JOURNAL
de Théorie des Nombres
de BORDEAUX
anciennement Séminaire de Théorie des Nombres de Bordeaux

Jérémy BERTHOMIEU, Joris VAN DER HOEVEN et Grégoire LECERF

Relaxed algorithms for p-adic numbers

Tome 23, n° 3 (2011), p. 541-577.

<http://jtnb.cedram.org/item?id=JTNB_2011__23_3_541_0>

Journal de Théorie des Nombres
de Bordeaux **23** (2011), 541-577

Relaxed algorithms for p-adic numbers

par Jérémy BERTHOMIEU, Joris VAN DER HOEVEN
et Grégoire LECERF

Résumé. Les implantations actuelles des nombres p-adiques reposent essentiellement sur des techniques dites *zélées*, où la précision de calcul doit être fixée à l'avance par l'utilisateur. Cette approche est très efficace du point de vue de la complexité asymptotique et elle est largement utilisée, par exemple dans des algorithmes de remontée de type Newton–Hensel.

Dans le contexte similaire des séries formelles, il existe des techniques alternatives, appelées *paresseuses*, qui ont l'avantage d'être plus naturelles : une série formelle y est représentée comme une suite de coefficients munie d'une méthode pour calculer le coefficient suivant et ce, à tout ordre. Cette approche facilite grandement la résolution d'équations implicites et retire tout soucis de choix de la précision des calculs à l'utilisateur. Pendant longtemps cette approche paresseuse était pénalisée par son manque d'efficacité. Les premières variantes rapides ont été développées dans les années 90 par van der Hoeven, et portent désormais la terminologie d'algorithmes *détendus* : ceux-ci combinent le confort de l'approche paresseuse avec l'efficacité des méthodes zélées.

Dans ce papier, nous montrons comment adapter l'algorithmique détendue des séries au cas des nombres p-adiques. Nos implantations sont disponibles dans la bibliothèque C++ algebramix de Mathemagix. Comparés à une itération de Newton classique, nous obtenons des gains de performances significatifs pour la résolution de certaines équations fonctionnelles p-adiques.

Abstract. Current implementations of p-adic numbers usually rely on so called *zealous* algorithms, which compute with truncated p-adic expansions at a precision that can be specified by the user. In combination with Newton-Hensel type lifting techniques, zealous algorithms can be made very efficient from an asymptotic point of view.

Manuscrit reçu le 26 mai 2010.
This work has been partly supported by the French ANR-09-JCJC-0098-01 MaGiX project, and by the Digiteo 2009-36HD grant of the Région Ile-de-France.
Mots clefs. p-adic numbers, power series, algorithms.
Classification math. 68W30, 11Y40, 11Y16.

In the similar context of formal power series, another so called *lazy* technique is also frequently implemented. In this context, a power series is essentially a stream of coefficients, with an effective promise to obtain the next coefficient at every stage. This technique makes it easier to solve implicit equations and also removes the burden of determining appropriate precisions from the user. Unfortunately, naive lazy algorithms are not competitive from the asymptotic complexity point of view. For this reason, a new *relaxed* approach was proposed by van der Hoeven in the 90's, which combines the advantages of the lazy approach with the asymptotic efficiency of the zealous approach.

In this paper, we show how to adapt the lazy and relaxed approaches to the context of p-adic numbers. We report on our implementation in the C++ library ALGEBRAMIX of MATHEMAGIX, and show significant speedups in the resolution of p-adic functional equations when compared to the classical Newton iteration.

1. Introduction

Let R be an *effective commutative ring*, which means that algorithms are available for all the ring operations. Let (p) be a proper principal ideal of R. Any element a of the completion R_p of R for the p-adic valuation can be written, in a non unique way, as a power series $\sum_{i\geqslant 0} a_i p^i$ with coefficients in R. For example, the completion of $\mathbb{K}[x]$ for the ideal (x) is the classical ring of power series $\mathbb{K}[[x]]$, and the completion of $\mathbb{Z}$ for any prime integer p is the ring of *p-adic integers* written $\mathbb{Z}_p$.

In general, elements in R_p give rise to infinite sequences of coefficients, which cannot be directly stored in a computer. Nevertheless, we can compute with finite but arbitrarily long expansions of p-adic numbers. In the so called *zealous* approach, the precision n of the computations must be known in advance, and fast arithmetic can be used for computations in $R/(p^n)$. In the *lazy* framework, p-adic numbers are really promises, which take a precision n on input, and provide an n-th order expansion on output.

In [Hoe97] appeared the idea that the lazy model actually allows for asymptotically fast algorithms as well. Subsequently [Hoe02], this compromise between the zealous and the naive lazy approaches has been called the *relaxed* model. The main aim of this paper is the design of *relaxed* algorithms for computing in the completion R_p. We will show that the known complexity results for power series extend to this setting. For more details on the power series setting, we refer the reader to the introduction of [Hoe02].

1.1. Motivation. Completion and deformation techniques come up in many areas of symbolic and analytic computations: polynomial factorization, polynomial or differential system solving, analytic continuation, etc. They make an intensive use of power series and p-adic integers.

1.1.1. *Recursive equations.* The major motivation for the relaxed approach is the resolution of algebraic or functional equations. Most of the time, such equations can be rewritten in the form

$$Y = \Phi(Y), \tag{1.1}$$

where the indeterminate Y is a vector in R_p^d and Φ some algebraic or more complicated expression with the special property that

$$\tilde{y} - y \in p^n R_p^d \implies \Phi(\tilde{y}) - \Phi(y) \in p^{n+1} R_p^d,$$

for all $y, \tilde{y} \in R_p^d$ and $n \in \mathbb{N}$. In that case, the sequence $0, \Phi(0), \Phi^2(0), \ldots$ converges to a solution $y \in R_p^d$ of (1.1), and we call (1.1) a recursive equation.

Using zealous techniques, the resolution of recursive equations can often be done using a Newton iteration, which doubles the precision at every step [BK78]. Although this leads to good asymptotic time complexities in n, such Newton iterations require the computation and inversion of the Jacobian of Φ, leading to a non trivial dependence of the asymptotic complexity on the size of Φ as an expression. For instance, at higher dimensions d, the inversion of the Jacobian usually involves a factor $O(d^3)$, whereas Φ may be of size $O(d)$. We shall report on such examples in Section 5.

The main *rationale* behind relaxed algorithms is that the resolution of recursive equations just corresponds to a relaxed evaluation of Φ at the solution itself. In particular, the asymptotic time complexity to compute a solution has a linear dependence on the size of Φ. Of course, the technique does require relaxed implementations for all operations involved in the expression Φ. The essential requirement for a *relaxed operation* φ is that $\varphi(y)_n$ should be available as soon as $y_0, \ldots, y_n$ are known.

1.1.2. *Elementary operations.* A typical implementation of the relaxed approach consists of a library of basic relaxed operations and a function to solve arbitrary recursive equations built up from these operations. The basic operations typically consist of linear operations (such as addition, shifting, derivation, etc.), multiplication and composition. Other elementary operations (such as division, square roots, higher order roots, exponentiation) are easily implemented by solving recursive equations. In several cases, the relaxed approach is not only elegant, but also gives rise to more efficient algorithms for basic operations.

Multiplication is the key operation and Sections 2, 3 and 4 are devoted to it. In situations were relaxed multiplication is as efficient as naive multiplication (e.g. in the naive and Karatsuba models), the relaxed strategy is optimal in the sense that solving a recursive equation is as efficient as verifying the validity of the solution. In the worst case, as we will see in Proposition 3.1, relaxed multiplication is $O(\log n)$ times more expensive than zealous multiplication modulo p^n. If $\mathbb{F}_p$ contains many 2^p-th roots of unity, then this overhead can be further reduced to $O(\mathrm{e}^{2\sqrt{\log 2 \log\log n}})$ using similar techniques as in [Hoe07b]. In practice, the overhead of relaxed multiplication behaves as a small constant, even though the most efficient algorithms are hard to implement.

In the zealous approach, the division and the square root usually rely on Newton iteration. In small and medium precisions the cost of this iteration turns out to be higher than a direct call to one relaxed multiplication or squaring. This will be illustrated in Section 6: if p is sufficiently large, then our relaxed division outperforms zealous division.

1.1.3. ***User-friendly interface.*** An important advantage of the relaxed approach is its user-friendliness. Indeed, the relaxed approach automatically takes care of the precision control during all intermediate computations. A central example is the Hensel lifting algorithm used in the factorization of polynomials in $\mathbb{Q}[x]$: one first chooses a suitable prime number p, then computes the factorization in $\mathbb{Z}/p\mathbb{Z}[x]$, lifts this factorization into $\mathbb{Q}_p[x]$, and finally one needs to discover how these p-adic factors recombine into the ones over $\mathbb{Q}$ (for details see for instance [GG03, Chapter 15]). Theoretically speaking, Mignotte's bound [GG03, Chapter 6] provides us with the maximum size of the coefficients of the irreducible factors, which yields a bound on the precision needed in $\mathbb{Q}_p$. Although this bound is sharp in the worst case, it is pessimistic in several particular situations. For instance, if the polynomial is made of small factors, then the factorization can usually be discovered at a small precision. Here the relaxed approach offers a convenient and efficient way to implement adaptive strategies. In fact we have already implemented the polynomial factorization in the relaxed model with success, as we intend to show in detail in a forthcoming paper.

1.2. Our contributions. The relaxed computational model was first introduced in [Hoe97] for formal power series, and further improved in [Hoe02, Hoe07b]. In this article, we extend the model to more general completions R_p. Although our algorithms will be represented for arbitrary rings R, we will mainly focus on the case $R = \mathbb{Z}$ when studying their complexities. In Section 2 we first present the relaxed model, and illustrate it on a few easy algorithms: addition, subtraction, and naive multiplications.

In Section 3, we adapt the relaxed product of [Hoe02, Section 4] to p-adic numbers. We first present a direct generalization, which relies on products of finite p-expansions. Such products can be implemented in various ways but essentially boil down to multiplying polynomials over R. We next focus on the case $R = \mathbb{Z}$ and how to take advantage of fast hardware arithmetic on small integers, or efficient libraries for computations with multiple precision integers, such as GMP [G+91]. In order to benefit from this kind of fast binary arithmetic, we describe a variant that internally performs conversion between p-adic and 2-adic numbers in an efficient way. We will show that the performance of p-adic arithmetic is similar to power series arithmetic over $R/(p)$.

For large precisions, such conversions between p-adic and 2-adic expansions involve an important overhead. In Section 4 we present yet another blockwise relaxed multiplication algorithm, based on the fact that $R_p \cong R_{p^k}$ for all $k \geqslant 1$. This variant even outperforms power series arithmetic over $R/(p)$. For large block sizes k, the performance actually gets very close to the performance of zealous multiplication.

In Section 5, we recall how to use the relaxed approach for the resolution of recursive equations. For small dimensions d, it turns out that the relaxed approach is already competitive with more classical algorithm based on Newton iteration. For larger numbers of variables, we observe important speed-ups.

Section 6 is devoted to division. For power series, relaxed division essentially reduces to one relaxed product [Hoe02, Section 3.2.2]. We propose an extension of this result to p-adic numbers. For medium precisions, our algorithm turns out to be competitive with Newton's method.

In Section 7, we focus on the extraction of r-th roots. We cover the case of power series in small characteristic, and all the situations within $\mathbb{Z}_p$. Common transcendental operations such as exponentiation and logarithm are more problematic in the p-adic setting than in the power series case, since the formal derivation of p-adic numbers has no nice algebraic properties. In this respect, p-adic numbers rather behave like floating point numbers. Nevertheless, it is likely that holonomic functions can still be evaluated fast in the relaxed setting, following [Bre76, CC90, Hoe99, Hoe01, Hoe07a]. We also refer to [Kob84, Kat07] for more results about exponentiation and logarithms in $\mathbb{Q}_p$.

Algorithms for p-adic numbers have been implemented in several libraries and computer algebra systems: P-PACK [Wan84], MAPLE, MAGMA, PARI/GP [PAR08], MATHEMATICA [DS04], SAGE [S+09], FLINT [HH09], etc. These implementations all use the zealous approach and mainly provide fixed-precision algorithms for $R = \mathbb{Z}$. Only SAGE also proposes a lazy

interface. However, this interface is not relaxed and therefore inefficient for large precisions.

Most of the algorithms presented in this paper have been implemented in the C++ open source library algebramix of Mathemagix [H+02] (revision 5342, freely available from `http://www.mathemagix.org`). Although we only report on timings for p-adic integers, our code provides support for general effective Euclidean domains R.

2. Data structures and naive implementations

In this section we present the data structures specific to the relaxed approach, and the naive implementations of the ring operations in R_p.

2.1. Finite p-adic expansions. As stated in the introduction, any element a of the completion R_p of R for the p-adic valuation can be written, in a non unique way, as a power series $\sum_{i\geqslant 0} a_i p^i$ with coefficients in R. Now assume that M is a subset of R, such that the restriction of the projection map $\pi : R \to R/(p)$ to M is a bijection between M and $R/(p)$. Then each element a admits a unique power series expansion $\sum_{i\geqslant 0} a_i p^i$ with $a_i \in M$. In the case when $R = \mathbb{Z}$ and $p \in \mathbb{N} \setminus \{0, 1\}$, we will always take $M = \{0, \ldots, p-1\}$.

For our algorithmic purposes, we assume that we are given quotient and remainder functions by p

$$\begin{aligned} \mathsf{quo}\,(\cdot, p) : R &\to R \\ \mathsf{rem}\,(\cdot, p) : R &\to M, \end{aligned}$$

so that we have

$$a = \mathsf{quo}\,(a, p)p + \mathsf{rem}\,(a, p),$$

for all $a \in R$.

Polynomials $\sum_{i=0}^{n-1} a_i p^i \in M[p]$ will also be called *finite p-adic expansions* at order n. In fact, finite p-adic expansions can be represented in two ways. On the one hand, they correspond to unique elements in R, so we may simply represent them by elements of R. However, this representation does not give us direct access to the coefficients a_i. By default, we will therefore represent finite p-adic expansions by polynomials in $M[p]$. Of course, polynomial arithmetic in $M[p]$ is not completely standard due to the presence of carries.

2.2. Classical complexities. In order to analyze the costs of our algorithms, we denote by $\mathsf{M}(n)$ the cost for multiplying two univariate polynomials of degree n over an arbitrary ring A with unity, in terms of the number of arithmetic operations in A. Similarly, we denote by $\mathsf{I}(n)$ the time needed to multiply two integers of bit-size at most n in the classical *binary representation*. It is classical [SS71, CK91, Für07] that $\mathsf{M}(n) = O(n \log n \log\log n)$

and $\mathsf{I}(n) = O(n \log n 2^{\log^* n})$, where $\log^*$ represents the iterated logarithm of n. Throughout the paper, we will assume that $\mathsf{M}(n)/n$ and $\mathsf{I}(n)/n$ are increasing. We also assume that $\mathsf{M}(O(n)) = O(\mathsf{M}(n))$ and $\mathsf{I}(O(n)) = O(\mathsf{I}(n))$.

In addition to the above complexities, which are classical, it is natural to introduce $\mathsf{I}_p(n)$ as the time needed to multiply two p-adic expansions in $\mathbb{Z}_p$ at order n with coefficients in the usual binary representation. When using Kronecker substitution for multiplying two finite p-adic expansions, we have $\mathsf{I}_p(n) = \mathsf{I}(n(\log p + \log n))$ [GG03, Corollary 8.27]. We will assume that $\mathsf{I}_p(n)/n$ is increasing and that $\mathsf{I}_p(O(n)) = O(\mathsf{I}_p(n))$.

It is classical that the above operations can all be performed using linear space. Throughout this paper, we will make this assumption.

2.3. The relaxed computational model. For the description of our relaxed algorithms, we will follow [Hoe02] and use a C++-style pseudo-code, which is very similar to the actual C++ implementation in MATHEMAGIX. As in [Hoe02], we will not discuss memory management related issues, which have to be treated with care in real implementations, especially when it comes to recursive expansions (see Section 5 below).

The main class Padic_p for p-adic numbers really consists of a pointer (with reference counting) to the corresponding abstract representation class $\mathsf{Padic\_rep}_p$. On the one hand, this representation class contains the computed coefficients $\varphi : M[p]$ of the number a up till a given order $n : \mathbb{N}$ (let us mention here that φ can eventually be used to store anticipated data, as in the algorithms of Section 3). On the other hand, it contains a "purely virtual method" next, which returns the next coefficient a_n:

```
class Padic_rep_p
    φ: M[p]
    n: ℕ
    virtual next ()
```

Following C++-terminology, the purely virtual function next is only defined in a concrete representation class which derives from $\mathsf{Padic\_rep}_p$. For instance, to construct a p-adic number from an element in M, we introduce the type $\mathsf{Constant\_Padic\_rep}_p$ that inherits from $\mathsf{Padic\_rep}_p$ (inheritance is represented by the symbol $\trianglerighteq$) in this way:

```
class Constant_Padic_rep_p ⊵ Padic_rep_p
    c: M
    constructor (c̃: M)
        c := c̃
    method next ()
        if n = 0 then return c else return 0
```

In this piece of code n represents the current precision inherited from $\mathsf{Padic\_rep}_p$. The user visible constructor is given by

$$\mathsf{padic}\ (c\colon M) \to \mathsf{Padic}_p := (\mathsf{Padic}_p)\ \mathsf{new}\ \mathsf{Constant\_Padic\_rep}_p\ (c).$$

This constructor creates a new object of type $\mathsf{Constant\_Padic\_rep}_p$ to represent c, after which its address can be casted to the type Padic_p of p-adic numbers. From now on, for the sake of conciseness, we no longer describe such essentially trivial user level functions anymore, but only the concrete representation classes.

It is convenient to define one more public top-level function for the extraction of the coefficient a_k, given an instance a of Padic_p and a positive integer $k : \mathbb{N}$. This function first checks whether k is smaller than the order $a.n$ of a. If so, then $a_k = (a.\varphi)_k$ is already available. Otherwise, we keep increasing $a.n$ while calling next, until a_k will eventually be computed. For more details, we refer to [Hoe02, Section 2]. We will now illustrate our computational model on the basic operations of addition and subtraction.

2.4. Addition. The representation class for sums of p-adic numbers, written $\mathsf{Sum\_Padic\_rep}_p$, is implemented as follows:

```
class Sum_Padic_rep_p ⊵ Padic_rep_p
    a, b: Padic_p
    γ: R
    constructor (ã: Padic_p, b̃: Padic_p)
        a := ã;  b := b̃;  γ := 0
    method next ()
        t := a_n + b_n + γ
        γ := quo (t, p)
        return rem (t, p)
```

In the case when $R = \mathbb{Z}$, we notice by induction over n that we have $\gamma \in \{0, 1\}$, each time that we enter next, since $0 \leqslant a_n + b_n + \gamma \leqslant 2p - 1$. In that case, it is actually more efficient to avoid the calls to quo and rem and replace the method next by

```
method next ()
    t := a_n + b_n + γ
    if t < p then
        γ := 0
        return t
    else
        γ := 1
        return t − p
```

Proposition 2.1. *Given two relaxed p-adic integers a and b, the sum $a+b$ can be computed up till precision n using $O(n \log p)$ bit-operations.*

Proof. Each of the additions $a_k + b_k + \gamma$ and subsequent reductions modulo p take $O(\log p)$ bit-operations. □

2.5. Subtraction in $\mathbb{Z}_p$. In general, the subtraction is the same as the addition, but for the special case when $R = \mathbb{Z}$, we may use the classical school book method. In our framework, this yields the following implementation:

```
class Sub_Padic_rep_p ⊵ Padic_p
    a, b: Padic_p
    γ: R
    constructor (ã: Padic_p, b̃: Padic_p)
        a := ã;  b := b̃;  γ := 0
    method next ()
        t := a_n − b_n − γ
        if t ⩾ 0 then
            γ := 0
            return t
        else
            γ := 1
            return t + p
```

Proposition 2.2. *Given two relaxed p-adic integers a and b, the difference $a - b$ can be computed up till precision n using $O(n \log p)$ bit-operations.*

Proof. Each call to the function next costs $O(\log p)$ bit-operations. □

2.6. Naive product. Here we consider the school book algorithm: each coefficient $(ab)_n$ is obtained from the sum of all products of the form $a_k b_{n-k}$ plus the carry involved by the products of the preceding terms. Carries are larger than for addition, so we have to take them into account carefully. The naive method is implemented in the following way:

```
class Naive_Mul_Padic_rep_p ⊵ Padic_rep_p
    a, b: Padic_p
    γ: a vector with entries in R, with indices starting at 0.
    constructor (ã: Padic_p, b̃: Padic_p)
        a := ã; b := b̃;
        Initialize γ with the empty vector
    method next ()
        Append γ_n = 0 at the end of γ
        t := 0
        for i from 0 to n do
            s := a_i b_{n−i} + γ_i
            t := t + s
            γ_i := quo (t, p)
            t := rem (t, p)
        return t
```

Note. Let us precise that, in the pseudo-code, the **for** loop means that i runs over all the integral values from 0 to n included.

Proposition 2.3. *Given two relaxed p-adic integers a and b, the product ab can be computed up till precision n using $O(n^2\mathsf{I}(\log p))$ bit-operations.*

Proof. We show by induction that, when entering in next to compute the coefficient $(ab)_n$, the vector γ has size n and entries in M. This clearly holds for $n = 0$. Assume that the hypothesis is satisfied until a certain value $n \geqslant 0$. When entering next the size of γ is increased by 1, so that it will be $n+1$ at the end. Then, at step $i \in \{0, \ldots, n\}$ of the loop we have $s \leqslant (p-1)^2 + p - 1 = p^2 - p$. Since $t \leqslant p-1$ it follows that $s + t \leqslant p^2 - 1$, whence $\gamma_i \leqslant p - 1$ on exit. Each of the $O(n^2)$ steps within the loop takes $O(\mathsf{I}(\log p))$ bit-operations, which concludes the proof. □

Since hardware divisions are more expensive than multiplications, performing one division at each step of the above loop turns out to be inefficient in practice. Especially when working with hardware integers, it is therefore recommended to accumulate as many terms $a_i b_{n-i}$ as possible in s before a division. For instance, if p fits 30 bits and if we use 64 bits hardware integers then we can do a division every 16 terms.

2.7. Lifting the power series product. In this subsection we assume that we are given an implementation of relaxed power series over R, as described in [Hoe02, Hoe07b]. The representation class is written $\mathsf{Series\_rep}_R$ and the user level class is denoted by Series_R, in the same way as for p-adic numbers. Another way to multiply p-adic numbers relies on the relaxed product in $R[[p]]$. This mainly requires a lifting algorithm of $M[[p]]$ into $R[[p]]$ and a projection algorithm of $R[[p]]$ onto $M[[p]]$, The lifting procedure is trivial:

class $\mathsf{Lift\_Series\_rep}_R \trianglerighteq \mathsf{Series\_rep}_R$
  a: Padic_p
  constructor ($\tilde{a}$: Padic_p)
    $a := \tilde{a}$
  method next ()
    return a_n

Let lift denote the resulting function that converts a p-adic number a: Padic_p into a series in Series_R. The reverse operation, project, is implemented as follows:

class $\mathsf{Project\_Padic\_rep}_p \trianglerighteq \mathsf{Padic\_rep}_p$
  f: Series_R
  γ: R
  constructor ($\tilde{f}$: Series_R)
    $f := \tilde{f}$; $\gamma := 0$

n	8	16	32	64	128	256	512	1 024
Naive_Mul_Padic$_p$	2.9	3.8	8.3	22	68	250	920	3 600
Maple 14	240	320	520	1 200	3 500	11 000	38 000	160 000
Pari/Gp	0.52	1.0	2.7	8.4	28	99	360	1 300

Table 2.1. Naive products, for $p = 536870923$, in microseconds.

method next ()
  $t := f_n + \gamma$
  $\gamma :=$ quo (t, p)
  return rem (t, p)

Finally the product $c = ab$ is obtained as project (lift(a)lift(b)).

Proposition 2.4. *Given relaxed p-adic integers a and b, the product ab can be computed up till precision n using* $O(\mathsf{I}(\log p + \log n)\mathsf{M}(n)\log n)$ *or* $O(\mathsf{I}(n(\log p + \log n))\log n)$ *bit-operations.*

Proof. The relaxed product of two power series in size n can be done with $O(\mathsf{M}(n)\log n)$ operations in R by [Hoe02, Section 4.3.2, Theorem 6]. In our situation the size of the integers in the product are in $O(\log p + \log n)$. Then, by induction, one can easily verify that the size of the carry γ does not exceed $O(\log p + \log n)$ during the final projection step. We are done with the first bound.

The second bound is a consequence of the classical Kronecker substitution: we can multiply two polynomials in $\mathbb{Z}[x]$ of size n and coefficients of bit-size $O(\log p)$ with $O(\mathsf{I}(n(\log p + \log n)))$ bit operations [GG03, Corollary 8.27]. □

This strategy applies in full generality and gives a "softly optimal algorithm". It immediately benefits from any improvements in the power series product. Nevertheless, when n is not much larger than p, practical implementations of this method involve a large constant overhead. In the next sections, we will therefore turn our attention to "native" counterparts of the relaxed power series products from [Hoe02, Hoe07b].

2.8. Timings. We conclude this section with some timings in Table 2.1 for our C++ implementation of naive multiplication inside the algebramix package of the Mathemagix system [H+02]. Timings are measured using one core of an Intel Xeon X5450 at 3.0 GHz running Linux and Gmp 5.0.0 [G+91]. As a comparison, we display timings on the same platform, obtained for the Maple 14 package Padic. Precisely, we created two random numbers to precision n, did their product via the function `evalp`, and then asked for the coefficient of order $n/2$. Notice that timings for small precisions are not very relevant for Maple because of the overhead due to the interpreter. As another comparison, we report on the performances of

Pari/Gp version 2.3.5. For all these three cases we observe that asymptotically fast algorithms are not used. In fact Pari/Gp carefully implements the zealous strategy on the binary representation modulo p^n: as expected, such timings are better than ours, but no so much neither. We shall come back in Section 4 on this critical issue of taking better advantage of native binary representations within the lazy model.

3. Relaxed product

In this section, we extend the relaxed product of [Hoe02, Section 4.3.1] to more general p-adic numbers. We also present a special version for $R = \mathbb{Z}$, which uses internal base conversions between base 2 and base p, and takes better advantage of the fast arithmetic in Gmp [G+91].

3.1. Fast relaxed multiplication algorithm. Let a and b denote the two p-adic numbers that we want to multiply, and let c be their product. Let us briefly explain the basic idea behind the speed-up of the relaxed algorithm with respect to naive lazy multiplication.

The first coefficient c_0 is simply obtained as the remainder of a_0b_0 in the division by p. The corresponding quotient is stored as a carry in a variable γ similar to the one used in $\mathsf{Naive\_Mul\_Padic\_rep}_p$. We next obtain c_1 by computing $a_0b_1 + a_1b_0 + \gamma$ and taking the remainder modulo p; the quotient is again stored in γ. At the next stage, which basically requires the computation of $a_0b_2+a_1b_1+a_2b_0+\gamma$, we do a little bit more than necessary: instead of a_1b_1, we rather compute $(a_1+a_2p)(b_1+b_2p)$. For c_3, it then suffices to compute a_0b_3 and a_3b_0 since $a_1b_2 + a_2b_1$ has already been computed as part of $(a_1+a_2p)(b_1+b_2p)$. Similarly, in order to obtain c_4, we only need to compute a_0b_4, a_4b_0, a_3b_1 and a_1b_3, since a_2b_2 is already known. In order to anticipate more future computations, instead of computing a_3b_1, a_1b_3, we compute $(a_1 + a_2p)(b_3 + b_4p)$ and $(a_3 + a_4p)(b_1 + b_2p)$.

In Figure 3.1 below, the contribution of each a_ib_j to the product c_{i+j}, corresponds to a small square with coordinates (i, j). Each such square is part of a larger square which corresponds to a product

$$(a_k + \cdots + a_{k+2^q-1}p^{2^q-1})(b_l + \cdots + b_{l+2^q-1}p^{2^q-1}).$$

The number inside the big square indicates the stage $k + l$ at which this product is computed. For instance the products

$$\begin{aligned}&\left(a_3 + a_4p + a_5p^2 + a_6p^3\right)\left(b_7 + b_8p + b_9p^2 + b_{10}p^3\right),\\ &\left(a_7 + a_8p + a_9p^2 + a_{10}p^3\right)\left(b_3 + b_4p + b_5p^2 + b_6p^3\right).\end{aligned}$$

correspond to the two 4×4 squares marked with 10 inside.

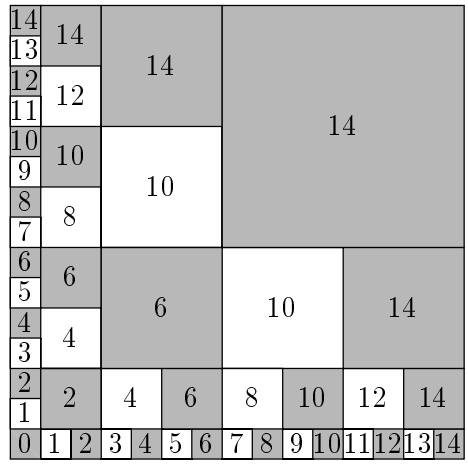

FIGURE 3.1. Relaxed product.

Given a p-adic number a, it will be convenient to denote

$$a_{i\ldots j} = a_i + a_{i+1}p + \cdots + a_{j-1}p^{j-1-i}.$$

For any integer n, we also define l_n to be the largest integer such that 2^{l_n} divides $n+2$ if $n+2$ is not a power of 2. Otherwise, if $n+2 = 2^m$, we let $l_n = m-1$. For instance, $l_0 = 0$, $l_1 = 0$, $l_2 = 1$, $l_3 = 0$, $l_4 = 1$, etc. In fact, this can be seen in Figure 3.1, where the greatest square with number n inside has size precisely $2^{l_n} \times 2^{l_n}$.

We can now describe our fast relaxed product. Recall that φ is the finite p-expansion inherited from $\mathsf{Padic\_rep}_p$ that stores the coefficients known to order n. In the present situation we also use φ for storing the anticipated products.

class $\mathsf{Relaxed\_Mul\_Padic\_rep}_p \trianglerighteq \mathsf{Padic\_rep}_p$
  a, b: Padic_p
  γ^a, γ^b: vectors of vectors over R, with indices starting from 0
  constructor ($\tilde{a}$: Padic_p, $\tilde{b}$: Padic_p)
    $a := \tilde{a}$, $b := \tilde{b}$
    Initialize γ^a and γ^b with the empty vector
  method next ()
    On entry, γ^a and γ^b have size $2n$; resize them to $2(n+1)$
    Initialize γ^a_{2n} and γ^b_{2n} with the zero vector of size $l_{2n}+1$
    Initialize γ^a_{2n+1} and γ^b_{2n+1} with the zero vector of size $l_{2n+1}+1$
    $t^a := 0$, $t^b := 0$
    for q **from** 0 **to** l_n **do**
      $k := (n+2)/2^q$

$t^a \mathrel{+}= \gamma^a_{n,q}$, $t^b \mathrel{+}= \gamma^b_{n,q}$
$t^a \mathrel{+}= a_{2^q-1\ldots 2^{q+1}-1} b_{(k-1)2^q-1\ldots k2^q-1}$
if $k = 2$ **then break**
$t^b \mathrel{+}= a_{(k-1)2^q-1\ldots k2^q-1} b_{2^q-1\ldots 2^{q+1}-1}$
$s^a := \varphi_{n\ldots n+2^{l_n+1}} + t^a$
for i **from** 0 **to** $2^{l_n+1}-1$ **do** $\varphi_{n+i} := s^a_i$
if $n+2 \neq 2^{l_n+1}$ **then** $\gamma^a_{n+2^{l_n+1},l_n} := s^a_{2^{l_n+1}}$
$s^b := \varphi_{n\ldots n+2^{l_n+1}} + t^b$
for i **from** 0 **to** $2^{l_n+1}-1$ **do** $\varphi_{n+i} := s^b_i$
if $n+2 \neq 2^{l_n+1}$ **then** $\gamma^b_{n+2^{l_n+1},l_n} := s^b_{2^{l_n+1}}$
return φ_n

Example. Let us detail how our algorithm works with the first four steps of the multiplication $c = ab$ with

$$a = 676 = 4 + 5 \times 7 + 6 \times 7^2 + 7^3,$$
$$b = -1 = 6 + 6 \times 7 + 6 \times 7^2 + 6 \times 7^3 + O\left(7^4\right).$$

Computation of c_0. Since $l_0 = l_1 = 0$, the entries γ^a_0, γ^b_0, γ^a_1, γ^b_1 are set to (0). In the **for** loop, q takes the single value 0, which gives $k = 2$, $t^a = a_0 b_0 = 3 + 3 \times 7$, and $t^b = 0$. Then we deduce $s^a = 3 + 3 \times 7$, and we set $\varphi_0 = 3$, $\varphi_1 = 3$. In return we have $c_0 = 3$.

Computation of c_1. We have $l_1 = 0$, $l_2 = 1$ and $l_3 = 0$, so that γ^a_2 and γ^b_2 are initialized with $(0, 0)$, while γ^a_3 and γ^b_3 are initialized with (0). In the **for** loop, q takes again the single value 0, and k is set to 3. We obtain $t^a = a_0 b_1 = 3 + 3 \times 7$ and $t^b = a_1 b_0 = 2 + 4 \times 7$. It follows that $s^a = 6 + 3 \times 7$, and then that $s^b = 1 + 1 \times 7 + 1 \times 7^2$. Finally we set $\varphi_1 = 1$, $\varphi_2 = 1$, $\gamma^b_{3,0} = s^b_2 = 1$, and we return $c_1 = 1$.

Computation of c_2. We have $l_2 = 1$, $l_4 = 1$ and $l_5 = 0$, so that γ^a_4 and γ^b_4 are initialized with $(0, 0)$ and γ^a_5 and γ^b_5 with (0). During the first step of the **for** loop we have $q = 0$, $k = 4$, $t^a = a_0 b_2 = 3 + 3 \times 7$ and $t^b = a_2 b_0 = 1 + 5 \times 7$. In the second step we have $q = 1$, $k = 2$, and we add $a_{1\ldots 3} b_{1\ldots 3} = (a_1 + a_2 \times 7)(b_1 + b_2 \times 7) = 2 + 4 \times 7^2 + 6 \times 7^3$ to t^a, its value becomes $5 + 3 \times 7 + 4 \times 7^2 + 6 \times 7^3$. Then we get $s^a = 6 + 3 \times 7 + 4 \times 7^2 + 6 \times 7^3$, and then $s^b = 2 \times 7 + 5 \times 7^2 + 6 \times 7^3$. Finally we set $\varphi_2 = 0$, $\varphi_3 = 2$, $\varphi_4 = 5$, $\varphi_5 = 6$, and return 0 for c_2.

Computation of c_3. We have $l_3 = 0$, $l_6 = 2$ and $l_7 = 0$, hence γ^a_6 and γ^b_6 are set to $(0, 0, 0)$, and γ^a_7 and γ^b_7 to (0). In the **for** loop, q takes the single value 0. We have $k = 5$, $t^a = a_0 b_3 = 3 + 3 \times 7$ and $t^b = \gamma^b_{3,0} + a_3 b_0 = 7$. Then we deduce $s^a = 5 + 7 + 7^2$ which yields $\gamma^a_{5,0} = 1$, and then $s^b = 5 + 2 \times 7$. In return we thus obtain $c_3 = 5$.

Proposition 3.1. *Given relaxed p-adic integers a and b, the product ab can be computed up till precision n using* $O(\mathsf{I}_p(n)\log n)$ *bit-operations. For this computation, the total amount of space needed to store the carries* γ^a *and* γ^b *does not exceed* $O(n)$.

Proof. The proof is essentially the same as for [Hoe02, Section 4.3.2, Theorem 6], but the carries require additional attention. We shall prove by induction that all the entries of γ^a and γ^b are always in $\{0,1\}$ when entering next for the computation of φ_n. This holds for $n=0$. Assume now that it holds for a certain $n \geqslant 0$. After the first step of the loop, namely for $q=0$, we have $t^a \leqslant (p-1)^2+1 \leqslant p^2-1$. After the second step, when $q=1$, we have $t^a \leqslant (p^2-1)^2+p^2-1+1 \leqslant p^4-1$. By induction, it follows that $t^a \leqslant p^{2^{q+1}}-1$, at the end of the q-th step.

At the end of the **for** loop, we thus get $t^a \leqslant p^{2^{l_n+1}}-1$. This implies $s^a \leqslant 2p^{2^{l_n+1}}-2$, whence $\gamma^a_{n+2^{l_n+1},l_n} \in \{0,1\}$. The same holds for superscripts b instead of a. Notice that if $n+2 \neq 2^{l_n+1}$ then $2^{l_n+1} \leqslant n+1$, hence $n+2^{l_n+1} \leqslant 2n+1$. This implies that $\gamma^a_{n+2^{l_n+1},l_n}$ and $\gamma^b_{n+2^{l_n+1},l_n}$ are well defined.

If $n+2=2^{l_n+1}$, then $s^a_{2^{l_n+1}} = s^b_{2^{l_n+1}} = 0$, since s^a and s^b are bounded by $(p^{n+1}-1)^2/p^n < p^{n+2}$. On the other hand, the map $n \mapsto (n+2^{l_n+1}, l_n)$ is injective, so that each entry of γ^a and γ^b can be set to 1 at most once. It thus follows that all the carries are carefully stored in the vectors γ^a and γ^b.

If $n+2=2^{l_n}u$, with $u \geqslant 2$, then $n+2^{l_n+1}+2 = 2^{l_n}(u+2)$, with $u+2 \geqslant 2$. This implies that, when we arrive at order $n' = n+2^{l_n+1}$, then the value $l_{n'}$ is at least l_n. Therefore all the carries are effectively taken into account. This proves the correctness of the algorithm.

The cost of the algorithm at order n is

$$O\left(\sum_{i=0}^{n}\sum_{q=0}^{l_i}\mathsf{I}_p(2^i)\right) = O\left(\sum_{2^q\leqslant n}\frac{n}{2^q}\mathsf{I}_p(2^q)\right) =$$

$$O\left(\mathsf{I}_p(n)\sum_{2^q\leqslant n}1\right) = O(\mathsf{I}_p(n)\log n),$$

using our assumption that $\mathsf{I}_p(n)/n$ is increasing. Finally,

$$O\left(\sum_{i=0}^{n}(1+l_i)\right) = O\left(\sum_{i=0}^{n}\sum_{q=0}^{l_i}1\right) = O\left(\sum_{2^q\leqslant n}\frac{n}{2^q}\right) = O(n)$$

provides enough space for storing all the carries. □

In practice, instead of increasing the sizes of carry vectors by one, we double these sizes, so that the cost of the related memory allocations and copies becomes negligible. The same remark holds for the coefficients stored in φ.

When multiplying finite p-adic expansions using Kronecker substitution, we obtain a cost similar to the one of Proposition 2.4. Implementing a good product for finite p-adic expansions requires some effort, since we cannot directly use binary arithmetic available in the hardware. In the next subsection, we show that minor modifications of our relaxed product allow us to convert back and forth between the binary representation in an efficient manner. Finally, notice that in the case when $R = \mathbb{K}[x]$ and $(p) = (x)$, the carries γ^a and γ^b are useless.

3.2. Variant with conversion to binary representation. In this subsection, we assume that $R = \mathbb{Z}$ and we adapt the above relaxed product in order to benefit from fast binary arithmetic available in the processor or Gmp. In fact we shall convert from base p to base 2 in order to perform most of the internal computations efficiently, but backward conversions are needed for the output. These conversions can be naturally integrated in an efficient manner as described in the following algorithm:

class Binary_Mul_Padic_rep$_p$ $\trianglerighteq$ Padic_rep$_p$
  a, b: Padic$_p$
  $\beta^a, \beta^b, \delta^a, \delta^b, \gamma$: vectors over $\mathbb{Z}$, with indices starting from 0.
  constructor ($\tilde{a}$: Padic$_p$, $\tilde{b}$: Padic$_p$)
    $a := \tilde{a}$, $b := \tilde{b}$
    Initialize $\beta^a, \beta^b, \delta^a, \delta^b, \gamma$ with empty vectors
  method next ()
    If $n + 2$ is a power of 2, then
      Resize $\beta^a, \beta^b, \delta^a, \delta^b$, and γ to $l_n + 1$
      Fill the new entries with zeros
    $\varepsilon^a := a_n$, $\varepsilon^b := b_n$, $\tau := 0$
    for q **from** 0 **to** l_n **do**
      if $q > 0$ **then**
        $\varepsilon^a := \beta^a_{q-1} + p^{2^{q-1}}\varepsilon^a$
        $\varepsilon^b := \beta^b_{q-1} + p^{2^{q-1}}\varepsilon^b$
      if $n + 2 = 2^{q+1}$ **then**
        $\delta^a_q := \varepsilon^a$, $\delta^b_q := \varepsilon^b$
      $\tau \mathrel{+}= \delta^a_q\varepsilon^b + \gamma_q$
      if $n + 2 = 2^{q+1}$ **then break**
      $\tau \mathrel{+}= \varepsilon^a\delta^b_q$
    $\beta^a_{l_n} := \varepsilon^a$, $\beta^b_{l_n} := \varepsilon^b$

for q **from** l_n **down to** 0 **do**
  $\gamma_q := \mathsf{quo}\,(\tau, p^{2^q})$, $\tau := \mathsf{rem}\,(\tau, p^{2^q})$
return τ

Proposition 3.2. *Given two relaxed p-adic integers a and b, the computation of the product ab up till precision n can be done with* $O\,(\mathsf{I}(n \log p) \log n)$ *bit-operations and* $O(n \log p)$ *bit-space.*

Proof. When computing φ_n, the vectors $\beta^a, \beta^b, \delta^a, \delta^b$, and γ are resized to r_n, where r_n is the largest integer such that $2^{r_n} \leqslant n+2$. From $2^{r_n+1} > n+2 \geqslant 2^{l_n+1}$, we deduce that $r_n > l_n$, which means that the read and write operations in these vectors are licit.

For any integers n and q such that $2^{q+1} < n+2$, we write $\mu(n,q)$ for the largest integer less than n such that $l_{\mu(n,q)} = q$. We shall prove by induction that, when entering next for computing φ_n, the following properties hold for all $q \geqslant 0$ such that $2^{q+1} < n+2$:

(a) $\beta^a_q = a_{(k-1)2^q-1\ldots k2^q-1} = a_{\mu(n,q)-2^q+1\ldots\mu(n,q)+1}$ where $k = \frac{\mu(n,q)+2}{2^q}$, and similarly for β^b_q,
(b) $\delta^a_q = a_{2^q-1\ldots 2^{q+1}-1}$, and similarly for δ^b_q, and
(c) $\gamma_q \leqslant 2p^{2^q}$.

These properties trivially hold for when $n = 0$. Let us assume that they hold for a certain $n \geqslant 0$.

Now we claim that, at the end of step q of the first loop, the value of ε^a_n is $a_{(k-1)2^q-1\ldots k2^q-1} = a_{n-2^q+1\ldots n+1}$ with $k = (n+2)/2^q$. This clearly holds for when $q = 0$ because $\varepsilon^a = a_n$ and $k = n+2$. Now assume that this claim holds until step $q-1$ for some $q \geqslant 1$. When entering step q, we have that $\mu(n, q-1) = n - 2^{q-1}$, and part (a) of the induction hypothesis gives us that $\beta^a_{q-1} = a_{n-2^q+1\ldots n-2^{q-1}+1}$. From these quantities, we deduce:

$$\begin{aligned}\beta^a_{q-1} + p^{2^{q-1}}\varepsilon^a &= a_{n-2^q+1\ldots n-2^{q-1}+1} + p^{2^{q-1}} a_{n-2^{q-1}+1\ldots n+1} \\ &= a_{n-2^q+1\ldots n+1} = a_{(k-1)2^q-1\ldots k2^q-1},\end{aligned}$$

with $k = (n+2)/2^q$, which concludes the claim by induction. If $n+2$ is not a power of 2 then part (a) is clearly ensured at the end of the computation of φ_n. Otherwise $n+2 = 2^{l_n+1}$, and $\beta^a_{l_n}$ is set to $a_{n-2^{l_n}+1\ldots n+1}$, and part (a) is again satisfied when entering the computation of φ_{n+1}.

When δ^a_q is set to ε^a, the value of ε^a is $a_{(k-1)2^q-1\ldots k2^q-1}$ with $k = 2$. This ensures that part (b) holds when entering the computation of φ_{n+1}.

As to (c), during step q of the first loop, the value of τ is incremented by at most

$$2(p^{2^q}-1)^2 + 2p^{2^q} \leqslant 2p^{2^{q+1}} - 2p^{2^q} + 2.$$

At the end of this loop, we thus have

$$\tau \leqslant 2p^{2^{l_n+1}} - 2p + 2(l_n+1).$$

It follows that $\tau/p^{2^{l_n}} < 2p^{2^{l_n}} + 2(l_n+1)/p^{2^{l_n}}$. If $l_n \in \{0,1\}$ then it is clear that $2(l_n+1) \leqslant p^{2^{l_n}}$, since $p \geqslant 2$. If $l_n \geqslant 1$ then $\frac{d(p^{2^{l_n}})}{dl_n} = \log(2)\log(p)2^{l_n}p^{2^{l_n}} \geqslant 8(\log 2)^2 \geqslant 2$. We deduce that $2(l_n+1) \leqslant p^{2^{l_n}}$ holds for all integer $l_n \geqslant 0$. Before exiting the function we therefore have that $\gamma_{l_n} \leqslant 2p^{2^{l_n}}$, $\gamma_{l_n-1} \leqslant p^{2^{l_n-1}} \leqslant 2p^{2^{l_n-1}}$, etc., which completes the induction.

Since $n+2 = 2^{l_n}u$, with $u \geqslant 2$, we have $n+2^k+2 = 2^k(2^{l_n-k}u+1)$, whence $l_{n+2^k} \geqslant k$, for any $k \leqslant l_n$. All the carries stored in γ are therefore properly taken into account. This proves the correctness of the algorithm.

At precision n, summing the costs of all the calls to next amounts to

$$\begin{aligned} O\left(\sum_{i=0}^{n}\sum_{q=0}^{l_i} \mathsf{I}(2^q \log p)\right) &= O\left(\sum_{2^q \leqslant n} \frac{n}{2^q}\mathsf{I}(2^q \log p)\right) \\ &= O\left(\mathsf{I}(n\log p)\sum_{2^q \leqslant n} 1\right) \\ &= O(\mathsf{I}(n\log p)\log n). \end{aligned}$$

Furthermore,

$$O\left(\sum_{2^{q+1} \leqslant n+2} 2^q \log p\right) = O(n \log p)$$

provides a bound for the total bit-size of the auxiliary vectors $\beta^a, \beta^b, \delta^a, \delta^b$, and γ. □

Again, in practice, one should double the allocated sizes of the auxiliary vectors each time needed so that the cost of the related memory allocations and copies becomes negligible. In addition, for efficiency, one should precompute the powers of p.

3.3. Timings. In following Table 3.1, we compare timings for power series over $\mathbb{F}_p$, and for p-adic integers *via* the technique of Section 2.7, called Series_Mul_Padic_rep$_p$, and *via* Binary_Mul_Padic_rep$_p$ of Proposition 3.2. In Series_Mul_Padic_rep$_p$ the internal series product is the relaxed one reported in the first line.

n	8	16	32	64	128	256	512	1 024	2 048	4 096
Rel. mul. in $\mathbb{F}_p[[x]]$	2	7	20	50	140	360	930	2 300	5 700	14 000
Series_Mul_Padic$_p$	19	59	160	420	1 100	2 600	6 200	14 000	34 000	79 000
Binary_Mul_Padic$_p$	8	16	37	89	170	360	800	1 900	4 300	10 000
Naive_Mul_Padic$_p$	2.9	3.8	8.3	22	68	250	920	3 600	14 000	56 000

Table 3.1. Fast relaxed products, and naive lazy product, for $p = 536870923$, in microseconds.

For convenience, we recall the timings for the naive algorithm of Section 2.6 in the last line of Table 3.1. We see that our Binary_Mul_Padic_rep$_p$ is faster from size 512 on. Since handling small numbers with GMP is expensive, we also observe some overhead for small sizes, compared to the fast relaxed product of formal power series. On the other hand, since the relaxed product for power series makes internal use of Kronecker substitution, it involves integers that are twice as large as those in Binary_Mul_Padic_rep$_p$. Notice finally that the lifting strategy Series_Mul_Padic_rep$_p$, described in Section 2.7, is easy to implement, but not competitive.

4. Blockwise product

As detailed in Section 4.1 below for $R = \mathbb{Z}$, the relaxed arithmetic is slower than direct computations modulo p^n in binary representation. In [Hoe07b], an alternative approach for relaxed power series multiplication was proposed, which relies on grouping blocks of k coefficients and reducing a relaxed multiplication at order n to a relaxed multiplication at order n/k, with FFT-ed coefficients in M^{2k-1}.

Unfortunately, we expect that direct application of this strategy to our case gives rise to a large overhead. Instead, we will now introduce a variant, where the blocks of size k are rather rewritten into an integer modulo p^k. This aims at decreasing the overhead involved by the control instructions when handling objects of small sizes, and also improving the performance in terms of memory management by choosing blocks well suited to the sizes of the successive cache levels of the platform being used.

We shall start with comparisons between the relaxed and zealous approaches. Then we develop a supplementary strategy for a continuous transition between the zealous and the relaxed models.

4.1. Relaxed versus zealous. The first line of Table 4.1 below displays the time needed for the product modulo p^n of two integers taken at random in the range $0, \ldots, p^n - 1$. The next line concerns the performance of our function binary that converts a finite p-expansion of size n into its binary representation. The reverse function, reported in the last line, and written expansion, takes an integer in $0, \ldots, p^n - 1$ in base 2 and returns its p-adic expansion.

n	8	16	32	64	128	256	512	1 024	2 048	4 096	8 192
mod. mul.	0.38	0.52	1.0	2.9	9.0	27	85	250	690	1 800	4 500
binary	1.0	2.2	4.5	9.8	20	44	100	250	560	1 400	3 300
expansion	2.5	5.2	10	22	46	96	220	490	1 200	3 000	7 300

TABLE 4.1. Zealous product modulo p^n and conversions, for $p = 536870923$, in microseconds.

Let us briefly recall that binary can be computed fast by applying the classical divide and conquer paradigm as follows:

$$\mathsf{binary}(a, p^n) = \mathsf{binary}(a_{0\ldots h}, p^h) + p^h \mathsf{binary}(a_{h\ldots n}, p^{n-h}), \text{where } h = \lfloor n/2 \rfloor,$$

which yields a cost in $O(\mathsf{I}(n \log p) \log n)$. Likewise, the same complexity bound holds for expansion. Within our implementation we have observed that these asymptotically fast algorithms outperform the naive ones whenever p^n is more than around 32 machine words.

Compared to Tables 2.1 and 3.1, these timings confirm the asymptotic theoretical bounds: the relaxed product does not compete with a direct modular computation in binary representation. This is partly due to the extra $O(\log n)$ factor for large sizes. But another reason is the overhead involved by the use of Gmp routines with integers of a few words. In Table 4.2, we report on the naive and the relaxed products in base p^{32}. Now we see that our naive product becomes of the same order of efficiency as the zealous approach up to precision 1024. The relaxed approach starts to win when the precision reaches 256 in base p^{32}.

kl	32	64	128	256	512	1 024	2 048	4 096	8 192
Naive_Mul_Padic$_{p^k}$	1.8	4.1	10	27	84	280	1 000	3 900	15 000
Binary_Mul_Padic$_{p^k}$	3.2	6.1	16	53	170	570	1 800	5 300	15 000

Table 4.2. Relaxed product modulo $(p^k)^l$, for $k = 32$ and $p = 536870923$, in microseconds.

4.2. Monoblock strategy. If one wants to compute the product ab of two p-adic numbers a and b, then: one can start by converting both of them into p^k-adic numbers A and B respectively, multiply A and B as p^k-adic numbers, and finally convert AB back into a p-adic number. The transformations between p-adic and p^k-adic numbers can be easily implemented:

class To_Blocks$_{p^k}$ $\trianglerighteq$ Padic_rep$_{p^k}$
  a: Padic$_p$
  constructor ($\tilde{a}$: Padic$_p$)
    $a := \tilde{a}$
  method next ()
    return binary$(a_{nk\ldots(n+1)k})$
class From_Blocks$_p$ $\trianglerighteq$ Padic_rep$_p$
  a: Padic$_{p^k}$
  b: p-expansion of size k
  constructor ($\tilde{a}$: Padic$_{p^k}$)
    $a := \tilde{a}$

method next ()
  if $n \bmod k{=}0$ **then** $b =$p_expansion$(a_{n/k}, p)$
  return $b_{n \bmod k}$

If to_blocks and from_blocks represent the top level functions then the product c of a and b can be simply obtained as

$$c = \mathsf{from\_blocks}\,(\mathsf{to\_blocks}(a)\mathsf{to\_blocks}(b))\,.$$

We call this way of computing products the *monoblock strategy*.

Notice that choosing k very large is similar to zealous computations. This monoblock strategy can thus be seen as a mix of the zealous and the relaxed approaches. However, it is only relaxed for p^k-expansions, not for p-expansions. In fact, let A and B still denote the respective p^k-adic representations of a and b, so that $c = \mathsf{from\_blocks}\,(C)$, for $C = AB$. Then the computation of c_0 requires the knowledge of $C_0 = A_0B_0$, whence it depends on all the coefficients $a_0, \ldots, a_{k-1}$ and $b_0, \ldots, b_{k-1}$, which breaks the main requirement on relaxed operations recalled in Section 1.1.1. In the next paragraphs we derive an actual relaxed product from this strategy, at the price of a reasonable overhead.

4.3. Relaxed blockwise product. We are now to present a relaxed p-adic blockwise product. This product depends on two integer parameters m and k. The latter still stands for the size of the blocks to be used, while the former is a threshold: below precision m one calls a given product on p-expansions, while in large precision an other product is used on p^k-expansions.

If a and b are the two numbers in R_p that we want to multiply as a p-expansions, then we first rewrite them $a = a_{0\ldots m} + p^m\bar{a}$ and $b = b_{0\ldots m} + p^m\bar{b}$, where

$$\bar{a} = a/p^m = \sum_{i=0}^{\infty} a_{m+i}p^i \quad \text{and} \quad \bar{b} = b/p^m = \sum_{i=0}^{\infty} b_{m+i}p^i.$$

Now multiplying a and b gives

$$c = a_{0\ldots m}b_{0\ldots m} + p^m\left(a_{0\ldots m}\bar{b} + \bar{a}b_{0\ldots m}\right) + p^{2m}\bar{a}\bar{b},$$

where the product $\bar{a}\bar{b}$ can be computed in base p^k, as it is detailed in the following implementation:

class Blocks_Mul_Padic_rep$_p$ $\trianglerighteq$ Padic_rep$_p$
  a, b, c, $\bar{a}$, $\bar{b}$: Padic$_p$
  $\bar{A}$, $\bar{B}$: Padic$_{p^k}$
  constructor ($\tilde{a}$: Padic$_p$, $\tilde{b}$: Padic$_p$)
    $a := \tilde{a}$, $b := \tilde{b}$
    $\bar{a} := a/p^m$, $\bar{b} := b/p^m$
    $\bar{A} :=$ to_blocks$(\bar{a})$, $\bar{B} :=$ to_blocks$(\bar{b})$

$\bar{c} := \mathsf{from\_blocks}(\bar{A}\bar{B})$
$c := a_{0\ldots m}b_{0\ldots m} + p^m\left(a_{0\ldots m}\bar{b} + \bar{a}b_{0\ldots m}\right) + p^{2m}\bar{c}$
method next ()
return c_n

In Figure 4.1 below, we illustrate the contribution of each a_ib_j to the product c_{i+j} computed with the present blockwise version. In both bases p and p^k the naive product is used, and the numbers inside the squares indicate the degrees at which the corresponding product is actually computed.

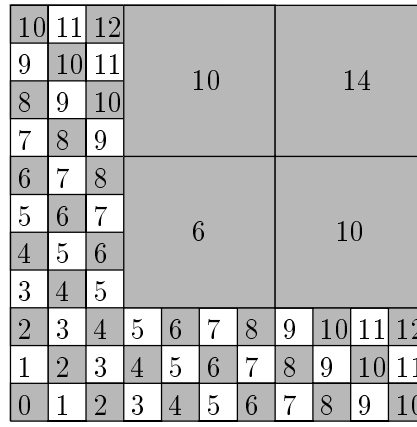

Figure 4.1. Blockwise product for $m = 3$ and $k = 4$.

Proposition 4.1. *If $m \geqslant k-1$, then* $\mathsf{Blocks\_Mul\_Padic\_rep}_p$ *is relaxed for base p.*

Proof. It is sufficient to show that the computation of $\bar{c}_{n-2m}$ only involves terms in a and b of degree at most n. In fact $\bar{c}_{n-2m}$ requires the knowledge of the coefficients of $\bar{A}$ and $\bar{B}$ to degree at most $l = \lfloor (n-2m)/k \rfloor$, hence the knowledge of the coefficients of a and b to degree $k(l+1)-1+m \leqslant n-2m+k-1+m = n+k-1-m$, which concludes the proof thanks to the assumption on m. Notice that the latter inequality is an equality whenever $n-2m$ is a multiple of k. Therefore $m \geqslant k-1$ is necessary to ensure the product to be relaxed. □

4.4. Timings. In following Table 4.3, we use blocks of size $k = 32$, and compare the blockwise versions of the naive product of Section 2.6 to the relaxed one of Section 3.2. The first line concerns the monoblock strategy: below precision 32 we directly use the naive p-adic product; for larger precisions we use the naive p^k-adic product. The second line is the same as the first one except that we use a relaxed p^k-adic product. In the third line the relaxed blockwise version is used with $m = 32$: we use the naive product

for both p- and p^k-adic expansions. The fourth line is similar except that the fast relaxed product is used beyond precision 32.

n	8	16	32	64	128	256	512	1 024	2 048	4 096
mono Naive_Mul_Padic$_p$	3.5	5.5	11	53	95	190	380	870	2 200	6 500
mono Binary_Mul_Padic$_p$	3.4	5.5	11	57	110	220	500	1 200	3 000	7 800
blocks Naive_Mul_Padic$_p$	8.0	11	17	46	140	320	700	1 600	3 700	9 400
blocks Binary_Mul_Padic$_p$	8.0	11	17	46	140	330	750	1 700	3 900	9 100

TABLE 4.3. Blockwise products to precision n, for $p = 536870923$ and $k = 32$, in microseconds.

When compared to Table 4.2, we can see that most of the time within the monoblock strategy is actually spent on base conversions. In fact, the strategy does not bring a significant speed-up for a single product, but for more complex computations, the base conversions can often be factored.

For instance, assume that a and b are two $d \times d$ matrices with entries in $\mathbb{Z}_p$. Then the multiplication $c = ab$ involves only $O(d^2)$ base conversions and $O(d^3)$ p^k-adic products. For large d, the conversions thus become inexpensive. In Section 7, we will encounter a similar application to root extraction.

5. Application to recursive p-adic numbers

A major motivation behind the relaxed computational model is the efficient expansion of p-adic numbers that are solutions to recursive equations. This section is an extension of [Hoe02, Section 4.5] to p-adic numbers.

Let us slightly generalize the notion of a recursive equation, which was first defined in the introduction, so as to accommodate for initial conditions. Consider a functional equation

$$Y = \Phi(Y), \tag{5.1}$$

where Y is a vector of d unknowns in R_p. Assume that there exist a $k \in \mathbb{N}^*$ and *initial conditions* $c_0, \ldots, c_{k-1} \in M^d$, such that for all $n \geqslant k$ and $y, \tilde{y} \in R_p^d$ with $y_0 = c_0, \ldots, y_{k-1} = c_{k-1}$, we have

$$\tilde{y} - y \in p^n R_p^d \implies \Phi(\tilde{y}) - \Phi(y) \in p^{n+1} R_p^d. \tag{5.2}$$

Stated otherwise, this condition means that each coefficient $b_n = \Phi(b)_n$ with $n \geqslant k$ only depends on previous coefficients $b_0, \ldots, b_{n-1}$. Therefore, setting $c = c_0 + c_1 p + \cdots + c_{k-1} p^{k-1}$, the sequence $c, \Phi(c), \Phi(\Phi(c))$ converges to a unique solution $b \in R_p^d$ of (5.1) with $b_0 = c_0, \ldots, b_{k-1} = c_{k-1}$. We will call (5.1) a *recursive equation* and the entries of the solution b *recursive p-adic numbers*.

5.1. Implementation. Since p induces an element $\boldsymbol{p} = (p, \ldots, p)$ in R^d and an isomorphism $R_p^d \cong (R^d)_{\boldsymbol{p}}$, we may reinterpret a solution $b = \Phi(b)$ as a $\boldsymbol{p}$-adic number over R^d. Using this trick, we may assume without loss of generality that $d = 1$. In our implementation, recursive numbers are instances of the following class that stores the initial conditions $b_0, \ldots, b_{k-1}$ and the equation Φ:

class Recursive_Padic_rep$_p$ $\trianglerighteq$ Padic_rep$_p$
  Φ: function from R_p to R_p
  $b_0, \ldots, b_{k-1}$: initial conditions in M
  constructor ($\tilde{\Phi}$: function, $\tilde{b}_0, \ldots, \tilde{b}_{k-1}$: M)
    $\Phi := \tilde{\Phi}$, $b_0 := \tilde{b}_0, \ldots, b_{k-1} := \tilde{b}_{k-1}$
  method next ()
    If $n < k$ **then return** b_n
    return $\Phi(\text{this})_n$

In the last line, the expression $\Phi(\text{this})$ means the evaluation of Φ at the concrete instance of the p-adic $b = \Phi(b)$ being currently defined.

Example. Consider $\Phi(b) = pb + 1$, with one initial condition $b_0 = 1$. It is clear that b is recursive, since the n first terms of pb can be computed from the only $n - 1$ first terms of b. We have $b_1 = b_2 = \cdots = 1$. In fact, $b = 1/(1 - p)$.

5.2. Complexity analysis. If Φ is an expression built from L constants, sums, differences, and products (all of arity two), then the computation of b simply consists in performing these L operations in the relaxed model. For instance, when using the relaxed product of Proposition 3.1, this amounts to $O(L\mathsf{I}_p(n) \log n)$ operations to obtain the n first terms of b.

This complexity bound is to be compared to the classical approach *via* the Newton operator. In fact, one can compute b with fixed-point p-adic arithmetic by evaluating the following operator $N_\Phi(z) = z - (z - \Phi(z))/(1 - \Phi'(z))$. There are several cases where the relaxed approach is faster than the Newton operator:

(1) The constant hidden behind the "O" of the Newton iteration is higher than the one with the relaxed approach. For instance, if b is really a vector in R_p^d, then the Newton operator involves the inversion of a $d \times d$ matrix at precision $n/2$, which gives rise to a factor $O(d^3)$ in the complexity (assuming the naive matrix product is used). The total cost of the Newton operator to precision n in $\mathbb{Z}_p$ is thus in $O((dL + d^3)\mathsf{I}_p(n))$. Here $O(dL)$ bounds the number of operations needed to evaluate the Jacobian matrix. In this situation, if $L \ll d^2$, and unless n is very large, the relaxed approach is faster. This will be actually illustrated in the next subsection.

(2) Even in the case $d = 1$, the "O" hides a non trivial constant factor due to a certain amount of "recomputations". For moderate sizes, when polynomial multiplication uses Karatsuba's algorithm, or the Toom-Cook method, the cost of relaxed multiplication also drops to a constant times the cost of zealous multiplication [Hoe02, Hoe07b]. In such cases, the relaxed method often becomes more efficient. This will be illustrated in Section 6 for the division.

(3) When using the blockwise method from Section 4 or [Hoe07b] for power series, the overhead of relaxed multiplication can often be further reduced. In practice, we could observe that this makes it possible to outperform Newton's method even for very large sizes.

For more general functional equations, where Φ involves non-algebraic operations, it should also be noticed that suitable Newton operators Φ are not necessarily available. For instance, if the mere definition of Φ involves p-expansions, then the Newton operator may be not defined anymore, or one needs to explicitly compute with p-expansions. This occurs for instance for $R = \mathbb{Z}$, when Φ involves the "symbolic derivation" $b \mapsto \sum_{n \geqslant 1} n b_n p^{n-1}$.

5.3. Timings. In order to illustrate the performance of the relaxed model with respect to Newton iteration, we consider the following family of systems of p-adic integers:

$$\Phi_{d,i}(x_1, \ldots, x_d) = 1 + p \sum_{k=1}^{d} (k+i) x_k^{(k+i) \bmod 3}, \qquad \text{for } i \in \{1, \ldots, d\}.$$

The number of p-adic products grows linearly with d. Yet, the total number of operations grows with d^2.

In Table 5.1, we compute the 256 first terms of the solution $b = \Phi_d(b)$ with the initial condition $b = (1, \ldots, 1) + O(p)$. We use the naive product of Section 2.6 and compare to the Newton iteration directly implemented on the top of the routines of Gmp. In fact the time we provide in the line "Matrix multiplication" does not correspond to a complete implementation of the iteration but only to two products of two $d \times d$ matrices with integers modulo p^{64}. These two operations necessarily occurs for inverting the Jacobian matrix to precision p^{128} when using the classical algorithm as described in [GG03, Algorithm 9.2]. This can be seen as a lower bound for any implementation of the Newton method. However the line "Newton implementation" corresponds to our implementation of this method, hence this is an upper bound. The next line of the table, named "Naive iteration", corresponds to the computation from $b = (1, \ldots, 1)$ of $\Phi_d(b)$ modulo p^2, then $\Phi_d(\Phi_d(b))$ modulo p^3, etc. This sequence converges linearly to the solution.

Although Newton iteration is faster for tiny dimensions $d \leqslant 2$, its cost growths as d^3 for larger d, whereas the relaxed approach reported on the

d	1	2	4	8	16	32	64	128
Matrix multiplication	0.002	0.014	0.12	0.9	7.2	56	460	3 600
Newton implementation	0.13	0.31	2.3	13	95	700	5 500	43 000
Naive iteration	3.8	7.7	17	40	110	330	1 100	4 000
Naive_Mul_Padic$_p$	0.34	0.42	1.4	3.3	8.7	26	94	420

Table 5.1. 256 first terms of $b = \Phi_d(b)$, for $p = 536870923$, in milliseconds.

last line only grows as d^2. For $d = 1$, we notice that the number b is computed with essentially one relaxed product. Notice that, due to the linear convergence, the naive iteration behaves well when the dimension is large and the precision relatively small.

In the next table we report of the same computations but with the relaxed product of Section 3.2 at precision 1024; the conclusions are essentially the same:

d	1	2	4	8	16	32	64	128
Matrix multiplication	0.014	0.12	0.98	7.9	62	490	4 000	31 000
Newton implementation	0.58	1.2	13	90	640	4 900	38 000	300 000
Naive iteration	110	220	450	930	2 000	4 800	13 000	37 000
Binary_Mul_Padic$_p$	2.4	2.6	9.4	21	52	150	480	2 300

Table 5.2. 1024 first terms of $b = \Phi_d(b)$, for $p = 536870923$, in milliseconds.

6. Relaxed division

We are now to present relaxed algorithms to compute the quotient of two p-adic numbers. The technique is similar to power series, as treated in [Hoe02], but with subtleties.

6.1. Division by a "scalar". The division of a power series in $\mathbb{K}[[x]]$ by an element of $\mathbb{K}$ is immediate, but it does not extend to p-adic numbers, because of the propagation of the carries. We shall introduce two new operations. Let $\beta \in M$ play the role of a "scalar". The first new operation, written $\mathsf{mul\_rem}$ (a: Padic_p, β: M), returns the p-adic number c with coefficients $c_n = \mathsf{rem}\,(\beta a_n, p)$. The second operation, written $\mathsf{mul\_quo}$ (a: Padic_p, β: R), returns the corresponding carry, so that

$$\begin{aligned}\beta a &= \mathsf{mul\_rem}(a,\beta) + p\mathsf{mul\_quo}(a,\beta)\\ &= \sum_{n=0}^{\infty} \mathsf{rem}\ (\beta a_n, p)\, p^n + \sum_{n=0}^{\infty} \mathsf{quo}\ (\beta a_n, p)\, p^{n+1}.\end{aligned}$$

These operations are easy to implement, as follows:

class Mul_Rem_Padic_rep$_p$ $\trianglerighteq$ Padic_rep$_p$
  a: Padic$_p$
  β: M
  constructor ($\tilde{a}$: Padic$_p$, $\tilde{\beta}$: M)
    $a := \tilde{a}$, $\beta := \tilde{\beta}$
  method next ()
    return rem $(\beta a_n, p)$
class Mul_Quo_Padic_rep$_p$ $\trianglerighteq$ Padic_rep$_p$
  a: Padic$_p$
  β: M
  constructor ($\tilde{a}$: Padic$_p$, $\tilde{\beta}$: M)
    $a := \tilde{a}$, $\beta := \tilde{\beta}$
  method next ()
    return quo $(\beta a_n, p)$

Proposition 6.1. *Let a be a relaxed p-adic number and let $\beta \in M$. If β is invertible modulo p, with given inverse $\gamma = \beta^{-1} \bmod p$, then the quotient $c{=}a/\beta$ is recursive and c satisfies the equation*

$$c = \mathsf{mul\_rem}\,(a - p\mathsf{mul\_quo}\,(c, \beta)\,, \gamma)\,, \qquad c_0 = \gamma a_0 \bmod p.$$

If $R = \mathbb{Z}$, then a/β can be computed up till precision n using $O(n\mathsf{I}(\log p))$ bit-operations.

Proof. It is clear from the definitions that the proposed formula actually defines a recursive number. Then, from

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

we deduce that $\beta c - p\mathsf{mul\_quo}\,(c, \beta) = \mathsf{mul\_rem}\,(c, \beta)$, hence

$$\begin{aligned} c &= \mathsf{mul\_rem}\,(\beta c - p\mathsf{mul\_quo}\,(c, \beta)\,, \gamma) \\ &= \mathsf{mul\_rem}\,(a - p\mathsf{mul\_quo}\,(c, \beta)\,, \gamma)\,. \end{aligned}$$

The functions mul_rem and mul_quo both take $O(n\mathsf{I}(\log p))$ bit-operations if $R = \mathbb{Z}$, which concludes the proof. □

6.2. Quotient of two *p*-adic numbers. Once the division by a "scalar" is available, we can apply a similar formula as for the division of power series of [Hoe02].

Proposition 6.2. *Let a and b be two relaxed p-adic numbers such that b_0 is invertible of given inverse $\gamma = b_0^{-1} \bmod p$. The quotient $c{=}a/b$ is recursive and satisfies the following equation:*

$$c = \frac{a - (b - b_0)c}{b_0}, \qquad c_0 = \gamma a_0 \bmod p.$$

In addition, if $R = \mathbb{Z}$, then a/b can be computed up till precision n using $O(\mathsf{I}(n \log p) \log n)$ bit-operations.

Proof. The last assertion on the cost follows from Proposition 3.2. □

Remark. Notice that p is not assumed to be prime, so that we can replace p by p^k, and thus benefit of the monoblock strategy of Section 4.2. This does not involve a large amount of work: it suffices to write

$$\mathsf{from\_blocks}(\mathsf{to\_blocks}(a)/\mathsf{to\_blocks}(b)).$$

Notice that this involves inverting $b_{0\ldots p^k}$ modulo p^k.

6.3. Timings. In Table 6.1 we display the computation time for our division algorithm. We compare several methods:

- The first line "Newton" corresponds to the classical Newton iteration [GG03, Algorithm 9.2] used in the zealous model.
- The second line corresponds to one call of Gmp's extended g.c.d. function.
- The third line is a comparison with Pari/Gp version 2.3.5.
- The next two lines $\mathsf{Naive\_Mul\_Padic}_p$ and $\mathsf{Binary\_Mul\_Padic}_p$ correspond to the naive product of Section 2.6, and the relaxed one of Section 3.2.
- The lines "mono $\mathsf{Naive\_Mul\_Padic}_p$" and "mono $\mathsf{Naive\_Mul\_Padic}_p$" correspond to the monoblock strategy from Section 4.2 with blocks of size 32.
- Then "blocks $\mathsf{Naive\_Mul\_Padic}_p$ and "blocks $\mathsf{Naive\_Mul\_Padic}_p$" correspond to the relaxed block strategy from Section 4.3 with blocks of size 32.
- Finally the last line corresponds to direct computations in base p^{32} (with no conversions from/to base p).

When compared to Tables 2.1 and 3.1, we observe that the cost of one division algorithm is merely that of one multiplication whenever the size becomes sufficiently large, as expected. We also observe that our "monoblock division" is faster than the zealous one for large sizes; this is even more true if we directly compute in base p^{32}.

7. Higher order roots

For power series in characteristic 0, the r-th root g of f is recursive, with equation $g = \int f'/(rg^{r-1})$ and initial condition $g_0 = f_0^{1/r}$ (see [Hoe02, Section 3.2.5] for details). This expression neither holds in small positive characteristic, nor for p-adic integers. In this section we propose new formulas for these two cases, which are compatible with the monoblock strategy of Section 4.2.

7.1. Regular case. In this subsection we treat the case when r is invertible modulo p.

n	8	16	32	64	128	256	512	1024	2048
Newton	3	4	7	18	49	140	430	1 300	3 700
GMP's extended g.c.d.	3	6	14	35	92	250	730	2 200	5 600
PARI/GP	0.68	1.1	2.8	8.5	28	99	360	1 300	4 800
Naive_Mul_Padic$_p$	4	7	15	35	95	300	1 000	3 700	14 000
Binary_Mul_Padic$_p$	9	21	44	93	200	420	920	2 000	4 800
mono Naive_Mul_Padic$_p$	6	10	20	95	160	280	540	1 200	2 800
mono Binary_Mul_Padic$_p$	6	10	20	110	170	320	660	1 500	3 600
blocks Naive_Mul_Padic$_p$	10	16	27	70	190	430	900	1 900	4 500
blocks Binary_Mul_Padic$_p$	10	16	27	65	180	410	900	2 000	4 500
Naive_Mul_Padic$_{p^{32}}$			6	22	42	88	200	500	1 500

TABLE 6.1. Divisions, for $p = 536870923$, in microseconds.

Proposition 7.1. *Assume that r is invertible modulo p, and let a be a relaxed invertible p-adic number in R_p such that a_0 is an r-th power modulo p. Then any r-th root b_0 of a_0 modulo p can be uniquely lifted into an r-th root b of a. Moreover, b is a recursive number for the equation*

$$b = \frac{a - b^r + rb_0^{r-1}b}{rb_0^{r-1}}. \tag{7.1}$$

The n first terms of b can be computed using

- $O(\log r \mathsf{M}(n) \log n)$ *operations in $\mathbb{K}$, if $R = \mathbb{K}[[x]]$ and $p = x$, or*
- $O(\log r \mathsf{I}(n \log p) \log n)$ *bit-operations, if $R = \mathbb{Z}$.*

Proof. Since r is invertible modulo p, the polynomial $x^r - a$ is separable modulo p. Any of its roots modulo p can be uniquely lifted into a root in R_p by means of the classical Newton operator [Lan02, Proposition 7.2].

Since a_0 is invertible, so is b_0. It is therefore clear that Equation (7.1) uniquely defines b, but it is not immediately clear how to evaluate it so that it defines a recursive number. For this purpose we rewrite b into $b_0 + c$, with c of valuation at least 1:

$$b = \frac{a - (b_0 + c)^r + rb_0^{r-1}(b_0 + c)}{rb_0^{r-1}} = \frac{a - \sum_{k=2}^{r} \binom{r}{k} b_0^{r-k} c^k + (r-1)\, b_0^r}{rb_0^{r-1}}.$$

Since r is invertible modulo p, we now see that it does suffice to know the terms to degree $n - 1$ of b in order to deduce b_n.

The latter expanded formula is suitable for an implementation but unfortunately the number of products to be performed grows linearly with r. Instead we modify the classical binary powering algorithm to compute the expression needed with $O(\log r)$ products only, as follows. In fact we aim at computing $\beta_r = (b_0 + c)^r - rb_0^{r-1}c - b_0^r$ in a way to preserve the recursiveness. We proceed by induction on r.

If $r = 1$, then $\beta_r = 0$. If $r = 2$ then $\beta_2 = c^2$. Assume that $r = 2h$, and that β_h is available by induction. From

$$\begin{aligned}\beta_h^2 &= (b_0 + c)^r + \left(hb_0^{h-1}c + b_0^h\right)^2 - 2\left(hb_0^{h-1}c + b_0^h\right)\left(\beta_h + hb_0^{h-1}c + b_0^h\right)\\ &= (b_0 + c)^r - \left(hb_0^{h-1}c + b_0^h\right)^2 - 2\left(hb_0^{h-1}c + b_0^h\right)\beta_h,\end{aligned}$$

we deduce that

$$\begin{aligned}\beta_r &= \beta_h^2 + \left(hb_0^{h-1}c + b_0^h\right)^2\\ &\quad + 2\left(hb_0^{h-1}c + b_0^h\right)\beta_h - rb_0^{r-1}c - b_0^r\\ &= \beta_h\left(\beta_h + 2\left(hb_0^{h-1}c + b_0^h\right)\right) + \left(hb_0^{h-1}c\right)^2.\end{aligned}$$

Since β_h and c have positive valuation, the recursiveness is well preserved through this intermediate expression.

On the other hand, if r is odd then we can write $r = h + 1$, with h even, and assume that β_h is available by induction. Then we have that:

$$\begin{aligned}\beta_r &= (b_0 + c)\beta_h + (b_0 + c)(hb_0^{h-1}c + b_0^h) - (h + 1)b_0^h c - b_0^{h+1}\\ &= (b_0 + c)\beta_h + hb_0^{h-1}c^2.\end{aligned}$$

Again the recursiveness is well preserved through this intermediate expression. The equation of b can finally be evaluated using $O(\log r)$ products and one division. By [Hoe02, Section 4.3.2, Theorem 6], this concludes the proof for power series. By Propositions 3.2 and 6.2, we also obtain the desired result for p-adic integers. □

For the computation of the r-th root in $\mathbb{Z}/p\mathbb{Z}$, we have implemented the algorithms of [GG03, Theorems 14.4 and 14.9]: each extraction can be done with $\tilde{O}(r \log p)$ bit-operations in average, with a randomized algorithm. This is not the bottleneck for our purpose, so we will not discuss this aspect longer in this paper.

Remark. Notice that (p) is not assumed to be prime in Proposition 7.1. Therefore, if we actually have an r-th root b of a modulo p^k, then b can be seen as a p^k-recursive number, still using Equation (7.1). Hence, one can directly apply the monoblock strategy of Section 4.2 to perform internal computations modulo p^k.

7.2. *p*-th roots. If $\mathbb{K}$ is a field of characteristic p, then $f \in \mathbb{K}[[x]]$ is a p-th power if, and only if, $f \in \mathbb{K}^p[[x^p]]$. If it exists, the p-th root of a power series is unique. Here, $\mathbb{K}^p$ represents the subfield of the p-th powers of $\mathbb{K}$. By the way, let us mention that, for a general effective field $\mathbb{K}$, Fröhlich and Shepherdson have shown that testing if an element is a p-th power is not decidable [FS56, Section 7] (see also the example in [Gat84, Remark 5.10]).

In general, for p-adic numbers, an r-th root extraction can be almost as complicated as the factorization of a general polynomial in $R_p[x]$. For instance, with $R = \mathbb{Z}[\sqrt{2}]$ and $p = \sqrt{2}$ we have that $r = 2 = p^2$ has valuation 2 in R_p. We will not cover such a general situation. We will only consider the case of the p-adic integers, that is for when $R = \mathbb{Z}$ and p is prime.

From now on, let a denote a p-adic integer in $\mathbb{Z}_p$ from which we want to extract the p-th root (if it exists). If the valuation of a is not a multiple of p, then a is not a p-th power. If it is a multiple of p, then we can factor out $p^{\operatorname{val} a}$ and assume that a has valuation 0. The following lemma is based on classical techniques, we briefly recall its proof for completeness:

Lemma 7.1. *Assume that p is prime, and let $a \in \mathbb{Z}_p$ be invertible.*

- *If $p \geqslant 3$, then a is a p-th power if, and only if, $a_0 + pa_1 = a_0^p$ modulo p^2. In this case there exists only one p-th root.*
- *If $p = 2$, then a is a p-th power if, and only if, $a_1 = a_2 = 0$. In this case there exist exactly two square roots.*

Proof. If $a = b^p$ in $\mathbb{Z}_p$ then $b_0 = a_0$. After the translation $x = a_0 + y$ in $x^p - a = 0$, we focus on the equation $(b_0 + y)^p - a = 0$, which expands to

$$h(y) = y^p + \sum_{i=1}^{p-1} \binom{p}{i} b_0^{p-i} y^i - (a - b_0^p) = 0. \tag{7.2}$$

For any $i \in \{1, \dots, p-1\}$, the coefficient $\binom{p}{i}$ has valuation at least one because p is prime. Reducing the latter equation modulo p^2, it is thus necessary that $a_0 + pa_1 = b_0^p$ modulo p^2.

Assume now that $a_0 + pa_1 = b_0^p$ holds modulo p^2. After the change of variables y by pz and division by p^2, we obtain

$$\tilde{h}(z) = p^{p-2} z^p + \sum_{i=2}^{p-1} \binom{p}{i} b_0^{p-i} p^{i-2} z^i + b_0^{p-1} z - \frac{a - b_0^p}{p^2} = 0. \tag{7.3}$$

We distinguish two cases: $p \geqslant 3$ and $p = 2$.

If $p \geqslant 3$, then any root of $\tilde{h}$ must be congruent to $(a - b_0^p)/(b_0^{p-1} p^2)$ modulo p. Since $\tilde{h}'\left((a - b_0^p)/(b_0^{p-1} p^2)\right) = b_0^{p-1}$ has valuation 0, the Newton operator again ensures that $\tilde{h}$ has exactly one root [Lan02, Proposition 7.2].

If $p = 2$, then $\tilde{h}(z)$ rewrites into $z^2 + b_0 z - \frac{a - b_0^2}{p^2} = z^2 + z - \frac{a-1}{4}$. Since $\widetilde{h'}(z) = b_0 \bmod p = 1 \bmod 2$, any root of $\tilde{h}$ modulo 2 can be lifted into a unique root of $\tilde{h}$ in $\mathbb{Z}_2$. The possible roots being 0 and 1, this gives the extra condition $a_2 = 0$. □

7.3. Square roots in base 2. In the following proposition we show that the square root of a 2-adic integer can be expressed into a recursive number that can be computed with essentially one relaxed product.

Proposition 7.2. *Let a be a relaxed 2-adic integer in $\mathbb{Z}_2$ with $a_0 = 1$ and $a_1 = a_2 = 0$. Let b be a square root of a, with $b_0 = 1$ and b_1 being 0 or 1, and let $c = (b - b_0 - 2b_1)/4$, and $\tilde{a} = (a - (b_0 + 2b_1)^2)/8$. Then c is a recursive number with initial condition $c_0 = \tilde{a}_0$ and equation*

$$c = \frac{\tilde{a} - 2c^2}{b_0 + 2b_1}. \tag{7.4}$$

In addition, the n first terms of b can be computed with $O(\mathsf{I}(n \log p) \log n)$ bit-operations.

Proof. Equation (7.4) simply follows from

$$(b_0 + 2b_1 + 4c)^2 - a = (b_0 + 2b_1)^2 + 8(b_0 + 2b_1)c + 16c^2 - a = 0.$$

The cost is a consequence of Propositions 3.2 and 6.2. □

Remark. As in the preceding regular case, we can see c as a p^k-recursive number as soon as c is known modulo p^k. In fact letting $C = C_0 + \tilde{C}$, with $C_0 = C \bmod p^k$, Equation (7.4) rewrites into

$$\begin{aligned}(b_0 + 2b_1)C &= \tilde{a} - 2(C_0 + (C - C_0))^2 \\ &= \tilde{a} - 2C_0^2 - 4C_0(C - C_0) - 2(C - C_0)^2,\end{aligned}$$

which gives

$$(b_0 + 2b_1 + 4C_0)C = \tilde{a} + 2C_0^2 - 2(C - C_0)^2.$$

The latter equation implies that C is p^k-recursive, so that we can naturally benefit of the monoblock strategy from Section 4.2.

7.4. p-th roots in base p. In this subsection we assume that p is an odd prime integer. We will show that the p-th root is recursive and can be computed using similar but slightly more complicated formulas than in the regular case.

Proposition 7.3. *Let a be an invertible relaxed p-adic integer in $\mathbb{Z}_p$ such that $a = a_0^p \bmod p^2$. Let b denote the p-th root of a, with $b_0 = a_0$, let $\tilde{a} = (a - (b_0 + pb_1)^p)/p^2$, and let $c = (b - b_0)/p$. Then c is a recursive number with initial condition $c_0 = b_1$ and equation*

$$c = c_0 + \frac{\tilde{a} - \gamma_p}{(b_0 + pc_0)^{p-1}}, \tag{7.5}$$

where

$$\gamma_r = \frac{(b_0 + pc)^r - rp(b_0 + pc_0)^{r-1}(c - c_0) - (b_0 + pc_0)^r}{p^2}, \text{ for all } r \geqslant 0.$$

The n first terms of b can be computed with $O(\log p \mathsf{I}(n \log p) \log n)$ bit-operations.

Proof. As a shorthand we let $\beta = b_0 + pc_0$ and $d = c - c_0$. Equation (7.5) simply follows from

$$\begin{aligned} b^p - a &= (\beta + pd)^p - a \\ &= p^2\gamma_p + p^2\beta^{p-1}d + \beta^p - a \\ &= p^2\gamma_p + p^2\beta^{p-1}d - p^2\tilde{a} = 0. \end{aligned}$$

Since

$$\gamma_p = \sum_{i=2}^{p} \binom{p}{i} \beta^{p-i} p^{i-2} d^i,$$

and since d has positive valuation, Equation (7.5) actually defines c as a recursive number.

As in the regular case, we need to provide an efficient way to compute γ_r. We proceed by induction on r. If $r = 1$, then $\gamma_r = 0$. If $r = 2$, then $\gamma_r = d^2$, which preserves the recursiveness. Assume now that $r = 2h$ and that γ_h is available by induction. From

$$\begin{aligned} (p^2\gamma_h)^2 &= (\beta + pd)^r + \left(hp\beta^{h-1}d + \beta^h\right)^2 \\ &\quad - 2\left(hp\beta^{h-1}d + \beta^h\right)\left(p^2\gamma_h + hp\beta^{h-1}d + \beta^h\right) \\ &= (\beta + pd)^r - \left(hp\beta^{h-1}d + \beta^h\right)^2 - 2\left(hp\beta^{h-1}d + \beta^h\right)p^2\gamma_h, \end{aligned}$$

we deduce that

$$\begin{aligned} p^2\gamma_r &= (p^2\gamma_h)^2 + \left(hp\beta^{h-1}d + \beta^h\right)^2 \\ &\quad + 2\left(hp\beta^{h-1}d + \beta^h\right)p^2\gamma_h - rp\beta^{r-1}d - \beta^r \\ &= p^2\gamma_h\left(p^2\gamma_h + 2\left(hp\beta^{h-1}d + \beta^h\right)\right) + \left(hp\beta^{h-1}d\right)^2, \end{aligned}$$

whence

$$\gamma_r = \gamma_h\left(p^2\gamma_h + 2\left(hp\beta^{h-1}d + \beta^h\right)\right) + \left(h\beta^{h-1}d\right)^2.$$

Since γ_h and d have positive valuation, the recursiveness is well preserved through this intermediate expression.

On the other hand, if r is odd, then we can write $r = h + 1$, with h even, and assume that γ_h is available by induction. Then we have that:

$$\begin{aligned} p^2\gamma_r &= p^2(\beta + pd)\gamma_h + (\beta + pd)(hp\beta^{h-1}d + \beta^h) - (h+1)p\beta^h d - \beta^{h+1} \\ &= p^2(\beta + pd)\gamma_h + hp^2\beta^{h-1}d^2, \end{aligned}$$

whence

$$\gamma_r = (\beta + pd)\gamma_h + h\beta^{h-1}d^2,$$

which again preserves the recursiveness. Finally the equation of d can be evaluated with $O(\log r)$ products and one division, which concludes the proof by Propositions 3.2 and 6.2. □

Remark. As in the regular case, we can see c as a p^k-recursive number as soon as c is known modulo p^k. In fact, letting $C = C_0 + D$, with $C_0 = C \bmod p^k$, Equation (7.5) rewrites into

$$C = C_0 + \frac{\tilde{A} - \Gamma_p}{(b_0 + pC_0)^{p-1}},$$

where $\tilde{A} = (a - (b_0 + pC_0)^p)/p^2$, and

$$\Gamma_r = \frac{(b_0 + pC)^r - rp(b_0 + pC_0)^{r-1}(C - C_0) - (b_0 + pC_0)^r}{p^2}, \text{ for all } r \geqslant 0.$$

Notice that division by p^2 in base p^k, with $k > 2$, is equivalent to multiplication by p^{k-2} and division by p^k, which corresponds to a simple shift. Then Γ_p can be computed by recurrence with the same formula as γ_p, *mutatis mutandis*. In this way C is p^k-recursive, so that we can naturally benefit of the monoblock strategy of Section 4.2.

7.5. Timings. In Table 7.1, we give the computation time of the square root using our fast relaxed product of Section 4.3 that has been reported in Table 4.3. Since, in terms of performances, the situation is very similar to the division, we only compare to the zealous implementation in Pari/Gp version 2.3.5.

n	8	16	32	64	128	256	512	1024	2048
blocks Binary_Mul_Padic$_p$	14	22	39	91	230	520	1 100	2 400	5 400
Pari/Gp	5	11	28	74	210	670	2 300	8 100	30 000

Table 7.1. Square root, for $p = 536870923$, in microseconds.

8. Conclusion

From more than a decade a major stream in complexity theory for computer algebra has spread the idea that high level algorithms must be parameterized in terms of a small number of elementary operations (essentially integer, polynomial and matrix multiplication), so that the main goal in algorithm design consists in reducing as fast as possible to these operations. Although many asymptotically efficient algorithms have been developed along these lines, an overly doctrinaire application of this philosophy tends to be counterproductive.

For example, when it comes to computations in completions, we have seen that there are two general approaches: Newton's method and the relaxed (or lazy) approach. It is often believed that Newton's method is simply the best, because it asymptotically leads to the same complexity as integer or polynomial multiplication. However, this "reduction" does not take into account possible sparsity in the data, non asymptotic input sizes, more involved types of equations (such as partial differential equations), etc.

In this paper, we have demonstrated that, in the area of computations with p-adic numbers, the relaxed approach can be more efficient than methods based on Newton iteration. The gains are sometimes important: in Tables 5.1 and 5.2, we have shown that Hensel lifting in high dimensions can become more than 100 times faster, when using the relaxed approach. At other moments, we were ourselves surprised: in Table 6.1, we see that, even for the division of p-adic numbers, a naive implementation of the relaxed product yields better performances than a straightforward use of GMP, whenever p is sufficiently large.

Of course, the detailed analysis of the mutual benefits of both approaches remains an interesting subject. On the one hand, Newton iteration can be improved using blockwise techniques [Ber00, Hoe10]. On the other hand, the relaxed implementation can be improved for small sizes by ensuring a better transition between hardware and long integers, and massive inlining. At huge precisions, the recursive blockwise technique from [Hoe07b] should also become useful. Finally, "FFT-caching" could still be used in a more systematic way, and in particular for the computation of squares.

To conclude our comparison between Newton iteration and the relaxed approach, we would like to stress that, under most favourable circumstances, Newton iteration can only be hoped to be a small constant times faster than the relaxed approach, since the overhead $O(\log n)$ of relaxed multiplication should really be read as $(1/2)\log(n/128)$ or less. In other extreme cases, Newton iteration is several hundreds times slower, or even does not apply at all (e.g. for the resolution of partial differential equations).

Let us finally mention that the relaxed resolution of recursive systems of equations has been extended to more general systems of implicit equations [Hoe09]. The computation of such local solutions is the central task of the polynomial system solver called KRONECKER (see [DL08] for an introduction). We are confident that the results of [Hoe09], which were presented in the power series context, extend to more general completions, and that the relaxed model will lead to an important speed-up.

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

References

[Ber00] D. Bernstein, *Removing redundancy in high precision Newton iteration.* Available from `http://cr.yp.to/fastnewton.html`, 2000.

[BK78] R. P. Brent and H. T. Kung, *Fast algorithms for manipulating formal power series.* Journal of the ACM **25** (1978), 581–595.

[Bre76] R. P. Brent, *The complexity of multiprecision arithmetic.* In R. S. Anderssen and R. P. Brent, editors, Complexity of computational problem solving, 126–165. University of Queensland Press, Brisbane, 1976.

[CC90] D. V. Chudnovsky and G. V. Chudnovsky, *Computer algebra in the service of mathematical physics and number theory (Computers in mathematics, Stanford, Ca, 1986).* In Lect. Notes in Pure and Applied Math. **125**, 109–232. Dekker, New-York, 1990.

[CK91] D. G. Cantor and E. Kaltofen, *On fast multiplication of polynomials over arbitrary algebras.* Acta Informatica **28** (1991), 693–701.

[DL08] C. Durvye and G. Lecerf, *A concise proof of the Kronecker polynomial system solver from scratch.* Expositiones Mathematicae **26(2)** (2008), 101 – 139.

[DS04] S. De Smedt, *p-adic arithmetic.* The Mathematica Journal **9(2)** (2004), 349–357.

[FS56] A. Fröhlich and J. C. Shepherdson, *Effective procedures in field theory.* Philos. Trans. Roy. Soc. London. Ser. A. **248** (1956), 407–432.

[Für07] M. Fürer, *Faster integer multiplication.* In Proceedings of the Thirty-Ninth ACM Symposium on Theory of Computing (STOC 2007), 57–66, San Diego, California, 2007.

[G+91] T. Granlund et al, *GMP, the GNU multiple precision arithmetic library.* Available from `http://www.swox.com/gmp`, 1991.

[Gat84] J. von zur Gathen, *Hensel and Newton methods in valuation rings.* Math. Comp., **42(166)** (1984), 637–661.

[GG03] J. von zur Gathen and J. Gerhard, *Modern computer algebra.* Cambridge University Press, Cambridge, second edition, 2003.

[H+02] J. van der Hoeven et al, *Mathemagix.* Available from `http://www.mathemagix.org`, 2002.

[HH09] W. B. Hart and D. Harvey, *FLINT 1.5.1: Fast library for number theory.* Available from `http://www.flintlib.org`, 2009.

[Hoe97] J. van der Hoeven, *Lazy multiplication of formal power series.* In W. W. Küchlin, editor, Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (ISSAC 1997), 17–20, Maui, Hawaii, July 1997.

[Hoe99] J. van der Hoeven, *Fast evaluation of holonomic functions.* Theoretical Computer Science, **210** (1999), 199–215.

[Hoe01] J. van der Hoeven, *Fast evaluation of holonomic functions near and in singularities.* J. Symbolic Comput. **31** (2001), 717–743.

[Hoe02] J. van der Hoeven, *Relax, but don't be too lazy.* J. Symbolic Comput. **34(6)** (2002), 479–542.

[Hoe07a] J. van der Hoeven, *Efficient accelero-summation of holonomic functions.* J. Symbolic Comput. **42(4)** (2007), 389–428.

[Hoe07b] J. van der Hoeven, *New algorithms for relaxed multiplication.* J. Symbolic Comput., **42(8)** (2007), 792–802.

[Hoe09] J. van der Hoeven, *Relaxed resolution of implicit equations.* Technical report, HAL, 2009. `http://hal.archives-ouvertes.fr/hal-00441977/fr/`.

[Hoe10] J. van der Hoeven, *Newton's method and FFT trading.* J. Symbolic Comput. **45(8)** (2010), 857–878.

[Kat07] S. Katok, *p-adic analysis compared with real.* Student Mathematical Library **37**. American Mathematical Society, Providence, RI, 2007.

[Kob84] N. Koblitz, *p-adic numbers, p-adic analysis, and zeta-functions.* Graduate Texts in Mathematics **58**. Springer-Verlag, New York, second edition, 1984.

[Lan02] S. Lang, *Algebra.* Graduate Texts in Mathematics **211**. Springer-Verlag, third edition, 2002.

[PAR08] The PARI Group, Bordeaux, *PARI/GP, version 2.3.5*, 2008. Available from `http://pari.math.u-bordeaux.fr/`.

[S+09] W. A. Stein et al, *Sage Mathematics Software (Version 4.2.1)*. The Sage Development Team, 2009. Available from `http://www.sagemath.org`.

[SS71] A. Schönhage and V. Strassen, *Schnelle Multiplikation grosser Zahlen*. Computing **7** (1971), 281–292.

[Wan84] P. S. Wang, *Implementation of a p-adic package for polynomial factorization and other related operations*. In EUROSAM 84 (Cambridge, 1984), Lecture Notes in Comput. Sci. **174**, 86–99. Springer, Berlin, 1984.

Jérémy Berthomieu
Laboratoire d'Informatique
UMR 7161 CNRS
École polytechnique
Route de Saclay
91128 Palaiseau Cedex
France
E-mail: `berthomieu@lix.polytechnique.fr`
URL: `http://www.lix.polytechnique.fr/~berthomieu`

Joris van der Hoeven
Laboratoire d'Informatique
UMR 7161 CNRS
École polytechnique
Route de Saclay
91128 Palaiseau Cedex
France
E-mail: `vdhoeven@lix.polytechnique.fr`
URL: `http://www.lix.polytechnique.fr/~vdhoeven`

Grégoire Lecerf
Laboratoire d'Informatique
UMR 7161 CNRS
École polytechnique
Route de Saclay
91128 Palaiseau Cedex
France
E-mail: `Gregoire.Lecerf@math.cnrs.fr`
URL: `http://lecerf.perso.math.cnrs.fr`