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 [1], 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 [12, 13, 14, 15]. 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]. Lazy power series are streams of coefficients and the expressions they are involved in are evaluated as soon as the needed coefficients are provided. It was used in [10] to minimize the number of operations in that setting. The main drawback of the lazy representation is the bad complexity at high order of some basic algorithms such as multiplication.

Relaxed algorithms for formal power series were introduced in [12]. They share with the lazy algorithms the property that the coefficients of the output are computed one after another and that only minimal knowledge of 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 multiplication: the so-called on-line multiplication for integers in [6]. Then came the on-line multiplication for real numbers in [22], and relaxed multiplication for power series [11, 12], improved in [13].

One important advantage of relaxed algorithms is to allow the computation of recursive power series or p-adic numbers in a good complexity for both theoretical and practical purposes. 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. Relaxed algorithms are well-suited when one wants to lift a p-adic number to a rational number with no sharp *a priori* estimation on the required precision.

On the other hand, there are *zealous* algorithms. The precision is fixed in advance in the computations. The Newton-Hensel operator allows one 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 [5, 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 requires that the derivative at the point we wish to lift is not zero.

Our algorithms are worse by a logarithmic factor in the precision compared to zealous Newton iteration. However, the constant factors hidden in the big-O notation are poten-

ISSAC '12 Grenoble, FRANCE

tially smaller. Moreover, we take advantage of the good evaluation properties of the implicit equations. For example, we found a quasi-optimal algorithm for solving linear systems over p-adics (see Proposition 21 for s constant). Another example concerns the multivariate Newton-Hensel operator which performs at each step an evaluation of the implicit equations and an inversion of its evaluated Jacobian matrix. In Theorems 24 and 26, we manage to save the cost of the inversion of the Jacobian matrix at full precision.

Finally, we implement these algorithms to obtain timings competitive with Newton and even 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 [15]. This 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 blockwise relaxed algorithms are often an efficient alternative.

2. PRELIMINARIES

Straight-line programs. In this paper, we will make use of the straight-line model of computation to describe the algorithms behind the algebraic and recursive equations. We give a short presentation of this notion and refer to [2] for more details. Let R be a ring and A an 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 $r \times \_$ by r. We denote the set of all these operations by R^c and R. Finally, let us denote by 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 $\_/s : A \times S \to A$, and we denote by S their set. Let us fix a set of operations Ω, usually $\Omega = \{+, -, \cdot\} \cup R \cup S \cup R^c$.

An 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 an 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 $\Gamma_1 = (\cdot; -1, -1)$, $\Gamma_2 = (\cdot; 1, 0)$, $\Gamma_3 = (1^c)$, $\Gamma_4 = (-; 2, 3)$, $\Gamma_5 = (3 \times \_; 1)$.*

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 $(X, Y, X^2, X^2Y, 1, X^2Y - 1, 3X^2)$.

REMARK 2. *For the sake of simplicity, we will associate an arithmetic expression with an 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 an 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 $\varphi : Z \mapsto (Z^2)^2 + 1$ can be represented by the s.l.p. with one argument 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; $\mathsf{quo}(\_, p)$ is a function from R to R and $\mathsf{rem}(\_, p)$ 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, 12]. 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. *Addition in $\mathbb{Z}_p$ is implemented in the class* $\mathsf{Sum\_Padic}_p$ *derived from* Padic_p *and therefore inherits 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 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 by $a_n \in M$ 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 $a_0b_1+a_1b_0+\gamma$ with both products a_0b_1 and a_1b_0. Then, the remainder of the division by p is assigned to c_1 while the quotient is to γ.

Some modifications happen 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 $a_0b_2 + a_2b_0 + (a_1 + a_2p)(b_1 + b_2p) + \gamma$ using two multiplications of elements of size 1 and one of elements of size 2: $(a_1 + a_2p)(b_1 + b_2p)$. 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 $a_0b_3+a_3b_0+\gamma$, with two multiplications of elements of size 1. Instead of the two products in $a_1b_2+a_2b_1$, we used fast algorithms on the bigger data $a_1 + a_2p$ and $b_1 + b_2p$ 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. Let us denote by $\mathsf{M}(n)$ 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 $\mathsf{M}(n) \in \mathcal{O}(n \log n \log \log n)$ [3].

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 [7], where $\log^*$ is the iterated logarithm.

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

Theorem 4 ([1, 6, 12]). *The complexity $\mathsf{R}(n)$ for multiplying two p-adic numbers at precision n, using the relaxed multiplication, is $\mathcal{O}(\mathsf{M}(n) \log n)$.*

The previous result was first discovered for integers in [6]. However the application to compute recursive power series or p-adic integers was seen for the first time in [12]. Article [1] generalizes relaxed algorithms for p-adic numbers.

Remark 5. *If $R = \mathbb{F}_p$ contains many 2^pth roots of unity, then two power series over $\mathbb{F}_p$ can be multiplied in precision n in $\mathcal{O}\left(\mathsf{M}(n)\, \mathrm{e}^{2\sqrt{\log 2 \log \log n}}\right)$ multiplications in $\mathbb{F}_p$, see [13].*

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 denote by $\nu_p(a)$ the valuation in p of the p-adic number a.

Definition 1. Let $\Phi \in R_p[Y]$, y be a fixed point of Φ, *i.e.* $y = \Phi(y)$, and let $y_0 = y \bmod p$, $y_0 \in M$. 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 $(y - \Phi^n(y_0)) \in p^{n+1}R_p$.

The vector case is included in the definition: 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.

Proposition 6. *Let $\Phi \in R_p[Y]$ with a fixed point y and let $y_0 = y \bmod p$. Suppose $\nu_p(\Phi'(y_0)) > 0$. Then Φ allows the computation of y.*

Moreover, for all $n \leqslant m \in \mathbb{N}^*$, $(\Phi(y))_n$ does not depend on y_m, i.e. *$(\Phi(y))_n = (\Phi(y + a))_n$ for any $a \in p^nR_p$.*

Proof. First, notice that $(y - y_0) \in pR_p$. Then, for all $y, z \in R_p$, there exists, by Taylor expansion of Φ at z, a p-adic $\Theta(y, z)$ such that $\Phi(y) - \Phi(z) = \Phi'(z)(y - z) + (y - z)^2\Theta(y, z)$. For all $n \in \mathbb{N}$, we set $y_{(n)} := \Phi^n(y_0)$ and we apply the previous statement to y itself and $z = y_{(n)}$:

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

By our assumption, $\nu_p\left(\Phi'\left(y_{(n)}\right)\right) > 0$, so that we have $\frac{y - y_{(n+1)}}{y - y_{(n)}} \in pR_p$, for all n in $\mathbb{N}$. For the second point, one has $\Phi(y+a) - \Phi(y) = \Phi'(y)a + a^2\Theta(y+a, y) \in p^{n+1}R_p$ since $\nu_p(\Phi'(y)) > 0$. $\square$

We refer to [1, Section 5.1] for a description of the implementation of a recursive p-adic number y with input Φ and y_0. We recall briefly the ideas to compute y from $\Phi(y)$ in the lazy framework. What we really compute is $\Phi(y)$. Suppose that we are at the point where we know $y_0, \ldots, y_{n-1}$ and $\Phi(y)$ has been computed up to its $(n-1)$st coefficient. Since in the lazy framework, the computation are done step by step, one can naturally ask for one more step of the computation of $\Phi(y)$. Also, from Proposition 6, $(\Phi(y))_n$ depends only on $y_0, \ldots, y_{n-1}$ so that we can compute it and deduce $y_n = (\Phi(y))_n$.

But one has to be cautious 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 fixed point of Φ satisfying $y_0 = 0$. Since $\Phi'(0) = 0$, Φ allows the computation of y.*

At the first step, we find $(\Phi(y))_1 = 1$. Then we compute $(\Phi(y))_2 = (y^2 + p)_2 = (y^2)_2$. At the second step of the relaxed multiplication algorithm, we compute y_0y_2 and $q = (y_1 + y_2p)(y_1 + y_2p)$. Here we face a problem. The p-adic number q 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 make the fact that y_n is not required in the computation of $(\Phi(y))_n$ explicit. This issue was never mentioned in the literature before. In this section, we solve this problem for polynomial operators Φ. For this matter, we introduce for all i in $\mathbb{N}^*$ two new operators:

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

Let Ω' be the set of operations $\{+, -, \cdot, p^i \times \_, \_/p^i\} \cup R \cup S \cup R^c$. In the next definition, we define a number, the *shift*, that will indicate which coefficients of an argument of an s.l.p. are involved in its output from a syntactic point of view.

Definition 2. Let $\Gamma = (\Gamma_1, \ldots, \Gamma_k)$ be an 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 its ith result b_i 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. 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 $\Gamma_i = (p^s \times \_; u)$, then $\mathrm{sh}(\Gamma, i, j) := \mathrm{sh}(\Gamma, u, j) + s$;
- if $\Gamma_i = (\_/p^s; u)$, 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.

With this definition, we can now prove that if an s.l.p. Ψ has a positive shift, then $(\Psi(y))_n$ does not depend on y_n.

Proposition 8. *With the notations of Definition 2, let $\mathbf{y} = (y_0, \ldots, y_{r-1}) \in (R_p)^r$ be such that Γ is executable on input $\mathbf{y}$ and $b_i \in R_p$ be the ith element of the result sequence. Then, for all $n \in \mathbb{N}$, the computation of $(b_i)_n$ involves at most the terms $(y_j)_\ell$ of the argument y_j where $0 \leqslant \ell \leqslant \max(0, n - \mathrm{sh}(\Gamma, i, j-(r-1)))$.*

Proof. By recursion on the second argument i of sh. When $-(\ell-1) \leqslant i \leqslant 0$, the result b_i equals to $y_{i+(r-1)}$ so that the proposition is easily checked. Now recursively for $i > 0$. If $\Gamma_i = (p^s \times \_; u)$, then for all $n \in \mathbb{N}$, $(b_i)_n = (p^s b_u)_n = (b_u)_{n-s}$ which, by assumption, depends at most on the coefficients $(y_j)_\ell$ of the argument y_j where $0 \leqslant \ell \leqslant \max(0, n-s-\mathrm{sh}(\Gamma, i, j-(r-1)))$. If $\Gamma_i = (\cdot; u, v)$, then for all $n \in \mathbb{N}$, $(b_i)_n = (b_u \cdot b_v)_n$. Since the nth term of the product b_i depends only on the terms up to n of its argument b_u and b_v, the proposition follows. The other cases can be treated similarly. □

Example 9. *We carry on with the notations of Warning 7. Recall that $(\Phi(y)_n)$ involved y_ℓ for $0 \leqslant \ell \leqslant n$. This is due to the fact that $\mathrm{sh}(\Gamma) = 0$, where Γ is the s.l.p. with one argument associated to the arithmetic expression $Z \mapsto Z^2 + p$ (see Remark 2). Now consider the s.l.p.*

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

Then $\mathrm{sh}(\Psi) = 1$, since $\mathrm{sh}(p^2 \times (Z/p)^2) = \mathrm{sh}((Z/p)^2) + 2 = \mathrm{sh}(Z/p) + 2 = \mathrm{sh}(Z) + 1$. So Proposition 8 ensures that the s.l.p. Ψ solves the problem raised in Warning 7.

Still, we detail the first steps of the new algorithm to convince even the most skeptical reader. At the nth step, we deduce y_n from $(\Psi(y))_n = ((y/p)^2)_{n-2} + p_n$. Therefore, $(\Psi(y))_1 = 0 + 1$. Then $(\Psi(y))_2 = ((y/p)^2)_0 + 0$ and the relaxed product algorithm will compute y_1^2. At step 3, we compute the term 1 of the product, which uses $y_0 y_1$. Finally at step 4, the relaxed product algorithm computes $y_1 y_3$ and $q = (y_2 + y_3 p)(y_2 + y_3 p)$. We have solved the dependency issue.

We are now able to express which s.l.p.'s Ψ are suited to the implementation of recursive p-adic numbers. Recall that S is the set of regular elements in R. We denote by $K := S^{-1} R$ 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 an s.l.p. with one argument and operations in Ω'.

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

Next proposition is the cornerstone of complexity estimates regarding recursive p-adic numbers.

Proposition 10. *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. Proposition 8 proved that we can compute y by computing $\Psi(y)$ in the relaxed framework. Therefore the cost of the computation of y is exactly 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 11. *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

In this subsection, we fix a polynomial P of degree d given in dense representation, that is 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. Recall, from Definition 2, that the shift of $\Phi(Y)$ is 0.

Lemma 12. *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 thus 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 13. *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 $\mathcal{O}(d)$ 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 12, $\mathrm{sh}\left(p^2 \times (\mathrm{sq}(Z)\cdot Q(Z))\right) = 1$, we have $\mathrm{sh}(\Psi) = 1$. Also, thanks to Lemma 12, 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 time $\mathcal{O}(d)$ *via* the naïve Euclidean division algorithm. Using Horner scheme to evaluate $Q(Z)$, we have $L^*(\Psi) = d-1$ and we can apply Proposition 10. □

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 a shifted algorithm was not introduced, an algorithm of multiplicative complexity $\mathcal{O}(L^*(P))$ was given. The shifted algorithm was only present in the implementation in Mathemagix [16]. We clarify and generalize this approach to any polynomial P given as an 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 an s.l.p. Γ 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 an s.l.p. ε with operations in $\Omega' := \left\{+,-,\cdot,p^i \times \_, \_/p^i\right\} \cup R \cup S \cup R^c$. 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 $\Gamma_i = (a \times \_; u)$, 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 by $\tau^u = (a, A), \tau^v = (b, B)$, then $\tau^i = (ab, aB + bA)$ and ε^i equals

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

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 14. *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 denote by $E^i := E_{P_i}$ 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

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

Also $\tau^i = (P_u(y_0)\,P_v(y_0), P_u(y_0)\,P_v'(y_0) + P_v(y_0)\,P_u'(y_0))$. 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 $(A\varepsilon^v(y) + B\varepsilon^u(y))/p \in R_p$. 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 15. *Let P be a univariate polynomial over R_p given as an s.l.p. whose 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 14 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 16. *By adding the square operation $\_^2$ to the set of operations Ω of P, we can save a few multiplications. In Definition 4, if $\Gamma_i = (\_^2; u)$ and $\tau^u = (a, A)$, then we define ε^i by $\varepsilon^u \cdot (\varepsilon^u + 2 \times (a + A \times (Z - y_0))) + p^2 \times \left(A^2 \times ((Z-y_0)/p)^2\right)$. Thereby, we reduce the multiplicative complexity of ε_P and Ψ by the number of square operations in P.*

Theorem 17. *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 by $y \in R_p$ the unique solution of P lifted from y_0. Assume that P is given as an 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 11 and 15, y can be computed as a recursive p-adic number with the shifted algorithm Ψ. Proposition 10 gives the announced complexity. □

Remark 18. *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.'s. Then the part $p\beta(Y)$ is already shifted. In this case, set $\tilde{\varepsilon}_P := \varepsilon_\alpha + p\beta$ so that*

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

is a shifted algorithm for P with multiplicative complexity $2L^*(\alpha) + L^*(\beta) + 1$.

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.

Any matrix $A \in \mathcal{M}_{r\times s}(R_p)$ 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 two operators $\mathsf{mul\_rem}$ and $\mathsf{mul\_quo}$ such that for all $\beta \in M$ and $a \in R_p$, $\mathsf{mul\_rem}(\beta, a)_n = \mathsf{rem}(\beta a_n, p)$ and $\mathsf{mul\_quo}(\beta, a)_n = \mathsf{quo}(\beta a_n, p) \in M$. Thus

$$\beta a = \mathsf{mul\_rem}(\beta, a) + p\,\mathsf{mul\_quo}(\beta, a).$$

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

$$BA = \mathsf{Mul\_rem}(B, A) + p\,\mathsf{Mul\_quo}(B, A).$$

Let us denote by $\mathsf{MM}(r, s)$ 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 $r \geqslant s$, 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 19. *Let B and A be two p-adic matrices such that $B \in \mathcal{M}_r(M)$ and $A \in \mathcal{M}_{r\times s}(R_p)$. Then, the computations of $\mathsf{Mul\_rem}(B, A)$, $\mathsf{Mul\_quo}(B, A)$, and therefore of BA, can be done to precision n in time $\mathcal{O}(\mathsf{MM}(r, s)n)$.*

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

Proposition 20. *Let A be a relaxed matrix of size $r \times s$ over R_p and let $B \in \mathcal{M}_r(M)$. If B is invertible modulo p and $\Gamma := B^{-1} \bmod p$, then the product $C = B^{-1}A$ satisfies*

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

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

Proof. First, Γ is computed in time $\mathcal{O}(\mathsf{MM}(r, r))$. Then $A = \mathsf{Mul\_rem}(B, C) + p\,\mathsf{Mul\_quo}(B, C)$ so we can deduce that $\mathsf{Mul\_rem}(B, C) = A - p\,\mathsf{Mul\_quo}(B, C)$. It remains to multiply both sides by Γ using $\mathsf{Mul\_rem}$ to prove equation (2).

From the definition of $\mathsf{Mul\_quo}$, 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 19. □

4.2 Inversion of a matrix over p-adics

We can now apply the division of matrices over p-adic integers, as in [12].

Proposition 21. *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*

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

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

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 $\mathsf{MM}(r, s)\,\mathsf{R}(n)$. The cost of the product by B_0^{-1} is estimated in Proposition 20. □

Remark 22. *Notice that if $B \in \mathcal{M}_r(R)$, then the p-adic multiplications in the proof of Proposition 21 are in $R \times R_p$, so that $C = B^{-1}A$ costs 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 $\mathcal{M}_r(R/(p))$. The 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 23. *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 11. 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

In this subsection, we assume that the algebraic system $\mathbf{P}$ is given in dense representation. We assume that $d \geqslant 2$, where $d := \max_{1 \leqslant i,j \leqslant r}(\deg_{X_j}(P_i))+1$, so that the dense size of $\mathbf{P}$ is bounded by rd^r.

As in the univariate case, the shift of $\mathbf{\Phi}(\mathbf{Y})$ is 0. We adapt Lemma 12 and Proposition 13 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 24. *Let $\mathbf{P} = (P_1, \ldots, P_r)$ be a polynomial system in $R[\mathbf{Y}]^r$ in dense representation, satisfying $d \geqslant 2$ where $d := \max_{1\leqslant i,j\leqslant r}(\deg_{X_j}(P_i)) + 1$, and let $\mathbf{y}_0$ be an approximate zero of $\mathbf{P}$.*

Then 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 computing $\mathbf{y}$ up to precision n costs $rd^r\mathsf{R}(n)+\mathcal{O}(r^2n+r^\omega)$.

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. The naïve algorithm does the first division in time $\mathcal{O}(rd^r)$. Then the second division costs $\mathcal{O}(r2d^{r-1})$ because we reduce a polynomial with less monomials. The third $\mathcal{O}(r2^2d^{r-2})$ and so on. At the end, all these divisions are done in time $\mathcal{O}(rd^r)$. 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 $\mathcal{O}(2^r) = o(rd^r)$.

Next, we have to evaluate $\mathbf{\Psi}_1$ at $\mathbf{y}$. Since the total numbers of monomials of $\mathbf{Q}^{(j,k)}(\mathbf{Y})$ for $1 \leqslant j \leqslant k \leqslant r$ is bounded by rd^r, Proposition 10 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 20 and Remark 22, and since we only lift a single root, it can be done at precision n in time $\mathcal{O}(r^2n + r^\omega)$. □

5.2 Algebraic systems as s.l.p.'s

In this subsection, we assume that the algebraic system $\mathbf{P}$ is given as an 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 $\tau_i \in R \times R_p, \varepsilon_i \in R_p$ for $1 \leqslant i \leqslant r$ with operations in $\Omega' := \{+, -, \cdot, p^j \times \_, \_/p^j\} \cup R \cup S \cup R^c$.

Initialize $\varepsilon_i^{-r+j} := 0, \tau_i^{-r+j} := (y_{j,0}, y_j - y_{j,0})$ for all $1 \leqslant j \leqslant r$. Then for $1 \leqslant j \leqslant k_i$ where k_i is the number of instructions in the s.l.p. P_i, we define ε_i^j and τ_i^j recursively on j by formulas similar to Definition 4. Let us detail the changes when $\Gamma_j = (\cdot; u, v)$:

Let $\tau_i^u = (a, A)$ and $\tau_i^v = (b, B)$, then define τ_i^j by $(ab, a \times B + b \times A)$ and ε_i^j by

$$p\times\left((a+A+\varepsilon_i^u)\cdot\frac{\varepsilon_i^v}{p}+(b+B)\cdot\frac{\varepsilon_i^u}{p}\right)+p^2\times\left(\frac{A}{p}\cdot\frac{B}{p}\right).$$

As before, we set $\varepsilon_{P_i} := \varepsilon_i^{k_i}$ and $\tau_{P_i} := \tau_i^{k_i}$.

Lemma 25. *If $\tau_{P_i} = (a, A)$ then $a = P_i(\mathbf{y}_0)$ and $A = \mathrm{d}(P_i)_{\mathbf{y}_0}(\mathbf{Y} - \mathbf{y}_0) \in R_p$. Besides, $\varepsilon_{\mathbf{P}} := (\varepsilon_{P_1}, \ldots, \varepsilon_{P_r})$ is a shifted algorithm for $\mathbf{E}_{\mathbf{P}}$ and $\mathbf{y}_0$ whose complexity is $3L^*$.*

Proof. Following the proof of Lemma 14, the first assertion is clear, as is the fact that $\varepsilon_{\mathbf{P}}$ is a shifted algorithm for $\mathbf{E}_{\mathbf{P}}$ and $\mathbf{y}_0$. Finally, for all instructions $\cdot$ in the s.l.p. P_i, ε_{P_i} adds three multiplications between p-adics (see operations $\cdot$ in formulas above). So $L^*(\varepsilon_{\mathbf{P}}) = 3L^*$. □

Theorem 26. *Let $\mathbf{P}$ be a polynomial system of r polynomials in r variables over R, given as an s.l.p. such that its multiplicative complexity is L^*. Let $\mathbf{y}_0 \in (R/(p))^r$ 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 by $\mathbf{y}$ the unique solution of $\mathbf{P}$ lifted from $\mathbf{y}_0$.*

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 $\mathbf{y}_0$. Thus, one can compute $\mathbf{y}$ to precision n in time $3L^*\mathsf{R}(n) + \mathcal{O}(r^2n + r^\omega)$.

Proof. Similarly to Proposition 15, $\mathbf{\Psi}$ is a shifted algorithm. By Proposition 10 and Lemma 25, one can evaluate $\varepsilon_{\mathbf{P}}(\mathbf{y})$ in time $3L^*\mathsf{R}(n) + \mathcal{O}(n)$. Besides, by Remark 22, the operation $\mathrm{d}\mathbf{P}_{\mathbf{y}_0}^{-1}(\_)$ costs $\mathcal{O}(r^2n + r^\omega)$. □

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 [9] and setting $p = 536871001$ a 30 bit prime number.

Our implementation is available in the files whose names begin with `series_carry` or `p_adic` in the C++ library algebramix of Mathemagix.

In the following tables, the first line, "Newton" corresponds to the classical Newton iteration [8, Algorithm 9.2] used in the zealous model. The second line "Relaxed" 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 the variant "N". Then, we say that our solution can be seen as a p^{32}-adic integer $F = F_0 + \cdots + F_n p^{32n} + \cdots$ 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 the variant "N", while "BR" means it is with the variant "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 the variant "N" and then, the p^{1024}-adic solution is computed with the variant "N" (resp. "R").

Polynomial root. This table corresponds to the lifting of a regular root from $\mathbb{F}_p$ to $\mathbb{Z}_p$ at precision n as in Section 3.1.

Dense polynomial of degree 127

n	512	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
Newton	17	48	140	380	1000	2500	5900
Relaxed	120	140	240	600	1600	4200	11000
Variant	R	BN	BN	BR	BR	BR	BR

In this table, the timings of "Newton" are always better than "Relaxed". However, if the unknown required precision is slightly above a power of 2, *e.g.* $2^\ell + 1$, then one needs to compute at precision $2^{\ell+1}$ with Newton algorithms. Whereas relaxed algorithms increase the precision one by one. So the timings of "Relaxed" are better on significant ranges after powers of 2.

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

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
Relaxed	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
Relaxed	3600	18000	53000	150000	1000000
Variant	N	N	N	BN	BN

Now, we solve integer linear systems and retrieve the solutions over $\mathbb{Q}$, using the rational number reconstruction [8, 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 variant "N". We compared with LinBox [19] and IML [4] but we do not display the timings of IML within LinBox because they are about 10 times slower. As LinBox and IML are designed for big matrices and small integers, it is not surprising that "Relaxed" performs better.

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
Relaxed	0.10	0.24	0.58	2.1	14	110	760

Integer linear system of size $r = 32$

j	0	2	4	6	8	10
LinBox	5.9	25	170	1900	27000	480000
Relaxed	24	150	360	2000	14000	90000

In fact, when $j \leqslant 3$, there is a major overhead coming from the use of Gmp. Indeed, in these cases, we transform q-adic numbers into p-adic numbers, compute up to the necessary precision and call 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 B. Boyer and J.-G. Dumas.

We would like to thank the three anonymous referees for their helpful comments.

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

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

[4] Z. Chen and A. Storjohann. IML, the Integer Matrix Library, 2004. Version 1.0.3.

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

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

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

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

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

[10] M. v. Hoeij. Formal solutions and factorization of differential operators with power series coefficients. *J. Symbolic Comput.*, 24(1):1–30, 1997.

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

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

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

[14] J. v. d. Hoeven. Relaxed resolution of implicit equations. Technical report, HAL, 2009.

[15] J. v. d. Hoeven. From implicit to recursive equations. Technical report, HAL, 2011.

[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.

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

[21] I. Newton. *The method of fluxions and infinite series: with its application to the geometry of curve-lines*. Henry Woodfall, 1736.

[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.