Relaxed p-adic Hensel lifting for algebraic systems

Jérémy Berthomieu, Romain Lebreton

Relaxed p-adic Hensel lifting for algebraic systems

Jérémy Berthomieu
Laboratoire de Mathématiques
Université de Versailles
Versailles, France
jeremy.berthomieu@uvsq.fr

Romain Lebreton
Laboratoire d'Informatique
École polytechnique
Palaiseau, France
lebreton@lix.polytechnique.fr

ABSTRACT

In a previous article, an implementation of lazy p-adic integers with a multiplication of quasi-linear complexity, the so-called relaxed product, was presented. Given a ring R and an element p in R, we design a relaxed Hensel lifting for algebraic systems from $R/(p)$ to the p-adic completion R_p of R. Thus, any root of linear and algebraic regular systems can be lifted with a quasi-optimal complexity. We report our implementations in C++ within the computer algebra system MATHEMAGIX and compare them with Newton operator. As an application, we solve linear systems over the integers and compare the running times with LINBOX and IML.

Keywords

Lazy p-adic numbers, power series, algebraic system resolution, relaxed algorithms, complexity, integer linear systems.

1. INTRODUCTION

Let R be an *effective* commutative ring with unit, which means that algorithms are given for any ring operation and for zero-testing. Given a proper principal ideal (p) with $p \in R$, we write R_p for the completion of the ring R for the p-adic valuation. Any element $a \in R_p$ 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 writing of elements in R_p, let us 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 $a_i \in M$.

Two classical examples are the completions $k[[X]]$ of the ring of polynomials $k[X]$ for the ideal (X) and $\mathbb{Z}_p$ of the ring of integers $\mathbb{Z}$ for the ideal (p), with p a prime number. In this paper, for $R = \mathbb{Z}$, we take $M = \{0, \ldots, p-1\}$.

Related Works. This paper is the natural extension of [1], which comes in the series of papers [13, 14, 15, 24]. These papers deal with lazy power series (or lazy p-adic numbers) and relaxed algorithms.

Lazy power series is the adaptation of the *lazy evaluation* (also known as call-by-need) function evaluation scheme for computer algebra [18]. It consists in delaying the evaluation of the arguments at most. It was used in [25] to minimize the number of operations in its settings. The main drawback of these objects is the bad complexity at high order of some basic algorithms such as the multiplication.

Relaxed algorithms for power series were introduced in [13]. They share with the lazy algorithms the property that the coefficients of the output are computed one after another and that only minimal knowledge on the input is required. However, *relaxed algorithms* differ in the sense that they do not try to minimize the number of operations of each step. Since they can anticipate some computations, they have better complexity. The first presented relaxed algorithm was for the multiplication: the so-called on-line multiplication for integer in [7]. Then, came the on-line multiplication for real numbers in [22], and relaxed multiplication for power series [12, 13], improved in [14].

One important advantage of relaxed algorithms is to allow the computation of recursive power series or p-adic numbers in a good complexity both theoretical and practical. Another advantage inherited from lazy power series is that the precision can be increased at any time and the computation resumes from its previous state. It is well-suited when one wants to lift a p-adic number to a rational number with no sharp *a priori* estimation on the precision required.

On the other hand, there are *zealous* algorithms. The precision is fixed in advance in the computations. The Newton-Hensel operator allows to solve implicit equations in $R/(p^n)$ for any $n \in \mathbb{N}$ [21]. It has been thoroughly studied and optimized in particular for linear system solving [6, 20, 23].

Our contribution. In this paper, we show how to transform algebraic equations into recursive equations. As a consequence, we can use relaxed algorithms to compute the Hensel lifting of a root from the residue ring $R/(p)$ to its p-adic ring R_p. We work under the hypothesis of Hensel's lemma, which states that the derivative at the point we wish to lift is not zero.

Our algorithms lose a logarithmic factor in the precision compared to zealous Newton iteration. However, the constant factors hidden in the big-O notation are potentially smaller. Moreover, we take advantage of the good evaluation properties of the implicit equations. For example, we rediscover the quadratic factor in the size of matrices for linear system solving [23]. Another example concerns the

ISSAC '12 Grenoble, FRANCE

multivariate Newton operator which performs at each step an evaluation of the implicit equations and an inversion of its evaluated Jacobian matrix. In Theorems 25 and 28, we manage to save the cost corresponding to the Jacobian matrix.

Finally, we implement these algorithms to obtain timings competitive with Newton and even significantly lower on wide ranges of input parameters. As an application, we solve linear systems over the integers and compare to LinBox and IML. We show that we improve the timings for small matrices and big integers.

Our results on the transformation of implicit equations to recursive equations were discovered independently at the same time by [24]. The latter paper deals with more general recursive power series defined by algebraic, differential equations or a combination thereof. However, its algorithms have yet to be implemented and only work in characteristic zero. Furthermore, since the carry is not dealt with, the blockwise product as presented in [1, Section 4] cannot be used. This is important because it is the most efficient algorithm for higher precision among relaxed algorithms.

2. PRELIMINARIES

Straight-line programs. In this paper, we will use the model of computation to describe the algorithms behind the algebraic and recursive equations. We give a short presentation of this notion and refer to [3] for more details. Let R be a ring and A a R-algebra.

A *straight-line program* (s.l.p.) 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 operations $+, -, \cdot : A^2 \to A$. We also define for $r \in R$ the 0-ary operations r^c whose output is the constant r and the unary scalar multiplication $r \times \cdot$ by r. We denote the set of all these operations R^c and R. Finally, let us denote S the set of *regular* elements in R, that is of non zero divisors in R. We consider for $s \in S$ the unary scalar division $\cdot/s : A \times S \to A$, and we still denote S their set. Let us fix a set of operations Ω, usually $\Omega = \{+, -, \cdot\} \cup R \cup S \cup R^c$.

A s.l.p. starts with a number ℓ of *input* parameters indexed from $-(\ell-1)$ to 0. It has k *instructions* $\Gamma_1, \ldots, \Gamma_k$ 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 $a = (a_0, \ldots, a_{\ell-1})$ with *result sequence* $b = (b_{-\ell+1}, \ldots, b_k) \in A^{\ell+k}$, if $b_i = a_{\ell-1+i}$ whenever $-(\ell-1) \leqslant i \leqslant 0$ and $b_i = \omega_i(b_{u,1}, \ldots, b_{u,r_i})$ with $(b_{u,1}, \ldots, b_{u,r_i}) \in \mathcal{D}_{\omega_i}$ whenever $1 \leqslant i \leqslant k$.

The *multiplicative complexity* $L^*(\Gamma)$ of a s.l.p. Γ is the number of operations ω_i that are multiplications $\cdot$ between elements of A.

Example 1. *Let $R = \mathbb{Z}$, $A = \mathbb{Z}[X, Y]$ and Γ be the s.l.p. with two input parameters indexed -1, 0 and $\Gamma_1 = (\cdot; -1, -1)$, $\Gamma_2 = (\cdot; 1, 0)$, $\Gamma_3 = (1^c)$, $\Gamma_4 = (-; 2, 3)$, $\Gamma_5 = (3 \times \cdot; 1)$.*

First, its multiplicative complexity is $L^*(\Gamma) = 2$. Then, Γ is executable on $(X, Y) \in A^2$, and for this input its result sequence is $\left(X, Y, X^2, X^2Y, 1, X^2Y - 1, 3X^2\right)$.

Remark 2. *For the sake of simplicity, we will associate an arithmetic expression with a s.l.p. It is the same operation as when one writes an arithmetic expression in a programming language,* e.g. *C, and a compiler turns it into a s.l.p. In our case, we fix an arbitrary compiler that starts by the left-hand side of an arithmetic expression.*

For example, the arithmetic expression $\varphi : Z \mapsto \left(Z^2\right)^2 + 1$ can be represented by the s.l.p. with one input and instructions $\Gamma_1 = (\cdot; 0, 0), \Gamma_2 = (\cdot; 1, 1), \Gamma_3 = (1^c), \Gamma_4 = (+; 2, 3)$.

Lazy framework and data structures on p-adics. We assume that two functions quo and rem are provided in order to deal with carries that appear when computing in $\mathbb{Z}_p$ for example. They are the quotient and the remainder functions by p; $\mathsf{quo}(\cdot, p)$ is a function from R to R and $\mathsf{rem}(\cdot, p)$ is a function from R to M such that for all a in R, $a = \mathsf{quo}(a, p)p + \mathsf{rem}(a, p)$.

Then, we give a short presentation of the lazy framework. It states that for any operation, the output at a given precision cannot require to know the input at a precision greater than necessary. For instance, when computing c_n in the product $c = a \times b$, one cannot make use of a_{n+1} and b_{n+1}. As a consequence, in the lazy framework, the coefficients of the output are computed one by one starting from the term of order 0.

Now let's recall the basics of the implementation of recursive p-adic numbers found in [1, 13]. We follow these papers notation and use a C++-style pseudo-code. The main class Padic_p contains the computed coefficients $\varphi : M[p]$ of the number a up till a given order n in $\mathbb{N}$. In addition, any class derived from Padic_p must contain a method next whose purpose is to compute the next coefficient a_n.

Example 3. *The addition in $\mathbb{Z}_p$ is implemented in the class $\mathsf{Sum\_Padic}_p$ derived from Padic_p and therefore inherits of the attributes $\varphi : M[p]$ and n. It has additional attributes a, b in Padic_p and carry γ in M set by default to zero. Finally, the function next adds a_n, b_n and γ, puts the quotient of the addition in γ and returns the remainder.*

The class Padic_p is also endowed with an accessor method $[]\,(n \in \mathbb{N})$ which calls $\mathsf{next}\,()$ until the precision reaches $n+1$ and then outputs φ_n.

Relaxed multiplication. Having a relaxed multiplication is very convenient for solving recursive algebraic equation in good complexity. As we will see, solving a recursive equation is very similar to checking it. Therefore, the cost of solving such an equation depends mainly on the cost of evaluating the equation. If, for example, the equations are sparse, we are able to take advantage of this sparsity.

Let $a, b \in R_p$ be two p-adic numbers. Denote $a_n \in M$ the coefficients of a in the decomposition $a = \sum_{n \in \mathbb{N}} a_n p^n$. We detail the computation of the first four terms of $c = a \times b$.

The coefficients c_n are computed one by one. The major difference with the naive algorithm comes from the use of fast multiplication algorithms on significant parts of forthcoming terms.

The zeroth coefficient $c_0 = \mathsf{rem}(a_0 b_0, p)$ is computed normally, a carry $\gamma = \mathsf{quo}(a_0 b_0, p)$ is then stored. For the first coefficient, one computes $a_0 b_1 + a_1 b_0 + \gamma$ with both products $a_0 b_1$ and $a_1 b_0$. Then, the remainder of the division by p is assigned to c_1 while the quotient is to γ.

Changes arise when computing c_2. By hypothesis, the coefficients $a_0, a_1, a_2, b_0, b_1, b_2 \in M$ are known. So one can compute $a_0 b_2 + a_2 b_0 + (a_1 + a_2 p)(b_1 + b_2 p) + \gamma$ using two

multiplications of elements of size 1 and one of elements of size 2: $(a_1 + a_2 p)(b_1 + b_2 p)$. Then, c_2 is just the remainder of the division of this number by p and γ is the quotient.

For the term of order 3, one computes $a_0 b_3 + a_3 b_0 + \gamma$, with two multiplications of elements of size 1. Instead of the two products in $a_1 b_2 + a_2 b_1$, we used fast algorithms on the bigger data $a_1 + a_2 p$ and $b_1 + b_2 p$ to save some multiplications.

Throughout this paper, we measure the cost of an algorithm by the number of arithmetic operations in $R/(p)$ it performs.

Notation 1. We note $\mathsf{M}(n)$ the number of arithmetic operations in $R/(p)$ needed to multiply two elements of $R/(p^n)$.

In particular, when $R = k[X]$ and $p = X$, it is classical that $\mathsf{M}(n) \in \mathcal{O}(n \log n \log\log n)$ [4].

If $R = \mathbb{Z}$, we rather specify the number of bit-operations. Two integers of bit-size less than m can be multiplied in $\mathsf{I}(m) \in \mathcal{O}(m \log m \; 2^{\log^* m})$ bit-operations [8], where $\log^*$ is the iterated logarithm.

We note $\mathsf{R}(n)$ the number of arithmetic operations in $R/(p)$ necessary to multiply two elements of R_p at precision n.

THEOREM 4 ([1, 7, 13]). *The complexity $\mathsf{R}(n)$ for multiplying two p-adic numbers at precision n is $\mathcal{O}(\mathsf{M}(n) \log n)$.*

This statement was discovered for integers in [7]. However the application to compute recursive power series or p-adic integers was seen for the first time in [13]. Article [1] generalizes the algorithm for p-adic numbers.

REMARK 5. *If $R = \mathbb{F}_p$ contains many 2^pth root of unity, then two power series over $\mathbb{F}_p$ can be multiplied in precision n in $\mathcal{O}\left(\mathsf{M}(n)\, e^{2\sqrt{\log 2 \log\log n}}\right)$ multiplications in $\mathbb{F}_p$, see [14].*

The relaxed algorithm will be used for multiplications in R_p. For divisions in R_p, we use the relaxed algorithm of [1].

Relaxed recursive p-adic numbers.

The relaxed model was motivated by its efficient implementation of recursive p-adic numbers. We will work with recursive p-adic numbers in a simple case and do not need the general context of recursive p-adic numbers [17, Definition 7]. We note $\nu_p(a)$ the valuation in p of the p-adic number a.

Definition 1. Let $\Phi \in R_p[Y]$ and y be a fixed point of Φ, *i.e.* $y = \Phi(y)$, such that $y_0 = y \bmod p$. Let us denote $\Phi^0 = \mathrm{Id}$ and, for all $n \in \mathbb{N}^*$, $\Phi^n = \Phi \circ \cdots \circ \Phi$ (n times). Then, y is a *recursive* p-adic number and Φ *allows the computation* of y if, for all $n \in \mathbb{N}$, we have $(y - \Phi^n(y_0)) \in p^{n+1} R_p$.

PROPOSITION 6. *Let $\Phi \in R_p[Y]$ with a fixed point y. If Φ is such that $\nu_p(\Phi'(y_0)) > 0$, where $y_0 = y \bmod p$, then Φ allows the computation of y.*

PROOF. At first, we notice that $(y - y_0) \in pR_p$. Then, for all p-adic number z, there exists $\Theta(z) \in R_p$ such that $\Phi(Y) - \Phi(z) = \Phi'(z)(Y - z) + (Y - z)^2 \Theta(z)$. Now for all $n \in \mathbb{N}$, we denote $y_{(n)} := \Phi^n(y_0)$ and we apply the previous statement to $z = y_{(n)}$ to get

$$\frac{y - y_{(n+1)}}{y - y_{(n)}} = \frac{\Phi(y) - \Phi(y_{(n)})}{y - y_{(n)}} = \Phi'(y_{(n)}) + (Y - y_{(n)})\Theta(y_{(n)}).$$

Since $\Phi'(y_{(n)}) = \Phi'(y_0) \bmod p = 0 \bmod p$, we have that $\frac{y - y_{(n+1)}}{y - y_{(n)}} \in pR_p$, for all n in $\mathbb{N}$. □

The definition of a recursive p-adic number is effective: given $y_0 \in R/(p)$ a root of the reduced polynomial $\bar{P} \in R/(p)[Y]$, we can compute recursively $y_1, y_2, \ldots$, thanks to Proposition 6.

Remark that the vector case is included in the definition. Indeed, if $\mathbf{p}$ denotes $(p, \ldots, p) \in R^r$, then $R_p^r \simeq (R^r)_{\mathbf{p}}$. Furthermore, the general case with more initial conditions $y_0, y_1, \ldots, y_\ell$ is not considered here but it is believed to be an interesting extension of these results.

We refer to [1] for a description of the implementation of a recursive p-adic number y. In a word, the method next derives the nth term by computing $\Phi(y)_n$, which only depends on the preceding terms. But one has to be cautious with Φ because, even if $\Phi(y)_n$ does not depend on y_n, the coefficient y_n could still be involved in the computation of this coefficient. Here is an example.

WARNING 7. *Take $R_p = \mathbb{Z}_p$ for any prime number p. Let $\Phi(Y) = Y^2 + p$, and y be the only solution of $Y = \Phi(Y)$ satisfying $y_0 = 0$. We can check that Φ allows the computation of y since $\Phi'(0) = 0$.*

At the first step, we find $y_1 = 1$. Then we compute $\Phi(y)_2 = (y^2 + p)_2 = (y^2)_2$. In the relaxed product algorithm, we compute $y_0 y_2$ and $q = (y_1 + y_2 p)(y_1 + y_2 p)$. Then $\Phi(y)_2 = 2y_0 y_2 + q_0 = 0 y_2 + q_0$. Here we face two problems.

First, y_2 is involved in the computation of $\Phi(y)_2$, although $\Phi(y)_2$ does not depend on y_2. More importantly, the p-adic number q involves and depends *on y_2. Since we do not know y_2 yet, we must proceed otherwise.*

Shifted algorithms.

Because of the issue raised in Warning 7, we need to force the shift inside Φ. In other terms, we must explicit the fact that y_n is not required in the computation of $(\Phi(y))_n$. For this matter, we introduce for all i in $\mathbb{N}^*$ two new shift operators:

$$\begin{array}{lccc} p^i \times \cdot : & R_p & \to & R_p \\ & a & \mapsto & p^i a, \end{array} \qquad \begin{array}{lccc} \cdot / p^i : & p^i R_p & \to & R_p \\ & a & \mapsto & a/p^i. \end{array}$$

Let Ω' be the set of operations $\{+, -, \cdot, p^i \times \cdot, \cdot / p^i\} \cup R \cup S \cup R^c$.

Definition 2. Let $\Gamma = (\Gamma_1, \ldots, \Gamma_k)$ be a s.l.p. over the R-algebra R_p with ℓ input parameters and operations in Ω'. For any i, j such that $-(\ell - 1) \leqslant i \leqslant k$ and $-(\ell - 1) \leqslant j \leqslant 0$, the *shift* $\mathrm{sh}(\Gamma, i, j)$ of the ith operation of Γ with respect to its jth argument is an element of $\mathbb{Z} \cup \{+\infty\}$. For $i \leqslant 0$, we define $\mathrm{sh}(\Gamma, i, j)$ by 0 if $i = j$ and $+\infty$ otherwise. Now, for $i > 0$:

- if $\Gamma_i = (\omega_i; u, v)$ with $\omega_i \in \{+, -, \cdot\}$, then we set $\mathrm{sh}(\Gamma, i, j) := \min(\mathrm{sh}(\Gamma, u, j), \mathrm{sh}(\Gamma, v, j))$;
- if $\Gamma_i = (r^c;)$, then $\mathrm{sh}(\Gamma, i, j) := +\infty$;
- if $\Gamma_i = (p^s \times \cdot; u)$, then $\mathrm{sh}(\Gamma, i, j) := \mathrm{sh}(\Gamma, u, j) + s$;
- if $\Gamma_i = (\cdot / p^s; u)$, then $\mathrm{sh}(\Gamma, i, j) := \mathrm{sh}(\Gamma, u, j) - s$;
- if $\Gamma_i = (\omega; u)$ with $\omega \in R \cup S$, then we set $\mathrm{sh}(\Gamma, i, j) := \mathrm{sh}(\Gamma, u, j)$.

We abbreviate $\mathrm{sh}(\Gamma) := \mathrm{sh}(\Gamma, k, 0)$ if Γ has one argument.

This definition simply formalizes which terms of the jth argument is involved in the result of the ith element of the result sequence from a syntactic point of view.

PROPOSITION 8. *With the notation of Definition 2, let* $\mathbf{y} = (y_1, \ldots, y_r) \in (R_p)^r$ *be such that* Γ *is executable on input* $\mathbf{y}$ *and* $b \in R_p$ *be the* i*th element of the result sequence. Then, for all* $n \in \mathbb{N}$*, the computation of* b_n *involves at most the terms* $(y_j)_l$ *of the* j*th argument* y_j *for* $0 \leqslant l \leqslant n - \mathrm{sh}(\Gamma, i, j-r)$.

EXAMPLE 9. *We carry on with Warning 7. For the natural s.l.p.* Γ *with one argument associated to the arithmetic expression* $\Phi : Z \mapsto Z^2 + p$*, we have* $\mathrm{sh}(\Gamma) = 0$*. This formalizes the previous remark that the computation of* $\Phi(y)_n$ *involves* y_l *for* $0 \leqslant l \leqslant n$.

Now take the s.l.p.

$$\Psi : Z \mapsto p^2 \times \left(\frac{Z}{p}\right)^2 + p$$

(see Remark 2). Then $\mathrm{sh}(\Psi) = 1$*, since* $\mathrm{sh}\left(p^2 \times (Z/p)^2\right) = \mathrm{sh}\left((Z/p)^2\right) + 2 = \mathrm{sh}(Z/p) + 2 = \mathrm{sh}(Z) + 1$*. Moreover,* Ψ *is executable on the solution* y *of* $Y = \Phi(Y)$ *since* $y \in pR_p$*. So this s.l.p.* Ψ *solves the problem raised in Warning 7.*

Nevertheless, we explicit the first steps of the new algorithm to convince even the most skeptical reader. So $\Psi(y)_2$ *executes* $\left((y/p)^2\right)_0 = y_1^2$*. Then* $\Psi(y)_3$ *does* $2y_1y_2$*. Finally* $\Psi(y)_4$ *computes* y_1y_3 *and* $q = (y_2 + y_3 p)(y_2 + y_3 p)$ *so that* $\Psi(y)_4 = 2y_1y_3 + q_0$*. We can check that there is no dependency issue here and that we have a shift of* $1 = \mathrm{sh}(\Psi)$ *in the indices of* y.

Thanks to this, we are now able to explicit which s.l.p. Ψ are suited to the implementation of recursive p-adic numbers. Recall that S is the set of regular elements in R, we denote $K := S^{-1}R$ the total ring of fractions of R.

Definition 3. Let y be a recursive p-adic and $\Phi \in K[Y]$ with denominators not in pR that allows the computation of y. Let Ψ be a s.l.p. with one input and operations in Ω'.

Then, Ψ is said to be a *shifted algorithm* for Φ and y_0 if $\mathrm{sh}(\Psi) \geq 1$, Ψ is executable on y over the R-algebra R_p and Ψ computes $\Phi(Y)$ on input Y over the R-algebra $K[Y]$.

REMARK 10. *There is no uniqueness of a shifted operator. For example, if* $\Phi(Y) = Y^3 + p \in \mathbb{Z}[Y]$ *and* $y_0 = 0$*, then* $\Psi : Z \to p^2 \times \left(\frac{Z}{p}\right)^2 Z + p$ *and* $\Psi_1 : Z \to p^3 \times \left(\frac{Z}{p}\right)^3 + p$ *are two distinct shifted algorithms for* Φ *and* $y_0 = 0$*. Indeed* $\mathrm{sh}(\Psi) = 1$, $\mathrm{sh}(\Psi_1) = 2$ *and they are executable on* y.

We have dealt with the algorithmic issues of relaxed recursive p-adic numbers. Now, we can assess the complexity.

PROPOSITION 11. *Let* Ψ *be a shifted algorithm for the recursive* p*-adic* y *whose multiplicative complexity is* L^**. Then, the relaxed* p*-adic* y *can be computed at precision* n *in time* $L^*\mathsf{R}(n) + \mathcal{O}(n)$.

PROOF. The cost of the computation of y is the cost of the evaluation of $\Psi(y)$ in R_p. We recall that addition in $R_p \times R_p$, subtraction in $R_p \times R_p$, multiplication in $R \times R_p$ (that is operations in R) and division in $R_p \times S$ (that is operations in S) up to the precision n can be computed in time $\mathcal{O}(n)$. Scalars from R are decomposed in R_p in constant complexity. Finally, multiplications in $R_p \times R_p$ are done in time $\mathsf{R}(n)$ (see [1]). Now the multiplicative complexity L^* of Ψ counts exactly the latter operation. □

3. UNIVARIATE ROOT LIFTING

In [1, Section 7], it is shown how to compute the dth root of a p-adic number a in a recursive relaxed way, d being relatively prime to p. In this section, we extend this result to the relaxed lifting of a simple root of any polynomial $P \in R[Y]$. Hensel's lemma ensures that from any modular simple root $y_0 \in R/(p)$ of $\bar{P} \in R/(p)[Y]$, there exists a unique lifted root $y \in R_p$ of P such that $y = y_0 \bmod p$.

From now on, P is a polynomial with coefficients in R and $y \in R_p$ is the unique root of P lifted from the modular simple root $y_0 \in R/(p)$.

PROPOSITION 12. *The polynomial*

$$\Phi(Y) := \frac{P'(y_0)\,Y - P(Y)}{P'(y_0)} \in K[Y]$$

allows the computation of y.

PROOF. It is clear that if $P(y) = 0$ and $P'(y_0) \neq 0$, then $y = \frac{P'(y_0)y - P(y)}{P'(y_0)} = \Phi(y)$. Furthermore, $\Phi'(y_0) = 0$. □

In the following subsections, we will derive some shifted algorithms associated to the recursive equation Φ depending on the representation of P.

3.1 Dense polynomials

We assume in this subsection that the polynomial P of degree d is given as the vector of its coefficients in the monomial basis $(1, Y, \ldots, Y^d)$. To have a shifted algorithm, we need to express $\Phi(Y)$ with a positive shift. Remark, from Definition 2, that the shift of $\Phi(Y)$ is 0.

LEMMA 13. *The s.l.p.* $\Gamma : Z \mapsto p^2 \times \left(\left(\frac{Z - y_0}{p}\right)^2 \cdot Z^k\right)$ *for* $k \in \mathbb{N} - \{0\}$ *is executable on* y *and* $\mathrm{sh}(\Gamma) = 1$.

PROOF. Since $y_0 = y \bmod p$, $\Gamma(y) \in R_p$ and Γ is executable on y. Furthermore, the shift $\mathrm{sh}(\Gamma)$ equals $2 + \min\left(\mathrm{sh}\left(\frac{Z - y_0}{p}\right), \mathrm{sh}(Z)\right) = 1$. □

We are now able to derive a shifted algorithm for Φ.

Algorithm 1 - Dense polynomial root lifting

Input: $P \in R[Y]$ with a simple root y_0 in $R/(p)$.
Output: A shifted algorithm Ψ associated to Φ and y_0.

1. Compute $Q(Y)$ the quotient of $P(Y)$ by $(Y - y_0)^2$
2. Let $\mathrm{sq}(Z) : Z \mapsto \left(\frac{Z - y_0}{p}\right)^2$
3. **return** the shifted algorithm Ψ :

$$Z \to \frac{-1}{P'(y_0)}\left(P(y_0) - P'(y_0)\,y_0 + p^2 \times (Q(Z) \cdot \mathrm{sq}(Z))\right).$$

PROPOSITION 14. *Given a polynomial* P *of degree* d *in dense representation and a modular simple root* y_0*, Algorithm 1 defines a shifted algorithm* Ψ *associated to* Φ*. The precomputation of such an operator involves* $\mathcal{O}(\mathsf{M}(d))$ *operations in* R*, while we can lift* y *at precision* n *in time* $(d-1)\mathsf{R}(n) + \mathcal{O}(n)$.

PROOF. First, Ψ is a shifted algorithm for Φ. Indeed since $\mathrm{sh}(P(y_0) - P'(y_0)\,y_0) = +\infty$ and, due to Lemma 13, $\mathrm{sh}\left(p^2 \times (\mathrm{sq}(Z) \cdot Q(Z))\right) = 1$, we have $\mathrm{sh}(\Psi) = 1$.

Also, thanks to Lemma 13, we can execute Ψ on y over the R-algebra R_p. Moreover, it is easy to see that $\Phi(Y) = \Psi(Y)$ over the R-algebra $K[Y]$.

The quotient polynomial Q is precomputed in $\mathcal{O}(\mathsf{M}(d))$ arithmetic operations in R. Using Horner scheme to evaluate $Q(Z)$, we have $L^*(\Psi) = d-1$ and we can apply Proposition 11. □

3.2 Polynomials as straight-line programs

In [1, Proposition 7.1], the case of the polynomial $P(Y) = Y^d - a$ was studied. Although the general concept of shifted algorithm was not introduced, an algorithm of multiplicative complexity $\mathcal{O}(L^*(P))$ was given. The shifts were only present in the implementation in MATHEMAGIX [16]. We clarify and generalize this approach to any polynomial P given as a s.l.p. and propose a shifted algorithm Ψ whose complexity is linear in $L^*(P)$.

In this subsection, we fix a polynomial P given as a s.l.p. with operations in $\Omega := \{+,-,\cdot\} \cup R \cup R^c$ and multiplicative complexity $L^* := L^*(P)$, and a modular simple root $y_0 \in R/(p)$ of P. Then, we define the polynomials $T_P(Y) := P(y_0) + P'(y_0)(Y - y_0)$ and $E_P(Y) := P(Y) - T_P(Y)$.

Definition 4. We define recursively a vector $\tau \in R^2$ and a s.l.p. ε with operations in $\Omega' := \{+,-,\cdot, p^i \times \cdot, \cdot/p^i\} \cup R \cup S \cup R^c$. Initially, $\varepsilon^0 := 0$ and $\tau^0 := (y_0, 1)$. Then, we define ε^i and τ^i recursively on i with $1 \leqslant i \leqslant k$ by:

- if $\Gamma_i = (a^c;)$, then $\varepsilon^i := 0$, $\tau^i := (a, 0)$;
- if $\Gamma_i = (a \times \cdot; u)$, then $\varepsilon^i := a \times \varepsilon^u$, $\tau^i := a\tau^u$;
- if $\Gamma_i = (\pm; u, v)$, then $\varepsilon^i := \varepsilon^u \pm \varepsilon^v$, $\tau^i := \tau^u \pm \tau^v$;
- if $\Gamma_i = (\cdot; u, v)$ and we denote $\tau^u = (a, c)$, $\tau^v = (b, d)$, then $\tau^i = (ab, ad + cb)$ and ε^i equals

$$\varepsilon^u \cdot \varepsilon^v + p \times (((c \times \varepsilon^v + d \times \varepsilon^u)/p) \cdot (Z - y_0)) + (a \times \varepsilon^v + b \times \varepsilon^u) + p^2 \times \left((cd) \times ((Z - y_0)/p)^2\right). \quad (1)$$

Recall that multiplications denoted by $\cdot$ are the ones between p-adics. Finally, we set $\varepsilon_P := \varepsilon^k$ and $\tau_P := \tau^k$ where k is the number of instructions in the s.l.p. P.

LEMMA 15. *The s.l.p. ε_P is a shifted algorithm for E_P and y_0. Its multiplicative complexity is bounded by $2L^* + 1$. Also, τ_P is the vector of coefficients of the polynomial T_P in the basis $(1, (Y - y_0))$.*

PROOF. Let us call P_i the ith result of the s.l.p. P on the input Y over $R[Y]$, with $0 \leqslant i \leqslant k$. We note $E^i := E_{P_i}$ and $T^i := T_{P_i}$ for all $0 \leqslant i \leqslant k$. Let us prove recursively that ε^i is a shifted algorithm for E^i and y_0, and that τ^i is the vector of coefficients of T^i in the basis $(1, (Y - y_0))$.

For the initial step $i = 0$, we have $P_0 = Y$ and we verify that $E^0(Y) = \varepsilon^0(Y) = 0$ and $T^0(Y) = y_0 + (Y - y_0)$. The s.l.p. ε_0 is executable on y over R_p and its shift is $+\infty$.

Now we prove the result recursively for $i > 0$. We detail the case when $\Gamma_i = (\cdot; u, v)$, the others cases being straightforward. Equation (1) corresponds to the last equation of

$$\begin{array}{rcl} P_i & = & P_u P_v \\ \Leftrightarrow\ E^i & = & (E^u + T^u)(E^v + T^v) - T^i \\ \Leftrightarrow\ E^i & = & E^u E^v + [E^u T^v + T^u E^v] + \left(T^u T^v - T^i\right) \\ \Leftrightarrow\ E^i & = & E^u E^v + [(P'_v(y_0) E^u + P'_u(y_0) E^v)(Y - y_0) \\ & & + (P_v(y_0) E^u + P_u(y_0) E^v)] \\ & & + P'_u(y_0) P'_v(y_0) (Y - y_0)^2. \end{array}$$

Also $\tau^i = (P_u(y_0) P_v(y_0), P'_u(y_0) P_v(y_0) + P_u(y_0) P'_v(y_0))$. The s.l.p. ε^i is executable on y over R_p because, for all $j < i$, $\mathrm{sh}(\varepsilon_j) > 0$ implies that $(c\varepsilon^v(y) + d\varepsilon^u(y))/p \in R_p$. Concerning the shifts, since $\mathrm{sh}(\varepsilon_u), \mathrm{sh}(\varepsilon_v) > 0$, we can check that every operand in equation (1) has a positive shift. So $\mathrm{sh}(\varepsilon^i) > 0$. Then, take $i = r$ to conclude the proof.

Concerning multiplicative complexity, we slightly change ε^0 such that it computes once and for all $((Y - y_0)/p)^2$ before returning zero. Then, for all multiplication instructions $\cdot$ in the s.l.p. P, the s.l.p. ε_P adds two multiplications $\cdot$ between p-adics (see equation (1)). So $L^*(\varepsilon_P) = 2L^* + 1$. □

PROPOSITION 16. *Let P be a univariate polynomial over R_p given as a s.l.p. such that its multiplicative complexity is L^*. Then, the following algorithm*

$$\Psi : Z \mapsto \frac{-P(y_0) + P'(y_0) y_0 - \varepsilon_P(Z)}{P'(y_0)}$$

is a shifted algorithm associated to Φ and y_0 whose multiplicative complexity is $2L^* + 1$.

PROOF. We have $\Phi(Y) = \Psi(Y)$ over the algebra $K[Y]$ because $\Phi(Y) = (-P(y_0) + P'(y_0) y_0 + E_P(Y))/P'(y_0)$. Because of Lemma 15 and $\nu_p(P'(y_0)) = 0$, the s.l.p. Ψ is executable on y over R_p and its shift is positive. We conclude with $L^*(\Psi) = L^*(\varepsilon_P) = 2L^* + 1$ as the division by $P'(y_0)$ is an operation in the set S. □

REMARK 17. *By adding the square operation $\cdot^2$ to the set of operations Ω of P, we can gain a few multiplications. In Definition 4, if $\Gamma_i = (\cdot^2; u)$ and $\tau^u = (a, c)$, then define ε^i by $\varepsilon^u \cdot (\varepsilon^u + 2 \times (a + c \times (Z - y_0))) + p^2 \times \left(c^2 \times ((Z - y_0)/p)^2\right)$. Thereby, we reduce the multiplicative complexity of ε_P and Ψ by the number of square operations in P.*

THEOREM 18. *Let $P \in R[Y]$ and $y_0 \in R/(p)$ be such that $P(y_0) = 0 \bmod p$ and $P'(y_0) \neq 0 \bmod p$. Denote $y \in R_p$ the unique solution of P lifted from y_0. Assume that P is given as a s.l.p. with operations in $\Omega := \{+,-,\cdot\} \cup R \cup R^c$ whose multiplicative complexity is L^*. Then, we can lift y up to precision n in time $(2L^* + 1)\mathsf{R}(n) + \mathcal{O}(n)$.*

PROOF. By Propositions 12 and 16, y can be computed as a recursive p-adic number with the shifted algorithm Ψ. Proposition 11 gives the announced complexity. □

REMARK 19. *We can improve the bound on the multiplicative complexity when the polynomial has a significant part with positive valuation. Indeed suppose that the polynomial P is given as $P(Y) = \alpha(Y) + p\beta(Y)$ with α and β two s.l.p.. Then the part $p\beta(Y)$ is already shifted. In this case, set $\tilde{\varepsilon}_P := \varepsilon_\alpha + p\beta$ so that the following is a shifted algorithm:*

$$\Psi : Z \mapsto \frac{-\alpha(y_0) + \alpha'(y_0) y_0 - \tilde{\varepsilon}_P(Z)}{\alpha'(y_0)}.$$

Its multiplicative complexity is $L^*(\alpha) + 2L^*(\beta) + 1$.

4. LINEAR ALGEBRA OVER P-ADICS

As an extension of the results of the previous section, we will lift a simple root of a system of r algebraic equations with r unknowns in Section 5. For this matter, one needs to solve a linear system based on the Jacobian matrix in a relaxed way, as we describe in this section.

For any matrix $A \in \mathcal{M}_{r\times s}(R_p)$, we will denote by a_{ij} the coefficient of A lying on the ith row and the jth column. Furthermore, A can be seen as a p-adic matrix, *i.e.* a p-adic number whose coefficients are matrices over M. In this case, the matrix of order n will be denoted by $A_n \in \mathcal{M}_{r\times s}(M)$, so that $A = \sum_{n=0}^{\infty} A_n p^n$.

4.1 Inversion of a "scalar" matrix

We can generalize the remark of [1, Section 6.1]: because of the propagation of the carries, the computation of the inverse of a regular $r \times r$ matrix with coefficients in M is not immediate in the p-adic case.

Let us recall the scalar case. We define mul_rem and mul_quo such that $\beta a = \mathsf{mul\_rem}(\beta, a) + p\,\mathsf{mul\_quo}(\beta, a)$ for all $\beta \in M$ and $a \in R_p$. The nth term of $\mathsf{mul\_rem}(\beta, a)$ is $\mathsf{rem}(\beta a_n, p) \in M$, while, for $\mathsf{mul\_quo}(\beta, a)$, the corresponding one is $\mathsf{quo}(\beta a_{n-1}, p) \in M$.

We shall introduce two operators Mul_rem and Mul_quo which are the matricial counterparts of both mul_rem and mul_quo. Let $\mathrm{B} \in \mathcal{M}_r(M)$ and $A \in \mathcal{M}_{r\times s}(R_p)$ seen as a p-adic matrix, then the nth term of $\mathsf{Mul\_rem}(\mathrm{B}, A)$ is $\mathsf{rem}(\mathrm{B}A_n, p) \in \mathcal{M}_{r\times s}(M)$, while the one of $\mathsf{Mul\_quo}(\mathrm{B}, A)$ corresponds to $\mathsf{quo}(\mathrm{B}A_{n-1}, p) \in \mathcal{M}_{r\times s}(R_p)$, so that we have $\mathrm{B}A = \mathsf{Mul\_rem}(\mathrm{B}, A) + p\mathsf{Mul\_quo}(\mathrm{B}, A)$.

Let us denote $\mathsf{MM}(r, s)$ the number of operations in the ground ring to multiply a square matrix of size r and a matrix of size $r \times s$. Recall that if $r \geq s$, then $\mathsf{MM}(r, s) \in \mathcal{O}\left(r^2 s^{\omega-2}\right)$ and otherwise $\mathsf{MM}(r, s) \in \mathcal{O}\left(r^{\omega-1} s\right)$, where ω is the exponent of matrix multiplication over any ring.

LEMMA 20. *Let* B *and* A *be two* p*-adic matrices such that* $\mathrm{B} \in \mathcal{M}_r(M)$ *and* $A \in \mathcal{M}_{r\times s}(R_p)$*. Then, the computations of* $\mathsf{Mul\_rem}(\mathrm{B}, A)$*,* $\mathsf{Mul\_quo}(\mathrm{B}, A)$*, and therefore of* $\mathrm{B}A$*, can be done to precision* n *in time* $\mathcal{O}(\mathsf{MM}(r, s)n)$*.*

PROOF. To compute $\mathsf{Mul\_rem}(\mathrm{B}, A)$, we multiply B and A as if B were over $R/(p)$ and A over $(R/(p))[[x]]$. To compute $\mathsf{Mul\_quo}(\mathrm{B}, A)$, for each $k < n$, we multiply B with A_{k-1} and only keep the quotient by p of this product. Therefore, the total cost is in $\mathcal{O}(\mathsf{MM}(r, s)n)$. □

PROPOSITION 21. *Let* A *be a relaxed matrix of size* $r \times s$ *over* R_p *and let* $\mathrm{B} \in \mathcal{M}_r(M)$*. If* B *is invertible modulo* p *and* $\Gamma := \mathrm{B}^{-1} \bmod p$*, then the product* $C = \mathrm{B}^{-1}A$ *satisfies*

$$C = \mathsf{Mul\_rem}(\Gamma, A - p\mathsf{Mul\_quo}(\mathrm{B}, C)) \tag{2}$$

with $C_0 = \Gamma A_0 \bmod p$*. Furthermore,* C *can be computed up until precision* n *in time* $\mathcal{O}(\mathsf{MM}(r, s)n)$*.*

PROOF. First, $A = \mathsf{Mul\_rem}(\mathrm{B}, C) + p\mathsf{Mul\_quo}(\mathrm{B}, C)$ so we can deduce that $\mathsf{Mul\_rem}(\mathrm{B}, C) = A - p\mathsf{Mul\_quo}(\mathrm{B}, C)$. It remains to multiply both sides by Γ using Mul_rem to prove equation (2).

From the definition of Mul_quo, we can see that the nth term of the right-hand side of equation (2) involves only C_{n-1}. So C is recursively computed with a cost evaluated in Lemma 20. □

4.2 Inversion of a matrix over p-adics

We can now apply the division of matrices over p-adic integers, as in [13].

PROPOSITION 22. *Let* $A \in \mathcal{M}_{r\times s}(R_p)$ *and* $B \in \mathcal{M}_r(R_p)$ *be two relaxed matrices such that* B_0 *is invertible of inverse* $\Gamma = B_0^{-1} \bmod p$*. Then, the product* $C = B^{-1}A$ *satisfies*

$$C = B_0^{-1}\left(A - p \times \left(\frac{(B - B_0)}{p} \cdot C\right)\right), \tag{3}$$

with $C_0 = \Gamma A_0 \bmod p$*. Thus,* C *can be computed up to precision* n *in time* $\mathsf{MM}(r, s)\mathsf{R}(n) + \mathcal{O}(n)$*.*

PROOF. The right-hand side of equation (3) is a shifted algorithm associated to $C \mapsto A - BC$ and Γ. The only p-adic matrix product $\cdot$ involves $\mathsf{MM}(r, s)$ p-adic multiplications and therefore a cost of $\mathsf{MM}(r, s)\mathsf{R}(n)$. Proposition 21 for the product by B_0^{-1} shows that its cost is not dominant. □

REMARK 23. *Note that if* $B \in \mathcal{M}_r(R)$*, then the matrix product* $((B - B_0)/p)\,C$ *can be computed up to precision* n *in time* $\mathcal{O}(\mathsf{MM}(r, s)n)$*. Therefore, so can* C*. This is analogous to the inversion of matrices with polynomial entries which can be done in time linear in the precision [20].*

5. MULTIVARIATE ROOT LIFTING

In this section, we lift a p-adic root $\mathbf{y} \in R_p^r$ of a polynomial system $\mathbf{P} = (P_1, \ldots, P_r) \in R[\mathbf{Y}]^r = R[Y_1, \ldots, Y_r]^r$ in a relaxed recursive way. We make the assumption that $\mathbf{y}_0 = (y_{1,0}, \ldots, y_{r,0}) \in (R/(p))^r$ is a regular modular root of $\mathbf{P}$, *i.e.* its Jacobian matrix $\mathrm{d}\mathbf{P}_{\mathbf{y}_0}$ is invertible in $\mathcal{M}_r(R/(p))$. Newton-Hensel operator ensures both the existence and the uniqueness of $\mathbf{y} \in R_p^r$ such that $\mathbf{P}(\mathbf{y}) = 0$ and $\mathbf{y}_0 = \mathbf{y} \bmod p$. From now on, $\mathbf{P}$ is a polynomial system with coefficients in R and $\mathbf{y} \in R_p^r$ is the unique root of $\mathbf{P}$ lifted from the modular regular root $\mathbf{y}_0 \in (R/(p))^r$.

PROPOSITION 24. *The polynomial system*

$$\mathbf{\Phi}(\mathbf{Y}) := \mathrm{d}\mathbf{P}_{\mathbf{y}_0}^{-1}\left(\mathrm{d}\mathbf{P}_{\mathbf{y}_0}(\mathbf{Y}) - \mathbf{P}(\mathbf{Y})\right) \in K[\mathbf{Y}]^r$$

allows the computation of $\mathbf{y}$*.*

PROOF. We adapt the proof of Proposition 12. Since $\mathrm{d}\mathbf{\Phi}_{\mathbf{y}_0} = 0$, $\mathbf{\Phi}$ allows the computation of $\mathbf{y}$. □

As in the univariate case, we have to introduce a positive shift in $\mathbf{\Phi}$. In the following, we present how to do so depending on the representation of $\mathbf{P}$.

5.1 Dense algebraic systems

Let $\mathbf{P}$ be given in dense representation. We assume that each P_i has total degree at most $d \geq 2$, so that its dense size is bounded by $(d+1)^r$. As in the univariate case, the shift of $\mathbf{\Phi}(\mathbf{Y})$ is 0. We adapt Lemma 13 and Proposition 14 to the multivariate polynomial case as follows. For $1 \leqslant j \leqslant k \leqslant r$, let $\mathbf{Q}^{(j,k)}$ be polynomial systems such that $\mathbf{P}(\mathbf{Y})$ equals

$$\mathbf{P}(\mathbf{y}_0) + \mathrm{d}\mathbf{P}_{\mathbf{y}_0}(\mathbf{Y}) + \sum_{1\leqslant j\leqslant k\leqslant r} \mathbf{Q}^{(j,k)}(\mathbf{Y})\,(Y_j - y_{j,0})\,(Y_k - y_{k,0}).$$

Algorithm 2 - Dense polynomial system root lifting

Input: $\mathbf{P} \in R[\mathbf{Y}]^r$ with a regular root $\mathbf{y}_0$ in $(R/(p))^r$.
Output: A shifted algorithm $\mathbf{\Psi}$ associated to $\mathbf{\Phi}$ and $\mathbf{y}_0$.

1. For $1 \leqslant j \leqslant k \leqslant r$, compute a $\mathbf{Q}^{(j,k)}(\mathbf{Y})$ from $\mathbf{P}(\mathbf{Y})$
2. For $1 \leqslant j \leqslant k \leqslant r$, let $\mathrm{pr}_{j,k}(\mathbf{Z}) := \left(\frac{Z_j - y_{j,0}}{p}\right)\left(\frac{Z_k - y_{k,0}}{p}\right)$
3. Let $\mathbf{\Psi}_1 : \mathbf{Z} \mapsto \sum_{1\leqslant j\leqslant k\leqslant r} \mathbf{Q}^{(j,k)}(\mathbf{Z}) \cdot \mathrm{pr}_{j,k}(\mathbf{Z})$
4. **return** the shifted algorithm

$$\mathbf{\Psi} : \mathbf{Z} \mapsto -\mathrm{d}\mathbf{P}_{\mathbf{y}_0}^{-1}\left(\mathbf{P}(\mathbf{y}_0) - \mathrm{d}\mathbf{P}_{\mathbf{y}_0}(\mathbf{y}_0) + p^2 \times \mathbf{\Psi}_1\right).$$

THEOREM 25. *Given $\mathbf{P} = (P_1, \ldots, P_r)$ a polynomial system in $R[\mathbf{Y}]$ in dense representation, such that each P_i has total degree at most d, and an approximate zero $\mathbf{y}_0$, Algorithm 2 outputs a shifted algorithm $\mathbf{\Psi}$ associated to $\mathbf{\Phi}$ and $\mathbf{y}_0$. The precomputation in $\mathbf{\Psi}$ costs $\tilde{\mathcal{O}}(rd^r)$, while the evaluation of y to precision n costs $rd^r\mathsf{R}(n) + \mathcal{O}(n)$.*

PROOF. First, for $j \leqslant r$, we perform the Euclidean division of $\mathbf{P}$ by $(Y_j - y_{j,0})^2$ to reduce the degree in each variable. We use Kronecker substitution [2, Chapter 1, Section 8] to obtain a quasi-linear complexity. By Kronecker substitution on the variables $Y_2, \ldots, Y_r$, $\mathbf{P}$ can be written as a bivariate polynomial system $\tilde{\mathbf{P}}(Y_1, U_1)$ of degree d^{r-1} in U_1. Then, one obtains $\tilde{\mathbf{Q}}^{(1,1)}(Y_1, U_1)$ by doing the Euclidean division of each $\tilde{P}_i(Y_1, U_1)$ by $(Y_1 - y_{1,0})^2$ and then retrieve $\mathbf{Q}^{(1,1)}(\mathbf{Y})$ as a r-variate polynomial system. The Euclidean division costs $\tilde{\mathcal{O}}(d^r)$ arithmetic operations for each P_i, for a total cost of $\tilde{\mathcal{O}}(rd^r)$. Next, the process is repeated on the remainders of the division. We write them as bivariate polynomials in Y_2 and U_2 with degree $2d^{r-2}$ in U_2 and divide them by $(Y_2 - y_{2,0})^2$ and so on. The total cost of this process is $\tilde{\mathcal{O}}(rd^r)$ arithmetic operations.

Then, for each P_i, it remains a polynomial with partial degree at most 1 in each variable. Necessary divisions by $(Y_j - y_{j,0})(Y_k - y_{k,0})$ are given by the presence of a multiple of Y_jY_k, which gives rise to a cost of $\mathcal{O}(2^r)$.

Next, we have to evaluate $\mathbf{\Psi}_1$ at $\mathbf{y}$. Since the total numbers of monomials of the $\mathbf{Q}^{(j,k)}(\mathbf{Y})$ for $1 \leqslant j \leqslant k \leqslant r$ is bounded by rd^r, Proposition 11 gives the desired cost estimate for the evaluation of y at precision n. Finally, we have to multiply this by the inverse of the Jacobian of $\mathbf{P}$ at $\mathbf{y}_0$, which is a matrix with coefficients in R. By Proposition 21 and Remark 23, and since we only lift a single root, it can be done at precision n in time $\mathcal{O}(r^2n)$. □

5.2 Algebraic systems as s.l.p.

We keep basically the same notations as in Section 3.2. Given an algebraic system $\mathbf{P}$, we define $\mathbf{T}_\mathbf{P}(\mathbf{Y}) := \mathbf{P}(\mathbf{y}_0) + \mathrm{d}\mathbf{P}_{\mathbf{y}_0}(\mathbf{Y} - \mathbf{y}_0)$ and $\mathbf{E}_\mathbf{P}(\mathbf{Y}) := \mathbf{P}(\mathbf{Y}) - \mathbf{T}_\mathbf{P}(\mathbf{Y})$. We adapt Definition 4 so that we may define τ and ε for multivariate polynomials.

Definition 5. We define recursively vectors $\tau_j \in R^{r+1}$ and s.l.p.s ε_j for $1 \leqslant j \leqslant r$ with operations in Ω', where $\Omega' := \{+, -, \cdot, p^i \times \cdot, \cdot/p^i\} \cup R \cup S \cup R^c$.

First, we initialize for all $1 \leqslant i \leqslant r$, $\varepsilon_j^{-r+i} := 0$, $\tau_j^{-r+i} := (y_{i,0}, 0, \ldots, 0, 1, 0, \ldots, 0)$ with 1 at index $i+1$. Then for $1 \leqslant i \leqslant k_j$ where k_j is the number of instructions in the s.l.p. P_j, we define ε_j^i and τ_j^i recursively on i by almost the same formulas as in Definition 4. Let us detail the changes when $\Gamma_i = (\cdot, u, v)$:

Let $\tau_j^u = (a_0, a_1, \ldots, a_r)$ and $\tau_j^v = (b_0, b_1, \ldots, b_r)$, then $\tau_j^i = (a_0b_0, a_0b_1 + a_1b_0, \ldots, a_0b_r + a_rb_0)$, and $\varepsilon_j^i := \varepsilon_j^u\varepsilon_j^v + p \times \varepsilon_1 + p^2 \times \varepsilon_2$ with ε_1 given by

$$\left(\sum_{\ell=1}^{r} a_\ell \times (Z_\ell - y_{0,\ell})\right) \cdot \frac{\varepsilon_j^v}{p} + \left(\sum_{\ell=1}^{r} b_\ell \times (Z_\ell - y_{0,\ell})\right) \cdot \frac{\varepsilon_j^u}{p},$$

$\varepsilon_2 = \sum_{1 \leqslant \ell_1, \ell_2 \leqslant r} a_{\ell_1} b_{\ell_2} \times ((Z_{\ell_1} - y_{0,\ell_1})/p) \cdot ((Z_{\ell_2} - y_{0,\ell_2})/p)$.

As before, we set $\varepsilon_{P_j} := \varepsilon_j^{k_j}$ and $\tau_{P_j} := \tau_j^{k_j}$.

LEMMA 26. *The s.l.p. $\varepsilon_\mathbf{P} := (\varepsilon_{P_1}, \ldots, \varepsilon_{P_r})$ is a shifted algorithm for $\mathbf{E}_\mathbf{P}$ and $\mathbf{y}_0$. Its complexity is $3L^* + \frac{r(r+1)}{2}$. Moreover, assuming $\mathbf{T}_\mathbf{P} = (T_{P_1}, \ldots, T_{P_r})$. Then, τ_{P_j} is the vector of coefficients of the polynomial T_{P_j} in the basis $(1, (Y_1 - y_{1,0}), \ldots, (Y_r - y_{r,0}))$.*

PROOF. From Lemma 15, it is clear that $\varepsilon_\mathbf{P}$ is a shifted algorithm for $\mathbf{E}_\mathbf{P}$ and $\mathbf{y}_0$. It is also clear that τ_{P_i} is the coefficients of T_{P_i} in the basis $(1, (Y_1 - y_{1,0}), \ldots, (Y_r - y_{r,0}))$.

Concerning the multiplicative complexity, we perform the same change as in Lemma 15 by computing $((Y_i - y_{i,0})/p) \cdot ((Y_j - y_{j,0})/p)$ once and for all in $\varepsilon_\mathbf{P}^0$. Therefore we have to perform $\frac{r(r+1)}{2}$ product of p-adics.

Moreover, for all instruction $\cdot$ in the s.l.p. P_j, ε_{P_j} adds three multiplications between p-adics (see operations $\cdot$ in formulas above). So $L^*(\varepsilon_\mathbf{P}) = 3L^* + \frac{r(r+1)}{2}$. □

PROPOSITION 27. *Let $\mathbf{P}$ be a polynomial system of r polynomials in r variables over R_p, given as a s.l.p. such that its multiplicative complexity is L^*. Then, the algorithm*

$$\mathbf{\Psi} : \mathbf{Z} \mapsto \mathrm{d}\mathbf{P}_{\mathbf{y}_0}^{-1}((-\mathbf{P}(\mathbf{y}_0) + \mathrm{d}\mathbf{P}_{\mathbf{y}_0}(\mathbf{y}_0)) - \varepsilon_\mathbf{P}(\mathbf{Z}))$$

is a shifted algorithm associated to $\mathbf{\Phi}$ and $\mathbf{y}_0$ whose evaluation complexity is $3L^* + \frac{r(r+1)}{2}$.

PROOF. We just need to prove the bound for the multiplicative complexity as the remaining part is straightforwardly analogous to Proposition 16.

As in the proof of Theorem 25, the evaluation of $\mathrm{d}\mathbf{P}_{\mathbf{y}_0}^{-1}(\cdot)$ consists of a product of the inverse of a matrix over R and of a vector over R_p, and does not contribute to the multiplicative complexity. Therefore, $L^*(\mathbf{\Psi}) = L^*(\varepsilon_\mathbf{P}) = 3L^* + \frac{r(r+1)}{2}$. □

THEOREM 28. *Let $\mathbf{P}$ be a system of r polynomials in r variables over R and $\mathbf{y}_0 \in (R/(p))^r$ be such that $\mathbf{P}(\mathbf{y}_0) = 0 \bmod p$ and $\det(\mathrm{d}\mathbf{P}(\mathbf{y}_0)) \neq 0 \bmod p$. Denote $\mathbf{y} \in R_p^r$ the unique solution of $\mathbf{P}$ lifted from $\mathbf{y}_0$. Assume that $\mathbf{P}$ is given as a s.l.p. with multiplicative complexity L^*. Then, one can compute $\mathbf{y}$ to precision n in time $\left(3L^* + \frac{r(r+1)}{2}\right)\mathsf{R}(n) + \mathcal{O}(n)$.*

PROOF. By Propositions 24 and 27, $\mathbf{y}$ can be computed as a p-adic vector with the shifted algorithm $\mathbf{\Psi}$. Proposition 11 gives the announced complexity. □

6. IMPLEMENTATION AND TIMINGS

In this section, we display computation times in milliseconds for the univariate polynomial root lifting and for the computation of the product of the inverse of a matrix with a vector or with another square matrix. Timings are measured using one core of an INTEL XEON X5650 at 2.67 GHz running LINUX, GMP 5.0.2 [11] and setting $p = 536871001$ a 29 bit prime number. In the following tables, the first line, "Newton" corresponds to the classical Newton iteration [9, Algorithm 9.2] used in the zealous model. The second line "Mmx" corresponds to our best variant. The last line gives a few details about which variant is used. We make use of the naive variant "N" and the relaxed variant "R". Furthermore, when the precision is high, we make use of blocks of size 32 or 1024. That means, that at first, we compute the solution f up to precision 32 as $F_0 = f_0 + \cdots + f_{31}p^{31}$ with "N". Then, we say that our solution can be seen as a p^{32}-adic integer $F = F_0 + \cdots + F_np^{32n} + \cdots$ and the algorithm runs with F_0 as the initial condition. Then, each F_n is decomposed in

base p to retrieve $f_{32n}, \ldots, f_{32n+31}$. Although it is competitive, the initialization of F can be quite expensive. "BN" means that F is computed with "N", while "BR" means it is with "R". Finally, if the precision is high enough, one may want to compute F with blocks of size 32, and therefore f with blocks of size 1024. "B^2N" (resp. "B^2R") means that f and F are computed up to precision 32 with "N" and then, the p^{1024}-adic solution is computed with "N" (resp. "R").

Polynomial root. These first two tables correspond to the lifting of a regular root from $\mathbb{F}_p$ to $\mathbb{Z}_p$ at precision n as in Section 3.

Dense polynomial of size 8

n	16	64	2^8	2^{10}	2^{12}	2^{14}	2^{16}
Newton	0.023	0.078	0.52	4.1	29	170	870
Mmx	0.052	0.29	0.60	2.9	27	120	1300
Variant	N	N	BN	BN	B^2N	B^2N	B^2N

Dense polynomial of size 128

n	4	16	64	2^8	2^{10}	2^{12}	2^{14}
Newton	0.21	0.90	7.9	86	720	5400	30000
Mmx	0.086	0.71	4.4	46	140	600	4200
Variant	N	N	N	N	BN	BR	BR

Linear algebra. The next two tables correspond to timings of computing $B^{-1}A$ at precision n, with $A, B \in \mathcal{M}_{r\times r}(\mathbb{Z}_p)$.

Square matrices of size $r = 8$

n	4	16	64	2^8	2^{10}	2^{12}	2^{14}	2^{16}
Newton	0.097	0.22	0.89	6.8	59	490	3400	20000
Mmx	0.15	0.61	3.1	8.1	38	335	1600	14000
Variant	N	N	N	BN	BN	BN	B^2N	B^2N

Square matrices of size $r = 128$

n	4	16	64	2^8	2^{10}
Newton	930	2600	14000	140000	1300000
Mmx	3600	18000	53000	150000	1000000
Variant	N	N	N	BN	BN

As above, we solve integer linear systems, however, now we retrieve the solutions over $\mathbb{Q}$, using the rational number reconstruction [9, Section 5.10]. We set q as p to the power 2^j and pick at random a square matrix B of size r with coefficients in $M = \{0, \ldots, q-1\}$. We solve $BC = A$ with a random vector A. Because we deal with q-adic numbers at low precision, we only use the naive variant in our timings. We wanted to compare to LinBox [19] and IML [5]. However, we do not display the timings of IML within LinBox because they are about 10 times slower. It goes against the impression of [10, page 148] that IML is better for large integers.

Integer linear system of size $r = 4$

j	0	2	4	6	8	10	12
LinBox	1.0	1.4	3.6	25	310	4700	77000
Mmx	0.10	0.24	0.58	0.96	5.3	39	290

Integer linear system of size $r = 32$

j	0	2	4	6	8	10
LinBox	5.9	25	170	1900	27000	48000
Mmx	24	150	360	2000	14000	90000

In fact, when j is small, there is a major overhead coming from the use of Gmp. Indeed, in our case, it is best to transform q-adic numbers into p-adic numbers, to compute up to the necessary precision and then retrieve the solutions as q-adic numbers before calling the rational reconstruction.

Acknowledgments

We would like to thank J. van der Hoeven, M. Giusti, G. Lecerf, M. Mezzarobba and É. Schost for their helpful comments and remarks. For their help with LinBox, we thank J.-G. Dumas and B. Boyer.

This work has been partly supported by the Digiteo 2009-36HD grant of the Région Île-de-France, and by the French ANR-09-JCJC-0098-01 MaGiX project.

7. REFERENCES

[1] J. Berthomieu, J. v. d. Hoeven, and G. Lecerf. Relaxed algorithms for p-adic numbers. *J. Théor. Nombres Bordeaux*, 23(3):541–577, 2011.

[2] D. Bini and V. Y. Pan. *Polynomial and matrix computations. Vol. 1.* Birkhäuser Boston Inc., Boston, MA, 1994.

[3] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic complexity theory*, volume 315. Springer-Verlag, Berlin, 1997.

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

[5] Z. Chen and A. Storjohann. IML, the Integer Matrix Library, 2004. Version 1.0.3. Available from `http://www.cs.uwaterloo.ca/~astorjoh/iml.html`.

[6] J. D. Dixon. Exact solution of linear equations using p-adic expansions. *Numer. Math.*, 40(1):137–141, 1982.

[7] M. J. Fischer and L. J. Stockmeyer. Fast on-line integer multiplication. *J. Comput. System Sci.*, 9:317–331, 1974.

[8] M. Fürer. Faster Integer Multiplication. In *Proceedings of STOC 2007*, pages 57–66, San Diego, California, 2007.

[9] J. v. z. Gathen and J. Gerhard. *Modern computer algebra.* Cambridge University Press, Cambridge, second edition, 2003.

[10] P. Giorgi. *Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque LinBox.* PhD thesis, ENS Lyon, France, 2004. Available from `http://tel.archives-ouvertes.fr/tel-00008951`.

[11] T. Granlund et al. GMP, the GNU multiple precision arithmetic library, 1991. Version 5.0.2. Available from `http://www.swox.com/gmp`.

[12] J. v. d. Hoeven. Lazy multiplication of formal power series. In W. W. Küchlin, editor, *Proceedings of ISSAC '97*, pages 17–20, Maui, Hawaii, 1997.

[13] J. v. d. Hoeven. Relax, but don't be too lazy. *J. Symb. Comput.*, 34(6):479–542, 2002.

[14] J. v. d. Hoeven. New algorithms for relaxed multiplication. *J. Symbolic Comput.*, 42(8):792–802, 2007.

[15] J. v. d. Hoeven. Relaxed resolution of implicit equations. Technical report, HAL, 2009. `http://hal.archives-ouvertes.fr/hal-00441977/fr/`.

[16] J. v. d. Hoeven et al. Mathemagix, 2002. SVN Version 6374. Available from `http://www.mathemagix.org`.

[17] G. Kapoulas. Polynomially time computable functions over p-adic fields. In *Computability and complexity in analysis*, volume 2064, pages 101–118. Springer, Berlin, 2001.

[18] J. Karczmarczuk. Generating power of lazy semantics. *Theoret. Comput. Sci.*, 187(1-2):203–219, 1997.

[19] The LinBox Group. *LinBox – Exact Linear Algebra over the Integers and Finite Rings*, 2008. SVN Version 4136. Available from `http://linalg.org`.

[20] R. T. Moenck and J. H. Carter. Approximate algorithms to derive exact solutions to systems of linear equations. In *EUROSAM '79, Internat. Sympos., Marseille*, volume 72, pages 65–73. Springer, Berlin, 1979.

[21] I. Newton. *La méthode des fluxions, et les suites infinies.* de Bure aîné, 1740. French traduction by G. Buffon of the 1671 paper. Available at `http://gallica.bnf.fr`.

[22] M. Schröder. Fast online multiplication of real numbers. In *STACS 97 (Lübeck)*, volume 1200 of *Lecture Notes in Comput. Sci.*, pages 81–92. Springer, Berlin, 1997.

[23] A. Storjohann. High-order lifting and integrality certification. *J. Symbolic Comput.*, 36(3-4):613–648, 2003. ISSAC'2002, Lille.

[24] J. van der Hoeven. From implicit to recursive equations. Technical report, HAL, 2011. `http://hal.archives-ouvertes.fr/hal-00583125/fr/`.

[25] M. van Hoeij. Factorization of differential operators with rational functions coefficients. *J. Symb. Comput.*, 24(5):537–561, 1997.