ZpL: a p-adic precision package

Xavier Caruso
Université Rennes 1;
xavier.caruso@normalesup.org

David Roe
MIT;
roed@mit.edu

Tristan Vaccon
Université de Limoges;
tristan.vaccon@unilim.fr

ABSTRACT

We present a new package ZpL for the mathematical software system SageMath. It implements a sharp tracking of precision on p-adic numbers, following the theory of ultrametric precision introduced in [4]. The underlying algorithms are mostly based on automatic differentiation techniques. We introduce them, study their complexity and discuss our design choices. We illustrate the benefits of our package (in comparison with previous implementations) with a large sample of examples coming from linear algebra, commutative algebra and differential equations.

CCS CONCEPTS

•**Computing methodologies** → **Algebraic algorithms;**

KEYWORDS

Algorithms, p-adic precision, Automatic Differentiation

1 INTRODUCTION

When computing with real and p-adic fields, exact results are usually impossible, since most elements have infinite decimal or p-adic expansions. Working with these fields thus requires an analysis of how precision evolves through the sequence of steps involved in carrying out a computation. In this paper, we describe a package for computing with p-adic rings and fields [13], based on a series of papers by the same authors [4, 5, 6, 7]. The core of the package is a method for tracking precision using p-adic lattices which can yield dramatically more precise results, at the cost of increased runtime and memory usage.

The standard method for handling precision when computing with real numbers is floating point arithmetic, which may also be used in p-adic computation. At a given precision level, a finite set of representable numbers are chosen, and arithmetic operations are defined to give a representable number that is close to the true result [1]. Floating point arithmetic has the benefit of efficient arithmetic operations, but users are responsible for tracking the precision of the results. Numerically unstable algorithms can lead to very inaccurate answers [9].

If provably correct results are desired, interval arithmetic provides an alternative to floating point. Instead of just tracking an approximation to the answer, the package also tracks a radius within which the true result lies. This method is commonly used for p-adic computations since the ultrametric property of p-adic fields frequently keeps the radius small. Computations remain fairly efficient with this approach, but numerical instability can still lead to dramatic losses in precision (see §2 for many examples). Tracking the precision of multiple variables concurrently, the set of possible true values associated to an inexact value takes the form of an ellipsoid with axes parallel to the coordinate axes.

For better control of precision, we may allow arbitrary axes. This change would have little utility for real numbers, since such ellipsoids are not preserved by most functions. For p-adic fields, in contrast, differentiable maps with surjective differential will send sufficiently small ellipsoids to other ellipsoids. From an algebraic perspective, these ellipsoids are just cosets of a lattice H inside a p-adic vector space, and the main result of [4] (see also Proposition 3.1 below) describes how the image of such a coset under a map f is given exactly by applying the differential of f to H.

In this paper, we describe an implementation of this idea in SageMath [12]. Rather than attaching a precision to each element, we store the precision of many elements together by tracking a precision module for the whole collection of variables. As variables are created and destroyed, we update a matrix whose rows represent the vectors in the module. Information about the precision of elements is extracted from the matrix as necessary.

The article is structured as follows. In §2 we provide a demonstration of the package, showing how it can provide more precise answers than the traditional methods for tracking p-adic precision. In particular, §2.1 describes elementary arithmetic and the SOMOS-4 sequence, §2.2 gives examples from linear algebra, §2.3 examples using polynomials, and §2.4 examples of differential equations.

In §3 we give more details on the implementation. §3.1 contains a brief overview on the theory of p-adic precision of [4]. In the next two subsections, we explain in more details how ZpLC and ZpLF work. §3.2 is devoted to the implementation of automatic differentiation leading to the actual computation of the module that models the precision. In §3.3, we explain how precision on any individual number can be recovered and discuss the validity of our results. The complexity overhead induced by our package is analyzed in §3.4.

Finally, §4 contains a discussion of how we see this package fitting into the existing p-adic implementations. While these methods do introduce overhead, they are well suited to exploring precision behavior when designing algorithms, and can provide hints as to when further precision analysis would be useful.

2 SHORT DEMONSTRATION

The first step is to define the parents: the rings of p-adic numbers we will work with.

```
In:  Z2 = ZpXX(2, print_mode='digits')
     Q2 = QpXX(2, print_mode='digits')
```


Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

ZpXX is a generic notation for ZpCR, ZpLC and ZpLF. The first, ZpCR, is the usual constructor for p-adic parents in SageMath. It tracks precision using interval arithmetic. On the contrary ZpLC and ZpLF are provided by our package. In the sequel, we will compare the outputs provided by each parent. Results for ZpLF are only displayed when they differ from ZpLC.

2.1 Elementary arithmetic

We begin our tour of the features of the ZpL package with some basic arithmetic computations. We first pick some random element x. The function `random_element` is designed so that it guarantees that the picked random element is the same for each constructor ZpCR, ZpLC and ZpLF.

```
   In:  x = random_element(Z3, prec=5); x
ZpCR:   ...11111
ZpLC:   ...11111
```

Multiplication by p (here 3) is a shift on the digits and thus leads to a gain of one digit in absolute precision. In the example below, we observe that when this multiplication is split into several steps, ZpCR does not see the gain of precision while ZpL does.

```
   In:  3*x                    In:  x + x + x
ZpCR:   ...111110           ZpCR:    ...11110
ZpLC:   ...111110           ZpLC:   ...111110
```

The same phenomenon occurs for multiplication.

```
   In:  x^3                    In:  x * x * x
ZpCR:   ...010101           ZpCR:    ...10101
ZpLC:   ...010101           ZpLC:   ...010101
```

ZpL is also well suited for working with coefficients with unbalanced precision.

```
   In:  x = random_element(Z2, prec=10)
        y = random_element(Z2, prec=5)
   In:  u, v = x+y, x-y
        u, v
ZpCR:   (...10111, ...01111)
ZpLC:   (...10111, ...01111)
```

Now, let us compute $u + v$ and compare it with $2x$ (observe that they should be equal).

```
   In:  u + v                  In:  2*x
ZpCR:   ...00110            ZpCR:   ...00110100110
ZpLC:   ...00110100110      ZpLC:   ...00110100110
```

Again ZpCR does not output the optimal precision when the computation is split into several steps whereas ZpL does. Actually, these toy examples illustrate quite common situations which often occur during the execution of many algorithms. For this reason, interval arithmetic often overestimates the losses of precision. Roughly speaking, the aim of our package is to "fix this misfeature". In the next subsections, we present a bunch of examples showing the benefit of ZpL in various contexts.

SOMOS 4. A first example is the SOMOS-4 sequence. It is defined by the recurrence:

$$u_{n+4} = \frac{u_{n+1}u_{n+3} + u_{n+2}^2}{u_n}$$

and is known for its high numerical instability (see [4]). Nevertheless, the ZpL package saves precision even when using a generic unstable implementation of the SOMOS iteration.

```
   In:  def somos4(u0, u1, u2, u3, n):
            a, b, c, d = u0, u1, u2, u3
            for _ in range(4, n+1):
                a, b, c, d = b, c, d, (b*d + c*c) / a
            return d
   In:  u0 = u1 = u2 = Z2(1,15); u3 = Z2(3,15)
        somos4(u0, u1, u2, u3, 18)
ZpCR:   ...11
ZpLC:   ...100000000000111
   In:  somos4(u0, u1, u2, u3, 100)
ZpCR:   PrecisionError: cannot divide by something
        indistinguishable from zero.
ZpLC:   ...001001001110001
```

2.2 Linear algebra

Many generic algorithms of linear algebra lead to quite important instability when they are used with p-adic numbers. In many cases, our package ZpL rubs this instability without having to change the algorithm, nor the implementation.

Matrix multiplication. As revealed in [5], a first simple example where instability appears is simply matrix multiplication. This might be surprising because no division occurs in this situation. Observe nevertheless the difference between ZpCR and ZpLC.

```
   In:  MS = MatrixSpace(Z2,2)
        M = random_element(MS, prec=5)
        for _ in range(25):
            M *= random_element(MS, prec=5)
        M
ZpCR:   [0 0]
        [0 0]
ZpLC:   [...100000000000   ...1000000000]
        [   ...010000000     ...00100000]
```

On the aforementioned example, we notice that ZpCR is unable to decide whether the product vanishes or not. Having good estimates on the precision is therefore very important in such situations.

Characteristic polynomials. Characteristic polynomials are notoriously hard to compute [5, 7]. We illustrate this with the following example (using the default algorithm of SageMath for the computation of the characteristic polynomial, which is a division free algorithm in this setting) :

```
   In:  M = random_element(MatrixSpace(Q2,3), prec=10)
        M.determinant()
ZpCR:   ...010000010
ZpLC:   ...010000010
   In:  M.charpoly()
ZpCR:   ...00000000000000000001*x^3 +
        ...1001011.011*x^2 + ...0111.01*x + 0
ZpLC:   ...00000000000000000001*x^3 +
        ...1001011.011*x^2 + ...11100111.01*x +
        ...010000010
```

We observe that ZpLC can guarantee 4 more digits on the x coefficient. Moreover, it recovers the correct precision on the constant coefficient (which is the determinant) whereas ZpCR is confused and cannot even certify that it does not vanish.

2.3 Commutative algebra

Our package can be applied to computation with p-adic polynomials.

Euclidean algorithm. A natural example is that of the computation of GCD, whose stability has been studied in [3]. A naive implementation of the Euclidean algorithm can produce different behavior depending on the type of implementation of the field of p-adic coefficients.

```
 In:  S.<x> = PolynomialRing(Z2)
      P = random_element(S, degree=10, prec=5)
      Q = random_element(S, degree=10, prec=5)
      D = x^5 + random_element(S, degree=4, prec=8); D
ZpCR: ...00000000000000000001*x^5 + ...11111010*x^4 +
      ...10000000*x^3 + ...11001111*x^2 +
      ...10000110*x + ...11100010
ZpLC: ...00000000000000000001*x^5 + ...11111010*x^4 +
      ...10000000*x^3 + ...11001111*x^2 +
      ...10000110*x + ...11100010
 In:  def euclidean(A,B):
          while B != 0:
              A, B = B, A % B
          return A.monic()
      euclidean(D*P, D*Q)
ZpCR: 0*x^9 + ...1*x^8 + 0*x^7 + 0*x^6 + 0*x^5 +
      0*x^4 + 0*x^3 + ...1*x^2 + ...10*x + ...10
ZpLC: ...00000000000000000001*x^5 + ...11111010*x^4 +
      ...10000000*x^3 + ...11001111*x^2 +
      ...10000110*x + ...11100010
```

With high probability, P and Q are coprime, implying that the gcd of DP is DQ is D. However, we observe that ZpCR output a quite different result. The point is that, in the ZpCR case, Euclidean algorithm stops prematurely because the test `B != 0` fails too early due to the lack of precision.

Gröbner bases. Our package can be applied on complex computations like that of Gröbner bases using generic Gröbner bases algorithms.

```
 In:  R.<x,y,z> = PolynomialRing(Q2, order='invlex')
      F = [ Q2(2,10)*x + Q2(1,10)*z,
            Q2(1,10)*x^2 + Q2(1,10)*y^2 - Q2(2,10)*z^2,
            Q2(4,10)*y^2 + Q2(1,10)*y*z + Q2(8,10)*z^2 ]
 In:  from sage.rings.polynomial.toy_buchberger\
          import buchberger_improved
      g = buchberger_improved(ideal(F))
      g.sort(); g
ZpCR: [x^3, x*y + ...1100010*x^2,
       y^2 + ...11001*x^2, z + ...0000000010*x]
ZpLC: [x^3, x*y + ...111100010*x^2,
       y^2 + ...1111111001*x^2, z + ...0000000010*x]
```

As we can see, some loss in precision occurs in the Buchberger algorithm and is avoided thanks to ZpL.

2.4 p-adic differential equations

In [10], the authors studied isogenies between elliptic curves over finite fields by applying the lattice precision model to p-adic differential equations. Specifically, they considered the equation $y'^2 = g \times h(y)$ with $g, h, y \in \mathbb{Z}_p[\![x]\!]$ such that $g(0) = h(0) = 1$ and $y(0) = 0$.

Their main result was that the intrinsic loss in precision when computing the coefficient x^n of y from g and h was in $\log_p(n)$ even though a naive analysis of the Newton method for solving the equation yield a loss in $\log_p(n)^2$.

We can reach this theoretical loss in precision using ZpL, while ZpCR does not perform as well. We apply `N` steps of the Newton method for $y' = g \times h(y)$ as described in [10], using a generic `Newton_Iteration_Solver(g,h,N)`.

```
 In:  S.<t> = PowerSeriesRing(Q2, 16)
      h = 1 + t + t^3
      y = t + t^2 * random_element(S, prec=10)
      g = y.derivative() / h(y)
      u = Newton_Iteration_Solver(g, h, 4); u[15]
ZpCR: ...1101
ZpLC: ...11011101
```

3 BEHIND THE SCENES

In this section, we explain how our package ZpL works and analyze its performance. The main theoretical result on which our package is based is the ultrametric precision theory developed in [4], which suggests tracking precision *via* lattices and differential computations. For this reason, our approach is very inspired by automatic differentiation techniques [11] and our implementation follows the usual operator overloading strategy. We will introduce two versions of our package, namely ZpLC and ZpLF: this former is safer while the latter is faster.

Remark about the naming. The letter `L`, which appears in the name of the package, comes from "lattices". The letters `C` (in ZpLC) and `F` (in ZpLF) stand for "cap" and "float" respectively.

3.1 The precision Lemma

In [4], we suggest the use of lattices to represent the precision of elements in $\mathbb{Q}_p$-vector spaces. This approach contrasts with the *coordinate-wise method* (of *e.g.* `Zp(5)`) that is traditionally used in SageMath where the precision of an element is specified by giving the precision of each coordinate separately and is updated after each basic operation.

Consider a finite dimensional normed vector space E defined over $\mathbb{Q}_p$. We use the notation $\|\cdot\|_E$ for the norm on E and $B_E^-(r)$ (resp. $B_E(r)$) for the open (resp. closed) ball of radius r centered at the origin. A *lattice* $L \subset E$ is a sub-$\mathbb{Z}_p$-module which generates E over $\mathbb{Q}_p$. Because of ultrametricity, the balls $B_E(r)$ and $B_E^-(r)$ are examples of lattices. Lattices can be thought of as special neighborhoods of 0, and therefore are good candidates to model precision data. Moreover, as revealed in [4], they behave quite well under (strictly) differentiable maps:

Proposition 3.1. Let E and F be two finite dimensional normed vector spaces over $\mathbb{Q}_p$ and $f : U \to F$ be a function defined on an open subset U of E. We assume that f is differentiable at some point $v_0 \in U$ and that the differential df_{v_0} is surjective. Then, for all $\rho \in (0, 1]$, there exists a positive real number δ such that, for all $r \in (0, \delta)$, any lattice H such that $B_E^-(\rho r) \subset H \subset B_E(r)$ satisfies:

$$f(v_0 + H) = f(v_0) + df_{v_0}(H). \tag{1}$$

This proposition enables the *lattice method* of tracking precision, where the precision of the input is specified as a lattice H and precision is tracked via differentials of the steps within a given algorithm. The equality sign in Eq. (1) shows that this method yields the optimum possible precision. We refer to [4, §4.1] for a more complete exposition.

3.2 Tracking precision

We now explain in more details the internal mechanisms ZpLC and ZpLF use for tracking precision.

In what follows, it will be convenient to use a notion of discrete time represented by the letter t. Rigorously, it is defined as follows: $t = 0$ when the p-adic ring ZpLC$(\cdots)$ or ZpLF$(\cdots)$ is created and increases by 1 each time a variable is created, deleted[1] or updated.

Let $\mathcal{V}_t$ be the set of alive variables at time t. Set $E_t = \mathbb{Q}_p^{\mathcal{V}_t}$; it is a finite dimensional vector space over $\mathbb{Q}_p$ which should be thought of as the set of all possible values that can be taken by the variables in $\mathcal{V}_t$. For $\mathtt{v} \in \mathcal{V}_t$, let $e_\mathtt{v} \in E_t$ be the vector whose coordinates all vanish except at position v which takes the value 1. The family $(e_\mathtt{v})_{\mathtt{v}\in\mathcal{V}_t}$ is obviously a basis of E_t; we will refer to it as the *canonical basis*.

3.2.1 The case of ZpLC. Following Proposition 3.1, the package ZpLC follows the precision by keeping track of a lattice H_t in E_t, which is a global object whose purpose is to model the precision on all the variables in $\mathcal{V}_t$ all together. Concretely, this lattice is represented by a matrix M_t in row-echelon form whose rows form a set of generators. Below, we explain how the matrices M_t are updated each time t increases.

Creating a variable. This happens when we encounter an instruction having one of the two following forms:

```
[Computation]    w = f(v_1, ..., v_n)
[New value]      w = R(value, prec)
```

In both cases, w is the newly created variable. The $\mathtt{v}_i$'s stand for already defined variables and f is some n-ary builtin function (in most cases it is just addition, subtraction, multiplication or division). On the contrary, the terms "value" and "prec" refer to user-specified constants or integral values which was computed earlier.

Let us first examine the first construction [Computation]. With our conventions, if t is the time just before the execution of the instruction we are interested in, the $\mathtt{v}_i$'s lie in $\mathcal{V}_t$ while w does not. Moreover $\mathcal{V}_{t+1} = \mathcal{V}_t \sqcup \{\mathtt{w}\}$, so that $E_{t+1} = E_t \oplus \mathbb{Q}_p e_\mathtt{w}$. The mapping taking the values of variables at time t to that at time $t+1$ is:

$$\begin{array}{rccc} F: & E_t & \longrightarrow & E_{t+1} \\ & \underline{x} & \mapsto & \underline{x} \oplus f(x_1, \ldots, x_n) \end{array}$$

where x_i is the $\mathtt{v}_i$-th coordinate of the vector $\underline{x}$. The Jacobian matrix of F at $\underline{x}$ is easily computed; it is the block matrix $J_{\underline{x}}(F) = \begin{pmatrix} I & L \end{pmatrix}$ where I is the identity matrix of size Card $\mathcal{V}_t$ and L is the column vector whose v-th entry is $\frac{\partial f}{\partial \mathtt{v}}(\underline{x})$ if v is one of the $\mathtt{v}_i$'s and 0 otherwise. Therefore, the image of H_t under $dF_{\underline{x}}$ is represented by the matrix $J_{\underline{x}}(F) \cdot M_t = \begin{pmatrix} M_t & C \end{pmatrix}$ where C is the column vector:

$$C = \sum_{i=1}^{n} \frac{\partial f}{\partial \mathtt{v}_i}(\underline{x}) \cdot C_i \tag{2}$$

where C_i is the column vector of M_t corresponding to the variable $\mathtt{v}_i$. Observe that the matrix $J_{\underline{x}}(F) \cdot M_t$ is no longer a square matrix; it has one extra column. This reflects the fact that $\dim E_{t+1} = \dim E_t + 1$. Rephrasing this in a different language, the image of H_t under $dF_{\underline{x}}$ is no longer a lattice in E_{t+1} but is included in an hyperplane.

The package ZpLC tackles this issue by introducing a cap: we do not work with $dF_{\underline{x}}(H_t)$ but instead define the lattice $H_{t+1} = dF_{\underline{x}}(H_t) \oplus p^{N_{t+1}}\mathbb{Z}_p e_\mathtt{w}$ where N_{t+1} is an integer, the so-called *cap*. Alternatively, one may introduce the map:

$$\begin{array}{rccc} \tilde{F}: & E_t \oplus \mathbb{Q}_p & \longrightarrow & E_{t+1} \\ & \underline{x} \oplus c & \mapsto & \underline{x} \oplus \big(f(x_1, \ldots, x_n) + c\big). \end{array} \tag{3}$$

The lattice H_{t+1} is then the image of $H_t \oplus p^{N_{t+1}}\mathbb{Z}_p$ under $d\tilde{F}_{(\underline{x},\star)}$ for any value of $\star$. The choice of the cap is of course a sensitive question. ZpLC proceeds as follows. When a ring is created, it comes with two constants (which can be specified by the user): a relative cap RELCAP and an absolute cap ABSCAP. With these predefined values, the chosen cap is:

$$N_{t+1} = \min\big(\text{ABSCAP},\ \text{RELCAP} + v_p(y)\big) \tag{4}$$

with $y = f(x_1, \ldots, x_n)$. In concrete terms, the lattice H_{t+1} is represented by the block matrix:

$$\begin{pmatrix} M_t & C \\ 0 & p^{N_{t+1}} \end{pmatrix}.$$

Performing row operations, we see then the entries of C can be reduced modulo $p^{N_{t+1}}$ without changing the lattice. In order to optimize the size of the objects, we perform this reduction and define M_{t+1} by:

$$M_{t+1} = \begin{pmatrix} M_t & C \bmod p^{N_{t+1}} \\ 0 & p^{N_{t+1}} \end{pmatrix}.$$

We observe in particular that M_{t+1} is still in row-echelon form.

Finally, we need to explain which value is set to the newly created variable w. We observe that it cannot be exactly $f(x_1, \ldots, x_n)$ because the latter is *a priori* a p-adic number which cannot be computed exactly. For this reason, we have to truncate it at some finite precision. Again we choose the precision $O(p^{N_{t+1}})$, *i.e.* we define $x_\mathtt{w}$ as $f(x_1, \ldots, x_n) \bmod p^{N_{t+1}}$. The congruence $\bar{x} \oplus f(x_1, \ldots, x_n) \equiv \bar{x} \oplus x_\mathtt{w} \pmod{H_{t+1}}$ (which holds thanks to the extra generator we have added) justifies this choice.

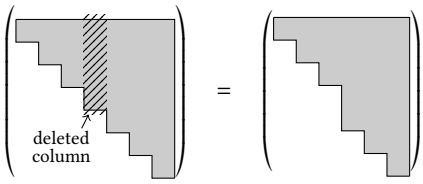
The second construction "w = R(value, prec)" is easier to handle since, roughly speaking, it corresponds to the case $n = 0$. In this situation, keeping in mind the cap, the lattice H_{t+1} is defined by $H_{t+1} = H_t + p^{\min(\text{prec}, N_{t+1})}\mathbb{Z}_p e_\mathtt{w}$ for the cap $N_{t+1} = \min\big(\text{ABSCAP},\ \text{RELCAP}+v_p(\text{value})\big)$. The corresponding matrix M_{t+1} is then given by:

$$M_{t+1} = \begin{pmatrix} M_t & 0 \\ 0 & p^{\min(\text{prec}, N_{t+1})} \end{pmatrix}.$$

Deleting a variable. Let us now examine the case where a variable w is deleted (or collected by the garbage collector). Just after the deletion, at time $t+1$, we then have $\mathcal{V}_{t+1} = \mathcal{V}_t \backslash \{\mathtt{w}\}$. Thus $E_t = E_{t+1} \oplus \mathbb{Q}_p e_\mathtt{w}$. Moreover, the deletion of w is modeled by the canonical projection $f : E_t \to E_{t+1}$. Since f is linear, it is its own differential (at each point) and we set $H_{t+1} = f(H_t)$. A matrix representing H_{t+1} is deduced from M_t by erasing the column corresponding to w. However the matrix we get this way is no longer in row-echelon form. We then need to re-echelonize it.

[1] The deletion can be explicit (through a call to the del operator) or implicit (handled by the garbage collector).

More precisely, the obtained matrix has this shape:

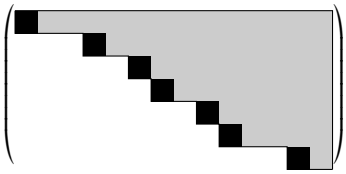


where a cell is colored when it can contain a non-vanishing entry. The top part of the matrix is then already echelonized, so that we only have to re-echelonize the bottom right corner whose size is the distance from the column corresponding to the erased variable to the end. Thanks to the particular shape of the matrix, the echelonization can be performed efficiently: we combine the first rows (of the bottom right part) in order to clear the first unwanted nonzero entry and then proceed recursively.

Updating a variable. Just like for creation, this happens when the program reaches an affectation "`w = ...`" where the variable `w` is already defined. This situation reduces to the creation of the temporary variable (the value of the right-hand-size), the deletion of the old variable `w` and a renaming. It can then be handled using the methods discussed previously.

3.2.2 The case of ZpLF. The way the package ZpLF tracks precision is based on similar techniques but differs from ZpLC in that it does not introduce a cap but instead allows H_t to be a sub-$\mathbb{Z}_p$-module of E_t of any codimension. This point of view is nice because it implies smaller objects and consequently leads to faster algorithms. However, it has a huge drawback; indeed, unlike lattices, submodules of E_t of arbitrary codimensions are *not* exact objects, in the sense that they cannot be represented by integral matrices in full generality. Consequently, they cannot be encoded on a computer. We work around this drawback by replacing everywhere exact p-adic numbers by floating point p-adic numbers (at some given precision) [2].

The fact that the lattice H_t can now have arbitrary codimension translates to the fact the matrix M_t can be rectangular. Precisely, we will maintain matrices M_t of the shape:



(5)

where only the colored cells may contain a nonzero value and the black cells —the so-called *pivots*— do not vanish. A variable whose corresponding column contains a pivot will be called a *pivot variable at time t*.

Creating a variable. We assume first that the newly created variable is defined through a statement of the form: "`w = f(v_1, ..., v_n)`". As already explained in the case of ZpLC, this code is modeled by the mathematical mapping:

$$\begin{array}{rccc} F: & E_t & \longrightarrow & E_{t+1} \\ & \underline{x} & \mapsto & \underline{x} \oplus f(x_1, \ldots, x_n). \end{array}$$

Here $\underline{x}$ represents the state of memory at time t, and x_i is the coordinate of $\underline{x}$ corresponding to the variable $\mathtt{v}_i$.

In the ZpLF framework, H_{t+1} is defined as the image of H_t under the differential $dF_{\underline{x}}$. Accordingly, the matrix M_{t+1} is defined as $M_{t+1} = \begin{pmatrix} M_t & C \end{pmatrix}$ where C is the column vector defined by Eq. (2). However, we insist on the fact that all the computations now take place in the "ring" of floating point p-adic numbers. Therefore, we cannot guarantee that the rows of M_{t+1} generate H_{t+1}. Nonetheless, they generate a module which is expected to be close to H_{t+1}.

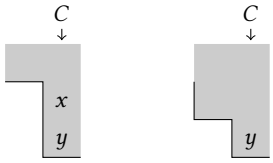
If `w` is created by the code "`w = R(value, prec)`", we define $H_{t+1} = H_t \oplus p^{\mathrm{prec}}\mathbb{Z}_p e_{\mathtt{w}}$ and consequently:

$$M_{t+1} = \begin{pmatrix} M_t & 0 \\ 0 & p^{\mathrm{prec}} \end{pmatrix}$$

If prec is $+\infty$ (or, equivalently, not specified), we agree that $H_{t+1} = H_t$ and $M_{t+1} = \begin{pmatrix} M_t & 0 \end{pmatrix}$.

Deleting a variable. As for ZpLC, the matrix operation implied by the deletion of the variable `w` is the deletion of the corresponding column of M_t. If `w` is not a pivot variable at time t, the matrix M_t keeps the form (5) after erasure; therefore no more treatment is needed in this case.

Otherwise, we re-echelonize the matrix as follows. After the deletion of the column $C_{\mathtt{w}}$, we examine the first column C which was located on the right of $C_{\mathtt{w}}$. Two situations may occur (depending on the fact that C was or was not a pivot column):



First case Second case

In the first case, we perform row operations in order to replace the pair (x, y) by $(d, 0)$ where d is an element of valuation $\min(v_p(x), v_p(y))$. Observe that y is necessarily nonzero in this case, so that d does not vanish as well. After this operation, we move to the next column and repeat the same process.

The second case is divided into two subcases. First, if y does not vanish, it can serve as a pivot and the obtained matrix has the desired shape. When this occurs, the echelonization stops. On the contrary, if $y = 0$, we just untint the corresponding cell and move to the next column without modifying the matrix.

3.3 Visualizing the precision

Our package implements several methods giving access to the precision structure. In the subsection, we present and discuss the most relevant features in this direction.

Absolute precision of one element. This is the simplest accessible precision datum. It is encapsulated in the notation when an element is printed. For example, the (partial) session:

```
  In:  v = Z2(173,10); v
ZpLC:  ...0010101101
```

indicates that the absolute precision on `v` is 10 since exactly 10 digits are printed. The method `precision_absolute` provides a more easy-to-use access to the absolute precision.

```
In:   v.precision_absolute()
ZpLC: 10
```

Both ZpLC and ZpLF compute the absolute precision of v (at time t) as the smallest valuation of an entry of the column of M_t corresponding to the variable v. Alternatively, it is the unique integer N for which $\pi_{\mathsf{v}}(H_t) = p^N\mathbb{Z}_p$ where $\pi_{\mathsf{v}} : E_t \to \mathbb{Q}_p$ takes a vector to its v-coordinate. This definition of the absolute precision sounds revelant because, if we believe that the submodule $H_t \subset E_t$ is supposed to encode the precision on the variables in $\mathcal{V}_t$, Proposition 3.1 applied with the mapping π_{v} indicates that a good candidate for the precision on e_{v} is $\pi_{\mathsf{v}}(H_t)$, that is $p^N\mathbb{Z}_p$.

About correctness. We emphasize that the absolute precision computed this way is *not* proved, either for ZpLF or ZpLC. However, in the case of ZpLC, one can be slightly more precise. Let $\mathcal{U}_t$ be the vector space of user-defined variables before time t and U_t be the lattice modeling the precision on them. The pair $(\mathcal{U}_t, U_t)$ is defined inductively as follows: we set $\mathcal{U}_0 = U_0 = 0$ and $\mathcal{U}_{t+1} = \mathcal{U}_t \oplus \mathbb{Q}_p e_{\mathsf{w}}$, $U_{t+1} = U_t \oplus p^{\text{prec}}\mathbb{Z}_p e_{\mathsf{w}}$ when a new variable w is created by "`w = R(value, prec)`"; otherwise, we put $\mathcal{U}_{t+1} = \mathcal{U}_t$ and $U_{t+1} = U_t$. Moreover the values entered by the user defines a vector (with integral coordinates) $\underline{u}_t \in \mathcal{U}_t$.

Similarly, in order to model the caps, we define a pair $(\mathcal{K}_t, K_t)$ by the recurrence $\mathcal{K}_{t+1} = \mathcal{K}_t \oplus \mathbb{Q}_p e_{\mathsf{w}}$, $K_{t+1} = K_t \oplus p^{N_{t+1}}\mathbb{Z}_p e_{\mathsf{w}}$ each time a new variable w is created. Here, the exponent N_{t+1} is the cap defined by Eq. (4). In case of deletion, we put $\mathcal{K}_{t+1} = \mathcal{K}_t$ and $K_{t+1} = K_t$.

Taking the compositum of all the functions $\tilde{F}$ (*cf* Eq. (3)) from time 0 to t, we find that the execution of the session until time t is modeled by a mathematical function $\Phi_t : \mathcal{U}_t \oplus \mathcal{K}_t \to E_t$. From the design of ZpLC, we deduce further that there exists a vector $\underline{k}_t \in K_t$ such that:

$$\Phi_t(\underline{u}_t \oplus \underline{k}_t) = \underline{x}_t \quad \text{and} \quad d\Phi_t(U_t \oplus K_t) = H_t$$

where the differential of Φ_t is taken at the point $\underline{u}_t \oplus \underline{k}_t$. Set $\Phi_{t,\mathsf{v}} = \pi_{\mathsf{v}} \circ \Phi_t$; it maps $\underline{u}_t \oplus \underline{k}_t$ to the v-coordinate $x_{t,\mathsf{v}}$ of $\underline{x}_t$ and satisfies $d\Phi_{t,\mathsf{v}}(U_t \oplus K_t) = \pi_{\mathsf{v}}(H_t) = p^N\mathbb{Z}_p$ where N is the value returned by `precision_absolute`. Thus, as soon as the assumptions of Proposition 3.1 are fulfilled, we derive $\Phi_{t,\mathsf{v}}\Big((\underline{u}_t + U_t) \oplus (\underline{k}_t + K_t)\Big) = x_{t,\mathsf{v}} + p^N\mathbb{Z}_p$. Noting that $k_t \in K_t$, we finally get:

$$\Phi_{t,\mathsf{v}}(\underline{u}_t + U_t) \subset \Phi_{t,\mathsf{v}}\Big((\underline{u}_t + U_t) \oplus K_t\Big) = x_{t,\mathsf{v}} + p^N\mathbb{Z}_p. \quad (6)$$

The latter inclusion means that the computed value $x_{t,\mathsf{v}}$ is accurate at precision $O(p^N)$, *i.e.* that the output absolute precision is correct.

Unfortunately, checking automatically the assumptions of Proposition 3.1 in full generality seems to be difficult, though it can be done by hand for many particular examples [4, 3, 10].

Remark 3.2. Assuming that Proposition 3.1 applies, the absolute precision computed as above is optimal if and only if the inclusion of (6) is an equality. Applying again Proposition 3.1 with the restricted mapping $\Phi_{t,\mathsf{v}} : \mathcal{U}_t \to \mathbb{Q}_p$ and the lattice U_t, we find that this happens if and only if $d\Phi_{t,\mathsf{v}}(U_t) = p^N\mathbb{Z}_p$.

Unfortunately, the latter condition cannot be checked on the matrix M_t (because of reductions). However it is possible (and easy) to check whether the weaker condition $d\Phi_{t,\mathsf{v}}(K_t) \subsetneq p^N\mathbb{Z}_p$. This checking is achieved by the method `is_precision_capped` (provided by our package) which returns true if $d\Phi_{t,\mathsf{v}}(K_t) = p^N\mathbb{Z}_p$. As a consequence, when this method answers FALSE, the absolute precision computed by the software is likely optimal.

Precision on a subset of elements. Our package implements the method `precision_lattice` through which we can have access to the joint precision on a set of variables: it outputs a matrix (in echelon form) whose rows generate a lattice representing the precision on the subset of given variables.

When the variables are "independent", the precision lattice is split and the method `precision_lattice` outputs a diagonal matrix:

```
In:   x = Z2(987,10); y = Z2(21,5)
In:   # We first retrieve the precision object
      L = Z2.precision()
In:   L.precision_lattice([x,y])
ZpLC: [1024    0]
      [   0   32]
```

However, after some computations, the precision matrix evolves and does not remain diagonal in general (though it is always triangular because it is displayed in row-echelon form):

```
In:   u, v = x+y, x-y
      L.precision_lattice([u,v])
ZpLC: [  32 2016]
      [   0 2048]
```

The fact that the precision matrix is no longer diagonal indicates that some well-chosen linear combinations of u and v are known with more digits than u and v themselves. In this particular example, the sum $u + v$ is known at precision $O(2^{11})$ while the (optimal) precision on u and v separately is only $O(2^5)$.

```
In:   u, v
ZpLC: (...10000, ...00110)
In:   u + v
ZpLC: ...11110110110
```

Diffused digits of precision. The phenomenon observed above is formalized by the notion of diffused digits of precision introduced in [5]. We recall briefly its definition.

Definition 3.3. Let E be a $\mathbb{Q}_p$-vector space endowed with a distinguished basis $(e_1, \dots, e_n)$ and write $\pi_i : E \to \mathbb{Q}_p e_i$ for the projections. Let $H \subset E$ be a lattice. The number of *diffused digits of precision* of H is the length of H_0/H where $H_0 = \pi_1(H) \oplus \cdots \oplus \pi_n(H)$.

If H represents the actual precision on some object, then H_0 is the smallest diagonal lattice containing H. It then corresponds to the maximal *coordinate-wise* precision we can reach on the set of n variables corresponding to the basis $(e_1, \dots, e_n)$.

The method `number_of_diffused_digits` computes the number of diffused digits of precision on a set of variables. Observe:

```
In:   L.number_of_diffused_digits([x,y])
ZpLC: 0
In:   L.number_of_diffused_digits([u,v])
ZpLC: 6
```

For the last example, we recall that the relevant precision lattice H is generated by the 2×2 matrix:

$$\begin{pmatrix} 2^5 & 2016 \\ 0 & 2^{11} \end{pmatrix}.$$

The minimal diagonal suplattice H_0 of H is generated by the scalar matrix $2^5 \cdot \mathrm{I}_2$ and contains H with index 2^6 in it. This is where the 6 digits of precision come from. There are easily visible here: the sum $u + v$ is known with 11 digits, that is exactly 6 more digits than the summands u and v.

3.4 Complexity

We now discuss the cost of the above operations. In what follows, we shall count operations in $\mathbb{Q}_p$. Although $\mathbb{Q}_p$ is an inexact field, our model of complexity makes sense because the size of the p-adic numbers we manipulate will all have roughly the same size: for ZpLF, it is the precision we use for floating point arithmetic while, for ZpLC, it is the absolute cap which was fixed at the beginning.

It is convenient to introduce a total order on $\mathcal{V}_t$: for $\mathtt{v}, \mathtt{w} \in \mathcal{V}_t$, we say that $\mathtt{v} <_t \mathtt{w}$ if v was created before w. By construction, the columns of the matrix M_t are ordered with respect to $<_t$. We denote by r_t (resp. c_t) the number of rows (resp. columns) of M_t. By construction r_t is also the cardinality of $\mathcal{V}_t$. We have $c_t \leq r_t$ and the equality always holds in the ZpLC case.

For $\mathtt{v} \in \mathcal{V}_t$, we define the *index* of v, denoted by $\mathrm{ind}_t(\mathtt{v})$ as the number of elements of $\mathcal{V}_t$ which are not greater than v. If we sort the elements of $\mathcal{V}_t$ by increasing order, v then appears in $\mathrm{ind}_t(\mathtt{v})$-th position. We also define the *co-index* of v by $\mathrm{coind}_t(\mathtt{v}) = r_t - \mathrm{ind}_t(\mathtt{v})$.

Similarly, for any variable $\mathtt{v} \in \mathcal{V}_t$, we define the *height* (resp. the *co-height*) of v at time t as the number of pivot variables w such that $\mathtt{w} \leq_t \mathtt{v}$ (resp. $\mathtt{w} >_t \mathtt{v}$). We denote it by $\mathrm{hgt}_t(\mathtt{v})$ (resp. by $\mathrm{cohgt}_t(\mathtt{v})$). Clearly $\mathrm{hgt}_t(\mathtt{v}) + \mathrm{cohgt}_t(\mathtt{v}) = c_t$. The height of v is the height of the significant part of the column of M_t which corresponds to v. In the case of ZpLC, all variables are pivot variables and thus $\mathrm{hgt}_t(\mathtt{v}) = \mathrm{ind}_t(\mathtt{v})$ and $\mathrm{cohgt}_t(\mathtt{v}) = \mathrm{coind}_t(\mathtt{v})$ for all v.

Creating a variable. With the notations of §3.2, it is obvious that creating a new variable w requires:

$$O\left(\sum_{i=1}^{n} \mathrm{hgt}_i(\mathtt{v}_i)\right) \subset O(n\, c_t)$$

operations in $\mathbb{Q}_p$. Here, we recall that n is the arity of the operation defining w. In most cases it is 2; thus the above complexity reduces to $O(c_t)$.

In the ZpLF context, c_t counts the number of user-defined variables. It is then expected to be constant (roughly equal to the size of the input) while running a given algorithm.

On the contrary, in the ZpLC context, c_t counts the number of variables which are alive at time t. It is no longer expected to be constant but evolves continuously when the algorithm runs. The tables of Figure 1 show the total number of created variables (which reflects the complexity) together with the maximum number of variables alive at the same time (which reflects the memory occupation) while executing two basic computations. The first one is the computation of the characteristic polynomial of a square matrix by the default algorithm used by SageMath for p-adic fields (which is a division-free algorithm of quartic complexity) while the second one is the computation of the gcd of two polynomials using a naive Euclidean algorithm (of quadratic complexity). We can observe that, for both of them, the memory usage is roughly equal to the square root of the complexity.

Dimension	2	5	10	20	50
Total	35	424	5 539	83 369	3 170 657
Simult.	17	65	225	845	5 101

Computation of characteristic polynomial

Degree	2	5	10	20	50	100
Total	54	130	332	1 036	4 110	10 578
Simult.	18	31	56	106	256	507

Naive Euclidean algorithm

Figure 1: Numbers of involved variables

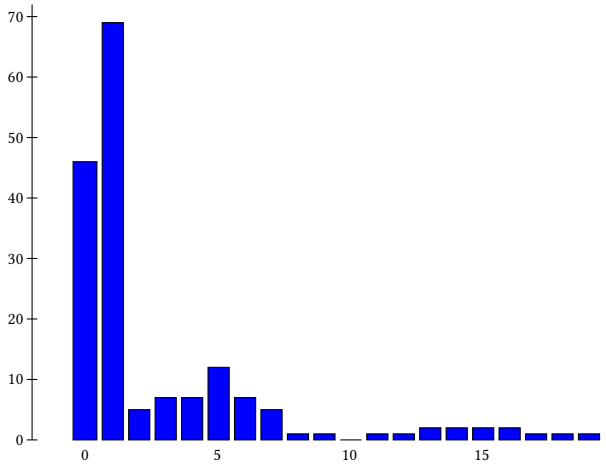


Figure 2: The distribution of $\mathrm{coind}_t(\mathtt{w})$

Deleting a variable. The deletion of the variable w induces the deletion of the corresponding column of M_t, possibly followed by a partial row-echelonization. In terms of algebraic complexity, the deletion is free. The cost of the echelonization is within $O\big(\mathrm{coind}_t(\mathtt{w}) \cdot \mathrm{cohgt}_t(\mathtt{w})\big)$ operations in $\mathbb{Q}_p$.

In the ZpLF case, we expect that, most of the time, the deleted variables were created after all initial variables were set by the user. This means that we expect $\mathrm{cohgt}_t(\mathtt{w})$ to vanish and so, the corresponding cost to be negligible.

In the ZpLC case, we always have $\mathrm{cohgt}_t(\mathtt{w}) = \mathrm{coind}_t(\mathtt{w})$, so that the cost becomes $O\big(\mathrm{coind}_t(\mathtt{w})^2\big)$. This does not look nice *a priori*. However, the principle of temporal locality [8] asserts that $\mathrm{coind}_t(\mathtt{w})$ tends to be small in general: destroyed variables are often variables that were created recently. As a basic example, variables which are local to a small piece of code (*e.g.* a short function or a loop) do not survive for a long time. It turns out that this behavior is typical in many implementations! The histogram of Figure 2 shows the distribution of $\mathrm{coind}_t(\mathtt{w})$ while executing the Euclidean algorithm (naive implementation) with two polynomials of degree 7 as input. The bias is evident: most of the time $\mathrm{coind}_t(\mathtt{w}) \leq 1$.

Summary: Impact on complexity. We consider the case of an algorithm with the following characteristics: its complexity is c operations in $\mathbb{Q}_p$ (without any tracking of precision), its memory

usage is m elements of $\mathbb{Q}_p$, its input and its output have size s_{in} and s_{out} (elements of $\mathbb{Q}_p$) respectively.

In the case of ZpLF, creating a variable has a cost $O(s_{\text{in}})$ whereas deleting a variable is free. Thus when executed with the ZpLF mechanism, the complexity of our algorithm becomes $O(s_{\text{in}}c)$.

In the ZpLC framework, creating a variable has a cost $O(m)$. The case of deletion is more difficult to handle. However, by the temporal locality principle, it seems safe to assume that it is not the bottleneck (which is the case in practice). Therefore, when executed with the ZpLF mechanism, the cost of our algorithm is expected to be roughly $O(mc)$. Going further in speculation, we might estimate the magnitude of m as about $s + \sqrt{c}$ with $s = \max(s_{\text{in}}, s_{\text{out}})$, leading to a complexity of $O(c^{3/2} + sc)$. For quasi-optimal algorithms, the term $sc \simeq c^2$ dominates. However, as soon as the complexity is at least quadratic in s, the dominant term is $c^{3/2}$ and the impact on the complexity is then limited.

4 CONCLUSION

The package ZpL provides powerful tools (based on automatic differentiation) to track precision in the p-adic setting. In many concrete situations, it greatly outperforms standard interval arithmetic, as shown in §2. The impact on complexity is controlled but nevertheless non-negligible (see §3.4). For this reason, it is unlikely that a fast algorithm will rely *directly* on the machinery proposed by ZpL, though it might do so for a specific part of a computation. At least for now, bringing together rapidity and stability still requires a substantial human contribution and a careful special study of all parameters.

Nevertheless, we believe that ZpL can be extremely helpful to anyone designing a fast and stable p-adic algorithm for a couple of reasons. First, it provides mechanisms to automatically detect which steps of a given algorithm are stable and which ones are not. In this way, it highlights the parts of the algorithm on which the researcher has to concentrate their effort. Second, recall that a classical strategy to improve stability consists in working internally at higher precision. Finding the internal increase in precision that best balances efficiency and accuracy is not an easy task in general. Understanding the number of diffused digits of precision gives very useful hints in this direction. For example, when there are no diffused digits of precision then the optimal precision completely splits over the variables and there is no need to internally increase the precision. On the contrary, when there are many diffused digits of precision, a large increment is often required. Since ZpL gives a direct access to the number of diffused digits of precision, it could be very useful to the designer who is concerned with the balance between efficiency and accuracy.

REFERENCES

[1] *754-2008 - IEEE Std. for Floating-Point Arithmetic*. IEEE, 2008.
[2] Xavier Caruso. Computations with p-adic numbers. pages 1–83, 2017.
[3] Xavier Caruso. Numerical stability of euclide algorithm over ultrametric fields. *J. Number Theor. Bordeaux*, 29:503–534, 2017.
[4] Xavier Caruso, David Roe, and Tristan Vaccon. Tracking p-adic precision. *LMS Journal of Computation and Mathematics*, 17(A):274–294, 2014.
[5] Xavier Caruso, David Roe, and Tristan Vaccon. p-Adic Stability In Linear Algebra. In *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '15, pages 101–108, New York, NY, USA, 2015. ACM.
[6] Xavier Caruso, David Roe, and Tristan Vaccon. Division and Slope Factorization of p-Adic Polynomials. In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, pages 159–166, New York, NY, USA, 2016. ACM.
[7] Xavier Caruso, David Roe, and Tristan Vaccon. Characteristic Polynomials of P-adic Matrices. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 389–396, New York, NY, USA, 2017. ACM.
[8] Peter Denning. The locality principle. *Commun. ACM*, 48:19–24, 2005.
[9] Nicholas Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 2nd ed. edition, 2002.
[10] Pierre Lairez and Tristan Vaccon. On p-adic differential equations with separation of variables. In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2016, Waterloo, ON, Canada, July 19-22, 2016*, pages 319–323, 2016.
[11] Louis Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, Berlin, 1981.
[12] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.1)*, 2018. http://www.sagemath.org.
[13] Trac #23505: Lattice precision for p-adics. http://trac.sagemath.org/ticket/23505, 2018.