Recursive p-adics

This chapter is based on a section of the paper *Relaxed Hensel p-adic lifting of algebraic systems* published with J. Berthomieu in the proceedings of *ISSAC'12* [BL12]. The present chapter contains additional details, proofs and examples.

One strength of relaxed algorithms is to allow the computation of recursive p-adics. The contribution of this chapter is to give a precise framework, based on our notion of shifted algorithms, to compute recursive p-adics. The *main result*, Proposition 2.17, is the building block of almost all relaxed algorithms in this thesis. Most of the following chapters are dedicated to the exploration of the consequences of this framework to further problems.

As we will see, solving a recursive equation is very similar to verifying it. Therefore, the cost of solving such an equation depends mainly on the cost of evaluating the equation.

2.1 Straight-line programs

Straight-line programs are a model of computation that consist in ordered lists of instructions without branching. We give a short presentation of this notion and refer to [BCS97] for more details. We will use this model of computation to describe and analyze the forthcoming recursive operators and shifted algorithms.

Let R be a ring and A an R-algebra. A *straight-line program* (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 operators $+, -, \cdot\colon \mathcal{D} = A^2 \to A$. We also define for $r \in R$ the 0-ary operations r^c whose output is the constant r and the unary scalar multiplication $r \times \_$ by r. We denote the set of all these operations by R^c and R. Let us fix a set of operations Ω, usually $\Omega = \{+, -, \cdot\} \cup R \cup R^c$.

An s.l.p. starts with a number ℓ of *input* parameters indexed from $-(\ell - 1)$ to 0. It has L *instructions* $\Gamma_1, \ldots, \Gamma_L$ with $\Gamma_i = (\omega_i; u_{i,1}, \ldots, u_{i,r_i})$ where $-\ell < u_{i,1}, \ldots, u_{i,r_i} < i$ and r_i is the arity of the operation $\omega_i \in \Omega$. The s.l.p. Γ is *executable* on $a = (a_0, \ldots, a_{\ell-1})$ with *result sequence* $b = (b_{-\ell+1}, \ldots, b_L) \in A^{\ell+L}$, 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 $1 \leqslant i \leqslant L$. We say that the s.l.p. Γ *computes* $b \in A$ on the entries $a_1, \ldots, a_\ell$ if Γ is executable on $a_1, \ldots, a_\ell$ over A and b is a member of the result sequence.

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 2.1. Let $R = \mathbb{Z}$, $A = \mathbb{Z}[X, Y]$ and Γ be the s.l.p. with two input parameters indexed -1, 0 and

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

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

Remark 2.2. For the sake of simplicity, we will associate a "canonical" 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. We use the binary powering algorithm to compute powers of an expression.

For example, the arithmetic expression $\varphi\colon Z \mapsto Z^4 + 1$ can be represented by the s.l.p. with one argument and instructions

$$\Gamma_1 = (\cdot; 0, 0), \quad \Gamma_2 = (\cdot; 1, 1), \quad \Gamma_3 = (1^c), \quad \Gamma_4 = (+; 2, 3).$$

2.2 Recursive p-adics

The study of on-line algorithms is motivated by its efficient implementation of recursive p-adics. To the best of our knowledge, the paper [Wat89] was the first to mention the lazy computation of power series which are solutions of a fixed point equation $y = \Phi(y)$. The paper [Hoe02], in addition to rediscovering the fast on-line multiplication algorithm of [FS74], connected for the first time this fast multiplication algorithm to the on-line computation of recursive power series. Van der Hoeven named these on-line algorithms, that use the fast on-line multiplication, *relaxed algorithms*. Article [BHL11] generalizes relaxed algorithms for p-adics.

We contribute by clarifying the setting in which recursive p-adics can be computed from their fixed point equations $y = \Phi(y)$ by an on-line algorithm. For this matter, we introduce the notion of *shifted algorithm*.

We will work with recursive p-adics in a simple case and do not need the general context of recursive p-adics [Kap01, Definition 7]. We denote by $\nu_p(a)$ the valuation in p of the p-adic a. For vectors or matrices $A \in \mathcal{M}_{r\times s}(R_p)$, we define $\nu_p(A) := \min_{i,j}(\nu_p(A_{i,j}))$. We start by giving a definition of recursive p-adics and their recursive equation that suits our needs.

Definition 2.3. *Let $\ell \in \mathbb{N}$, $\boldsymbol{\Phi} \in (R_p[Y_1, \ldots, Y_\ell])^\ell$, $\boldsymbol{y} \in (R_p)^\ell$ be a fixed point of $\boldsymbol{\Phi}$,* i.e. *$\boldsymbol{y} = \boldsymbol{\Phi}(\boldsymbol{y})$. We write $\boldsymbol{y} = \sum_{i\in\mathbb{N}} \boldsymbol{y}_i\, p^i$ the p-adic decomposition of $\boldsymbol{y}$. Let us denote $\boldsymbol{\Phi}^0 = \mathrm{Id}$ and, for all $n \in \mathbb{N}^*$, $\boldsymbol{\Phi}^n = \boldsymbol{\Phi} \circ \cdots \circ \boldsymbol{\Phi}$ (n times).*

Then, we say that the coordinates $(y_1, \ldots, y_\ell)$ of $\boldsymbol{y}$ are recursive p-adics *and that the recursive operator $\boldsymbol{\Phi}$* allows the computation *of $\boldsymbol{y}$ if, for all $n \in \mathbb{N}$, we have $\nu_p(\boldsymbol{y} - \boldsymbol{\Phi}^n(\boldsymbol{y}_0)) \geqslant n + 1$.*

The general case with more initial conditions $\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_s$ is not considered here but we believe it would to be an interesting extension of these results.

Proposition 2.4. *Let $\boldsymbol{\Phi} \in (R_p[Y_1, \ldots, Y_\ell])^\ell$ with a fixed point $\boldsymbol{y} \in R_p^\ell$ and let $\boldsymbol{y}_0 = \boldsymbol{y} \bmod p$. Suppose $\nu_p(\mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_0)) > 0$. Then $\boldsymbol{\Phi}$ allows the computation of $\boldsymbol{y}$.*

Moreover, for all $n \leqslant m \in \mathbb{N}^*$, the p-adic coefficient $(\boldsymbol{\Phi}(\boldsymbol{y}))_n$ does not depend on the coefficient $\boldsymbol{y}_m$, i.e. *$(\boldsymbol{\Phi}(\boldsymbol{y}))_n = (\boldsymbol{\Phi}(\boldsymbol{y} + \boldsymbol{a}))_n$ for any $\boldsymbol{a} \in (p^n\, R_p)^\ell$.*

Proof. We prove by induction on n that $\nu_p(\boldsymbol{y}-\boldsymbol{\Phi}^n(\boldsymbol{y}_0)) \geqslant n+1$. First, notice that $\nu_p(\boldsymbol{y}-\boldsymbol{y}_0) \geqslant 1$. Let us prove the claim for $n+1$, assuming that it is verified for n. For all $\boldsymbol{y}, \boldsymbol{z} \in R_p^\ell$, there exists, by Taylor expansion of $\boldsymbol{\Phi}$ at $\boldsymbol{z}$, vectors of p-adics $\boldsymbol{\Theta}_{i,j}(\boldsymbol{y},\boldsymbol{z}) \in R_p^\ell$ for $1 \leqslant i \leqslant j \leqslant \ell$ such that

$$\boldsymbol{\Phi}(\boldsymbol{y})-\boldsymbol{\Phi}(\boldsymbol{z}) = \mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{z})\,(\boldsymbol{y}-\boldsymbol{z}) + \sum_{1\leqslant i\leqslant j\leqslant \ell} (y_i-z_i)\,(y_j-z_j)\,\boldsymbol{\Theta}_{i,j}(\boldsymbol{y},\boldsymbol{z}).$$

For all $n \in \mathbb{N}$, we set $\boldsymbol{y}_{(n)} := \boldsymbol{\Phi}^n(\boldsymbol{y}_0)$ and we apply the previous statement to $\boldsymbol{y}$ itself and $\boldsymbol{z} = \boldsymbol{y}_{(n)}$:

$$\begin{aligned} \boldsymbol{y}-\boldsymbol{y}_{(n+1)} &= \boldsymbol{\Phi}(\boldsymbol{y})-\boldsymbol{\Phi}(\boldsymbol{y}_{(n)}) \\ &= \mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_{(n)})\,(\boldsymbol{y}-\boldsymbol{y}_{(n)}) + \sum_{1\leqslant i\leqslant j\leqslant \ell} (y_i-y_{(n),i})\,(y_j-y_{(n),j})\,\boldsymbol{\Theta}_{i,j}(\boldsymbol{y},\boldsymbol{y}_{(n)}). \end{aligned}$$

By the induction hypothesis, $\nu_p(\boldsymbol{y}-\boldsymbol{y}_{(n)}) \geqslant n+1$. Also $\nu_p(\mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_{(0)})) > 0$ implies $\nu_p(\mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_{(n)})) > 0$. As a consequence, one has $\nu_p(\boldsymbol{y}-\boldsymbol{y}_{(n+1)}) \geqslant n+2$.

For the second point, remark that if $\boldsymbol{a} \in (p^n R_p)^\ell$, then

$$\boldsymbol{\Phi}(\boldsymbol{y}+\boldsymbol{a})-\boldsymbol{\Phi}(\boldsymbol{y}) = \mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y})\,\boldsymbol{a} + \sum_{1\leqslant i\leqslant j\leqslant \ell} a_i\,a_j\,\boldsymbol{\Theta}_{i,j}\,(\boldsymbol{y}+\boldsymbol{a},\boldsymbol{y}) \in (p^{n+1}R_p)^\ell$$

since $\nu_p(\mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y})) > 0$ and $\nu_p(a_i) > 0$ because $n \in \mathbb{N}^*$. □

On-line computation of recursive p-adics Let us recall the ideas to compute $\boldsymbol{y}$ from $\boldsymbol{\Phi}(\boldsymbol{y})$ in the on-line framework. Let $\boldsymbol{\Phi}$ be given as an s.l.p. with operations in $\Omega = \{+,-,\cdot\} \cup R \cup R^c$. First, if $\boldsymbol{a} \in R_p^\ell$, we evaluate $\boldsymbol{\Phi}(\boldsymbol{a})$ in an on-line manner by performing the arithmetic operations of the s.l.p. $\boldsymbol{\Phi}$ with on-line algorithms. Let OnlineAddStep (resp. OnlineMulStep) be the step of any on-line addition (resp. multiplication) algorithm.

Algorithm `OnlineEvaluationStep`

Input: an s.l.p. $\boldsymbol{\Phi}$, $\boldsymbol{a} = (a_1,\ldots,a_\ell) \in (R_p)^\ell$, $\left[c_1^{(0)},\ldots,c_L^{(0)}\right] \in (R_p)^L$ and $i \in \mathbb{N}$
Output: $[c_1,\ldots,c_L] \in (R_p)^L$

1. $[c_{-\ell+1},\ldots,c_0] = [a_1,\ldots,a_\ell]$
2. $[c_1,\ldots,c_L] = \left[c_1^{(0)},\ldots,c_L^{(0)}\right]$
3. **for** j from 1 to L
   if $(\Gamma_j = (\text{'}+\text{'};u,v))$
   $c_j = \texttt{OnlineAddStep}(c_u, c_v, c_j, i)$
   if $(\Gamma_j = (\text{'}-\text{'};u,v))$
   $c_j = \texttt{OnlineAddStep}(c_u, -c_v, c_j, i)$
   if $(\Gamma_j = (\text{'}\cdot\text{'};u,v))$
   $c_j = \texttt{OnlineMulStep}(c_u, c_v, c_j, i)$
   if $(\Gamma_j = (r \times \_;u))$
   $c_j = \texttt{OnlineMulStep}(r, c_u, c_j, i)$
   if $(\Gamma_j = (r;))$
   $c_j = r$
4. **return** $[c_1,\ldots,c_L]$

We see that Algorithm `OnlineEvaluationStep` computes the result sequence $[c_1, ..., c_L] \in (R_p)^L$ of the s.l.p. $\mathbf{\Phi}$ on the input $\boldsymbol{a} \in (R_p)^\ell$. If one wants to evaluate $\mathbf{\Phi}$ on $\boldsymbol{a}$, it remains to loop on Algorithm `OnlineEvaluationStep`.

Algorithm `OnlineEvaluation`

Input: an s.l.p. $\mathbf{\Phi}$, $\boldsymbol{a} = (a_1, ..., a_\ell) \in (R_p)^\ell$ and $N \in \mathbb{N}$
Output: $[c_1, ..., c_L] \in (R_p)^L$

1. $[c_1, ..., c_L] = [0, ..., 0]$
2. **for** i from 0 to N
   $[c_1, ..., c_L] = \texttt{OnlineEvaluationStep}(\mathbf{\Phi}, \boldsymbol{a}, [c_1, ..., c_L], i)$
3. **return** $[c_1, ..., c_L]$

As expected, `OnlineEvaluation` is an on-line algorithm.

Proposition 2.5. *For any $N \in \mathbb{N}$, any s.l.p. $\mathbf{\Phi}$ and $\boldsymbol{a} \in (R_p)^\ell$, the output $[c_1, ..., c_L]$ of* `OnlineEvaluation`$(\mathbf{\Phi}, \boldsymbol{a}, N)$ *coincides at precision $N+1$ with the result sequence of the s.l.p. $\mathbf{\Phi}$ on the input $\boldsymbol{a}$.*

Moreover, Algorithm `OnlineEvaluation`$(\mathbf{\Phi}, \boldsymbol{a}, N)$ *is on-line with respect to its input $\boldsymbol{a}$.*

Now that we have this algorithm, we want to use the relation $\boldsymbol{y} = \mathbf{\Phi}(\boldsymbol{y})$ to compute the recursive p-adics $\boldsymbol{y}$. What we really compute is $\mathbf{\Phi}(\boldsymbol{y})$: suppose that we are at the point where we know the p-adic coefficients $\boldsymbol{y}_0, ..., \boldsymbol{y}_{N-1}$ of $\boldsymbol{y}$ and $\mathbf{\Phi}(\boldsymbol{y})$ has been computed up to its $(N-1)$st coefficient. Since in the on-line framework, the computation is done step by step, one can naturally ask for one more step of the computation of $\mathbf{\Phi}(\boldsymbol{y})$. Also, from Proposition 2.4, $(\mathbf{\Phi}(\boldsymbol{y}))_N$ depends only on $\boldsymbol{y}_0, ..., \boldsymbol{y}_{N-1}$ so that we should be able to compute it and deduce $\boldsymbol{y}_N = (\mathbf{\Phi}(\boldsymbol{y}))_N$.

We denote by $i_1, ..., i_\ell$ the indices of the outputs of the s.l.p. $\mathbf{\Phi}$.

Algorithm `OnlineRecursivePadic`

Input: an s.l.p. $\mathbf{\Phi}$, $\boldsymbol{y}_0 \in M^\ell$ and $N \in \mathbb{N}$
Output: $\boldsymbol{a} \in (R_p)^\ell$

1. $\boldsymbol{a} = \boldsymbol{y}_0$
2. $[c_1, ..., c_L] = [0, ..., 0]$
3. **for** i from 0 to N
   a. $[c_1, ..., c_L] = \texttt{OnlineEvaluationStep}(\mathbf{\Phi}, \boldsymbol{a}, [c_1, ..., c_L], i)$
   b. $\boldsymbol{a} = [c_{i_1}, ..., c_{i_\ell}]$
4. **return** $\boldsymbol{a}$

One's hope is that, with the notations of Definition 2.3, the output $\boldsymbol{a}$ of Algorithm `OnlineRecursivePadic` coincides with the recursive p-adic $\boldsymbol{y}$ at precision $N+1$. But one has to be cautious because, even if $(\mathbf{\Phi}(\boldsymbol{y}))_N$ does not depend on $\boldsymbol{y}_N$, the coefficient $\boldsymbol{y}_N$ could still be involved in anticipated computations at step N and may introduce mistakes in the following coefficients.

Here is an example of this issue that has never been raised before.

Warning 2.6. Take $R=\mathbb{Q}[X]$ and $p=X$ so that $R_p=\mathbb{Q}[[X]]$. Let Φ associated to the arithmetic expression $Y\mapsto Y^2+X$, that is the s.l.p. with one input and output and instructions

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

Let y be the only fixed point of Φ satisfying $y_0=0$, that is

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

Since $\Phi'(0)=0$, Φ allows the computation of y.

Let us specialize Algorithm `OnlineRecursivePadic` in our case. We choose to take Algorithm `LazyAddStep` for the addition and Algorithm `RelaxedProductStep` for multiplication.

Algorithm 2.1
Input: $N\in\mathbb{N}$ **Output:** $a\in R_p$
1. $a=0\,(=y_0)$
2. $c=0$
3. **for** i from 0 to N
a. $c=$`RelaxedProductStep`(c,a,a,i)
b. $a=$`LazyAddStep`(a,c,X,i)
4. **return** a

Since we already have $a_0=y_0$ before step 0, the purpose of this step is to initialize the computations. Then at the first step, we do the computations

$$c=c+2\,a_0\,a_1\,X=0,\quad a=a+(c_1+1)\,X=X.$$

So after step 1, we get $a_1=1$, which is correct, *i.e.* $a_1=y_1$. Now at step 2 we know that a_0 and a_1 are correct and we do

$$c=c+(2\,a_0\,a_2+(a_1+a_2\,X)^2)\,X^2,\quad a=a+c_2\,X^2.$$

Even if $a_2\neq y_2$, the computations produce $c=X^2$ and $a=X+X^2$ which is correct at precision 3. As predicted by Proposition 2.4, the incorrect coefficient $(a_2=0)\neq(y_2=1)$ at the beginning of step 2 did not impact the correct result $a_2=(\Phi(a))_2=1$ at the end. However the incorrect $a_2\neq y_2$ is involved in some anticipated computations of future terms.

An error appears at step 3: we do

$$c=c+(2\,a_0\,a_3)\,X^3,\quad a=a+c_3\,X^3.$$

This gives $c = X^2$ and $a = X + X^2$ which differs from the correct result $X + X^2 + 2\,X^3$ at precision 4.

Remark 2.7. We have just seen that `OnlineRecursivePadic` do *not* work for any on-line addition and multiplication algorithms. As it turns out, it does for lazy addition and multiplication algorithms, no matter the recursive operator $\mathbf{\Phi}$ as in Proposition 2.4. Indeed, lazy algorithms do at step N the computations for $(\mathbf{\Phi}(\boldsymbol{y}))_N$, and only them. So when we begin to compute $(\mathbf{\Phi}(\boldsymbol{y}))_N$, that is at step N, we know $\boldsymbol{y}_0, ..., \boldsymbol{y}_{N-1}$ and the unknown value of $\boldsymbol{y}_N$ does not change the result. Therefore, $(\mathbf{\Phi}(\boldsymbol{y}))_N$ is computed correctly for all $N \in \mathbb{N}$.

This may explain why the issue was not spotted before by papers dealing only with lazy algorithms [Wat89].

As a conclusion, even if $(\mathbf{\Phi}(\boldsymbol{y}))_N$ does not depend on the p-adic coefficient $\boldsymbol{y}_N$, the coefficient $\boldsymbol{y}_N$ can be involved in anticipated computations leading to errors later. Since we do not know $\boldsymbol{y}_N$ at step N, we must proceed otherwise. Given a recursive operator $\mathbf{\Phi} \in R_p[Y_1, ..., Y_\ell]^\ell$, we create another s.l.p. $\mathbf{\Psi}$ that computes the same polynomials $\mathbf{\Phi}(Y_1, ..., Y_\ell)$ but does not read the p-adic coefficient $\boldsymbol{y}_N$ at step N.

2.3 Shifted algorithms

Because of the issue raised in Warning 2.6, we need to make explicit the fact that $\boldsymbol{y}_N$ is not read at step N of the computation of $\mathbf{\Phi}(\boldsymbol{y})$. This issue was never mentioned in the literature before. In this section, we define the notion of shifted algorithms and prove that these algorithms compute correctly recursive p-adics by the on-line method of previous section.

We introduce for all s in $\mathbb{N}^*$ two new operators:

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

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

Algorithm `OnlineShiftStep`
Input: $a, c \in R_p$, $s \in \mathbb{Z}$ and $i \in \mathbb{N}$ **Output:** $c \in R_p$
1. $c = c + a_{i-s}\, p^i$
2. **return** c

Let Ω' be the set of operations $\{+, -, \cdot, p^s \times \_, \_/p^s\} \cup R \cup R^c$. We update the definition of Algorithm `OnlineEvaluationStep` to accept s.l.p.'s with operations in Ω'.

Algorithm `OnlineEvaluationStep`

Input: an s.l.p. $\mathbf{\Phi}$, $\boldsymbol{a} = (a_1, ..., a_\ell) \in (R_p)^\ell$, $\left[c_1^{(0)}, ..., c_L^{(0)}\right] \in (R_p)^L$ and $i \in \mathbb{N}$
Output: $[c_1, ..., c_L] \in (R_p)^L$

1. $[c_{-\ell+1}, ..., c_0] = [a_1, ..., a_\ell]$
2. $[c_1, ..., c_L] = \left[c_1^{(0)}, ..., c_L^{(0)}\right]$
3. **for** j from 1 to L
   if $(\Gamma_j = (\text{'}+\text{'}; u, v))$
   $c_j = \texttt{OnlineAddStep}(c_u, c_v, c_j, i)$
   if $(\Gamma_j = (\text{'}-\text{'}; u, v))$
   $c_j = \texttt{OnlineAddStep}(c_u, -c_v, c_j, i)$
   if $(\Gamma_j = (\text{'}\cdot\text{'}; u, v))$
   $c_j = \texttt{OnlineMulStep}(c_u, c_v, c_j, i)$
   if $(\Gamma_j = (r \times \_; u))$
   $c_j = \texttt{OnlineMulStep}(r, c_u, c_j, i)$
   if $(\Gamma_j = (r;))$
   $c_j = r$
   if $(\Gamma_j = (p^s \times \_; u))$
   $c_j = \texttt{OnlineShiftStep}(c_u, c_j, s, i)$
   if $(\Gamma_j = (\_/p^s; u))$
   $c_j = \texttt{OnlineShiftStep}(c_u, c_j, -s, i)$
4. **return** $[c_1, ..., c_L]$

In the next definition, we define a number, the *shift*, that will indicate which coefficients of an input of an s.l.p. are read at any step.

Definition 2.8. *Let us consider a Turing machine* T *with* n *inputs in* Σ^* *and one output in* Δ^**, where* Σ *and* Δ *are sets. We denote by* $\boldsymbol{a} = (a^1, ..., a^\ell)$ *an input sequence of* T *and, for all* $1 \leqslant i \leqslant \ell$*, we write* $a^i = a_0^i a_1^i ... a_n^i$ *with* $a_j^i \in \Sigma$*. We denote by* $c_0 c_1 ... c_n$ *the corresponding output, where* $c_k \in \Delta$*.*

For all input index i *with* $1 \leqslant i \leqslant \ell$*, we define the* set of shifts $\mathcal{S}(\mathrm{T}, i) \subseteq \mathbb{Z}$ *as the set of integers* $s \in \mathbb{Z}$ *such that, for all input sequences* $\boldsymbol{a}$*, the Turing machine produces* c_k *before reading* a_j^i *for* $0 \leqslant k < j + s \leqslant n$*.*

Also, we define the set of shifts $\mathcal{S}(\mathrm{T}) \subseteq \mathbb{Z}$ *by*

$$\mathcal{S}(\mathrm{T}) := \bigcap_{1 \leqslant i \leqslant \ell} \mathcal{S}(\mathrm{T}, i).$$

If $s \in \mathcal{S}(\mathrm{T})$*, we say that* T *has* shift s*.*

Algorithms do not have a unique shift: if $s \in \mathcal{S}(\mathrm{T}, i)$ then $s' \in \mathcal{S}(\mathrm{T}, i)$ for all integers $s' \leqslant s$. The definition of shift for a Turing machine is a generalization of the notion of on-line algorithms.

Corollary 2.9. *A Turing machine* T *is on-line if and only if* $0 \in \mathcal{S}(\mathrm{T})$*. Its* i*th input is an on-line argument if and only if* $0 \in \mathcal{S}(\mathrm{T}, i)$*.*

Example 2.10. Let $s \in \mathbb{Z}$ and denote by $\texttt{OnlineShift}(a, c, s, N)$ the algorithm that put $\texttt{OnlineShiftStep}(a, c, s, i)$ in a loop with i varying from 0 to $N \in \mathbb{N}$. This construction is similar to Algorithm $\texttt{Loop}_{\texttt{Algo}}$ in Chapter 1. Then Algorithm $\texttt{OnlineShift}(a, c, s, N)$ has shift s with respect to its input a.

Let us now focus on the rules to compute a shift. Let $\mathbf{\Phi}$ be a s.l.p. and N be an integer. We are interested in the shift of Algorithm $\texttt{OnlineEvaluation}(\mathbf{\Phi}, \boldsymbol{a}, N)$ with respect to its p-adic input $\boldsymbol{a}$. Let $\texttt{OnlineEvaluation}(\mathbf{\Phi}, \_, N)$ denote the partial algorithm which maps $\boldsymbol{a}$ to $\texttt{OnlineEvaluation}(\mathbf{\Phi}, \boldsymbol{a}, N)$. Recall that Algorithm $\texttt{OnlineEvaluation}(\mathbf{\Phi}, \_, N)$ merely executes the operations of the s.l.p. $\mathbf{\Phi}$ with on-line algorithms. For this reason we are able to define an integer $\mathrm{sh}(\Gamma, j, h)$ for each output index j and input index h, that will be a shift of Algorithm $\texttt{OnlineEvaluation}(\mathbf{\Phi}, \_, N)$ with respect to this input and this output.

Definition 2.11. *Let $\Gamma = (\Gamma_1, \ldots, \Gamma_L)$ be an s.l.p. over the R-algebra R_p with ℓ input parameters and operations in Ω'. For any operation index j such that $-(\ell-1) \leqslant j \leqslant L$ and for any input index h such that $-(\ell-1) \leqslant h \leqslant 0$, the* shift $\mathrm{sh}(\Gamma, j, h)$ *of its jth result b_j with respect to its hth input argument is an element of $\mathbb{Z} \cup \{+\infty\}$ defined as follows.*

If j corresponds to an input, i.e. *$j \leqslant 0$, we define for all $-(\ell-1) \leqslant h \leqslant 0$*

$$\mathrm{sh}(\Gamma, j, h) = \begin{cases} 0 & \text{if } j = h \\ +\infty & \text{if } j \neq h \end{cases}.$$

If j corresponds to an operation, i.e. *$j > 0$, then for all $-(\ell-1) \leqslant h \leqslant 0$*

- *if $\Gamma_j = (\omega_j; u, v)$ with $\omega_j \in \{+, -, \cdot\}$, then we set*

$$\mathrm{sh}(\Gamma, j, h) := \min\left(\mathrm{sh}(\Gamma, u, h), \mathrm{sh}(\Gamma, v, h)\right);$$

- *if $\Gamma_j = (r^c;)$, then $\mathrm{sh}(\Gamma, j, h) := +\infty$;*
- *if $\Gamma_j = (p^s \times \_; u)$, then $\mathrm{sh}(\Gamma, j, h) := \mathrm{sh}(\Gamma, u, h) + s$;*
- *if $\Gamma_j = (\_/p^s; u)$, then $\mathrm{sh}(\Gamma, j, h) := \mathrm{sh}(\Gamma, u, h) - s$;*
- *if $\Gamma_j = (\omega; u)$ with $\omega \in R$, then we set $\mathrm{sh}(\Gamma, j, h) := \mathrm{sh}(\Gamma, u, h)$.*

Finally if Γ has r outputs indexed by $j_1, \ldots, j_r$ in the result sequence, then we define

$$\mathrm{sh}(\Gamma) := \min\left(\{\mathrm{sh}(\Gamma, j_k, h) \mid 0 \leqslant k \leqslant r, -(\ell-1) \leqslant h \leqslant 0\}\right).$$

The following proposition proves that Algorithm $\texttt{OnlineEvaluation}(\Gamma, \_, N)$ has shift $\mathrm{sh}(\Gamma, j, h)$ with respect to its hth input and its jth output.

Proposition 2.12. *With the notations of Definition 2.11, let $\boldsymbol{y} = (y_0, \ldots, y_{\ell-1}) \in (R_p)^\ell$ be such that Γ is executable on input $\boldsymbol{y}$. Let $N \in \mathbb{N}$ and $c_1, \ldots, c_L$ be the output of $\texttt{OnlineEvaluation}(\Gamma, \boldsymbol{y}, N)$. Let $0 \leqslant h < \ell$ and $\bar{h} = h - (\ell-1)$ be the index of the input y_h in the result sequence.*

Then, the computation of $(c_j)_N$ reads at most the terms $(y_h)_i$ of the argument y_h where $0 \leqslant i \leqslant \max\left(0, N - \mathrm{sh}(\Gamma, j, \bar{h})\right)$.

Proof. By induction on the index j in the result sequence. When j corresponds to an input, *i.e.* $-(\ell-1)\leqslant j\leqslant 0$, the result c_j equals to the input $y_{j+(\ell-1)}$ so that the proposition is easily checked.

Now recursively for indices j corresponding to operations, *i.e.* $j>0$. If $\Gamma_j=(p^s\times\_;u)$, then for all $N\in\mathbb{N}$, $(c_j)_N=(p^s\,c_u)_N=(c_u)_{N-s}$ which, by assumption, reads at most the p-adic coefficients $(y_h)_i$ of the argument y_h where $0\leqslant i\leqslant\max(0,N-s-\mathrm{sh}(\Gamma,j,\bar{h}))$. So the definition matches.

If $\Gamma_j=(\cdot;u,v)$, then for all $N\in\mathbb{N}$, $(c_j)_N=(c_u\cdot c_v)_N$. Since the product $c_u\cdot c_v$ is done in Algorithm `OnlineEvaluation` by an on-line algorithm, the term $(c_j)_N$ depends only on the terms up to N of c_u and c_v, and the proposition follows.

The other cases can be treated similarly. □

Given any s.l.p. Γ, its shift index $\mathrm{sh}(\Gamma)$ can be computed automatically thanks to Definition 2.11. As an important consequence of Proposition 2.12, if an s.l.p. $\boldsymbol{\Psi}$ has a positive shift, then the computation of $(\boldsymbol{\Psi}(\boldsymbol{y}))_N$ does not read $\boldsymbol{y}_N$.

Example 2.13. We carry on with the notations of Warning 2.6. Recall that we remarked in Warning 2.6 that $(\Phi(y)_N)$ involved y_ℓ for $0\leqslant\ell\leqslant N$. We have now the tools to explain this. The shift of the s.l.p. Γ with one argument associated to the arithmetic expression $Z\mapsto Z^2+X$ (see Remark 2.2) satisfies

$$\begin{aligned}\mathrm{sh}(\Gamma) &= \min(\mathrm{sh}(Z\mapsto Z^2),\mathrm{sh}(Z\mapsto X))\\ &= \min(\min(\mathrm{sh}(Z\mapsto Z),\mathrm{sh}(Z\mapsto Z)),+\infty)\\ &= \min(\min(0,0),+\infty)\\ &= 0.\end{aligned}$$

Hence Proposition 2.12 gives that the computation of the ith term output of Φ: $Z\mapsto Z^2+X$ reads the jth term of the input with $0\leqslant j\leqslant i$, as observed in Warning 2.6.

Example 2.14. Here is a solution to the issue raised in Warning 2.6. Consider the s.l.p. deduced from the expression

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

Then $\mathrm{sh}(\Psi)=1$, since

$$\begin{aligned}\mathrm{sh}(Z\mapsto X^2\times(Z/X)^2) &= \mathrm{sh}(Z\mapsto(Z/X)^2)+2\\ &= \mathrm{sh}(Z\mapsto Z/X)+2\\ &= \mathrm{sh}(Z\mapsto Z)+1.\end{aligned}$$

So Proposition 2.12 ensures that the s.l.p. Ψ solves the problem raised in Warning 2.6.

Still, we detail the first steps of the new algorithm to convince even the most skeptical reader. Again, let us specialize Algorithm `OnlineRecursivePadic` in our case. The divisions and multiplications by X induce directly a shift in the step of the relaxed multiplication.

Algorithm 2.2

Input: $N \in \mathbb{N}$
Output: $a \in R_p$

1. $a = 0\ (= y_0)$
2. $[c_1, ..., c_3] = [0, 0, 0]$
3. **for** i from 1 to N
   a. $c_1 = a/X$
   b. $c_2 = \texttt{RelaxedProductStep}(c_2, c_1, c_1, i-2)$
   c. $c_3 = X^2 \times c_2$
   d. $a = \texttt{LazyAddStep}(a, c_3, X, i)$
4. **return** a

At Step 0 on the example, we do

$$c_1 = a/X = 0,\ c_2 = c_2,\ c_3 = X^2 \times c_2 = 0,\ a = a + (c_3)_0 + 0 = 0.$$

Then at Step 1, the following computations are done

$$\begin{array}{cc} c_1 = a/X = 0, & c_2 = c_2, \\ c_3 = X^2 \times c_2 = 0, & a = a + ((c_3)_1 + 1)\, X = X. \end{array}$$

Step 2 computes

$$\begin{array}{cc} c_1 = a/X = 1, & c_2 = c_2 + ((c_1)_0)^2 = 1, \\ c_3 = X^2 \times c_2 = X^2, & a = a + ((c_3)_2 + 0)\, X^2 = X + X^2. \end{array}$$

Step 3 computes

$$\begin{array}{cc} c_1 = a/X = 1 + X, & c_2 = c_2 + 2\,(c_1)_0\,(c_1)_1\, X = 1 + 2\,X, \\ c_3 = X^2 \times c_2 = X^2 + 2\,X^3, & a = a + (c_3)_3\, X^3 = X + X^2 + 2\,X^3. \end{array}$$

Finally Step 4 computes

$$\begin{array}{c} c_1 = a/X = 1 + X + 2\,X, \\ c_2 = c_2 + (2\,(c_1)_0\,(c_1)_2 + ((c_1)_1 + (c_1)_2\, X)^2)\, X^2 = 1 + 2\,X + 5\,X^2 + 4\,X^3 + 4\,X^4, \\ c_3 = X^2 \times c_2 = X^2 + 2\,X^3 + 5\,X^4 + 4\,X^5 + 4\,X^6, \\ a = a + (c_3)_4\, X^4 = X + X^2 + 2\,X^3 + 5\,X^4 \end{array}$$

which is still correct. If you look at Step 4 in terms of coefficients of a, we see that the shift is 1 because we do not read a_4:

$$\begin{array}{c} c_1 = a/X = a_1 + a_2\, X + \cdots, \\ c_2 = a_1^2 + 2\,a_1\,a_2\, X + (2\,a_1\,a_3 + (a_2 + a_3\, X)^2)\, X^2, \\ c_3 = a_1^2\, X^2 + 2\,a_1\,a_2\, X^3 + (2\,a_1\,a_3 + (a_2 + a_3\, X)^2)\, X^4, \\ a = a + (2\,a_1\,a_3 + a_2^2)\, X^4. \end{array}$$

We use only the coefficients a_1, a_2, a_3 at Step 4, which coincide with y_1, y_2, y_3. Therefore no error is introduced, even in the anticipated computations. In a word, we have solved the dependency issue in this example.

We are now able to express which s.l.p.'s $\mathbf{\Psi}$ are suited to the implementation of recursive p-adic numbers.

Definition 2.15. *Let $\boldsymbol{y} \in (R_p)^\ell$ be a vector of p-adics and $\mathbf{\Psi}$ be an s.l.p. with ℓ inputs, ℓ outputs and operations in Ω'.*

Then, $\mathbf{\Psi}$ is said to be a shifted algorithm *that compute $\boldsymbol{y}$ if*

- $\mathrm{sh}(\mathbf{\Psi}) \geqslant 1$,
- *$\mathbf{\Psi}$ is executable on $\boldsymbol{y}$ over the R-algebra R_p.*

A shifted algorithm is a recursive operator, but with tighter conditions.

Proposition 2.16. *If $\mathbf{\Psi}$ is a shifted algorithm that computes $\boldsymbol{y}$ then $\boldsymbol{y}$ are recursive p-adics and $\mathbf{\Psi}$ is a recursive operator that allows the computation of $\boldsymbol{y}$.*

Proof. We prove that the output of the on-line algorithm $\mathbf{\Psi}^n = \mathbf{\Psi} \circ \cdots \circ \mathbf{\Psi}$ on the input $\boldsymbol{y}_0$ coincides with $\boldsymbol{y}$ at precision $n+1$. This result is true for $n=0$. We prove it recursively on n.

Assume the claim is verified for n and let us prove it for $n+1$. If we denote by $\boldsymbol{y}_{(n)} := \mathbf{\Psi}^n(\boldsymbol{y}_0)$, we know that $\nu_p(\boldsymbol{y} - \boldsymbol{y}_{(n)}) \geqslant n+1$. Now in the steps $0, \ldots, n+1$ of the on-line computation of $\mathbf{\Psi}(\boldsymbol{y}_{(n)})$, only the p-adic coefficients of $\boldsymbol{y}_{(n)}$ in p^i are read for $i \leqslant n$ because $\mathrm{sh}(\mathbf{\Psi}) \geqslant 1$. So one has the following equalities between p-adic coefficients

$$(\boldsymbol{y}_{(n+1)})_i = (\mathbf{\Psi}(\boldsymbol{y}_{(n)}))_i = (\mathbf{\Psi}(\boldsymbol{y}))_i = \boldsymbol{y}_i$$

for $i \leqslant n$ and finally $\nu_p(\boldsymbol{y} - \boldsymbol{y}_{(n+1)}) \geqslant n+2$. □

Next proposition is the cornerstone of complexity estimates regarding recursive p-adics. We denote by $\mathsf{R}(N)$ the cost of multiplying two elements of R_p at precision N by an on-line algorithm (see Chapter 1).

Proposition 2.17. *Let $\mathbf{\Psi}$ be a shifted algorithm for recursive p-adics $\boldsymbol{y}$ whose length is L and multiplicative complexity is L^*. Then, the vector of p-adics $\boldsymbol{y}$ can be computed at precision N in time $L^*\,\mathsf{R}(N) + \mathcal{O}(L\,N)$.*

Proof. We use Algorithm `OnlineRecursivePadic` to compute $\boldsymbol{y}$. We have to prove that this algorithm is correct if $\mathbf{\Psi}$ is a shifted algorithm.

For this matter it is sufficient to prove that in the loop of Algorithm `OnlineRecursivePadic`, the correct p-adic coefficients of $\boldsymbol{y}$ are written in $\boldsymbol{a}$ before they are read by a call to `OnlineEvaluationStep`.

Since $\mathrm{sh}(\mathbf{\Psi}) \geqslant 1$, Proposition 2.12 tells us that the Nth p-adic coefficients of $\boldsymbol{a}$ are not read before step $N+1$ of `OnlineEvaluationStep`. At step 0 of the loop of Algorithm `OnlineRecursivePadic`, the p-adic coefficient $\boldsymbol{a}_0$ equals to $\boldsymbol{y}_0$. Therefore the computations of `OnlineEvaluationStep`$(\mathbf{\Psi}, \boldsymbol{a}, [c_1, \ldots, c_L], 0)$ are correct, *i.e.* they are the same than if $\boldsymbol{y}$ was given in input instead of $\boldsymbol{a}$.

At step 1, the call to `OnlineEvaluationStep`$(\mathbf{\Psi}, \boldsymbol{a}, [c_1, \ldots, c_L], 1)$ will only read $\boldsymbol{a}_0$ and carry correct computations, giving $\boldsymbol{y}_1 = (\mathbf{\Psi}(\boldsymbol{a}))_1$. At step 2, the call to `OnlineEvaluationStep`$(\mathbf{\Psi}, \boldsymbol{a}, [c_1, \ldots, c_L], 2)$ is known to read at most $\boldsymbol{a}_0, \boldsymbol{a}_1$, which coincide with $\boldsymbol{y}_0, \boldsymbol{y}_1$. So we will have $\boldsymbol{y}_2 = (\mathbf{\Psi}(\boldsymbol{a}))_2$. And so on.

The key point of our demonstration is that at each step, since the p-adic coefficients of $\boldsymbol{a}$ which are read in the call to `OnlineEvaluationStep` coincides with the ones of $\boldsymbol{y}$, Algorithm `OnlineEvaluationStep` does the same computation as if $\boldsymbol{y}$ was given in input instead of $\boldsymbol{a}$, and so computes correctly $\boldsymbol{\Psi}(\boldsymbol{y})$.

Therefore the cost of the computation of $\boldsymbol{y}$ *is exactly the cost of the evaluation* of $\boldsymbol{\Psi}(\boldsymbol{y})$ in R_p. We recall that addition in $R_p \times R_p$, subtraction in $R_p \times R_p$ and multiplication in $R \times R_p$ (that is operations in R) 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 $\mathsf{R}(N)$. Now the multiplicative complexity L^* of $\boldsymbol{\Psi}$ counts exactly the latter operation. □

Of course, if some multiplications in the evaluation of $\boldsymbol{\Psi}$ are between finite length p-adics, they cost less than $\mathsf{R}(N)$. An important special case concerns multiplications between a p-adic and another p-adic of length d, which can be done in time $\mathcal{O}(N\,\mathsf{R}(d)/d)$ instead of $\mathsf{R}(N)$.

Remark 2.18. The important property used in the proof of Proposition 2.17 is that Algorithm `OnlineEvaluation`($\boldsymbol{\Phi}$, _, N) has shift 1. We can extend the set of operations Ω' of our s.l.p.'s and adapt the rules of computation of $\mathrm{sh}(\boldsymbol{\Psi})$ consequently, Proposition 2.17 will remain correct as long as Algorithm `OnlineEvaluation`($\boldsymbol{\Phi}$, _, N) has shift 1.

Newton iteration Under the assumptions of Proposition 2.4, we can use the Newton iteration algorithm (also called Hensel lifting) to compute $\boldsymbol{y}$. Let us recall the mechanism of this lifting method.

If $\boldsymbol{f} := \mathrm{Id} - \boldsymbol{\Phi} \in R_p[Y_1, ..., Y_\ell]^\ell$, then $\boldsymbol{y}$ is a zero of the polynomials $\boldsymbol{f}$. Moreover since $\mathrm{Id} - \mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{y}_0) = \mathrm{Jac}_{\boldsymbol{\Phi}}(\boldsymbol{y}_0)$ has positive valuation, the Jacobian matrix $\mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{y})$ is invertible over R_p. Then we define recursively $\boldsymbol{y}_{(0)} = \boldsymbol{y}_0$ and for all $N \in \mathbb{N}$

$$\boldsymbol{y}_{(N+1)} = \boldsymbol{y}_{(N)} - \mathrm{Jac}_{\boldsymbol{f}}(\boldsymbol{y}_{(N)})^{-1}\, \boldsymbol{f}(\boldsymbol{y}_{(N)}) \in (R_p)^\ell.$$

It can be shown that for all $N \in \mathbb{N}$, $\nu_p(\boldsymbol{y}_{(N)} - \boldsymbol{y}) \geqslant 2^N$ [GG03].

The Newton iteration algorithm, as well as our on-line lifting algorithm for recursive p-adics, applies to more general operators than polynomial function $\boldsymbol{\Phi}$ and $\boldsymbol{f}$. For example on power series, the operator $\boldsymbol{\Phi}$ can use differentiation and integration. The notion of shift and shifted algorithms can be extended to s.l.p.'s with these new operators.

Space complexity One drawback of the relaxed method for computing recursive p-adics is the space complexity. We have seen that we store the current state of each computation of $\boldsymbol{\Psi}$ in Algorithm `OnlineRecursivePadicStep`. This leads to a space complexity $\mathcal{O}(N\,L)$ to compute the recursive p-adic at precision N where L is the size of $\boldsymbol{\Psi}$.

The zealous approach to evaluate $\boldsymbol{\Psi}$ could use significantly less memory by freeing the result of a computation as soon as it is used for the last time. For this reason, zealous lifting based on Newton iteration should consume less memory.

Partie II

Lifting of linear equations

Chapitre 3

Linear algebra over p-adics

This chapter deals with the resolution of linear systems over the p-adics. Linear algebra problems are often classified into broad categories, depending on whether the matrix of the system is dense, sparse, structured, ... In the context of solving over the p-adics, most previous algorithms rely on lifting techniques using either Dixon's / Moenck-Carter's algorithm, or Newton iteration, and can to some extent exploit the structure of the given matrix.

In this chapter, we introduce an algorithm based on the p-recursive framework of Chapter 2, which can in principle be applied to all above families of matrices. We will focus on two important cases, *dense* and *structured* matrices, and show how our algorithm can improve on existing techniques in these cases.

The relaxed linear system solver applied to dense matrices is a common work with J. Berthomieu, published as a part of [BL12]. The application to structured matrices is a joint work in progress with É. Schost.

3.1 Overview

Assumptions on the base ring Throughout this chapter, we continue using some notation and assumptions introduced in Chapter 1: R is our base ring (typically, $\mathbb{Z}$ or $\Bbbk[X]$), p is a non-zero element in R (typically, a prime in $\mathbb{Z}$ or $X \in \Bbbk[X]$) and R_p is the completion of R for the p-adic topology (so we get for instance the p-adic integers, or the power series ring $\Bbbk[[X]]$). In order to simplify some considerations below regarding the notion of rank of a matrix over a ring, we will make the following assumption in all this chapter: *both R and R_p are domains*; this is the case in the examples above.

As before, we fix a set M of representatives of $R/(p)$, which allows us to define the *length* $\lambda(a)$ of a non zero p-adic $a \in R_p$; recall that we make the assumption that the elements of $R \subset R_p$ have finite length. We generalize the length function to vectors or matrices of p-adics by setting $\lambda(A) := \max_{1 \leqslant i \leqslant r, 1 \leqslant j \leqslant s} (\lambda(A_{i,j}))$ if $A \in \mathcal{M}_{r \times s}(R_p)$.

Problem statement We consider a linear system of the form $A = B \cdot C$, where A and B are known, and C is the unknown. The matrix A belongs to $\mathcal{M}_{r\times s}(R_p)$ and $B \in \mathcal{M}_{r\times r}(R_p)$ is invertible; we solve the linear system $A = B \cdot C$ for $C \in \mathcal{M}_{r\times s}(R_p)$. We make the natural assumption that $s \leqslant r$; the most interesting cases are $s = 1$ (which amounts to linear system solving) and $s = r$, which contains in particular the problem of inverting B (our algorithm handles both cases in a uniform manner).

A major application of p-adic linear system solving is actually to solve systems over R (in the two contexts above, this means systems with integer, resp. polynomial coefficients), by means of lifting techniques (the paper [MC79] introduced this idea in the case of integer linear systems). In such cases, the solution C belongs to $\mathcal{M}_{r\times s}(Q)$, where Q is the fraction field of R, with a denominator invertible modulo p. Using p-adic techniques, we can compute the expansion of C in $\mathcal{M}_{r\times s}(R_p)$, from which C itself can be reconstructed by means of rational reconstruction — we will focus on the lifting step, and we will not detail the reconstruction step here.

In order to describe such situations quantitatively, we will use the following parameters: the length of the entries of A and B, that is, $d := \max(\lambda(A), \lambda(B))$, and the precision N to which we require C; thus, we will always be able to suppose that $d \leqslant N$. The case $N = d$ corresponds to the resolution of p-adic linear systems proper, whereas solving systems over R often requires to take a precision $N \gg d$. Indeed, in that case, we deduce from Cramer's formulas that the numerators and denominators of C have length $\mathcal{O}(r\,(d + \log(r)))$, so that we take N of order $\mathcal{O}(r\,(d + \log(r)))$ in order to make rational reconstruction possible.

For computations with structured matrices, we will use a different, non-trivial representation for B, by means of its "generators"; then, we will denote by d' the length of these generators. Details are given below.

Complexity model Throughout this chapter, we represent all p-adics through their base-M expansion, and we measure the cost of an algorithm by the number of arithmetic operations on p-adics of length 1 (*i.e.* with only a constant coefficient) it performs, as explained in Chapter 1.

The algorithms in this chapter will rely on the notion of *shifted decomposition*: a shifted decomposition of a p-adic $a \in R_p$ is simply a pair $(\sigma_a, \delta_a) \in R_p^2$ such that $a = \sigma_a + p\,\delta_a$. A simple particular case is $(a \bmod p, a \text{ quo } p)$; this is by no means the only choice. This notion carries over to matrices without difficulty.

We denote by $\mathsf{I}(N)$ the cost of multiplication of two p-adics at precision N and we let $\mathsf{R}(N)$ be the cost of multiplying two p-adics at precision N by an on-line algorithm. As in Chapter 1, we let further $\mathsf{M}(d)$ denote the arithmetic complexity of multiplication of polynomials of degree at most d over any ring (we will need this operation for the multiplication of structured matrices). Remark that when $R = \Bbbk[X]$, I and M are the same thing, but this may not be the case anymore over other rings, such as $\mathbb{Z}$.

Let next $\mathsf{I}(r, d)$ be the cost of multiplying two polynomials in $R_p[Y]$ with degree at most r and coefficients of length at most d. Since the coefficients of the product polynomial have length at most $2\,d + \lceil \log_2(r) \rceil$, we deduce that we can take

$$\mathsf{I}(r, d) = \mathcal{O}(\mathsf{M}(r)\ \mathsf{I}(d + \log(r)))$$

by working modulo p to the power the required precision; over $R_p = \Bbbk[[X]]$, the $\log(r)$ term vanishes since no carry occurs.

Let us focus on the corresponding on-line algorithm. We consider these polynomials as p-adics of polynomials, *i.e.* p-adic whose coefficients are polynomials in M. We denote by $\mathsf{R}(r, N)$ the cost of an on-line multiplication at precision N of polynomials of degrees at most r. As in Chapter 1, this cost is bounded by $\mathsf{R}(r, N) = \mathcal{O}(\mathsf{I}(r, N)\log(N))$ in the case of power series rings or p-adic integers. If the length d' of the coefficients of one operand is less than N, the cost reduces to $\mathcal{O}(N\,\mathsf{R}(r, d')/d')$.

Now, let us turn to matrix arithmetic. We let ω be such that we can multiply $r \times r$ matrices within $\mathcal{O}(r^\omega)$ ring operations over any ring. The best known bound on ω is $\omega \leqslant 2.3727$ [CW90, Sto10, VW11]. It is known that, if the base ring is a field, we can invert any invertible matrix in time $\mathcal{O}(r^\omega)$ base field operations. We will further denote by $\mathsf{MM}(r, s, d)$ the cost of multiplication of matrices A, B of sizes $(r \times r)$ by $(r \times s)$ over R_p, for inputs of length at most d. In our case $s \leqslant r$, and taking into account the growth of the length in the output, we obtain that $\mathsf{MM}(r, s, d)$ satisfies

$$\mathsf{MM}(r, s, d) = \mathcal{O}(r^2\, s^{\omega-2}\, \mathsf{I}(d + \log(r))),$$

since $\lambda(A \cdot B) \leqslant 2\,d + \lceil \log_2(r) \rceil$; the exponents on r and s are obtained by partitioning A and B into square blocks of size s.

Let us now consider the relaxed product of p-adic matrices, *i.e.* p-adic whose coefficients are matrices over M. We denote by $\mathsf{MMR}(r,\ s,\ N)$ the cost of the relaxed multiplication of a p-adic matrix of size $r \times r$ by a p-adic matrix of size $r \times s$ at precision N. As in Chapter 1, we can connect the cost of off-line and on-line multiplication algorithms by

$$\mathsf{MMR}(r, s, N) = \mathcal{O}(\mathsf{MM}(r, s, N)\log(N))$$

in the case of power series rings or p-adic integers. Likewise, we also notice that the relaxed multiplication of two matrices $A, B \in (\mathcal{M}_{r\times s}(R))_{(p)}$ at precision N with $d := \lambda(A) \leqslant N$ takes time $\mathcal{O}(N\,\mathsf{MMR}(r, s, d)/d)$.

Previous work The first algorithm we will mention is due to Dixon [Dix82]; it finds one p-adic coefficient of the solution C at a time and then updates the matrix A. On the other side of the spectrum, one finds Newton's iteration, which doubles the precision of the solution at each step (and can thus benefit from fast p-adic multiplication); however, this algorithm computes the whole inverse of B at precision N, which can be too costly when we only want one vector solution.

Moenck-Carter's algorithm [MC79] is a variant of Dixon's algorithm that works with p^ℓ-adics instead of p-adics. It takes advantages of fast truncated p-adic multiplication but requires that we compute the inverse of B at precision d (for which Newton iteration is used).

Finally, Storjohann's high-order lifting algorithm [Sto03] can be seen as a fast version of Moenck-Carter's algorithm, well-suited to cases where $d \ll N$. That algorithm was presented for $R = \Bbbk[X]$ and the result was extended to the integer case in [Sto05]. We believe that the result could carry over to any p-adic ring.

Historically, these algorithms were all introduced for dense matrices; however, most of them can be adapted to work with structured matrices. The exception is Storjohann's high-order lifting, which does not seem to carry over in a straightforward manner.

Main results The core of this chapter is an algorithm to solve linear systems by means of relaxed techniques; it is obtained by proving that the entries of the solution $C = B^{-1} \cdot A$ are p-recursive. In other words, we show that C is a fixed point for a suitable shifted operator.

This principle can be put to use for several families of matrices; we detail it for dense and structured matrices. Taking for instance $s = 1$, to compute C at precision N, the cost of the resulting algorithm will (roughly speaking) involve the following:

- the inversion of B modulo (p),
- $\mathcal{O}(N)$ matrix-vector products using the inverse of B modulo (p), with a right-hand side vector whose entries have length 1,
- $\mathcal{O}(1)$ matrix-vector product using B, with a right-hand side vector whose entries are relaxed p-adics.

Tables 3.1 and 3.2 give the resulting running time for the case of dense matrices, together with the results based on previous algorithms mentioned above; recall that $d = \lambda(B)$ and that N is the target precision. In the first table, we are in the general case $1 \leqslant s \leqslant r$; in the second one, we take $R = \Bbbk[X]$ and $s = 1$, and we choose two practically meaningful values for N, respectively $N = d$ and $N = r\,d$ (which was mentioned above). For the high-order lifting, the * indicates that the result is formally proved only for $R_p = \Bbbk[[X]]$ and $R = \mathbb{Z}$. The complexity $\mathsf{MM}(r, s\,N/d, 1)$ that appears in this case is bounded by $\mathsf{MM}(r, s\,N/d, 1) = r^{\omega-1}\,s\,N/d$.

Most previous complexity results are present in the literature, so we will not reprove them all; we only do it in cases where small difficulties may arise. For instance, Newton's algorithm and its cost analysis extend in a straightforward manner, since we only do computations modulo powers of p, which behave over general p-adics as they do over e.g. $R_p = \Bbbk[[X]]$; thus, we will not reprove the running time in this case. On the other hand, we will re-derive the cost of Dixon's and Moenck-Carter's algorithms, since they involve computations in R_p itself (*i.e.*, without reduction modulo a power of p), and considerations about the lengths of the operands play a role.

In most entries (especially in the first table), two components appear: the first one involves inverting the matrix B modulo (p), or a higher power of p and is independent of N; the second one describes the lifting process itself. In some cases, the cost of the first step can be neglected compared to the cost of the second one.

It appears in the last table that for solving up to precision $N = d$, our algorithm is the fastest among the ones we compare; for $N = r\,d$, Storjohann's high-order lifting does best (as it is specially designed for such large precisions).