

With this notation being introduced, we can state our first result, which deals with lifting of triangular representations.

Theorem 3. *Let $\boldsymbol{f} = (f_1, \ldots, f_s)$ be a polynomial system in $R[X_1, \ldots, X_s]$ given by a straight-line program Γ of size L. Suppose that $\boldsymbol{f}$ satisfies assumption* (H) *and admits a triangular representation $\boldsymbol{t}$ with coefficients in Q and multi-degree $\boldsymbol{d}$.*

Suppose that assumption $(\mathsf{H}')_{\boldsymbol{t},p}$ holds. Given $\boldsymbol{t}_0 = \boldsymbol{t} \bmod p$, one can compute $\boldsymbol{t} \bmod p^n$ in time

$$\mathcal{O}\left(\left(\mathsf{IT}(\boldsymbol{d}) + s^{\Omega}\mathsf{MT}(\boldsymbol{d})\right) + (sL + s^2)\mathsf{R}(n)\mathsf{MT}(\boldsymbol{d})\right).$$

In the running time, the first term corresponds to the inversion of the Jacobian matrix of $\boldsymbol{f}$ modulo $\mathcal{U}_0$, with coefficients in $R/(p)$; this allows us to initialize the lifting. Then, to reach precision n, most of the work consists in relaxed multiplications for p-adics in R_p at precision n, coupled with polynomial multiplication modulo a triangular set of multi-degree $\boldsymbol{d}$. The overall cost is quasi-linear in n.

For comparison, an adaptation of Newton's iteration for triangular sets (Schost, 2002) computes the same output in time

$$\mathcal{O}\left(\left(\mathsf{IT}(\boldsymbol{d}) + s^{\Omega}\mathsf{MT}(\boldsymbol{d})\right) + (sL + s^{\omega})\mathsf{M}(n)\mathsf{MT}(\boldsymbol{d})\right).$$

As the term $s^{\omega}\mathsf{M}(n)\mathsf{MT}(\boldsymbol{d})$ suggests, this algorithm requires one to do matrix multiplications, with entries that are multivariate polynomials of multi-degree $\boldsymbol{d}$, having themselves power series coefficients of precision n. Our solution requires no such full-precision matrix multiplication.

Let us next turn to the case of univariate representations. Our main result for this problem is the following.

Theorem 4. *Let $\boldsymbol{f} = (f_1, \ldots, f_s)$ be a polynomial system in $R[X_1, \ldots, X_s]$ given by a straight-line program Γ of size L. Suppose that $\boldsymbol{f}$ satisfies assumption* (H) *and admits a univariate representation $\mathcal{U}$ with coefficients in Q and degree d.*

Suppose that assumption $(\mathsf{H}')_{\mathcal{U},p}$ holds. Given $\mathcal{U}_0 = \mathcal{U} \bmod p$, one can compute $\mathcal{U} \bmod p^n$ in time

$$\mathcal{O}\left(\left(\mathsf{M}(d)\log(d) + s^{\Omega}\mathsf{M}(d)\right) + \left((L + s^2)\mathsf{R}(n) + sLn\right)\mathsf{M}(d)\right).$$

A univariate representation can be seen as a particular case of a triangular one, through the introduction of an extra unknown Z, and the equation expressing Z as a linear form in $X_1, \ldots, X_s$. However, Theorem 4 is not obtained by specializing the triangular case to the univariate one; further considerations are actually needed. We develop in Section 4.2 a technique suited for small number of *essential variables*, that is of variables $\{X_j | d_j > 1\}$ that actually appears in normal forms modulo a triangular set. Univariate representations are a typical application since they involve only one essential variable T.

The other known algorithm to perform this kind of task is a suitable version of Newton iteration, introduced in (Giusti et al., 2001; Heintz et al., 2001). To compute the same output, it runs in time

$$\mathcal{O}\left(\left(\mathsf{M}(d)\log(d) + s^{\Omega}\mathsf{M}(d)\right) + (sL + s^{\omega})\mathsf{M}(n)\mathsf{M}(d)\right).$$

As in the triangular case, the relaxed approach avoids matrix multiplication with p-adic multivariate entries.

As a general remark, relaxed lifting algorithms carry out (asymptotically in the precision) less relaxed multiplications than off-line (not relaxed) algorithms do off-line multiplications. In counterpart, the plain p-adic multiplication of Newton's iteration are slightly faster than the relaxed product used in our algorithm. Thus, there exists a trade-off between the dependencies in s and n for both algorithms.

1.5. Outline of the paper

One major advantage of online algorithms is their suitability to compute recursive p-adics. We present in Section 2 the framework of this application of online algorithms. In Section 3, we focus on arithmetics modulo a triangular set. Anticipating future needs, we develop an algorithm `Rem_quo` computing efficiently both the quotient and remainder modulo a triangular set. Section 4 provides shifted algorithms for a triangular set, which is the requirement to lift this triangular set using the framework of Section 2. Finally, we report on implementations in Section 5.

2. Online framework for recursive p-adics

This section presents online (or relaxed) algorithms and their use for computing recursive p-adics. We start by recalling the definition of online algorithms in Section 2.1. For this matter, we introduce a notion of *shift* of an algorithm that takes words as input and output. The rest of the section is devoted to the online solver of fixed point equations $\boldsymbol{y} = \boldsymbol{\Phi}(\boldsymbol{y})$. Recursive p-adics $\boldsymbol{y}$ are defined as solutions of special fixed point equations $\boldsymbol{y} = \boldsymbol{\Phi}(\boldsymbol{y})$ in Section 2.2. When $\boldsymbol{\Phi}$ is equipped with an evaluation algorithm of shift 1, we can use the relation $\boldsymbol{y} = \boldsymbol{\Phi}(\boldsymbol{y})$ to compute $\boldsymbol{y}$ from its first coefficient $\boldsymbol{y}_0$. The online algorithm that is behind this process is presented in Section 2.3. We present in Section 2.4 the framework used to build evaluation algorithms of shift 1 for $\boldsymbol{\Phi}$. Finally we recall in Section 2.5 the online linear algebra system solver, which is an application of the previous online solver.

The whole Section 2 is a natural extension of (Berthomieu and Lebreton, 2012, Section 2). We contribute by extending the framework to many recursive p-adics, by giving a more educational description of the online solver and by completing its proof.

2.1. Online algorithms

The idea behind online and shifted algorithms is to control the access to the p-adic coefficients of the input during the computation. The following definition of the shift of an algorithm is a generalization of the original definition of online algorithms given by (Hennie, 1966; Fischer and Stockmeyer, 1974).

Definition 5. Let us consider a Turing machine T with n inputs in Σ^* and one output in Δ^*, where Σ and Δ are sets. We denote by $\boldsymbol{a} = (a^{(1)}, \ldots, a^{(\ell)})$ an input sequence of T and, for all $1 \leqslant i \leqslant \ell$, we write $a^{(i)} = a_0^{(i)} a_1^{(i)} \ldots a_n^{(i)}$ with $a_j^{(i)} \in \Sigma$. We denote by $c_0 c_1 \ldots c_n$ the corresponding output, where $c_k \in \Delta$.

For an input index i, we say that T has *shift s with respect to its ith input* if T produces c_k before reading $a_j^{(i)}$ for all $0 \leqslant k < j + s$ and all input $\boldsymbol{a}$.

Also, we say that T has *shift s* if this is the case with respect to all its inputs.

Considering p-adics as the sequence of their coefficients, we obtain a notion of *shift* for algorithms with p-adic input and output.

Algorithms do not have a unique shift: if T has shift s with respect to its ith input, then any integer lesser than s is such a shift. In the last definition, we use Turing machines as a formal computation model that includes reads and writes on the input and output tapes.

The notion of *online* (or equivalently *relaxed*) algorithms is a specialization of the notion of shift.

Definition 6. A Turing machine T is online if and only if it has shift 0. Its ith input is an online argument if and only if T has shift 0 for this input.

Algorithms that are not online are called *off-line*, or equivalently *zealous* following the terminology of (van der Hoeven, 2002).

Let us give a few examples of online algorithms. For the sake of clarity, we will write our online algorithms iteratively, by regrouping the computations that lead to the production of a new coefficient of the output. This is particularly convenient to see the shift of an algorithm. It will also help us understand the order of the computations and therefore the correctness of the online recursive p-adics framework.

However, this presentation does not reflect how the algorithms are implemented in MATHEMAGIX. We refer to (van der Hoeven, 2002) and (Berthomieu et al., 2011, Section 5.1) for a description of the actual `C++` implementation of recursive p-adics in MATHEMAGIX.

Example 7. The first example of an online algorithm is the addition of p-adics. For computing the addition of p-adics a and b, we use a subroutine that takes as input another $c \in R_p$ and an integer i. The extra p-adic c stores the current state of the computation, while the integer i indicates the step of the computation we are at. Computations at step i are responsible for producing c_{i-1}; they complete those of previous steps to get a result at precision i, *i.e.* modulo p^i. Step 1 is additionally responsible for the initialization.

Algorithm 1: `NaiveAddStep`

Input: $a, b, c \in R_p$ and $i \in \mathbb{Z}$
Output: $c \in R_p$

1: **if** $i = 1$ **then** $c = 0$ \\ Initialization
2: **if** $i \geqslant 1$ **then** $c = c + (a_{i-1} + b_{i-1})p^{i-1}$
3: **return** c

The addition algorithm itself follows:

This addition algorithm is online: it outputs the coefficient c_i of the addition $c = a + b$ without using any a_j or b_j of index $j > i$. After each step i, c represents the sum of $a \bmod p^i$ and $b \bmod p^i$; thus, the result is correct modulo p^i.

Example 8. Let us introduce two new operators: for all s in $\mathbb{N}^*$, define

$$\begin{aligned} p^s \times \_ : R_p &\to R_p & \_/p^s : p^s R_p &\to R_p \\ a &\mapsto p^s a, & a &\mapsto a/p^s. \end{aligned}$$

Algorithm 2: `NaiveAdd`

Input: $a, b \in R_p$ and $n \in \mathbb{N}$
Output: $c \in R_p$ such that $c = (a + b) \bmod p^n$

1: **for** i **from** 1 **to** n **do**
2: $\quad c = \texttt{NaiveAddStep}(a, b, c, i)$
3: **return** c

The implementation of these operators just moves (or shifts) the coefficients of the input. It does not call any multiplication algorithm.

Algorithm 3: `OnlineShiftStep`

Input: $a, c \in R_p$, $s \in \mathbb{Z}$ and $i \in \mathbb{Z}$
Output: $c \in R_p$

1: **if** $i = 1$ **then** $c = 0$ \\ Initialization
2: **if** $i \geqslant \max(1, 1 + s)$ **then** $c = c + a_{(i-1)-s} p^{i-1}$
3: **return** c

Let $s \in \mathbb{Z}$ and denote by $\texttt{OnlineShift}(a, c, s, n)$ the algorithm that puts $\texttt{OnlineShiftStep}(a, c, s, i)$ in a loop with i varying from 1 to $n \in \mathbb{N}$. Then Algorithm $\texttt{OnlineShift}(a, c, s, n)$ has shift s with respect to its input a, and is online with respect to a if and only if $s \geqslant 0$.

2.2. Recursive p-adics

The study of online algorithms is motivated by their efficient implementation of recursive p-adics, which we introduce in this section. To the best of our knowledge, the paper (Watt, 1989) was the first to mention the lazy computation of power series which are solutions of a fixed point equation $y = \Phi(y)$. The paper (van der Hoeven, 2002), in addition to rediscovering the fast online multiplication algorithm of (Fischer and Stockmeyer, 1974), connected for the first time this fast online multiplication algorithm to the online computation of recursive power series. The article (Berthomieu et al., 2011) generalizes this setting to general p-adic rings. An issue about the correctness of the online recursive p-adics framework was raised and solved in (Berthomieu and Lebreton, 2012), leading to the notion of *shifted algorithm*.

We start by giving a definition of a vector of recursive p-adics.

Definition 9. Let $\ell \in \mathbb{N}$, $\boldsymbol{\Phi}$ a function from $(R_p)^\ell$ to $(R_p)^\ell$, $\boldsymbol{y} \in (R_p)^\ell$ be a fixed point of $\boldsymbol{\Phi}$, *i.e.* $\boldsymbol{y} = \boldsymbol{\Phi}(\boldsymbol{y})$. We write $\boldsymbol{y} = \sum_{i \in \mathbb{N}} \boldsymbol{y}_i p^i$ the p-adic decomposition of $\boldsymbol{y}$.

Then, we say that the coordinates $(y_1, \ldots, y_\ell)$ of $\boldsymbol{y}$ are *recursive* p-adics if for all $n \leqslant m \in \mathbb{N}^*$, the p-adic coefficient $(\boldsymbol{\Phi}(\boldsymbol{y}))_n$ does not depend on the coefficient $\boldsymbol{y}_m$, *i.e.* $(\boldsymbol{\Phi}(\boldsymbol{y}))_n = (\boldsymbol{\Phi}(\boldsymbol{y} + p^n \boldsymbol{a}))_n$ for any $\boldsymbol{a} \in (R_p)^\ell$.

The vector of p-adics $\boldsymbol{y}$ is called recursive since it can be computed recursively from its first coefficient $\boldsymbol{y}_0$ and $\boldsymbol{\Phi}$. Our definition of recursive p-adics extends the definition of (Berthomieu and Lebreton, 2012) to many recursive p-adics.

Example 10. Take $R = \mathbb{Q}[T]$ and $p = T$ so that $R_p = \mathbb{Q}[[T]]$. Consider the power series polynomial $\Phi(Y) = Y^2 + T \in (\mathbb{Q}[[T]])[Y]$.

The rational $y_0 = 0$ is a modular fixed point of Φ, *i.e.* $y_0 = \Phi(y_0) \bmod T$. Hensel's Lemma states that there exists a unique power series fixed point of Φ satisfying $y_0 = 0$, that is

$$y = \frac{1 - \sqrt{1 - 4T}}{2} = T + T^2 + 2T^3 + 5T^4 + \mathcal{O}(T^5).$$

Let us prove that the power series y is recursive. For any $b \in \mathbb{Q}[[T]]$, there exists, by Taylor expansion at y, a power series $\Theta(y, b)$ such that $\Phi(y+b)-\Phi(y) = \Phi'(y)b+\Theta(y, b)b^2$. Taking $b = T^n a$ and noticing that $(\Phi'(y))_0 = (\Phi'(y_0))_0 = 0$, we get

$$\Phi(y + T^n a) - \Phi(y) = \Phi'(y)T^n a + T^{2n}\Theta(y, T^n a) \in T^{n+1}\mathbb{Q}[[T]].$$

More generally, if $\Phi \in R_p[X_1, \ldots, X_\ell]^\ell$, $\Phi(y_0) = y_0 \bmod p$ and $\Phi'(y_0) = 0 \bmod p$, then the unique lifted fixed point y from y_0 is a recursive p-adic.

2.3. Online framework for computing recursive p-adics

In this section, we introduce the notion of *shifted algorithm* and present how it is used to compute a vector of recursive p-adics $\boldsymbol{y}$ from the relation $\boldsymbol{y} = \boldsymbol{\Phi}(\boldsymbol{y})$ when $\boldsymbol{\Phi}$ is a shifted algorithm.

Definition 11. Let $\boldsymbol{y} \in (R_p)^\ell$ be a vector of p-adics and $\boldsymbol{\Psi}$ be a Turing machine with ℓ inputs and ℓ outputs.

Then, $\boldsymbol{\Psi}$ is said to be a *shifted algorithm* that computes $\boldsymbol{y}$ if
(1) $\boldsymbol{y} = \boldsymbol{\Psi}(\boldsymbol{y})$,
(2) $\boldsymbol{\Psi}$ has shift 1.

The existence of a shifted algorithm $\boldsymbol{\Psi}$ for a vector of p-adics $\boldsymbol{y}$ implies the recursivity of $\boldsymbol{y}$. Indeed, the condition that $\boldsymbol{\Psi}$ has shift 1 implies that $(\boldsymbol{\Psi}(\boldsymbol{y}))_n = (\boldsymbol{\Psi}(\boldsymbol{y} + p^n\boldsymbol{a}))_n$ for any $\boldsymbol{a} \in (R_p)^\ell$ and $n \in \mathbb{N}$.

We now get down to the computation of $\boldsymbol{y}$ using the relation $\boldsymbol{y} = \boldsymbol{\Psi}(\boldsymbol{y})$. The sketch of the computation is the following: suppose that we are at the point where we know the p-adic coefficients $\boldsymbol{y}_0, \ldots, \boldsymbol{y}_{n-1}$ of $\boldsymbol{y}$ and $\boldsymbol{\Psi}(\boldsymbol{y})$ has been computed up to its $(n-1)$th coefficient. Since in the online framework, the computation is done step by step, one can naturally ask for one more step of the evaluation of $\boldsymbol{\Psi}$ at $\boldsymbol{y}$. Because $\boldsymbol{\Psi}$ has shift 1, the nth step of the computation of $\boldsymbol{\Psi}(\boldsymbol{y})$, which completes the computation of $(\boldsymbol{\Psi}(\boldsymbol{y})) \bmod p^{n+1}$, reads at most $\boldsymbol{y}_0, \ldots, \boldsymbol{y}_{n-1}$ and thus computes correctly. Now that we know $(\boldsymbol{\Psi}(\boldsymbol{y}))_n$, we can deduce $\boldsymbol{y}_n = (\boldsymbol{\Psi}(\boldsymbol{y}))_n$.

We sum this up in the next algorithm. We slice the computations of $\boldsymbol{\Psi}$ with respect to the coefficients of the output. For $i \geqslant 1$, the ith step of the computations of $\boldsymbol{\Psi}$ regroups all computations made strictly after the writing of the $(i-2)$th coefficient of the output until the writing of the $(i-1)$th coefficient of the output. Therefore, the ith step of $\boldsymbol{\Psi}$ is responsible for completing the computations of previous step in order to get the result modulo p^i.

It may seem that our algorithm allows to lift $\boldsymbol{y}$ from $\boldsymbol{\Psi}$ without requiring $\boldsymbol{y}_0$. In fact, $\boldsymbol{y}_0$ is encoded in $\boldsymbol{\Psi}$ since for any $a \in R_p$, $\boldsymbol{y}_0 = (\boldsymbol{\Psi}(\boldsymbol{y}))_0 = (\boldsymbol{\Psi}(\boldsymbol{y} + \boldsymbol{a}))_0$ which implies $\boldsymbol{y}_0 = (\boldsymbol{\Psi}(0))_0$ for instance.

Next proposition is the cornerstone of complexity estimates of recursive p-adics. It is a slight generalization of (Berthomieu and Lebreton, 2012, Proposition 10).

Algorithm 4: `OnlineRecursivePadic`

Input: a shifted algorithm $\mathbf{\Psi}$ and $n \in \mathbb{N}$
Output: $\boldsymbol{y} \in (R_p)^\ell$ at precision n

1: $\boldsymbol{a} = [0, \ldots, 0]$
2: **for** i **from** 1 **to** n **do**
3: Perform the ith step of the evaluation of $\mathbf{\Psi}$ at $\boldsymbol{a}$
4: Copy the p-adic coefficient $(\mathbf{\Psi}(\boldsymbol{a}))_{i-1}$ to $\boldsymbol{a}_{i-1}$
5: **return** $\boldsymbol{a}$

Proposition 12. *Let $\mathbf{\Psi}$ be a shifted algorithm for recursive p-adics $\boldsymbol{y}$. Then, Algorithm* `OnlineRecursivePadic` *is correct and computes the vector of recursive p-adics $\boldsymbol{y}$ at precision n in the time necessary to evaluate $\mathbf{\Psi}$ at $\boldsymbol{y}$ at precision n.*

Proof. We start with the proof of correctness of Algorithm `OnlineRecursivePadic`: let us show by induction for i from 1 to $n+1$ that $\boldsymbol{a}$ and $\boldsymbol{y}$ coincide modulo p^{i-1} before step i. We consider that the beginning of the loop for $i = n+1$ coincides with the end of the loop for $i = n$. For $i = 1$, it is trivial since the equality is modulo 1. Let us now assume the inductive hypothesis proved until step $i-1$. Therefore $\boldsymbol{a}$ and $\boldsymbol{y}$ coincide modulo p^{i-1} at the beginning of step i. Since the ith step of $\mathbf{\Psi}$ is known to read at most $\boldsymbol{a}_0, \ldots, \boldsymbol{a}_{i-2}$, which coincide with $\boldsymbol{y}_0, \ldots, \boldsymbol{y}_{i-2}$, we deduce that the evaluation of Step 3 produces the same output $\mathbf{\Psi}(\boldsymbol{y}) \bmod p^i$ than if it were executed on the solution $\boldsymbol{y}$. Therefore $(\mathbf{\Psi}(\boldsymbol{a}))_{i-1} = (\mathbf{\Psi}(\boldsymbol{y}))_{i-1} = \boldsymbol{y}_{i-1}$ and our inductive hypothesis follows.

As a consequence, the cost of the computation of $\boldsymbol{y}$ *is exactly the cost of the evaluation* of $\mathbf{\Psi}(\boldsymbol{y})$ in R_p. Indeed reads and writes of Step 4 are not accounted for in our complexity model. □

2.4. Creating shifted algorithms

This section is devoted to the process of creating shifted algorithms. We start by recalling the basics of straight line programs (s.l.p.s). Using these s.l.p.s, we build evaluation algorithms over p-adics based on online arithmetics, which are good candidates to be shifted algorithms. Finally, we introduce the notion of shift index of an s.l.p., which is the key to checking that our candidates are shifted algorithms.

Straight-line programs Straight-line programs are a model of computation that consists in ordered lists of instructions without branching. We give a short presentation of this notion and refer to (Bürgisser et al., 1997) for more details. We will use this model of computation to describe and analyze the forthcoming recursive operators and shifted algorithms.

Let R be a ring and A an R-algebra. A *straight-line program* is an ordered sequence of operations between elements of A. An *operation* of *arity* r is a map from a subset $\mathcal{D}$ of A^r to A. We usually work with the binary arithmetic operators $+, -, * : \quad \mathcal{D} = A^2 \to A$. We also define for $r \in R$ the 0-ary operations r^c whose output is the constant r and denote the set of all these operations by R^c. For $s \in \mathbb{N}^*$, we consider the unary operators $p^s \times \_ : A \to A$ and $\_/p^s : p^s A \to A$ defined in Example 8. Let us fix a set of operations Ω, *e.g.* $\Omega = \{+, -, *, p^s \times \_, \_/p^s\} \cup R^c$.

An s.l.p. starts with a number ℓ of *input* parameters indexed from $-(\ell-1)$ to 0. It has L *instructions* $\Gamma_1, \ldots, \Gamma_L$ with $\Gamma_i = (\omega_i; u_{i,1}, \ldots, u_{i,r_i})$ where $-\ell < u_{i,1}, \ldots, u_{i,r_i} < i$ and r_i is the arity of the operation $\omega_i \in \Omega$. The s.l.p. Γ is *executable* on $\boldsymbol{a} = (a_0, \ldots, a_{\ell-1})$ with *result sequence* $\boldsymbol{c} = (c_{-\ell+1}, \ldots, c_L) \in A^{\ell+L}$, if $c_i = a_{\ell-1+i}$ whenever $-(\ell-1) \leqslant i \leqslant 0$ and $c_i = \omega_i(c_{u,1}, \ldots, c_{u,r_i})$ with $(c_{u,1}, \ldots, c_{u,r_i}) \in \mathcal{D}_{\omega_i}$ whenever $1 \leqslant i \leqslant L$. Finally, the s.l.p. has a number r of outputs indexed by $1 \leqslant j_1, \ldots, j_r \leqslant L$. We then say that Γ computes $c_{j_1}, \ldots, c_{j_r}$ on the input $\boldsymbol{a}$.

Shift index of an s.l.p. Let $\Gamma = (\Gamma_1, \ldots, \Gamma_L)$ be an s.l.p. over the R-algebra R_p with ℓ input parameters and operations in Ω. We define the weighted directed acyclic graph $\mathcal{G}_\Gamma$ of Γ as the graph with $\ell + L$ nodes indexed by $-(\ell-1), \ldots, L$ and directed edges (u, j) for each input index u of the jth operation Γ_j. The edge (u, j) has weight 0 if $\omega_j \in \{+, -, *\}$. For any $s \in \mathbb{N}$, the edge (u, j) has weight s if $\omega_j = p^s \times \_$ and weight $-s$ if $\omega_j = \_/p^s$. Finally, we define the weight of any path $((v_0, v_1), \ldots, (v_{r-1}, v_r))$ as the sum of the weights of its edges (v_i, v_{i+1}).

The next definition of shift index is an extension of (Berthomieu and Lebreton, 2012, Definition 2) to s.l.p.s with many inputs and from a more global point of view.

Definition 13. For any pair (h, j) such that $-\ell < h \leqslant j \leqslant L$, we define the *shift index* $\mathrm{sh}(\Gamma, h \rightsquigarrow j) \in \mathbb{Z} \cup \{+\infty\}$ of (h, j) as the minimal weight of any path from h to j in the weighted directed graph associated to Γ. If there exists no such path, we set $\mathrm{sh}(\Gamma, h \rightsquigarrow j) = 0$ if $h = j$ and $\mathrm{sh}(\Gamma, h \rightsquigarrow j) = +\infty$ otherwise.

Moreover, we define

$$\begin{aligned} \mathrm{sh}(\Gamma, h \rightsquigarrow .) &:= \min\{\mathrm{sh}(\Gamma, h \rightsquigarrow j) | j \text{ an output index}\} \\ \mathrm{sh}(\Gamma) &:= \min\{\mathrm{sh}(\Gamma, h \rightsquigarrow .) | h \text{ an input index}\}. \end{aligned}$$

Example 14. This example is the continuation of Example 10. We still consider the power series polynomial $\Phi(Y) = Y^2 + T \in (\mathbb{Q}[[T]])[Y]$. We can naturally associate to Φ the s.l.p. with one input, one output, and instructions

$$\Gamma_1 = (*; 0, 0), \quad \Gamma_2 = (T^c), \quad \Gamma_3 = (+; 1, 2).$$

Let Γ' be the s.l.p. associated to $\Psi(Y) = T^2 \times ((Y/T)^2) + T$, that is the s.l.p. with one input, one output, and instructions

$$\Gamma'_1 = (\_/T; 0), \quad \Gamma'_2 = (*; 1, 1), \quad \Gamma'_3 = (T^2 \times \_; 2), \quad \Gamma'_4 = (T^c), \quad \Gamma'_5 = (+; 3, 4).$$

Then Γ' describes another way to compute the polynomial $Y^2 + T$.

We draw the directed acyclic graphs $\mathcal{G}_\Gamma$ and $\mathcal{G}_{\Gamma'}$ associated to Φ and Ψ in Figure 1. The weight of the unique path from the input (drawn by a double circle) to the output (drawn with an external arrow) is 0 for Φ and 1 for Ψ. So we have $\mathrm{sh}(\Phi) = 0$ and $\mathrm{sh}(\Psi) = 1$.

A convenient way to compute shift indices is the following. Let $-\ell < h \leqslant j \leqslant L$ and consider $\mathrm{sh}(\Gamma, h \rightsquigarrow j)$. The last edge of any path from h to j must originate from an input u of the jth operation as its last step. So

$$\mathrm{sh}(\Gamma, h \rightsquigarrow j) = \min\{\mathrm{sh}(h \rightsquigarrow u) + \mathrm{weight}(u, j) \mid u \text{ input index of the } j\text{th operation}\}.$$

Note that this is consistent with (Berthomieu and Lebreton, 2012, Definition 2).

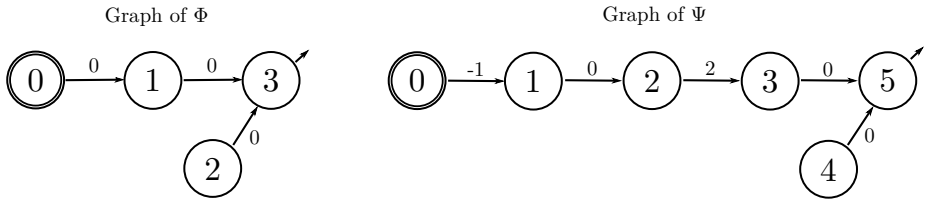

Figure 1. Weighted directed acyclic graphs of Φ and Ψ in Example 14.

Shifted evaluation algorithms Let Γ be given as an s.l.p. with operations in $\Omega = \{+, -, *, p^s \times \_, \_/p^s\} \cup R^c$. The upcoming Algorithm `ShiftedEvaluationStep` evaluates Γ at $\boldsymbol{a}$ in a way that controls the shift of the algorithm. Arithmetic operations are performed using online algorithms. Let `OnlineAddStep` (resp. `OnlineSubStep`, `OnlineMulStep`) be the step routine of any online addition (resp. subtraction, multiplication) algorithm (see Examples 7 and 8). Only online multiplication actually offers multiple implementation choices (see for instance (Lebreton and Schost, 2013, Section 2.3)).

Notice that for any operation j and any input index u of the jth operation, our choice of implementation is such that the algorithm has shift weight(u, j) with respect to its input u.

Algorithm 5: `OperationStep`

Input: an s.l.p. Γ, $[c_{-\ell+1}, \ldots, c_L] \in (R_p)^{\ell+L}$ and $i \in \mathbb{Z}$
Output: $[c_1, \ldots, c_L] \in (R_p)^L$

1: **for** j **from** 1 **to** L **do**
2: $\quad i' := i - \mathrm{sh}(\Gamma, j \rightsquigarrow .)$
3: $\quad$ **if** $(\Gamma_j = (\text{'}+\text{'}; u, v))$ **then** $c_j =$ `OnlineAddStep`(c_u, c_v, c_j, i')
4: $\quad$ **if** $(\Gamma_j = (\text{'}-\text{'}; u, v))$ **then** $c_j =$ `OnlineSubStep`(c_u, c_v, c_j, i')
5: $\quad$ **if** $(\Gamma_j = (\text{'}*\text{'}; u, v))$ **then** $c_j =$ `OnlineMulStep`(c_u, c_v, c_j, i')
6: $\quad$ **if** $(\Gamma_j = (p^s \times \_; u))$ **then** $c_j =$ `OnlineShiftStep`(c_u, c_j, s, i')
7: $\quad$ **if** $(\Gamma_j = (\_/p^s; u))$ **then** $c_j =$ `OnlineShiftStep`$(c_u, c_j, -s, i')$
8: $\quad$ **if** $(\Gamma_j = (r^c;))$ **then** $c_j = c_j + r_{i'} p^{i'}$
9: **return** $[c_1, \ldots, c_L]$

Algorithm 6: `ShiftedEvaluationStep`

Input: an s.l.p. Γ, $\boldsymbol{a} = (a_0, \ldots, a_{\ell-1}) \in (R_p)^\ell$, $[c_1, \ldots, c_L] \in (R_p)^L$ and $i \in \mathbb{Z}$
Output: $[c_1, \ldots, c_L] \in (R_p)^L$

1: **if** $i = 1$ **then** \\ Initialization
2: $\quad [c_{-\ell+1}, \ldots, c_0] = [a_0, \ldots, a_{\ell-1}]$; $[c_1, \ldots, c_L] = [0, \ldots, 0]$
3: $\quad$ **for** k **from** $1 + \min_{-\ell<j\leqslant L} (\mathrm{sh}(\Gamma, j \rightsquigarrow .))$ **to** 0 **do**
4: $\quad\quad [c_1, \ldots, c_L] =$ `OperationStep`$(\Gamma, [c_{-\ell+1}, \ldots, c_L], k)$
5: **if** $i \geqslant 1$ **then**
6: $\quad [c_1, \ldots, c_L] =$ `OperationStep`$(\Gamma, [c_{-\ell+1}, \ldots, c_L], i)$
7: **return** $[c_1, \ldots, c_L]$

Algorithm 7: `ShiftedEvaluation`

Input: an s.l.p. Γ, $\boldsymbol{a} = (a_0, \ldots, a_{\ell-1}) \in (R_p)^\ell$ and $n \in \mathbb{N}$
Output: $[c_1, \ldots, c_L] \in (R_p)^L$

1: **for** i **from** 1 **to** n **do**
2: $\quad [c_1, \ldots, c_L] =$ `ShiftedEvaluationStep`$(\Gamma, \boldsymbol{a}, [c_1, \ldots, c_L], i)$
3: **return** $[c_1, \ldots, c_L]$

Once again, Algorithm `ShiftedEvaluationStep` is a subroutine that corresponds to one step of following Algorithm `ShiftedEvaluation`, that computes the result sequence $[c_1, \ldots, c_L] \in (R_p)^L$ of the s.l.p. Γ on the input $\boldsymbol{a} \in (R_p)^\ell$. Note that we may need to perform a few more steps of `OperationStep` at initialization in order to guarantee that every step subroutine (*e.g.* `OnlineAddStep`) starts from $i' = 1$ even if $\mathrm{sh}(\Gamma, j \rightsquigarrow .) < 0$. We recall that our step routines do nothing when $i' \leqslant 0$ by design.

We can now prove that shift index and shift are compatible; the shift index of an s.l.p. Γ is a shift of Algorithm `ShiftedEvaluation`$(\Gamma, \boldsymbol{a}, n)$ with respect to its input $\boldsymbol{a}$.

Proposition 15. *Let $b_{-\ell+1}, \ldots, b_L$ be the result sequence of the s.l.p. Γ on input $\boldsymbol{a}$ and $j_1, \ldots, j_r$ be the indices of its output. Then Algorithm* `ShiftedEvaluation` *has shift* $\mathrm{sh}(\Gamma)$ *with respect to its input $\boldsymbol{a}$. Moreover, Algorithm* `ShiftedEvaluation` *outputs the result sequence at precision n,* i.e. *$c_{j_k} = b_{j_k} \bmod p^n$ for all $1 \leqslant k \leqslant r$.*

Proof. For the purposes of the proof, we introduce Algorithm `ShiftedEvaluationBis`. It is a variant of Algorithm `ShiftedEvaluation` that differs only in terms of presentation of the initialization step 1, which is now split in many steps i from $1 + \min_{-\ell<j\leqslant L}(\mathrm{sh}(\Gamma, j \rightsquigarrow .))$ to 1.

Algorithm 8: `ShiftedEvaluationBis`

Input: an s.l.p. Γ, $\boldsymbol{a} = (a_0, \ldots, a_{\ell-1}) \in (R_p)^\ell$ and $n \in \mathbb{N}$
Output: $[c_1, \ldots, c_L] \in (R_p)^L$

1: $\quad [c_{-\ell+1}, \ldots, c_0] = [a_0, \ldots, a_{\ell-1}]$; $[c_1, \ldots, c_L] = [0, \ldots, 0]$
2: $\quad$ **for** i **from** $1 + \min_{-\ell<j\leqslant L}(\mathrm{sh}(\Gamma, j \rightsquigarrow .))$ **to** n **do**
3: $\quad\quad [c_1, \ldots, c_L] =$ `OperationStep`$(\Gamma, [c_{-\ell+1}, \ldots, c_L], k)$
4: $\quad$ **return** $[c_1, \ldots, c_L]$

Let $(c_j)_{-\ell<j\leqslant L}$ be the variables of `ShiftedEvaluationBis` and denote by s the integer $-\min_{-\ell<j\leqslant L}(\mathrm{sh}(\Gamma, j \rightsquigarrow .)) \geqslant 0$. For any pair $(i, j) \in [\![1 - s; n]\!] \times [\![-\ell; L]\!]$, we let $c_{(i,j)}$ be the truncated p-adic $c_j \bmod p^{i'}$ where $i' := i - \mathrm{sh}(\Gamma, j \rightsquigarrow .)$. If $i' \leqslant 0$, we set $c_{(i,j)} = 0$ by convention. Algorithm `ShiftedEvaluationBis` computes progressively the p-adics $c_{(i,j)}$ for $(i, j) \in [\![1 - s; n]\!] \times [\![1; L]\!]$ following the lexicographic order $<_{\mathrm{lex}}$ with $(0, 1) <_{\mathrm{lex}} (1, 0)$. Let us prove by induction on $(i, j) \in [\![1 - s; n]\!] \times [\![-\ell; L]\!]$ that $c_{(i,j)} = b_{(i,j)}$ and, if $j \geqslant 0$, that the (i, j)th step of the computation reads at most $c_{(k,u)}$ for $(k, u) <_{\mathrm{lex}} (i, j)$.

We initiate the recursion: $c_{(1-s,-\ell+1)}$ is an input of Γ, so we have the equalities $c_{-\ell+1} = a_{-\ell+1} = b_{-\ell+1}$, which also hold modulo any power of p. Let us now assume that the inductive hypothesis is verified for all $(k, m) <_{\mathrm{lex}} (i, j)$. If $j \leqslant 0$ then $c_{(i,j)}$ corresponds to an input and $c_{(i,j)} = b_{(i,j)}$ as before. Otherwise $j > 0$ and $c_{(i,j)}$ is obtained from the

i'th step of the jth operation. Let $u < j$ be the index of an input of the jth operation. Then, the current computation reads at most $c_{(u,k)}$ for $k' \leqslant i' - \mathrm{weight}(u,j)$. Since

$$\mathrm{sh}(\Gamma, u \rightsquigarrow .) \leqslant \mathrm{weight}(u,j) + \mathrm{sh}(\Gamma, j \rightsquigarrow .),$$

we obtain $k \leqslant i$ and therefore $(k,u) \leqslant_{\mathrm{lex}} (i,u) <_{\mathrm{lex}} (i,j)$ which proves the second part of our induction.

By the inductive hypothesis, all the coefficients $c_{(k,u)}$ that are read by the jth operation up to step i coincide with $b_{(k,u)}$. Therefore the jth operation produces the same output than if it were executed on the inputs $b_{-\ell+1}, \ldots, b_L$. Since the loop on i starts from $1-s$, we have performed all the steps from 1 to i' of the jth operation, whose output c_j consequently equals to b_j modulo $p^{i'}$, *i.e.* $c_{(i,j)} = b_{(i,j)}$. This concludes our induction.

Using our induction, the ith step of `ShiftedEvaluationBis` reads at most $a_j \bmod p^{i-\mathrm{sh}(\Gamma, j\rightsquigarrow .)}$ for any input index $-\ell < j \leqslant 0$. The first part of the proposition follows from $\mathrm{sh}(\Gamma) \leqslant \mathrm{sh}(\Gamma, j \rightsquigarrow .)$ for any input index $-\ell < j \leqslant 0$. The second part of the proposition is a direct consequence of our induction on couples (n, j_k) for all $1 \leqslant k \leqslant r$. □

Corollary 16. *Let $\boldsymbol{y} \in (R_p)^\ell$ be a vector of p-adics and $\boldsymbol{\Psi}$ be an s.l.p. of positive shift index $\mathrm{sh}(\boldsymbol{\Psi}) \geqslant 1$ such that $\boldsymbol{y} = \boldsymbol{\Psi}(\boldsymbol{y})$. Then Algorithm `ShiftedEvaluation`$(\boldsymbol{\Psi}, \_, n)$ is a shifted algorithm that computes $\boldsymbol{y}$.*

Example 17. This example is the continuation of Examples 10 and 14. In particular, we keep the notations Φ and Ψ for the two s.l.p.s respectively associated to $Y^2 + T$ and $T^2 \times ((Y/T)^2) + T$. The two s.l.p.s are related: if $a = 0 \bmod T$, then $\Phi(a) = \Psi(a)$. In particular, Φ and Ψ give fixed point equations for the recursive p-adic y, *i.e.* $\Psi(y) = \Phi(y) = y$.

Since $\mathrm{sh}(\Psi) = 1$, the s.l.p. Ψ satisfies the conditions of Corollary 16 and can be used to compute y accordingly to Proposition 12. On the contrary, $\mathrm{sh}(\Phi) = 0$ and its associated evaluation algorithm `ShiftedEvaluation`$(\Phi, \_, n)$ does not introduce the shift in the coefficients of y necessary to be able to compute y.

Remark 18. One drawback of the online method for computing recursive p-adics is the space complexity. We have seen that we store the current state of each computation of Γ in Algorithm `ShiftedEvaluation`. This leads to a space complexity $\mathcal{O}(nL)$ to compute the recursive p-adic at precision n where L is the size of Γ.

In Newton's iteration approach, the required evaluation of Γ could use significantly less memory by freeing the result of a computation as soon as it is used for the last time. For this reason, Newton's iteration may consume less memory.

2.5. Online linear algebra system solver

One of the building blocks that will be required to lift triangular sets is an online p-adic linear system solver. In this section, we recall the main result concerning the online linear system solver of (Berthomieu and Lebreton, 2012, Section 4). This solver is built upon the online recursive p-adic framework. We refer to the original paper for more details.

We consider a linear system of the form $\boldsymbol{A} = \boldsymbol{B} \cdot \boldsymbol{C}$, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are known, and $\boldsymbol{C}$ is the unknown. The matrix $\boldsymbol{A}$ belongs to $\mathcal{M}_{m,1}(R_p)$ and $\boldsymbol{B} \in \mathcal{M}_m(R_p)$ is invertible; we solve the linear system $\boldsymbol{A} = \boldsymbol{B} \cdot \boldsymbol{C}$ for $\boldsymbol{C} \in \mathcal{M}_{m,1}(R_p)$.

Proposition 19 (Berthomieu and Lebreton, 2012, Proposition 21)**.** *With the previous notations and given* $\boldsymbol{C}_0$ *and* $\Gamma := (\boldsymbol{B}_0)^{-1} \bmod p$*, there exists an algorithm* `LinearSolver` *online with respect to inputs* $\boldsymbol{A}$ *and* $\boldsymbol{B}$ *that outputs the linear system solution* $\boldsymbol{C}$ *at precision* n *in time* $\mathcal{O}\left(m^2\mathsf{R}(n)\right)$*.*

We extend the set of operations Ω of our s.l.p.s with this new operation `LinearSolver` and, if $\Gamma_j = (\texttt{LinearSolver}; u, v)$, we set the weight of the edges (u, j) and (v, j) as 0. Since this Algorithm `LinearSolver` is online, this choice of shift index is consistent with Proposition 15 and Corollary 16.

3. Quotient and remainder modulo a triangular set

This section deals with Euclidean division modulo a triangular set. In this section, we denote by $\boldsymbol{t} = (t_1, \ldots, t_s)$ a triangular set in $R[X_1, \ldots, X_s]$, with R being any ring. Computing remainders is a basic operation necessary to be able to compute with the quotient algebra $A := R[X_1, \ldots, X_s]/(\boldsymbol{t})$. We are also interested in the quotients of the division since we will need them later.

We start by defining quotients and remainder of the Euclidean division by $\boldsymbol{t}$ in a unique manner in Section 3.1. Then we focus on computing these objects. We circumvent the fact that the size of the quotients is exponential in the size $d_1 \cdots d_s$ of the quotient algebra A by computing only reductions of the quotients modulo a triangular set. This leads us to Algorithms `Rem` and `Rem_quo` of Section 3.2. In this section, we will consider only arithmetic complexity over R.

3.1. Canonical quotients and remainder

For any $g \in R[X_1, \ldots, X_s]$, the existence of $r, q_1, \ldots, q_s \in R[X_1, \ldots, X_s]$ satisfying $g = r + q_1 t_1 + \cdots + q_s t_s$ and $\deg_{X_i}(r) < d_i$ is guaranteed because triangular sets are Gröbner bases. However, the quotients $q_1, \ldots, q_s$ are not necessarily unique. For $1 \leqslant i < j \leqslant s$, let $z_{i,j}$ be the vector of $R[X_1, \ldots, X_s]^s$ with only t_j in the ith position and $-t_i$ in the jth position. We can add to any choice of quotients $(q_1, \ldots, q_s)$ an element of the syzygy $R[X_1, \ldots, X_s]$-module spanned by the $(z_{i,j})_{1 \leqslant i < j \leqslant s}$ in $R[X_1, \ldots, X_s]^s$. Nevertheless, a canonical choice of quotient can be made.

Lemma 20. *For all* $g \in R[X_1, \ldots, X_s]$*, there exists a unique vector of polynomials* $(r, q_1, \ldots, q_s)$ *in* $R[X_1, \ldots, X_s]^{s+1}$ *such that*

$$g = r + q_1 t_1 + \cdots + q_s t_s$$

and for all $1 \leqslant i \leqslant s$*,* $\deg_{X_i}(r) < d_i$ *and for all* $1 \leqslant i < j \leqslant s$*,* $\deg_{X_j}(q_i) < d_j$*.*

Proof. Take any Euclidean decomposition $g = r + q_1 t_1 + \cdots + q_s t_s$ with $\deg_{X_i}(r) < d_i$. Then use the syzygies $(z_{1,i})_{1<i\leqslant s}$ to reduce the degree of q_1 in $X_2, \ldots, X_s$. Again use the syzygies $(z_{2,i})_{2<i\leqslant s}$ to reduce the degree of q_2 in $X_3, \ldots, X_s$. This last action does not change q_1. Continuing the process until we reduce the degree of q_{s-1} in X_s by $z_{s-1,s}$, we have exhibited a Euclidean decomposition satisfying the hypothesis of the lemma.

Now let us prove the uniqueness of $(r, q_1, \ldots, q_s)$. Because r is unique, we have to prove that if $q_1 t_1 + \cdots + q_s t_s = 0$ with $\deg_{X_j}(q_i) < d_j$ for all $1 \leqslant i < j \leqslant s$, then $q_1 = \cdots = q_s = 0$. By contradiction, we suppose there is such a decomposition with a

non-zero q_i. Let j be the maximal index of a non-zero q_j. Then $\deg_{X_j}(q_j t_j) \geqslant d_j$ and at the same time

$$\begin{aligned}\deg_{X_j}(q_j t_j) &= \deg_{X_j}(q_1 t_1 + \cdots + q_{j-1} t_{j-1}) \\ &\leqslant \max_{i<j}(\deg_{X_j}(q_i t_i)) \\ &\leqslant \max_{i<j}(\deg_{X_j}(q_i)) \\ &< d_j.\end{aligned}$$

Contradiction. □

We call canonical quotients and remainder, those which satisfy the conditions of Lemma 20. We denote by $g \operatorname{rem} \boldsymbol{t}$ the canonical remainder of g modulo $\boldsymbol{t}$. We will not compute the whole quotients q_i because they suffer from the phenomenon of *intermediate expression swell*; in the computations of the remainder of a polynomial modulo a triangular set, the size of intermediate expressions, e.g. the size of the quotients, increases too much. A quick estimate gives that the number of monomials of the quotients is exponential in the size $d_1 \cdots d_s$ of the quotient algebra A (Langemyr, 1991).

Instead, we will show how to compute modular reductions of the quotients.

3.2. Fast multivariate Euclidean division by a triangular set

Recall that d_i denotes the degree in X_i of t_i. For any triangular set $\boldsymbol{t}$ in $R[X_1, \ldots, X_s]$, we say that the variables $\{X_j | d_j > 1\}$ are the *essential variables* and denote by $e := \#\{i | d_i > 1\}$ their number. Only those variables play a true role in the quotient algebra $A := R[X_1, \ldots, X_s]/(\boldsymbol{t})$. If r is a reduced normal form modulo $\boldsymbol{t}$ (seen as a Gröbner basis for the lexicographic order $X_1 \ll \cdots \ll X_s$), then r is written on the e essential variables. For the sake of simplicity in our forthcoming algorithms, we will assume that the set of indices $\{i | d_i > 1\}$ of essential variables is $\{1, \ldots, e\}$, so that any reduced normal form r belongs to $R[X_1, \ldots, X_e]$.

We start with Algorithm `Rem`, which is a variant of the algorithm of (Li et al., 2009). It is designed to reduce the product of two reduced elements, therefore we suppose that the input polynomial $g \in R[X_1, \ldots, X_e]$ satisfies $\deg_{X_i}(g) \leqslant 2(d_i - 1)$. We denote by $\operatorname{rev}_{X_i}(g)$ the reverse polynomial of $g \in R[X_1, \ldots, X_s]$ with respect to X_i, *i.e.* $\operatorname{rev}_{X_i}(g) := X_i^{\deg_{X_i}(g)} g(1/X_i)$.

The precomputation of step 4 means that, as the object I depends only on $\boldsymbol{t}$, we compute it once and for all at the first call of Algorithm `Rem`.

Let $\mathsf{M}(d_1, \ldots, d_s) = \mathsf{M}(\boldsymbol{d})$ denote the cost of multiplication of dense multivariate polynomials $g \in R[X_1, \ldots, X_s]$ satisfying $\deg_{X_i}(g) < d_i$ for all $1 \leqslant i \leqslant s$. By Kronecker substitution, we get that $\mathsf{M}(\boldsymbol{d}) = \mathcal{O}(\mathsf{M}(2^s d_1 \cdots d_s))$ (von zur Gathen and Gerhard, 2013).

We define $\mathsf{Rem}(d_1, \ldots, d_s)$ to be the arithmetic cost of reducing polynomials $g \in R[X_1, \ldots, X_s]$ satisfying $\deg_{X_i}(g) \leqslant 2(d_i - 1)$ modulo $\boldsymbol{t}$. The number e of *essential variables* plays an important role because $\mathsf{Rem}(\boldsymbol{d})$ is exponential in e.

Proposition 21. *The algorithm `Rem` is correct and has cost*

$$\mathsf{Rem}(d_1, \ldots, d_e) = \mathcal{O}(3^e d \log(d) \log(\log(d))).$$

Algorithm 9: `Rem`

Input: $g \in R[X_1, \ldots, X_e]$ and $\boldsymbol{t}$ such that $\deg_{X_i}(g) \leqslant 2(d_i - 1)$
Output: $r = g \operatorname{rem} \boldsymbol{t} \in R[X_1, \ldots, X_e]$

1: Let $\boldsymbol{t}' := (t_1, \ldots, t_{e-1})$ and $R' := R[X_1, \ldots, X_{e-1}]/(\boldsymbol{t}')$
2: Compute the quotient q_e of g by t_e in $R'[X_e]$:
3: $q_e := \sum_{i=d_e}^{2d_e-1} \mathtt{Rem}(g[X_e^i], \boldsymbol{t}')X_e^i$
4: Precompute $I := 1/\operatorname{rev}_{X_e}(t_e) \operatorname{rem} X_e^{d_e-1}$ in $R'[X_e]$
5: $q_e := \operatorname{rev}_{X_e}(q_e) I \operatorname{rem} X_e^{d_e}$ in $R'[X_e]$
6: $q_e := \operatorname{rev}_{X_e}(q_e)$
7: $r := (g - q_e t_e) \operatorname{rem} X_e^{d_e}$ in $R[X_1, \ldots, X_{e-1}][X_e]$
8: $r := \sum_{i=0}^{d_e-1} \mathtt{Rem}(r[X_e^i], \boldsymbol{t}')X_e^i$
9: **return** r

Proof. The algorithm is a multivariate generalization of the fast univariate division with remainder (von zur Gathen and Gerhard, 2013, Section 9.1). We refer to (Li et al., 2009) for the proof of correction of our algorithm.

Let us focus on the complexity. Step 5 involves a multiplication in $R[X_1, \ldots, X_e]$ and a reduction by $\boldsymbol{t}'$ of the coefficients in X_e^i for $i < d_e$. So multivariate multiplications are used in steps 5 and 7 and d_e reductions by $\boldsymbol{t}'$ are done in steps 3, 5 and 7. Thus the complexity analysis becomes

$$\begin{aligned}
\mathsf{Rem}(d_1, \ldots, d_e) &= 3d_e\mathsf{Rem}(d_1, \ldots, d_{e-1}) + 2\mathsf{M}(d_1, \ldots, d_e) \\
&= 3d_e\mathsf{Rem}(d_1, \ldots, d_{e-1}) + 2\mathsf{M}(2^e d_1 \cdots d_e) \\
&= 3d_e\mathsf{Rem}(d_1, \ldots, d_{e-1}) + \mathcal{O}(2^e d \log(2^e d) \log(\log(2^e d)))
\end{aligned}$$

Unrolling the recurrence, we get

$$\begin{aligned}
\mathsf{Rem}(d_1, \ldots, d_e) = \mathcal{O}\big(&2^e d \log(2^e d) \log(\log(2^e d)) + \\
&+3 \cdot 2^{e-1} d \log(2^{e-1} d/d_e) \log(\log(2^{e-1} d/d_e)) + \cdots \\
&+3^{e-1} d \log(d_1) \log(\log(d_1))\big) \\
= \mathcal{O}(&3^e d \log(d) \log(\log(d))).
\end{aligned}$$

The precomputation of step 4 is an inversion of power series done by Newton iteration. The initialization is trivial; indeed $1/\operatorname{rev}_{X_e}(t_e) = 1 \bmod X_e$ since t_e is monic. The cost of the Newton iteration is big-O of the cost of the last step, which is $\mathsf{M}(d_1, \ldots, d_e) + d_e\mathsf{Rem}(d_1, \ldots, d_{e-1})$. So the precomputation can be done in time $\mathcal{O}\left(\mathsf{Rem}(d_1, \ldots, d_e)\right)$. □

Our algorithm is a slight improvement of the algorithm of (Li et al., 2009); the exponential factor in the complexity is 3^e instead of 4^e.

We obtain the complexity estimate of modular multiplication given in Proposition 2 of the introduction as corollary of Proposition 21:

$$\mathsf{MT}(\boldsymbol{d}) = \mathsf{M}(\boldsymbol{d}) + \mathsf{Rem}(\boldsymbol{d}) = \mathcal{O}\left(3^e d \log(d) \log(\log(d))\right).$$

Remark that non-essential variables do not impact the complexity of our algorithm, *i.e.* $\mathsf{Rem}(\boldsymbol{d}) = \mathsf{Rem}(d_1, \ldots, d_e)$, and the same for the modular multiplication.

Now we adapt Algorithm `Rem` to keep the quotients modulo another triangular set $\boldsymbol{t}^{(2)} = (t_1^{(2)}, \ldots, t_s^{(2)})$. In order to simplify the algorithm and because it suits our future needs, we assume that for all i, $\deg_{X_i}(t_i^{(1)}) = \deg_{X_i}(t_i^{(2)})$ and still denote by d_i this degree. Once again, our algorithm is designed to take the product of two reduced elements as input, so we suppose that $\deg_{X_i}(g) \leqslant 2(d_i - 1)$.

Algorithm 10: `Rem_quo`

Input: $g \in R[X_1, \ldots, X_e]$ s.t. $\deg_{X_i}(g) \leqslant 2(d_i - 1)$ and triangular sets $\boldsymbol{t}^{(1)}$, $\boldsymbol{t}^{(2)}$.
Output: $r, q_1, \ldots, q_e \in R[X_1, \ldots, X_e]$ reduced modulo $\boldsymbol{t}^{(1)}$ such that

$$g = r + \sum_{i=1}^{e} q_i t_i^{(1)} \quad \text{modulo}(\boldsymbol{t}^{(1)})(\boldsymbol{t}^{(2)}).$$

1: Let $\boldsymbol{t}^{(2)\prime} := (t_1^{(2)}, \ldots, t_{e-1}^{(2)})$ and $R' := R[X_1, \ldots, X_{e-1}]/(\boldsymbol{t}^{(2)\prime})$.
2: Compute the quotient q_e of g by $t_e^{(1)}$ in $R'[X_e]$:
3: $\quad q_e := \sum_{i=d_e}^{2d_e-1} \mathsf{Rem}(g[X_e^i], \boldsymbol{t}^{(2)\prime}) X_e^i$
4: $\quad$ Precompute $I := 1/\operatorname{rev}_{X_e}(t_e^{(1)}) \operatorname{rem} X_e^{d_e-1}$ in $R'[X_e]$
5: $\quad q_e := (\operatorname{rev}_{X_e}(q_e) I) \operatorname{rem} X_e^{d_e}$ in $R'[X_e]$
6: $\quad q_e := \operatorname{rev}_{X_e}(q_e)$
7: $r := (g - q_e t_e^{(1)})$ in $R[X_1, \ldots, X_e]$
8: $r_1 := r \operatorname{rem} X_e^{d_e}$, $r_2 = r - r_1$
9: $0, q_1, \ldots, q_{e-1} := \sum_{i=d_e}^{2d_e-1} \mathtt{Rem\_quo}(r_2[X_e^i], \boldsymbol{t}^{(2)\prime}, (t_1^{(1)}, \ldots, t_{e-1}^{(1)})) X_e^i$
10: **for** i **from** 1 **to** $e-1$ **do**
11: $\quad q_i' := \mathsf{Rem}(q_i, \boldsymbol{t}^{(1)})$
12: $r := r_1 + q_1' t_1^{(2)} + \cdots + q_{e-1}' t_{e-1}^{(2)}$ in $R[X_1, \ldots, X_e]$
13: $r, q_1, \ldots, q_{e-1} := \sum_{i=0}^{d_e-1} \mathtt{Rem\_quo}(r[X_e^i], \boldsymbol{t}^{(1)\prime}, \boldsymbol{t}^{(2)\prime}) X_e^i$
14: **return** $r, q_1, \ldots, q_{e-1}, q_e$

Lemma 22. *If r is reduced modulo $\boldsymbol{t}^{(1)}$ and $g = r + \sum_{i=1}^{e} q_i t_i^{(1)}$ modulo the product ideal $(\boldsymbol{t}^{(1)})(\boldsymbol{t}^{(2)})$, then r is the reduced normal form of g modulo $\boldsymbol{t}^{(1)}$ and*

$$g = r + \sum_{i=1}^{e} q_i t_i^{(1)} \quad \text{modulo } \boldsymbol{t}^{(2)}.$$

Proof. Since the product ideal $(\boldsymbol{t}^{(1)})(\boldsymbol{t}^{(2)})$ is included in both the ideals $(\boldsymbol{t}^{(1)})$ and $(\boldsymbol{t}^{(2)})$, the relation $g = r + \sum_{i=1}^{e} q_i t_i^{(1)}$ stands modulo both these ideals. In particular $g = r$ modulo $\boldsymbol{t}^{(1)}$ and, since r is reduced modulo $\boldsymbol{t}^{(1)}$, it is the reduced normal form of g modulo $\boldsymbol{t}^{(1)}$. □

We denote by $\mathsf{RemQuo}(d_1, \ldots, d_e)$ the complexity of `Rem_quo` for triangular sets $\boldsymbol{t}^{(1)}$ and $\boldsymbol{t}^{(2)}$ of same degrees $d_1, \ldots, d_e$.

Proposition 23. *Algorithm* `Rem_quo` *is correct and its cost is*

$$\mathsf{RemQuo}(d_1, \ldots, d_e) = \mathcal{O}\left(e\mathsf{Rem}(d_1, \ldots, d_e)\right) = \mathcal{O}\left(e\mathsf{MT}(d_1, \ldots, d_e)\right).$$

Proof. We proceed recursively on the number e on variables involved in g. In one variable, our algorithm coincides with the fast univariate division with remainder (see (von zur Gathen and Gerhard, 2013, Section 9.1)). So it is correct and $\mathsf{RemQuo}(d_1) = \mathsf{Rem}(d_1)$.

Let's suppose that we have proved our claims in less that e variables. Since q_e is the quotient of g by $t_e^{(1)}$ in $(R[X_1, \ldots, X_{e-1}]/(\boldsymbol{t}^{(2)\prime}))[X_e]$, we have $r_2 = 0$ in $(R[X_1, \ldots, X_{e-1}]/(\boldsymbol{t}^{(2)\prime}))[X_e]$. By assumption, the recursive call of step 9 gives the decomposition

$$r_2 = \sum_{i=1}^{e-1} q_i t_i^{(2)} \quad \text{modulo}(\boldsymbol{t}^{(1)\prime})(\boldsymbol{t}^{(2)\prime})$$

and also modulo $(\boldsymbol{t}^{(1)})(\boldsymbol{t}^{(2)})$. The reduction of the quotient of step 11 gives

$$r_2 = \sum_{i=1}^{e-1} q_i' t_i^{(2)} \quad \text{modulo}(\boldsymbol{t}^{(1)})(\boldsymbol{t}^{(2)})$$

where the polynomials q_i' are reduced modulo $\boldsymbol{t}^{(1)}$. Therefore the polynomial $r_2' := \sum_{i=1}^{e-1} q_i' t_i^{(2)}$ has degree $\deg_{X_e}(r_2') < d_e$ and $\deg_{X_i}(r_2') < 2d_i$ for $i < e$. Because r_1 satisfies the same degree conditions, they are still satisfied by $r = r_1 + r_2'$. By the induction hypothesis, at step 13, we have for all $0 \leqslant i < d_e$ and $1 \leqslant j \leqslant e-1$, that $q_j[X_e^i]$ is reduced modulo $\boldsymbol{t}^{(1)\prime}$. Since q_j has degree less than d_e in X_e, it is reduced modulo $\boldsymbol{t}^{(1)}$. The last quotient q_e is also reduced because it was computed in $(R[X_1, \ldots, X_{e-1}]/(\boldsymbol{t}^{(2)\prime}))[X_e]$ and $\deg_{X_e}(q_e) = \deg_{X_e}(g) - \deg_{X_e}(t_e^{(1)}) < d_e$. Finally

$$\begin{aligned} g = r_1 + r_2 + q_e t_e^{(1)} &= (r_1 + r_2') + q_e t_e^{(1)} \,\text{modulo}(\boldsymbol{t}^{(1)})(\boldsymbol{t}^{(2)}) \\ &= \left(r + \sum_{i=1}^{e-1} q_i t_i^{(1)}\right) + q_e t_e^{(1)} \,\text{modulo}(\boldsymbol{t}^{(1)})(\boldsymbol{t}^{(2)}). \end{aligned}$$

Concerning the complexity analysis, we have

$$\begin{aligned} \mathsf{RemQuo}(d_1, \ldots, d_e) = {} & 2d_e\mathsf{RemQuo}(d_1, \ldots, d_{e-1}) + 2d_e\mathsf{Rem}(d_1, \ldots, d_{e-1}) + \\ & (e-1)\mathsf{Rem}(d_1, \ldots, d_e) + (e+1)\mathsf{M}(d_1, \ldots, d_e) \end{aligned}$$

which gives

$$\mathsf{RemQuo}(d_1, \ldots, d_e) = \mathcal{O}\left(e\mathsf{Rem}(d_1, \ldots, d_e)\right). \qquad \square$$

As for the remainder algorithm, we have $\mathsf{RemQuo}(\boldsymbol{d}) = \mathsf{RemQuo}(d_1, \ldots, d_e)$, so that $\mathsf{RemQuo}(\boldsymbol{d}) = \mathcal{O}\left(e\mathsf{Rem}(\boldsymbol{d})\right)$.

Finally, we mention that there exists a different approach for computing remainders modulo a triangular set (Bostan et al., 2011): it relies roughly on an evaluation / interpolation on the points of the variety defined by the triangular set. Because this approach can not be adapted to obtain the quotients, we did not consider it here.

3.3. Shift index of remainder and quotient

We now specify the base ring to be the ring of p-adics R_p.

Lemma 24. *If arithmetic operations are performed in R_p by online algorithms, then both Algorithms* `Rem` *and* `Rem_quo` *are online.*

Proof. We prove it for `Rem_quo`, the other case being similar. We proceed by induction on the number s of variables involved in the input of $\boldsymbol{t}^{(1)}$. If no variables are involved, then the result is trivial. From now on, let us assume that the result is valid for input of less than s variables.

First, we prove that the computations that leads to $I := 1/\operatorname{rev}_{X_s}(t_s) \operatorname{rem} X_s^{d_s-1}$ in $R'[X_s]$ are online. Define $I_0 := 1$ and $I_\ell := I_{\ell-1} - I_{\ell-1}(\operatorname{rev}_{X_\ell}(t_\ell) I_{\ell-1} - 1)$ in $R'[X_s]/(X_s^\ell)$. Thereby $I = I_{\lceil \log_2(d_s-1) \rceil}$ modulo $X_s^{d_s-1}$. We conclude by noticing that I_ℓ is obtained from $I_{\ell-1}$ by an online multiplication and reductions modulo $\boldsymbol{t}'$ which are also online.

Our algorithm uses only recursive calls in less variables, online additions and multiplications. Since the composition of online algorithms is online, the lemma is proved. □

We extend one last time our set of operations Ω with the new operations `Rem` and `Rem_quo`. Let Γ be an s.l.p. whose jth operation is either `Rem` and `Rem_quo` and let u be one of the input indices, then we define $\operatorname{weight}(u, j) := 0$. Since these algorithms are online by Lemma 24, this choice of shift index is consistent with Proposition 15 and Corollary 16.

4. Relaxed lifting of triangular sets

In this section we detail two relaxed algorithms that lift triangular sets. The first algorithm of Section 4.1 should be used for triangular set of generic multi-degree. We refine this algorithm in Section 4.2 for triangular sets with few essential variables, which is the case of univariate representations.

Suppose that $\boldsymbol{f}$ satisfies assumption (H) and admits a triangular representation $\boldsymbol{t}$ with coefficients in Q and multi-degree $\boldsymbol{d}$. Suppose that assumption $(\mathsf{H}')_{\boldsymbol{t},p}$ holds and that we know $\boldsymbol{t}_0 = \boldsymbol{t} \bmod p$. Hypothesis $(\mathsf{H}')_{\boldsymbol{t},p}$ implies that the Jacobian matrix $\operatorname{Jac}(\boldsymbol{f}_0)$ is invertible over $(R/(p))[X_1, \ldots, X_s]/(\boldsymbol{t}_0)$. Our objective is to compute $\boldsymbol{t}$ from $\boldsymbol{f}$ and $\boldsymbol{t}_0$.

Define the R-algebra $B := R[X_1, \ldots, X_s]/(\boldsymbol{t}_0)$, $B_0 := (R/(p))[X_1, \ldots, X_s]/(\boldsymbol{t}_0)$ its reduction modulo p and $B_p := R_p[X_1, \ldots, X_s]/(\boldsymbol{t}_0)$ its p-adic completion.

Throughout this section, we let $\boldsymbol{f}$ be given as an s.l.p. Γ with inputs $X_1, \ldots, X_s$ and s outputs corresponding to $f_1, \ldots, f_s$. The s.l.p. Γ has operations in $\{+, -, *\}$ and can use constants in B. Denote by $c_{-s}, \ldots, c_L$ the result sequence of the s.l.p. Γ on the input $X_1, \ldots, X_s$. Let r_i and $\boldsymbol{b}_i$ be the canonical remainder and quotients of c_i for $-s \leqslant i \leqslant L$. So we have $c_i = r_i + {}^t\boldsymbol{b}_i \cdot \boldsymbol{t} \in R[X_1, \ldots, X_s]$. Let $i_1, \ldots, i_s$ be the indices of the s outputs of Γ, so that we have $f_j = c_{i_j}$ for $1 \leqslant j \leqslant s$. We denote by $\boldsymbol{B} \in \mathcal{M}_s(R[X_1, \ldots, X_s])$ the matrix whose jth row is $\boldsymbol{b}_{i_j}$. Therefore one has $\boldsymbol{f} = \boldsymbol{B} \cdot \boldsymbol{t} \in \mathcal{M}_{s,1}(R[X_1, \ldots, X_s])$.

Let us consider vectors as $s \times 1$ matrices. We fix the notation $\boldsymbol{A} \cdot \boldsymbol{v}$ for the matrix vector product and similarly ${}^t\boldsymbol{v}_1 \cdot \boldsymbol{v}_2$ for the inner product of two vectors, ${}^t\boldsymbol{v}$ being the transposed vector of $\boldsymbol{v}$. We also let $c \times \boldsymbol{v}$ be the coefficientwise product of a scalar c by a vector $\boldsymbol{v}$.

4.1. Using the quotient matrix

We define two maps σ and δ from R_p to R_p by $\sigma(a) = a_0$ and $\delta(a) := \frac{a-a_0}{p}$ for any $a = \sum_{i\in\mathbb{N}} a_i p^i \in R_p$. For any $a \in R_p$, we have $a = \sigma(a) + p\delta(a)$. We extend the definition of σ and δ to polynomials by acting coefficientwise and to matrices by acting componentwise. Thus $\sigma(\boldsymbol{t})$, also denoted by $\boldsymbol{t}_0$, verifies $\sigma(\boldsymbol{t}) = (\sigma(t_1), \ldots, \sigma(t_s))$.

Recursive formula for $\boldsymbol{t}$. The triangular set $\boldsymbol{t}$ is a recursive p-adic vector of polynomials.

Lemma 25. *The matrix $\sigma(\boldsymbol{B})$ is invertible modulo $\boldsymbol{t}_0$. Let $\boldsymbol{\Gamma}$ be a representative in $\mathcal{M}_s(R_p[X_1, \ldots, X_s])$ of $\sigma(\boldsymbol{B})^{-1} \bmod \boldsymbol{t}_0$.*

Then the triangular set $\boldsymbol{t}$ satisfies the recursive equation in $\mathcal{M}_{s,1}(R_p[X_1, \ldots, X_s])$

$$\boldsymbol{t} - \boldsymbol{t}_0 = [\boldsymbol{\Gamma} \cdot (\boldsymbol{f} - p^2(\delta(\boldsymbol{B}) \cdot \delta(\boldsymbol{t})))] \operatorname{rem} \boldsymbol{t}_0. \tag{1}$$

Proof. By differentiating the equality $\boldsymbol{f}_0 = \sigma(\boldsymbol{B}) \cdot \boldsymbol{t}_0 \in \mathcal{M}_{s,1}(R/(p)[X_1, \ldots, X_s])$ and then reducing modulo $\boldsymbol{t}_0$, we get $\mathrm{Jac}(\boldsymbol{f}_0) = \sigma(\boldsymbol{B}) \cdot \mathrm{Jac}(\boldsymbol{t}_0)$ in $\mathcal{M}_s(B_0)$. Since $\mathrm{Jac}(\boldsymbol{f}_0)$ is invertible in $\mathcal{M}_s(B_0)$ by hypothesis, $\boldsymbol{B}_0$ and $\mathrm{Jac}(\boldsymbol{t}_0)$ are also invertible

$$\sigma(\boldsymbol{B}) = \mathrm{Jac}(\boldsymbol{f}_0) \cdot \mathrm{Jac}(\boldsymbol{t}_0)^{-1} \in \mathcal{M}_s(B_0) \tag{2}$$

Then $\sigma(\boldsymbol{B})$ is invertible in $\mathcal{M}_s(B_p)$ since it is invertible modulo p.

For any $g \in R[X_1, \ldots, X_s]$, let r and $\boldsymbol{a}$ be the canonical remainder and quotients of g by $\boldsymbol{t}$ so that

$$g - r = {}^t\boldsymbol{a} \cdot \boldsymbol{t} = {}^t\sigma(\boldsymbol{a}) \cdot \boldsymbol{t} + p({}^t\delta(\boldsymbol{a}) \cdot \boldsymbol{t}) = {}^t\sigma(\boldsymbol{a}) \cdot \boldsymbol{t} + p({}^t\delta(\boldsymbol{a}) \cdot \boldsymbol{t}_0) + p^2({}^t\delta(\boldsymbol{a}) \cdot \delta(\boldsymbol{t})).$$

Thus we have ${}^t\sigma(\boldsymbol{a}) \cdot \boldsymbol{t} = g - (r + p^2({}^t\delta(\boldsymbol{a}) \cdot \delta(\boldsymbol{t}))) \in B_p$. Now if $g \in (\boldsymbol{t})$, then $r = 0$. We apply this to the equations $\boldsymbol{f}$ and get

$$\sigma(\boldsymbol{B}) \cdot \boldsymbol{t} = \boldsymbol{f} - p^2(\delta(\boldsymbol{B}) \cdot \delta(\boldsymbol{t})) \in \mathcal{M}_{s,1}(B_p). \tag{3}$$

It remains to invert $\sigma(\boldsymbol{B})$ to get

$$\boldsymbol{t} - \boldsymbol{t}_0 = \boldsymbol{t} \operatorname{rem} \boldsymbol{t}_0 = [\boldsymbol{\Gamma} \cdot (\boldsymbol{f} - p^2(\delta(\boldsymbol{B}) \cdot \delta(\boldsymbol{t})))] \operatorname{rem} \boldsymbol{t}_0 \in \mathcal{M}_{s,1}(R_p[X_1, \ldots, X_s]). \qquad \square$$

Recursive Equation (1) will be our shifted algorithm that will compute $\boldsymbol{t}$ thanks to Proposition 12. Indeed, taking the coefficient in p^m of Equation (1), we get that the p-adic coefficient $\boldsymbol{t}_m := (t_{1,m}, \ldots, t_{s,m})$ depends only on the p-adic coefficients $\boldsymbol{B}_i$ and $\boldsymbol{t}_i$ with $i < m$. We will now see how to compute $\boldsymbol{B}$ from $\boldsymbol{f}$ and $\boldsymbol{t}$ by a relaxed algorithm, so that $\boldsymbol{B}_i$ depends only on $\boldsymbol{t}_j$ with $j \leqslant i$.

Computation of $\boldsymbol{B}$ *modulo* $\boldsymbol{t}_0$. In this paragraph, we explain how to compute the remainder r_i and quotients $\boldsymbol{b}_i$ by $\boldsymbol{t}$ of any element c_i of the result sequence. Since Equation (1) is modulo $\boldsymbol{t}_0$, this quantities are only required modulo $\boldsymbol{t}_0$. We proceed recursively on the index i for $-s \leqslant i \leqslant L$.

First, we start with the inputs $X_1, \ldots, X_s$ on our s.l.p. Γ. For $-s < i \leqslant 0$, we have $c_i = X_{\bar{\imath}}$ where $\bar{\imath} := i + s$ and we distinguish two cases:
- if $d_{\bar{\imath}} > 1$, then $X_{\bar{\imath}}$ is already reduced modulo $\boldsymbol{t}$, we put $r_i := X_{\bar{\imath}}$ and $\boldsymbol{b}_i := (0, \ldots, 0)$.
- if $d_{\bar{\imath}} = 1$, then $\boldsymbol{b}_i := (0, \ldots, 0, 1, 0, \ldots, 0)$ with only a one in position $\bar{\imath}$ and $r_i = X_{\bar{\imath}} - t_{\bar{\imath}}$.

Let us now turn to operations of the s.l.p. indexed by $0 < i \leqslant L$. If the ith result c_i is a constant in B, then it is reduced modulo $\boldsymbol{t}$ because $\boldsymbol{t}$ and $\boldsymbol{t}_0$ have the same multi-degree. Consequently, we take $r_i := c_i$ and $\boldsymbol{b}_i := (0, \ldots, 0)$.

Otherwise $c_i = c_j \operatorname{op} c_k$ with $\operatorname{op} \in \{+, -, *\}$ and $j, k < i$. If op is an addition then we set

$$\begin{aligned} r_i &:= r_j + r_k \\ \boldsymbol{b}_i &:= \boldsymbol{b}_j + \boldsymbol{b}_k. \end{aligned}$$

Subtractions are dealt similarly. Let us consider the case of the multiplication. Define $r, \boldsymbol{q} := \texttt{Rem\_quo}(r_j r_k, \boldsymbol{t}, \boldsymbol{t}_0)$ be the reductions modulo $\boldsymbol{t}_0$ of the canonical remainder and quotients of $r_j r_k$ by $\boldsymbol{t}$. By Lemma 22, $r = r_j r_k \operatorname{rem} \boldsymbol{t}$ and $r_j r_k = r + {}^t\boldsymbol{q} \cdot \boldsymbol{t} \in B_p$. So one has

$$\begin{aligned} c_j c_k &= r_j r_k + {}^t[(r_j + {}^t\boldsymbol{b}_j \cdot \boldsymbol{t}) \times \boldsymbol{b}_k + r_k \times \boldsymbol{b}_j] \cdot \boldsymbol{t} \\ &= r + {}^t[\boldsymbol{q} + (r_j + {}^t\boldsymbol{b}_j \cdot \boldsymbol{t}) \times \boldsymbol{b}_k + r_k \times \boldsymbol{b}_j] \cdot \boldsymbol{t} \end{aligned}$$

in B_p and we set

$$\begin{aligned} r_i &:= r \\ \boldsymbol{b}_i &:= \boldsymbol{q} + (r_j + {}^t\boldsymbol{b}_j \cdot \boldsymbol{t}) \times \boldsymbol{b}_k + r_k \times \boldsymbol{b}_j. \end{aligned} \tag{4}$$

Shifted algorithm. We put all these formulas together to form an algorithm that computes all the remainders r_i and quotients $\boldsymbol{b}_i$ modulo $\boldsymbol{t}_0$. We describe this algorithm as a straight-line program, in order to prove that it is a part of a shifted algorithm.

Let L be the length of the s.l.p. Γ of $\boldsymbol{f}$. We define recursively in i such that $-s < i \leqslant L$ some s.l.p.'s ε^i with s inputs. These s.l.p.'s ε^i compute, on the entries $\boldsymbol{t}$ given as the list of their polynomial coefficients, the remainders r_j and quotients $\boldsymbol{b}_j$ of c_j for $j < i$. We call ρ^i and $\boldsymbol{\alpha}^i = (\alpha_1^i, \ldots, \alpha_s^i)$ the outputs of ε^i corresponding to r_i and $\boldsymbol{b}_i$.

Definition 26. Let us initiate the induction for $-s < i \leqslant 0$ and $\bar{\imath} := i + s$:
- if $d_{\bar{\imath}} = 1$, then we define $\varepsilon^i := (r_{\bar{\imath}}, 0, 1)$ where $r_{\bar{\imath}} := X_{\bar{\imath}} - t_{\bar{\imath}}$. The output ρ^i points to $r_{\bar{\imath}}$ and α_m^i points to 0 if $m \neq \bar{\imath}$ or 1 otherwise;
- if $d_{\bar{\imath}} > 1$, then we define $\varepsilon^i := (X_{\bar{\imath}}, 0)$. The output ρ^i points to $X_{\bar{\imath}}$ and α_m^i points to 0 for any $1 \leqslant m \leqslant s$.

Now recursively for $0 < i \leqslant L$, depending on the operation type of Γ_i:
- if $\Gamma_i = (g^c)$ with g reduced modulo $\boldsymbol{t}_0$, then we define $\varepsilon^i := (g, 0)$. The output ρ^i points to P and α_m^i points to 0 for any $1 \leqslant m \leqslant s$;

- if $\Gamma_i = (+; u, v)$, then we build ε^i on top of ε^u and ε^v in such a manner that one has $\rho^i := \rho^u + \rho^v$ and $\boldsymbol{\alpha}^i := \boldsymbol{\alpha}^u + \boldsymbol{\alpha}^v$;
- if $\Gamma_i = (-; u, v)$, then we build ε^i on top of ε^u and ε^v in such a manner that one has $\rho^i := \rho^u - \rho^v$ and $\boldsymbol{\alpha}^i := \boldsymbol{\alpha}^u - \boldsymbol{\alpha}^v$;
- if $\Gamma_i = (*; u, v)$, we define ε^i accordingly to formula (4). First, we compute $r, \boldsymbol{q} := \texttt{Rem\_quo}(\rho^u(\boldsymbol{t})\rho^v(\boldsymbol{t}), \boldsymbol{t}, \boldsymbol{t}_0)$. Then $\rho^i := r$ and $\boldsymbol{\alpha}^i$ is defined by

$$\boldsymbol{q} + (\rho^u(\boldsymbol{t}) + {}^t\boldsymbol{\alpha}^u(\boldsymbol{t}) \cdot \boldsymbol{t}) \times \boldsymbol{\alpha}^v(\boldsymbol{t}) + \rho^v(\boldsymbol{t}) \times \boldsymbol{\alpha}^u(\boldsymbol{t}), \boldsymbol{t}_0.$$

Finally, we set $\varepsilon = \varepsilon^L$.

The s.l.p. ε computes the matrix $\boldsymbol{B}$ on the entries $\boldsymbol{t}$. We build an s.l.p. $\boldsymbol{\Lambda}$ on top of ε such that

$$\boldsymbol{\Lambda} : \boldsymbol{t} \mapsto \boldsymbol{t}_0 + \texttt{LinearSolver}(\boldsymbol{f} - p^2 \times (\delta(\boldsymbol{B}) \cdot \delta(\boldsymbol{t})), \sigma(\boldsymbol{B})).$$

Since we need the right-hand side of Formula (1) modulo $\boldsymbol{t}_0$, all computations in the s.l.p.s ε and $\boldsymbol{\Lambda}$ take place in B_p. There are two notable exceptions: the product $r_j r_k$ in $r, \boldsymbol{q} := \texttt{Rem\_quo}(r_j r_k, \boldsymbol{t}, \boldsymbol{t}_0)$ should be in $R_p[X_1, \ldots, X_s]$. Otherwise, r is not the canonical remainder of c_i modulo $\boldsymbol{t}$ and Equation (3) does not hold. The second exception is the addition by $\boldsymbol{t}_0$ in $\boldsymbol{\Lambda}$ which should be done in $R_p[X_1, \ldots, X_s]$ since Formula (1) holds there.

Lemma 27. *Algorithm* $\texttt{ShiftedEvaluation}(\boldsymbol{\Lambda}, \_, n)$ *is a shifted algorithm for the recursive p-adics* $\boldsymbol{t}$.

Proof. To prove our lemma, we will rely on Corollary 16. Note that the triangular set $\boldsymbol{t}$ is a fixed point of the s.l.p. $\boldsymbol{\Lambda}$ because of Equation (1). It remains to prove that $\mathrm{sh}(\boldsymbol{\Lambda}) \geqslant 1$.

Let us consider the partial s.l.p. $\boldsymbol{t} \mapsto \boldsymbol{B}$ of $\boldsymbol{\Lambda}$ whose output is $\boldsymbol{B}$. Since the s.l.p. ε uses only additions, subtractions, multiplications, `Rem` and `Rem_quo`, and since all these operations preserve a zero shift index (see Definition 13), we know that $\mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{B}) = 0$. Besides

$$\begin{aligned}
\mathrm{sh}(\boldsymbol{t} \mapsto p^2 \times (\delta(\boldsymbol{B}) \cdot \delta(\boldsymbol{t}))) &= 2 + \mathrm{sh}(\boldsymbol{t} \mapsto \delta(\boldsymbol{B}) \cdot \delta(\boldsymbol{t})) \\
&= 2 + \min\left(\mathrm{sh}(\boldsymbol{t} \mapsto \delta(\boldsymbol{B})), \mathrm{sh}(\boldsymbol{t} \mapsto \delta(\boldsymbol{t}))\right) \\
&= 1 + \min\left(\mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{B}), \mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{t})\right) \\
&= 1.
\end{aligned}$$

Furthermore, notice that $\boldsymbol{f} \operatorname{rem} \boldsymbol{t}_0$ and $\sigma(\boldsymbol{B})$ depend only on $\boldsymbol{t}_0$. Finally the resolution of the linear system preserves the shift index (Proposition 19), hence we have $\mathrm{sh}(\boldsymbol{\Lambda}) = 1$. □

Using the online recursive p-adic framework, we have our relaxed p-adic triangular set Hensel lifting algorithm. We can now prove our first main result and state the complexity of our algorithm.

Proof. [of Theorem 3] Lemma 27 and Proposition 12 show that we can compute $\boldsymbol{t}$ at precision n in the time necessary to apply Algorithm $\texttt{ShiftedEvaluation}(\boldsymbol{\Lambda}, \_, n)$.
Cost of ε. The dominant cost when evaluating ε corresponds to multiplications $*$ in $\boldsymbol{f}$. These steps make one call to Algorithm `Rem_quo`, one inner product of vectors modulo $\boldsymbol{t}_0$,

and a few additions and multiplications modulo $\boldsymbol{t}_0$. Thereby the total cost of evaluating ε is $\mathcal{O}\left(L\mathsf{R}(n)\left(\mathsf{RemQuo}(\boldsymbol{d}) + s\mathsf{MT}(\boldsymbol{d})\right)\right) = \mathcal{O}\left(sL\mathsf{R}(n)\mathsf{MT}(\boldsymbol{d})\right)$ (see Proposition 23).
Cost of $\boldsymbol{f} \operatorname{rem} \boldsymbol{t}_0$. Evaluating the s.l.p. $\boldsymbol{f}$ over B_p costs $\mathcal{O}\left(L\mathsf{R}(n)\mathsf{MT}(\boldsymbol{d})\right)$.
Cost of $p^2 \times (\delta(\boldsymbol{B}) \cdot \delta(\boldsymbol{t}))$. The matrix vector product costs $\mathcal{O}\left(s^2\mathsf{R}(n)\mathsf{MT}(\boldsymbol{d})\right)$. Since $s \leqslant L$, this cost is dominated by the cost due to ε.
Resolution of the linear system in $\sigma(\boldsymbol{B})$. Proposition 19 solves the linear system in time $\mathcal{O}\left(s^2\mathsf{R}(n)\right)$. We need to provide the algorithm with $\sigma(\boldsymbol{B})^{-1} \in \mathcal{M}_s(B_0)$, which costs $\mathcal{O}\left(\mathsf{IT}(\boldsymbol{d}) + s^\Omega\mathsf{MT}(\boldsymbol{d})\right)$. □

4.2. By-passing the whole quotient matrix

The algorithm of Section 4.1 computes the whole quotient $\boldsymbol{B}$, leading to a component $\mathcal{O}\left(sL\mathsf{R}(n)\mathsf{MT}(\boldsymbol{d})\right)$ in the complexity. However, when $e \ll s$, we can benefit from not computing $\boldsymbol{B}$. We present in this section a new method to compute $\delta(\boldsymbol{B}) \cdot \delta(\boldsymbol{t})$ avoiding $\boldsymbol{B}$, thus leading to an asymptotic complexity of $\mathcal{O}\left(eL\mathsf{R}(n)\mathsf{Rem}(\boldsymbol{d})\right)$. In return, we increase the asymptotic subdominant part in the precision n of the complexity.

Indeed, this new method makes it harder to deal with the carries involved in the computation of $\boldsymbol{B}$. We introduce the notion of shifted decomposition to solve this issue.

Shifted decomposition. Recall that σ and δ were defined by $\sigma(a) = a_0$ and $\delta(a) := \frac{a-a_0}{p}$ and that, for any $a \in R_p$, we have $a = a_0 + p\delta(a)$. To our great regret, σ and δ are not ring homomorphisms. To remedy this fact, we call a *shifted decomposition* of $a \in R_p$ a pair $(\sigma_a, \delta_a) \in R_p^2$ such that $a = \sigma_a + p\delta_a$. Shifted decompositions are not unique. For any $a \in R_p$, the pair $(\sigma(a), \delta(a))$ is called the *canonical* shifted decomposition of a. Because σ and δ are not ring homomorphisms, we will use another shifted decomposition that behaves better with respect to arithmetic operations.

Lemma 28. *Let* $(\sigma_a, \delta_a), (\sigma_b, \delta_b) \in R_p^2$ *be shifted decompositions of* $a, b \in R_p$. *Then*
(1) $(\sigma_a + \sigma_b, \delta_a + \delta_b)$ *is a shifted decomposition of* $a + b$;
(2) $(\sigma_a - \sigma_b, \delta_a - \delta_b)$ *is a shifted decomposition of* $a - b$;
(3) $(\sigma_a\sigma_b, \delta_a\sigma_b + a\delta_b)$ *and* $(\sigma_a\sigma_b, \delta_a b + \sigma_a\delta_b)$ *are shifted decompositions of* ab.

The notion of shifted decomposition extends naturally to polynomials and matrices.

Recursive formula for $\boldsymbol{t}$. The recursive Formula (1) for $\boldsymbol{t}$ adapts well to shifted decomposition.

Lemma 29. *Let* $(\sigma_{\boldsymbol{B}}, \delta_{\boldsymbol{B}})$ *be a shifted decomposition of the* $\boldsymbol{B}$. *Then the matrix* $\sigma_{\boldsymbol{B}}$ *is invertible modulo* $\boldsymbol{t}_0$.

Let $\boldsymbol{\Gamma}$ *be a representative in* $\mathcal{M}_s(R_p[X_1, \ldots, X_s])$ *of* $\sigma(\boldsymbol{B})^{-1} \bmod \boldsymbol{t}_0$. *Then* $\boldsymbol{t}$ *satisfies the recursive equation*

$$\boldsymbol{t} - \boldsymbol{t}_0 = \left[\boldsymbol{\Gamma} \cdot \left(\boldsymbol{f} - p^2(\delta_{\boldsymbol{B}} \cdot \delta(\boldsymbol{t}))\right)\right] \quad \text{rem} \quad \boldsymbol{t}_0. \tag{5}$$

Proof. The proof of Formula (5) is similar to the proof of Lemma 25. Concerning $\sigma_{\boldsymbol{B}}$, its zeroth p-adic coefficient is the one of $\boldsymbol{B}$, which in invertible in $\mathcal{M}_s(B_0)$ (see the proof of Lemma 25). Therefore $\sigma_{\boldsymbol{B}}$ is invertible in $\mathcal{M}_s(B_p)$. □

Computation of r, $\sigma_{\boldsymbol{B}}$ and $\delta_{\boldsymbol{B}} \cdot \delta(t)$. For every multiplication of the s.l.p. Γ of $\boldsymbol{f}$, the computation of the quotients $\boldsymbol{b}_i$ were calling in Subsection 4.1 s times `Rem` and one time `Rem_quo`. Here, we present a method that does only $\mathcal{O}(1)$ calls to `Rem` and one call to `Rem_quo` in the same situation.

We denote by $(\sigma_{\boldsymbol{b}_i}, \delta_{\boldsymbol{b}_i})$ a shifted decomposition of the quotients $\boldsymbol{b}_i$. The main idea of our new method is to deal with ${}^t\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) \in R_p$ instead of $\delta_{\boldsymbol{b}_i} \in (R_p)^s$. Let us explain how to compute r_i, $\sigma_{\boldsymbol{b}_i}$ and ${}^t\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t})$ recursively on the index i for $-s < i \leqslant L$.

First, for an index i corresponding to an input, *i.e.* $-s < i \leqslant 0$, we set $\bar{\imath} := i + s$. Therefore $c_i = X_{\bar{\imath}}$ and we distinguish two cases:

- if $d_{\bar{\imath}} > 1$, then $X_{\bar{\imath}}$ is reduced modulo $\boldsymbol{t}$ and we take

$$r_i := X_{\bar{\imath}}, \quad \sigma_{\boldsymbol{b}_i} := (0, \ldots, 0), \quad {}^t\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) = 0. \tag{6}$$

- if $d_{\bar{\imath}} = 1$, then we set $r_i := X_{\bar{\imath}} - t_{\bar{\imath}}$, $\sigma_{\boldsymbol{b}_i} := (0, \ldots, 0, 1, 0, \ldots, 0)$ with only a one at position $\bar{\imath}$ and ${}^t\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) = 0$.

Now let $0 < i \leqslant L$ that corresponds to operations in Γ. If the ith result c_i is a constant reduced modulo $\boldsymbol{t}_0$, then, as before, we take

$$r_i := c_i, \quad \sigma_{\boldsymbol{b}_i} := (0, \ldots, 0), \quad {}^t\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) = 0. \tag{7}$$

Consider the final case when $c_i = c_j \,\mathrm{op}\, c_k$ with $\mathrm{op} \in \{+, -, *\}$ and $j, k < i$. The case where op is the addition is straightforward; using Lemma 28, one takes

$$\begin{aligned} r_i &:= r_j + r_k \\ \sigma_{\boldsymbol{b}_i} &:= \sigma_{\boldsymbol{b}_j} + \sigma_{\boldsymbol{b}_k} \\ {}^t\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) &:= {}^t\delta_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t}) + {}^t\delta_{\boldsymbol{b}_k} \cdot \delta(\boldsymbol{t}). \end{aligned} \tag{8}$$

The case of subtraction is similar. Let us deal with the more complicated case of multiplication. We start by computing the remainder and quotients $r, \boldsymbol{q} := \texttt{Rem\_quo}(r_j r_k, \boldsymbol{t}, \boldsymbol{t}_0)$ of $r_j r_k$ by $\boldsymbol{t}$ modulo $\boldsymbol{t}_0$. By Lemma 22, they satisfy $r_j r_k = r \operatorname{rem} \boldsymbol{t}$ and $r_j r_k = r + {}^t\boldsymbol{q} \cdot \boldsymbol{t} \in B_p$.

Thus we still take

$$r_i := r \tag{9}$$

and we have the same relation

$$\boldsymbol{b}_i = \boldsymbol{q} + (r_j + {}^t\boldsymbol{b}_j \cdot \boldsymbol{t}) \times \boldsymbol{b}_k + r_k \times \boldsymbol{b}_j.$$

Now we use Lemma 28 to compute the shifted decomposition of $\boldsymbol{b}_i$ from shifted decompositions of its operands. We choose to take the canonical shifted decomposition for $r_j, r_k, \boldsymbol{q}$ and $\boldsymbol{t}$. Since the scalar multiplication operator $\times$ and the inner product $\cdot$ are made of additions and multiplications, the following is a shifted decomposition of $\boldsymbol{b}_i$

$$\begin{aligned} \sigma_{\boldsymbol{b}_i} &= \boldsymbol{q}_0 + ((r_j)_0 + {}^t\sigma_{\boldsymbol{b}_j} \cdot \boldsymbol{t}_0) \times \sigma_{\boldsymbol{b}_k} + (r_k)_0 \times \sigma_{\boldsymbol{b}_j} \\ \delta_{\boldsymbol{b}_i} &= \delta(\boldsymbol{q}) + (\delta(r_j) + {}^t\delta_{\boldsymbol{b}_j} \cdot \boldsymbol{t}_0 + {}^t\boldsymbol{b}_j \cdot \delta(\boldsymbol{t})) \times \boldsymbol{b}_k + ((r_j)_0 + {}^t\sigma_{\boldsymbol{b}_j} \cdot \boldsymbol{t}_0) \times \delta_{\boldsymbol{b}_k} + \\ &\qquad \delta(r_k) \times \boldsymbol{b}_j + (r_k)_0 \times \delta_{\boldsymbol{b}_j}. \end{aligned}$$

Because we work modulo $\boldsymbol{t}_0$, this decomposition simplifies

$$\begin{aligned}\sigma_{\boldsymbol{b}_i} &:= \boldsymbol{q}_0 + (r_j)_0 \times \sigma_{\boldsymbol{b}_k} + (r_k)_0 \times \sigma_{\boldsymbol{b}_j} \\ \delta_{\boldsymbol{b}_i} &:= \delta(\boldsymbol{q}) + \delta(r_k) \times \boldsymbol{b}_j + (r_k)_0 \times \delta_{\boldsymbol{b}_j} + (\delta(r_j) + {}^t\boldsymbol{b}_j \cdot \delta(\boldsymbol{t})) \times \boldsymbol{b}_k + (r_j)_0 \times \delta_{\boldsymbol{b}_k}.\end{aligned} \tag{10}$$

Now that we have computed r_i and $\sigma_{\boldsymbol{b}_i}$, it remains to compute $\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t})$. Using $\sigma_{\boldsymbol{b}_j}, \sigma_{\boldsymbol{b}_k}, \delta_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t}), \delta_{\boldsymbol{b}_k} \cdot \delta(\boldsymbol{t})$ and other known polynomials, we compute

$$\begin{aligned}{}^t\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t}) := {}^t\delta(\boldsymbol{q}) \cdot \delta(\boldsymbol{t}) + \delta(r_k)({}^t\boldsymbol{b}_j \cdot \delta(\boldsymbol{t})) + (r_k)_0({}^t\delta_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t})) + \\ (\delta(r_j) + {}^t\boldsymbol{b}_j \cdot \delta(\boldsymbol{t}))({}^t\boldsymbol{b}_k \cdot \delta(\boldsymbol{t})) + (r_j)_0({}^t\delta_{\boldsymbol{b}_k} \cdot \delta(\boldsymbol{t}))\end{aligned} \tag{11}$$

where ${}^t\boldsymbol{b}_j \cdot \delta(\boldsymbol{t}) := {}^t\sigma_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t}) + p({}^t\delta_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t}))$ and the same for ${}^t\boldsymbol{b}_k \cdot \delta(\boldsymbol{t})$. This latest formula admits no equivalents for canonical shifted decompositions when the p-adics have carries.

Shifted algorithm. We sum up all these computations in an algorithm. We define recursively for $-s < i \leqslant L$ some s.l.p.'s ξ^i with s inputs. These s.l.p.'s ξ^i compute, on the entries $\boldsymbol{t}$ given as the list of their polynomial coefficients, the remainder r_j and the quantities $\sigma_{\boldsymbol{b}_j}$ and ${}^t\delta_{\boldsymbol{b}_j} \cdot \delta(\boldsymbol{t})$ for $j < i$. We name ρ^i, $\boldsymbol{\alpha}^i = (\alpha_1^i, \ldots, \alpha_s^i)$ and θ^i the outputs of ξ^i corresponding to r_i, $\sigma_{\boldsymbol{b}_i}$ and ${}^t\delta_{\boldsymbol{b}_i} \cdot \delta(\boldsymbol{t})$.

Definition 30. Let us initiate the induction for $-s < i \leqslant 0$ and $\bar{\imath} := i + s$:
- if $d_{\bar{\imath}} = 1$, then we define $\xi^i := (r_{\bar{\imath}}, 0, 1)$ where $r_{\bar{\imath}} := X_{\bar{\imath}} - t_{\bar{\imath}}$. The output ρ^i points to $r_{\bar{\imath}}$, α_m^i points to 0 if $m \neq \bar{\imath}$ or 1 otherwise and θ^i points to 0;
- if $d_{\bar{\imath}} > 1$, then we define $\xi^i := (X_{\bar{\imath}}, 0)$. The output ρ^i points to $X_{\bar{\imath}}$, θ^i and α_m^i points to 0 for any $1 \leqslant m \leqslant s$.

Now recursively for $0 < i \leqslant L$, depending on the operation type of Γ_i:
- if $\Gamma_i = (g^c)$ with g reduced modulo $\boldsymbol{t}_0$, then we define $\xi^i := (g, 0)$. The output ρ^i points to P and α_m^i points to 0 for any $1 \leqslant m \leqslant s$;
- if $\Gamma_i = (+; u, v)$, then we build ξ^i on top of ξ^u and ξ^v in such a manner that one has $\rho^i := \rho^u + \rho^v$, $\boldsymbol{\alpha}^i := \boldsymbol{\alpha}^u + \boldsymbol{\alpha}^v$ and $\theta^i := \theta^u + \theta^v$;
- if $\Gamma_i = (-; u, v)$, then we build ξ^i on top of ξ^u and ξ^v in such a manner that one has $\rho^i := \rho^u - \rho^v$, $\boldsymbol{\alpha}^i := \boldsymbol{\alpha}^u - \boldsymbol{\alpha}^v$ and $\theta^i := \theta^u - \theta^v$;
- if $\Gamma_i = (*; u, v)$, we define ξ^i accordingly to Formulas (9, 10, 11). First, we compute $r, \boldsymbol{q} := \texttt{Rem\_quo}(\rho^u(\boldsymbol{t})\rho^v(\boldsymbol{t}), \boldsymbol{t}, \boldsymbol{t}_0)$. Then $\rho^i := r$, $\boldsymbol{\alpha}^i$ is defined by

$$\sigma(\boldsymbol{q}) + (\rho^u(\boldsymbol{t}) + {}^t\boldsymbol{\alpha}^u(\boldsymbol{t}) \cdot \boldsymbol{t}) \times \boldsymbol{\alpha}^v(\boldsymbol{t}) + \rho^v(\boldsymbol{t}) \times \boldsymbol{\alpha}^u(\boldsymbol{t})$$

  and θ^i is defined by

$${}^t\delta(\boldsymbol{q}) \cdot \delta(\boldsymbol{t}) + \delta(\rho^v)(\Theta^u) + (\rho^v)_0(\theta^u) + (\delta(\rho^u) + \Theta^u)(\Theta^v) + (\rho^u)_0(\theta^v)$$

  where $\Theta^u := {}^t\boldsymbol{\alpha}^u \cdot \delta(\boldsymbol{t}) + p \times \theta^u$ and the same for Θ^v.

Finally, we set $\xi = \xi^L$.

Since ξ computes $\sigma_{\boldsymbol{B}}$ and $\delta_{\boldsymbol{B}} \cdot \delta(\boldsymbol{t})$, we can build an s.l.p. $\boldsymbol{\Delta}$ on top of ξ such that

$$\boldsymbol{\Delta} : \boldsymbol{t} \mapsto \boldsymbol{t}_0 + \texttt{LinearSolver}(\boldsymbol{f} - p^2 \times (\delta_{\boldsymbol{B}} \cdot \delta(\boldsymbol{t})), \sigma_{\boldsymbol{B}}).$$

Once again, computations take place in B_p except for the input of `Rem_quo` and the final addition by $\boldsymbol{t}_0$. We will now compute the shift index of $\boldsymbol{\Delta}$.

Lemma 31. *The s.l.p. ξ has shift index 0 with respect to its outputs ρ^L and $\boldsymbol{\alpha}^L$ and shift index -1 with respect to its output θ^L.*

Proof. We prove recursively on i for $-s < i \leqslant L$ that ξ^i has shift index 0 with respect to its outputs ρ^i and $\boldsymbol{\alpha}^i$ and shift index -1 with respect to its output θ^i.

We initialize the induction for any $-s < i \leqslant 0$. One has

$$\mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{\alpha}^i(\boldsymbol{t})) = \mathrm{sh}(\boldsymbol{t} \mapsto \theta^i(\boldsymbol{t})) = +\infty, \quad \mathrm{sh}(\boldsymbol{t} \mapsto \rho^i(\boldsymbol{t})) = \begin{cases} +\infty & \text{if } d_{\bar{\imath}} > 1 \\ 0 & \text{if } d_{\bar{\imath}} = 1 \end{cases}.$$

Now recursively for $0 < i \leqslant L$, depending on the type of the ith operation of Γ:

- if $\Gamma_i = (g^c)$ with g reduced modulo $\boldsymbol{t}_0$, one has
$$\mathrm{sh}(\boldsymbol{t} \mapsto \rho^i(\boldsymbol{t})) = \mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{\alpha}^i(\boldsymbol{t})) = \mathrm{sh}(\boldsymbol{t} \mapsto \theta^i(\boldsymbol{t})) = +\infty.$$
- if $\Gamma_i = (\omega; u, v)$ with $\omega \in \{+, -, *\}$ then we proceed as follows. The s.l.p. ξ^i uses only additions, subtractions, multiplications, shifts $p \times \_$ by p, and calls to `Rem` and `Rem_quo`. These operations are online algorithms and
$$\mathrm{sh}(\boldsymbol{t} \mapsto \rho^i(\boldsymbol{t})) = 0, \quad \mathrm{sh}(\boldsymbol{t} \mapsto \boldsymbol{\alpha}^i(\boldsymbol{t})) = 0.$$
Now θ^i is an arithmetic expression in $\delta(\boldsymbol{q})$, $\delta(\boldsymbol{t})$, $(\rho^u)_0$, $\delta(\rho^u)$, $\boldsymbol{\alpha}^u$, θ^u, $p \times \theta^u$ and the same for v. The operator $\delta : a \mapsto (a - a_0)/p$ decreases the shift index by 1. Therefore all elements of the result sequence have a shift index greater or equal to -1 and so it is for θ^i. □

Lemma 32. *Algorithm* `ShiftedEvaluation`$(\boldsymbol{\Delta}, \_, n)$ *is a shifted algorithm of which $\boldsymbol{t}$ is a fixed point.*

Proof. The s.l.p. $\boldsymbol{\Delta}$ satisfies $\boldsymbol{\Delta}(\boldsymbol{t}) = \boldsymbol{t}$ thanks to Lemma 29 and because the formulas that define ξ in Definition 30 match formulas (6) to (11).

A direct consequence of Lemma 31 is that

$$\mathrm{sh}(\boldsymbol{t} \mapsto p^2 \times (\delta_{\boldsymbol{B}} \cdot \delta(\boldsymbol{t}))) = (2 + \mathrm{sh}(\boldsymbol{t} \mapsto \delta_{\boldsymbol{B}} \cdot \delta(\boldsymbol{t}))) = 1.$$

Since $\boldsymbol{f} \operatorname{rem} \boldsymbol{t}_0$ and $\sigma_{\boldsymbol{B}}$ depend only on $\boldsymbol{t}_0$, and since the resolution of the linear system does not impact the shift, we have proved $\mathrm{sh}(\boldsymbol{\Delta}) = 1$. We conclude using Corollary 16. □

We can now state our second main theorem on the relaxed lifting of triangular sets.

Theorem 33. *Let $\boldsymbol{f} = (f_1, \ldots, f_s)$ be a polynomial system in $R[X_1, \ldots, X_s]$ given by a straight-line program Γ of size L. Suppose that $\boldsymbol{f}$ satisfies assumption* (H) *and admits a triangular representation $\boldsymbol{t}$ with coefficients in Q and multi-degree $\boldsymbol{d}$.*

Suppose that assumption $(\mathsf{H}')_{\boldsymbol{t},p}$ holds. Given $\boldsymbol{t}_0 = \boldsymbol{t} \bmod p$, one can compute $\boldsymbol{t} \bmod p^n$ in time

$$\mathcal{O}\left(\mathsf{IT}(\boldsymbol{d}) + s^{\Omega}\mathsf{MT}(\boldsymbol{d})\right) + \left((eL + s^2)\mathsf{R}(n) + sLn\right)\mathsf{MT}(\boldsymbol{d}).$$

Proof. (of Theorem 33) By Lemma 32 and Proposition 12, we can compute $\boldsymbol{t}$ in time necessary to compute $\texttt{ShiftedEvaluation}(\boldsymbol{\Delta}, \_, n)$, whose cost we now detail.
Cost of computing $\sigma_{\boldsymbol{B}}$. Recall that we compute recursively $\sigma_{\boldsymbol{b}_i}$ using Formula (10)

$$\sigma_{\boldsymbol{b}_i} = \boldsymbol{q}_0 + (r_j)_0 \times \sigma_{\boldsymbol{b}_k} + (r_k)_0 \times \sigma_{\boldsymbol{b}_j}.$$

The multiplication of two p-adics with one input of length 1 has a linear cost in the precision n. Consequently, we compute $\sigma_{\boldsymbol{B}}$ for a total cost of $\mathcal{O}(sLn\mathsf{MT}(\boldsymbol{d}))$.
Computation of $\boldsymbol{f}$ rem $\boldsymbol{t}_0$. It costs $\mathcal{O}(L\mathsf{R}(n)\mathsf{MT}(\boldsymbol{d}))$.
Computation of $\delta_{\boldsymbol{B}} \cdot \delta(\boldsymbol{t})$. We focus on multiplications in the s.l.p. of $\boldsymbol{f}$ because they induce the more operations in $\boldsymbol{\Delta}$. The remainder r and quotients $\boldsymbol{q}$ of $r_j r_k$ require a call to Algorithm `Rem_quo`. Then $\delta_{\boldsymbol{B}} \cdot \delta(\boldsymbol{t})$ use an inner product $\delta(\boldsymbol{q}) \cdot \delta(\boldsymbol{t})$ and $\mathcal{O}(1)$ multiplications in B_p. The inner product costs less than `Rem_quo`, since this latter algorithm performs such an inner product. Summing up, the total cost is $\mathcal{O}(L\mathsf{R}(n)(\mathsf{RemQuo}(\boldsymbol{d}) + \mathsf{MT}(\boldsymbol{d})))$, that is $\mathcal{O}(eL\mathsf{R}(n)\mathsf{MT}(\boldsymbol{d}))$.
Resolution of the linear system in $\sigma(\boldsymbol{B})$. Proposition 19 solves the linear system in time $\mathcal{O}\left(s^2\mathsf{R}(n)\right)$. We need to provide the algorithm with $\sigma(\boldsymbol{B})^{-1} \in \mathcal{M}_s(B_0)$, which costs $\mathcal{O}\left(\mathsf{IT}(\boldsymbol{d}) + s^{\Omega}\mathsf{MT}(\boldsymbol{d})\right)$. □

Finally, Theorem 4 is a direct corollary of previous Theorem 33.

5. Implementation in Mathemagix

The computer algebra software Mathemagix (van der Hoeven et al., 2002) provides a `C++` library named Algebramix implementing relaxed power series or p-adic numbers and the relaxed framework for recursive p-adics (van der Hoeven, 2002, 2007; Berthomieu et al., 2011; Berthomieu and Lebreton, 2012). Our implementation is built upon this base. It is available inside the file `lifting_fiber_relaxed` in the `C++` library Geomsolvex of Mathemagix. The relaxed algorithm has been plugged into an implementation of the geometric resolution algorithm inside the library Geomsolvex of Mathemagix by G. Lecerf.

5.1. Benchmarks

We have implemented the lifting of univariate representations over the power series ring $\mathbb{F}_p[[T]]$ for both the off-line and the relaxed approach. The implementations over the p-adic integers $\mathbb{Z}_p$ are still in progress. Although they work, they still require some efforts to be competitive.

For benchmark purposes, we decide to consider the computation of geometric resolutions, which is a good example of "real world" use of liftings of univariate representations. One of the most expensive computation in the geometric resolution algorithm is its last lifting: given a polynomial system $\boldsymbol{f} = (f_1, \ldots, f_s)$ in $\mathbb{F}_p[X_1, \ldots, X_s]$, we need to lift a univariate representation of $(f_1, \ldots, f_{s-1})$ over $\mathbb{F}_p[[X_1]][X_2, \ldots, X_s]$ (after a generic change of coordinates).

We report on the timings of our implementation of this lifting inside the geometric resolution algorithm. Timings are given in milliseconds. They are measured using one core of an Intel Xeon X5650 at 2.67 GHz running Linux 64 bits, Gmp 5.0.2 (Granlund et al., 1991) and setting $p = 1048583$ a 21-bit prime number. We indicate the number

of variables s, the required precision n of the p-adics and the degree d of the univariate representation. Our relaxed lifting algorithm is compared to the classical off-line variant based on Newton's iteration as detailed in (Giusti et al., 2001; Heintz et al., 2001).

We have tried our algorithm on two family of polynomial systems that have a small complexity of evaluation. The KATSURA polynomials systems comes from a problem of magnetism in physics (Katsura, 1990). The system KATSURA-s has $s+1$ unknowns $X_0, \ldots, X_s$ and $s+1$ equations:

$$\text{for } 0 \leqslant m < s, \quad \sum_{\ell=-s}^{s} X_{|\ell|} X_{|m-\ell|} = X_m$$

and $X_0 + 2\sum_{\ell=1}^{s} X_\ell = 1$. The best algorithm for each problem is written using a bold font.

KATSURA-$(s-1)$						
s (n,d)	4 $(8,16)$	5 $(16,32)$	6 $(32,64)$	7 $(64,128)$	8 $(128,256)$	9 $(256,512)$
off-line	50	220	520	2 600	14 000	77 000
relaxed	21	87	220	1 400	10 000	85 000

Table 1. Timings of off-line/relaxed lifting of univariate representations for KATSURA-s.

The other family of polynomial system MULLINFORM-s has s unknowns and s equations of the form

$$(\lambda_1 X_1 + \cdots + \lambda_s X_s)({}_1 X_1 + \cdots + {}_s X_s) = \alpha$$

where the $\lambda_i, {}_i$ and α are random coefficients in $\mathbb{F}_p$.

MULLINFORM-$(s-1)$						
s (n,d)	4 $(8,16)$	5 $(16,32)$	6 $(32,64)$	7 $(64,128)$	8 $(128,256)$	9 $(256,512)$
off-line	125	360	1 600	9 000	52 000	290 000
relaxed	52	230	910	7 000	60 000	560 000

Table 2. Timings of off-line/relaxed lifting of univariate representations for the polynomial systems MULLINFORM-s.

The timings of the relaxed algorithm depends strongly on the performance of the relaxed multiplication. For our computations, we need a relaxed multiplication of implementation of power series of polynomial in $(\mathbb{F}_p[X])[[T]]$. The relaxed multiplication algorithm reduces to multiplications in $(\mathbb{F}_p[X])[T]$. We classically used a Kronecker substitution to reduce these products to multiplications of univariate polynomials in $\mathbb{F}_p[Z]$.

We compared relaxed and not relaxed product in $(\mathbb{F}_p[[T]])[Y]$; relaxed algorithms are always slower by a ratio up to 15 in practice. Besides, this ratio grows logarithmically in the precision of the power series as expected. As a future work, we will apply the new relaxed multiplications of (van der Hoeven, 2007, 2012) or of (Lebreton and Schost, 2013) to keep the ratio $\mathsf{R}(n)/\mathsf{I}(n)$ smaller.

5.2. Conclusion

As a conclusion, we remark that a relaxed approach has generated new algorithms for the lifting of triangular sets. Our hope was to save the cost of linear algebra in the dominant part of the complexity when the precision n tends to infinity. Besides, previous experiences, with e.g. the relaxed lifting of regular roots, showed that we could expect to do less multiplications in the relaxed model than in the zealous one. Therefore, whenever the precision n gives a measured ratio of complexity between relaxed and zealous multiplication, we can expect better timings from the relaxed approach.

In view of our hopes, we are not completely satisfied with the lifting of general triangular sets, for it does more multiplications when $sL \geqslant s^\omega$. On the contrary, the lifting of univariate representations always improve the asymptotic number of p-adic multiplications and saves the cost of linear algebra. Thus, there exists a trade-off between the dependencies in s and n for both relaxed and classical algorithms.

Acknowledgments

I thank Éric Schost for his help in the preparation of this paper and the numerous conversations on polynomial systems. I am very grateful to Jérémy Berthomieu, Joris van der Hoeven and Grégoire Lecerf for showing me the potential of relaxed algorithms. Finally, this paper has benefited from comments of Kazuhiro Yokoyama on shifted algorithms and proofreadings of the anonymous reviewers.

References

Aubry, P., Lazard, D., Moreno Maza, M., 1999. On the theories of triangular sets. J. Symbolic Comput. 28 (1-2), 105–124, polynomial elimination—algorithms and applications.

Berkowitz, S. J., 1984. On computing the determinant in small parallel time using a small number of processors. Inform. Process. Lett. 18 (3), 147–150.

Berthomieu, J., Lebreton, R., 2012. Relaxed p-adic Hensel lifting for algebraic systems. In: Proceedings of ISSAC'12. ACM Press, pp. 59–66.

Berthomieu, J., van der Hoeven, J., Lecerf, G., 2011. Relaxed algorithms for p-adic numbers. J. Théor. Nombres Bordeaux 23 (3), 541–577.

Bostan, A., Chowdhury, M. F., van der Hoeven, J., Schost, E., 2011. Homotopy techniques for multiplication modulo triangular sets. Journal of Symbolic Computation 46 (12), 1378 – 1402.

Bostan, A., Chowdhury, M. F. I., Lebreton, R., Salvy, B., Schost, E., 2012. Power series solutions of singular (q)-differential equations. In: Proceedings of ISSAC'12. ACM Press, pp. 107–114.

Bürgisser, P., Clausen, M., Shokrollahi, M. A., 1997. Algebraic complexity theory. Vol. 315 of Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]. Springer-Verlag, Berlin, with the collaboration of Thomas Lickteig.

Cantor, D. G., Kaltofen, E., 1991. On fast multiplication of polynomials over arbitrary algebras. Acta Inform. 28 (7), 693–701.

Coppersmith, D., Winograd, S., 1990. Matrix multiplication via arithmetic progressions. J. Symb. Comp. 9 (3), 251–280.

Dahan, X., Jin, X., Moreno Maza, M., Schost, E., 2008. Change of ordering for regular chains in positive dimension. Theoretical Computer Science 392 (1-3), 37–65.

Dahan, X., Moreno Maza, M., Schost, E., Wu, W., Xie, Y., 2005. Lifting techniques for triangular decompositions. In: ISSAC'05. ACM Press, pp. 108–115.

Dahan, X., Moreno Maza, M., Schost, E., Xie, Y., 2006. On the complexity of the D5 principle. In: Transgressive Computing. pp. 149–168.

De, A., Kurur, P., Saha, C., Saptharishi, R., 2008. Fast integer multiplication using modular arithmetic. In: STOC'08. ACM, New York, pp. 499–505.

Fischer, M. J., Stockmeyer, L. J., 1974. Fast on-line integer multiplication. J. Comput. System Sci. 9, 317–331.

Fürer, M., 2007. Faster Integer Multiplication. In: Proceedings of STOC 2007. San Diego, California, pp. 57–66.

Gianni, P., Mora, T., 1989. Algebraic solution of systems of polynomial equations using Groebner bases. In: Applied algebra, algebraic algorithms and error-correcting codes (Menorca, 1987). Vol. 356 of Lecture Notes in Comput. Sci. Springer, Berlin, pp. 247–257.

Giusti, M., Heintz, J., 1991. Algorithmes—disons rapides—pour la décomposition d'une variété algébrique en composantes irréductibles et équidimensionnelles. In: MEGA. Vol. 94 of Progr. Math. Birkhäuser, pp. 169–194.

Giusti, M., Heintz, J., Hägele, K., Morais, J. E., Pardo, L. M., Montaña, J. L., 1997a. Lower bounds for Diophantine approximations. J. Pure Appl. Algebra 117/118, 277–317, algorithms for algebra (Eindhoven, 1996).

Giusti, M., Heintz, J., Morais, J. E., Pardo, L. M., 1997b. Le rôle des structures de données dans les problèmes d'élimination. C. R. Acad. Sci. Paris Sér. I Math. 325 (11), 1223–1228.

Giusti, M., Lecerf, G., Salvy, B., 2001. A Gröbner free alternative for polynomial system solving. J. Complexity 17 (1), 154–211.

Granlund, T., et al., 1991. GMP, the GNU multiple precision arithmetic library. Version 5.0.2.

Heintz, J., Matera, G., Waissbein, A., 2001. On the time-space complexity of geometric elimination procedures. Appl. Algebra Engrg. Comm. Comput. 11 (4), 239–296.

Hennie, F. C., 1966. On-line turing machine computations. Electronic Computers, IEEE Transactions on EC-15 (1), 35 –44.

Kalkbrener, M., 1993. A generalized Euclidean algorithm for computing triangular representations of algebraic varieties. J. Symb. Comp. 15, 143–167.

Kaltofen, E., 1992. On computing determinants of matrices without divisions. In: Wang, P. S. (Ed.), Proc. 1992 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'92). ACM Press, New York, N. Y., pp. 342–349.

Kaltofen, E., Villard, G., 2004. On the complexity of computing determinants. Comput. Complexity 13 (3-4), 91–130.

Katsura, S., 1990. Spin glass problem by the method of integral equation of the effective field. New Trends in Magnetism, 110–121.

König, J., 1903. Aus dem Ungarischen übertragen vom Verfasser. B. G. Teubner, Leipzig.

Kronecker, L., 1882. Grundzüge einer arithmetischen theorie des algebraischen grössen. J. reine angew. Math. 92, 1–122.

Langemyr, L., 1991. Algorithms for a multiple algebraic extension. In: Effective methods in algebraic geometry). Vol. 94 of Progr. Math. Birkhäuser, pp. 235–248.

Lazard, D., 1991. A new method for solving algebraic systems of positive dimension. Discrete Appl. Math. 33 (1-3), 147–160, applied algebra, algebraic algorithms, and error-correcting codes (Toulouse, 1989).

Lebreton, R., 2012. Contributions to relaxed algorithms and polynomial system solving. Ph.D. thesis, École Polytechnique.

Lebreton, R., Schost, E., 2013. A simple and fast online power series multiplication and its analysis.

Li, X., Moreno Maza, M., Schost, E., 2009. Fast arithmetic for triangular sets: from theory to practice. J. Symbolic Comput. 44 (7), 891–907.

Ritt, J. F., 1966. Differential algebra. Dover Publications Inc., New York.

Rouillier, F., 1999. Solving zero-dimensional systems through the rational univariate representation. Appl. Algebra Engrg. Comm. Comput. 9 (5), 433–461.

Schönhage, A., Strassen, V., 1971. Schnelle Multiplikation grosser Zahlen. Computing 7, 281–292.

Schost, E., 2002. Degree bounds and lifting techniques for triangular sets. In: Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation. ACM, New York, pp. 238–245 (electronic).

Schost, E., 2003a. Complexity results for triangular sets. J. Symbolic Comput. 36 (3-4), 555–594, international Symposium on Symbolic and Algebraic Computation (ISSAC'2002) (Lille).

Schost, E., 2003b. Computing parametric geometric resolutions. Appl. Algebra Engrg. Comm. Comput. 13 (5), 349–393.

Schröder, M., 1997. Fast online multiplication of real numbers. In: STACS 97 (Lübeck). Vol. 1200 of Lecture Notes in Comput. Sci. Springer, Berlin, pp. 81–92.

Stothers, A., 2010. On the complexity of matrix multiplication. Ph.D. thesis, University of Edinburgh.

van der Hoeven, J., 1997. Lazy multiplication of formal power series. In: ISSAC '97. Maui, Hawaii, pp. 17–20.

van der Hoeven, J., 2002. Relax, but don't be too lazy. J. Symb. Comput. 34 (6), 479–542.

van der Hoeven, J., 2007. New algorithms for relaxed multiplication. J. Symbolic Comput. 42 (8), 792–802.

van der Hoeven, J., 2010. Newton's method and FFT trading. JSC 45 (8), 857–878.

van der Hoeven, J., 2011. From implicit to recursive equations. Tech. rep., HAL.

van der Hoeven, J., 2012. Faster relaxed multiplication. Tech. rep., HAL.

van der Hoeven, J., Lecerf, G., Mourrain, B., et al., 2002. Mathemagix. Available from `http://www.mathemagix.org`.

Vassilevska Williams, V., 2011. Breaking the Coppersmith-Winograd barrier.

von zur Gathen, J., Gerhard, J., 2013. Modern computer algebra, 3rd Edition. Cambridge University Press, Cambridge.

Watt, S., 1989. A fixed point method for power series computation. In: Gianni, P. (Ed.), Symbolic and Algebraic Computation. Vol. 358 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 206–217.

Wu, W. J., 1984. Basic principles of mechanical theorem proving in elementary geometries. J. Systems Sci. Math. Sci. 4 (3), 207–235.