Relaxed p-adic Hensel lifting for algebraic systems

This chapter is the main part of the homonym paper published with J. Berthomieu in the proceedings of *ISSAC'12* [BL12].

In this chapter, 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. This chapter can be seen as a special and simpler case of lifting of triangular set done in Chapter 6.

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 Newton iteration. However, the constant factors hidden in the big-O notation are potentially smaller. Moreover, our algorithm's cost is roughly the cost of evaluating the implicit equation by on-line algorithms. This can lead to further savings compared to the cost of off-line methods. For example, consider 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 5.11 and 5.14, 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 [Hoe11]. 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 [BHL11, Section 4] cannot be used. This is important because blockwise relaxed algorithms are often an efficient alternative.

5.1 Univariate root lifting

In [BHL11, 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 5.1. *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.

5.1.1 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 2.11, that the shift of $\Phi(Y)$ is 0. In this chapter, for any two p-adics a and b, we denote by $a \cdot b$ their multiplication. If at least one of them has finite precision, we denote by $a\,b$ their multiplication.

Lemma 5.2. *The s.l.p.* $\Gamma\colon Z \mapsto p^2 \times \left(\left(\frac{Z - y_0}{p}\right)^2 \cdot Z^k\right)$ *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 - 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 $\mathrm{sq}(Z)\colon Z \mapsto \left(\frac{Z - y_0}{p}\right)^2$
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 5.3. *Given a polynomial P of degree d in dense representation and a modular simple root y_0, Algorithm 5.2.1 defines a shifted algorithm Ψ associated to Φ. The precomputation of such an operator involves $\mathcal{O}(d)$ operations in R. If λ is the length of $P'(y_0)$, then we can lift y at precision N in time*

$$(d-1)\,\mathsf{R}(N) + \mathcal{O}(N\,d + N\,\mathsf{R}(\lambda)/\lambda)$$

or equivalently

$$(d-1)\,\mathsf{R}(N) + \mathcal{O}(N\,d) + N \log(\lambda)^{\mathcal{O}(1)}.$$

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 5.2, $\mathrm{sh}(p^2 \times (\mathrm{sq}(Z) \cdot Q(Z))) = 1$, we have $\mathrm{sh}(\Psi) = 1$. Also, thanks to Lemma 5.2, 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 2.17. Note that by Proposition 3.6 for $r = 1$, the inversion of $P'(y_0)$ costs $\mathcal{O}(N\,\mathsf{R}(\lambda)/\lambda)$. Finally, the evaluation of Q also involves $\mathcal{O}(d)$ on-line additions which cost $\mathcal{O}(N\,d)$. □

In comparison, Newton iteration lifts y at precision N in time $(3\,d + \mathcal{O}(1))\,\mathsf{I}(N) + \mathcal{O}(d\,N)$ (see [GG03, Theorem 9.25]). Here, the universal constant in the $\mathcal{O}(1)$ corresponds to p-adic inversion and can be taken less than 4. The reminder on Newton iteration can be found in Section 6.3.1.

So the first advantage of our on-line algorithm is that it does asymptotically less on-line multiplications than Newton iteration does off-line multiplications. Also, we can expect better timings from the on-line method for the Hensel lifting of y when the precision N satisfies $\mathsf{R}(N) \leqslant 3\,\mathsf{I}(N)$.

5.1.2 Polynomials as straight-line programs

In [BHL11, 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 [HLM+02]. 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 L 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 5.4. *We define recursively a vector $\tau \in R^2$ and an s.l.p. ε with operations in $\Omega' := \{+, -, \cdot, p^i \times \_, \_/p^i\} \cup R \cup R^c$. Initially, $\varepsilon^0 := 0$ and $\tau^0 := (y_0, 1)$. Then, we define ε^i and τ^i recursively on i with $1 \leqslant i \leqslant L$ 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* $\tau^i = (a\,b,\ a\,B + b\,A)$ *and* ε^i *equals*

$$\varepsilon^u \cdot \varepsilon^v + p \times (((A \times \varepsilon^v + B \times \varepsilon^u)/p) \cdot (Z - y_0)) + (a \times \varepsilon^v + b \times \varepsilon^u) + p^2 \times ((A\,B) \times ((Z - y_0)/p)^2). \tag{5.1}$$

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

Lemma 5.5. *The s.l.p.* ε_P *is a shifted algorithm for* E_P *and* y_0. *Its multiplicative complexity is bounded by* $2\ L^* + 1$. *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 $0 \leqslant i \leqslant L$. We denote by $E^i := E_{P_i}$ and $T^i := T_{P_i}$ for all $0 \leqslant i \leqslant L$. 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 (5.1) corresponds to the last equation of

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

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 (5.1) 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 (5.1)). So $L^*(\varepsilon_P) = 2\,L^* + 1$. □

Proposition 5.6. *Let* P *be a univariate polynomial over* R_p *given as an s.l.p. whose multiplicative complexity is* L^*. *Then, the following algorithm*

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

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

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 5.5 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 $L^*(\Psi) = L^*(\varepsilon_P) = 2\,L^* + 1$ as the on-line division by $P'(y_0)$ does not require any multiplication between full precision p-adics (see Chapter 3). □

Remark 5.7. By adding the square operation $\_^2$ to the set of operations Ω of P, we can save a few multiplications. In Definition 5.4, if $\Gamma_i = (\_^2; u)$ and $\tau^u = (a, A)$, then we define ε^i by $\varepsilon^u \cdot (\varepsilon^u + 2 \times (a + A \times (Z - y_0))) + p^2 \times (A^2 \times ((Z - y_0)/p)^2)$. Thereby, we reduce the multiplicative complexity of ε_P and Ψ by the number of square operations in P.

Theorem 5.8. *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^*. Let λ be a bound on the length of all elements $P_i(y_0)$ in the result sequence of the evaluation of P at y_0 and on all $r \in R$ such that $r \times \_$ is an operation of the s.l.p. P.*

Then, we can lift y up to precision N in time

$$(2\,L^* + 1)\,\mathsf{R}(N) + \mathcal{O}(N\,L\,\mathsf{R}(\lambda)/\lambda),$$

that is

$$(2\,L^* + 1)\,\mathsf{R}(N) + N\,L \log(\lambda)^{\mathcal{O}(1)}.$$

Proof. By Propositions 5.1 and 5.6, y can be computed as a recursive p-adic number with the shifted algorithm Ψ. Proposition 2.17 gives that the cost of lifting y up to precision N is the cost of evaluating $\Psi(y)$ at precision N. This evaluation requires $(2\,L^* + 1)$ on-line multiplications, $\mathcal{O}(L)$ additions, $\mathcal{O}(L)$ multiplications between p-adics with one operand of finite length $\mathcal{O}(\lambda)$ (coming either from operations $r \times \_$ or $\cdot$ in P) and a division by $P'(y_0)$ for a total cost of

$$(2\,L^* + 1)\,\mathsf{R}(N) + \mathcal{O}(N\,L + N\,L\,\mathsf{R}(\lambda)/\lambda + N\,\mathsf{R}(\lambda)/\lambda). \qquad \square$$

In this case, Newton iteration costs $(4\,L^* + \mathcal{O}(1))\,\mathsf{I}(N) + \mathcal{O}(L\,N)$. To prove this claim, we have to show that the evaluation of P at precision N costs $L^*\,\mathsf{I}(N) + \mathcal{O}(L\,N)$, and that the evaluation of P' at precision $N/2$ costs $2\,L^*\,\mathsf{I}(N/2) + \mathcal{O}(L\,N)$. One way to compute $(P(y), P'(y))$ is to evaluate P at $y + \varepsilon$ in the ring of tangent numbers $R_p[\varepsilon]/\varepsilon^2$. Then $P(y + \varepsilon) = P(y) + \varepsilon\,P'(y)$. Note that

$$\begin{aligned} (a + b\,\varepsilon) + (c + d\,\varepsilon) &= (a + c) + (b + d)\,\varepsilon \\ (a + b\,\varepsilon)\,(c + d\,\varepsilon) &= a\,c + (b\,c + a\,d)\,\varepsilon \end{aligned}$$

in $R_p \oplus R_p\,\varepsilon = R_p[\varepsilon]/\varepsilon^2$. Consequently a multiplication in $R_p[\varepsilon]/\varepsilon^2$ costs 3 multiplications in R_p. But because we want the coefficient in ε at precision only $N/2$, we require b and d at precision $N/2$. Therefore by evaluating P at $y + \varepsilon$ in $R/(p^N) \oplus R/(p^{N/2})\ \varepsilon$, we obtain $P(y)$ at precision N and $P'(y)$ at precision $N/2$ in time $2\,L^*\,\mathsf{I}(N/2) + L^*\,\mathsf{I}(N) + \mathcal{O}(L\,N)$. The inversion of $P'(y)$ costs $\mathcal{O}(\mathsf{I}(N))$.

Therefore the last step of Newton iteration costs $(2\,L^* + \mathcal{O}(1))\,\mathsf{I}(N) + \mathcal{O}(L\,N)$. Finally, the whole Newton iteration involves the steps $N, N/2, N/4, \ldots$ for a total cost of $(4\,L^* + \mathcal{O}(1))\,\mathsf{I}(N) + \mathcal{O}(L\,N)$.

Remark 5.9. 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

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

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

5.2 Multivariate root lifting

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

Proposition 5.10. *The polynomial system*

$$\boldsymbol{\Phi}(\boldsymbol{Y}) := \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)^{-1}(\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\,\boldsymbol{Y} - \boldsymbol{P}(\boldsymbol{Y})) \in K[\boldsymbol{Y}]^r$$

allows the computation of $\boldsymbol{y}$.

Proof. We adapt the proof of Proposition 5.1. Since $\mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_0) = \boldsymbol{0}$, $\boldsymbol{\Phi}$ allows the computation of $\boldsymbol{y}$. □

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

5.2.1 Dense algebraic systems

In this subsection, we assume that the algebraic system $\boldsymbol{P}$ 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 $\boldsymbol{P}$ is bounded by $r\,d^r$.

As in the univariate case, the shift of $\boldsymbol{\Phi}(\boldsymbol{Y})$ is 0. We adapt Lemma 5.2 and Proposition 5.3 to the multivariate polynomial case as follows. For $1 \leqslant j \leqslant k \leqslant r$, let $\boldsymbol{Q}^{(j,k)}$ be polynomial systems such that $\boldsymbol{P}(\boldsymbol{Y})$ equals

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

Algorithm - Dense polynomial system root lifting

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

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

   $\boldsymbol{\Psi} \colon \boldsymbol{Z} \mapsto -\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)^{-1}(\boldsymbol{P}(\boldsymbol{y}_0) - \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\, \boldsymbol{y}_0 + p^2 \times \boldsymbol{\Psi}_1)$.

Theorem 5.11. *Let $\boldsymbol{P} = (P_1, \ldots, P_r)$ be a polynomial system in $R[\boldsymbol{Y}]^r$ in dense representation, satisfying $d \geqslant 3$ where $d := \max_{1 \leqslant i,j \leqslant r} (\deg_{X_j}(P_i)) + 1$, and let $\boldsymbol{y}_0$ be an approximate zero of $\boldsymbol{P}$. Let λ be a bound on the length of the polynomial coefficients of $\boldsymbol{P}$ and on the entries of $\mathrm{Jac}_{\boldsymbol{P}}(y_0)$.*

Then Algorithm 5.2.1 outputs a shifted algorithm $\boldsymbol{\Psi}$ associated to $\boldsymbol{\Phi}$ and $\boldsymbol{y}_0$. The precomputation in $\boldsymbol{\Psi}$ costs $\mathcal{O}(r\, d^r)$ operations in R, while computing $\boldsymbol{y}$ up to precision N costs

$$d^r\, \mathsf{R}(N) + \mathcal{O}(N\, [r\, d^r\, \mathsf{R}(\lambda)/\lambda + \mathsf{MMR}(r, 1, \lambda)/\lambda] + r^\omega),$$

that is

$$d^r\, \mathsf{R}(N) + N\, r\, d^r \log(\lambda)^{\mathcal{O}(1)} + \mathcal{O}(r^\omega).$$

Proof. First, for $j \leqslant r$, we perform the Euclidean division of $\boldsymbol{P}$ by $(Y_j - y_{j,0})^2$ to reduce the degree in each variable. The naïve algorithm does the first division in time $\mathcal{O}(r\, d^r)$. Then the second division costs $\mathcal{O}(r\, 2\, d^{r-1})$ because we reduce a polynomial with less monomials. The third $\mathcal{O}(r\, 2^2\, d^{r-2})$ and so on. At the end, all these divisions are done in time $\mathcal{O}(r\, d^r)$. 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 $Y_j\, Y_k$, which gives rise to a cost of $\mathcal{O}(2^r) = o(r\, d^r)$. Finally, the entries of the Jacobian matrix $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$ are obtained as the coefficients in $(Y_j - y_{j,0})$ of the resulting polynomial and $\boldsymbol{P}(\boldsymbol{y}_0)$ as the constant coefficient. The multiplication $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\, \boldsymbol{y}_0$ takes $\mathcal{O}(r^2) = o(r\, d^r)$ operations in R.

Next, we have to evaluate $\boldsymbol{\Psi}_1$ at $\boldsymbol{y}$. We start by computing the evaluation at $\boldsymbol{y}$ of all the monomials appearing in $\boldsymbol{\Psi}_1$. There are at most d^r monomials. Since each monomial, except 1, is obtained as the product of another monomial by one Z_j with $1 \leqslant j \leqslant r$, all these evaluations take d^r on-line multiplications.

Then, for each component of the vector $\boldsymbol{\Psi}_1$, we multiply the monomials by the corresponding polynomial coefficient in R and had these terms together. These coefficients have length λ, hence a cost $\mathcal{O}(N\, r\, d^r\, \mathsf{R}(\lambda)/\lambda + N\, r\, d^r)$.

Finally, we have to multiply this by the inverse of the Jacobian of $\boldsymbol{P}$ at $\boldsymbol{y}_0$, which is a matrix with coefficients in R of length λ. By Proposition 3.6, and since we only lift a single root, it can be done at precision N in time $\mathcal{O}(N\, \mathsf{MMR}(r, 1, \lambda)/\lambda + r^\omega)$. We conclude with the relation $\mathsf{MMR}(r, 1, \lambda) = \tilde{\mathcal{O}}(r^2 \log(\lambda)^{\mathcal{O}(1)})$. □

Once again, we compare with Newton iteration which performs at each step an evaluation of the polynomial equations and of their Jacobian matrix, and an inversion of its evaluated Jacobian matrix. This would amount to a cost $\mathcal{O}((r\,d^r + r^\omega)\,\mathsf{I}(N))$, since both the evaluations cost $\mathcal{O}(r\,d^r)$ arithmetic operations on p-adics. The latter theorem shows that we manage to save the cost of the inversion of the Jacobian matrix at full precision with on-line algorithms.

This latter advantage is meaningful when the cost of evaluation of the system is lower than the cost of linear algebra. Therefore we adapt our on-line approach to polynomials given as straight-line programs.

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

In this subsection, we assume that the algebraic system $\boldsymbol{P}$ is given as an s.l.p. We keep basically the same notations as in Section 5.1.2. Given an algebraic system $\boldsymbol{P}$, we define $\boldsymbol{T_P}(\boldsymbol{Y}) := \boldsymbol{P}(\boldsymbol{y}_0) + \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\,(\boldsymbol{Y} - \boldsymbol{y}_0)$ and $\boldsymbol{E_P}(\boldsymbol{Y}) := \boldsymbol{P}(\boldsymbol{Y}) - \boldsymbol{T_P}(\boldsymbol{Y})$. We adapt Definition 5.4 so that we may define τ and ε for multivariate polynomials.

Definition 5.12. *We define recursively $\tau_i \in R \times R_p$, $\varepsilon_i \in R_p$ for $1 \leqslant i \leqslant r$ with operations in $\Omega' := \{+, -, \cdot, p^j \times \_, \_/p^j\} \cup R \cup R^c$.*

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 $1 \leqslant j \leqslant L_i$ where L_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 5.4. 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 $(a\,b, a \times B + b \times A)$ 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 $\varepsilon_{P_i} := \varepsilon_i^{L_i}$ and $\tau_{P_i} := \tau_i^{L_i}$.

Lemma 5.13. *If $\tau_{P_i} = (a,\ A)$ then $a = P_i(\boldsymbol{y}_0)$ and $A = \mathrm{Jac}_{P_i}(\boldsymbol{y}_0)\ (\boldsymbol{Y} - \boldsymbol{y}_0) \in R_p$. Besides, $\boldsymbol{\varepsilon_P} := (\varepsilon_{P_1}, \ldots, \varepsilon_{P_r})$ is a shifted algorithm for $\boldsymbol{E_P}$ and $\boldsymbol{y}_0$ whose complexity is $3\,L^*$.*

Proof. Following the proof of Lemma 5.5, the first assertion is clear, as is the fact that $\boldsymbol{\varepsilon_P}$ is a shifted algorithm for $\boldsymbol{E_P}$ and $\boldsymbol{y}_0$. 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 $L^*(\boldsymbol{\varepsilon_P}) = 3\,L^*$. □

Theorem 5.14. *Let $\boldsymbol{P}$ 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 $\boldsymbol{y}_0 \in (R/(p))^r$ be such that $\boldsymbol{P}(\boldsymbol{y}_0) = 0 \bmod p$ and $\det(\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)) \neq 0 \bmod p$. Denote by $\boldsymbol{y}$ the unique solution of $\boldsymbol{P}$ lifted from $\boldsymbol{y}_0$. Let λ be a bound on the length of all $r \in R$ such that $r \times \_$ is an operation of $\boldsymbol{P}$, all elements $P_i(\boldsymbol{y}_0)$ in the result sequence of the evaluation of $\boldsymbol{P}$ at $\boldsymbol{y}_0$ and all entries of $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$.*

Then, the algorithm

$$\boldsymbol{\Psi}\colon \boldsymbol{Z} \mapsto \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)^{-1}(-\boldsymbol{P}(\boldsymbol{y}_0) + \mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\,\boldsymbol{y}_0 - \boldsymbol{\varepsilon}_{\boldsymbol{P}}(\boldsymbol{Z}))$$

is a shifted algorithm associated to $\boldsymbol{\Phi}$ *and* $\boldsymbol{y}_0$*. This algorithm requires a precomputation of* $\mathcal{O}(r\,L + r^2)$ *operations in* R*. Then, one can compute* $\boldsymbol{y}$ *to precision* N *in time*

$$3\,L^*\,\mathsf{R}(N) + \mathcal{O}(N\,[L\,\mathsf{R}(\lambda)/\lambda + \mathsf{MMR}(r,1,\lambda)/\lambda] + r^\omega),$$

or equivalently,

$$3\,L^*\,\mathsf{R}(N) + N\,(L + r^2 \log(r)^{\mathcal{O}(1)}) \log(\lambda)^{\mathcal{O}(1)} + \mathcal{O}(r^\omega).$$

Proof. Similarly to Proposition 5.6, $\boldsymbol{\Psi}$ is a shifted algorithm. In terms of operations in R, the evaluation of $\boldsymbol{P}(\boldsymbol{y}_0)$ and $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$ costs $\mathcal{O}(r\,L)$ operations by [BS83], and $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)\ \boldsymbol{y}_0$ requires $\mathcal{O}(r^2)$ more operations. By Lemma 5.13, the evaluation of $\boldsymbol{\varepsilon}_{\boldsymbol{P}}(\boldsymbol{y})$ cost $3\,L^*$ on-line multiplications, $\mathcal{O}(L)$ on-line additions, $\mathcal{O}(L)$ multiplications between p-adics with one operand of finite length $\mathcal{O}(\lambda)$ (coming either from operations $r \times \_$ or $\cdot$ in $\boldsymbol{P}$) and a division by $\mathrm{Jac}_{\boldsymbol{P}}(\boldsymbol{y}_0)$ for a total cost of

$$3\,L^*\,\mathsf{R}(N) + \mathcal{O}(N\,L + N\,L\,\mathsf{R}(\lambda)/\lambda + r^\omega + N\,\mathsf{MMR}(r,1,\lambda)/\lambda). \qquad \square$$

In this case, Newton iteration costs $\mathcal{O}(r\,L^* + r^\omega)\,\mathsf{I}(N) + \mathcal{O}(N\,L)$. Hence our on-line approach is particularly well-suited to systems that can be evaluated cheaply, e.g. sparse polynomial systems.

5.3 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 [G+91] 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 off-line Newton iteration [GG03, Algorithm 9.2]. 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". These variants differ only by the on-line multiplication algorithm used in Algorithm `OnlineEvaluationStep` inside Algorithm `OnlineRecursivePadic` to compute the recursive p-adics (see Section 2.2.2). The naive variant calls Algorithm `LazyMulStep` of Section 1.1.1.3, whereas the relaxed variant calls Algorithm `RelaxedProductStep` of Section 1.1.3.4. In fact, since we work on p-adic integers, the relaxed version uses an implementation of Algorithm $\texttt{Binary\_Mul\_Padic}_p$ from [BHL11, Section 3.2], which is a p-adic integer variant of Algorithm `RelaxedProductStep`.

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^{2}N" (resp. "B^{2}R") 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 5.1.1.

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

Table 5.1. Dense polynomial of degree 127

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. Notice that this remark is only valid when the required precision N is not known in advance. Otherwise, we can adapt Newton's iteration to end with precision N or $N + 1$.

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.

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.