p-adic Precision

David Roe

Department of Mathematics
Massachusetts Institute of Technology

Simons Collaboration Meeting

May 7, 2021

Outline

p-adic precision in Sage

This work is joint with Xavier Caruso and Tristan Vaccon.

Throughout the talk I'll be using $\mathbb{Q}_p$ as an example, but the methods apply to any complete discrete valuation field.

Ball arithmetic

Sage, and most other p-adic systems, normally tracks precision by attaching it to each element and propagating it through arithmetic.

Ball arithmetic

The *absolute precision* of $p^v u + O(p^N)$ is N, and the *relative precision* is $N - v$.

- The absolute precision of the sum or difference of two numbers is the minimum of their absolute precisions.
- The relative precision of the product of quotient of two number is the minimum of their relative precisions.

Standard because you get provably correct results, and the precision behavior is often better than $\mathbb{R}$ (for example, you don't get gradually accumulating error from adding elements).

Floating point

Problem: sometimes you lose all your precision, especially if you ignore numerical stability. Can emulate methods over $\mathbb{R}$ and drop precision tracking completely.

Floating point arithmetic

Pick a finite collection of representable numbers, e.g. $p^v u$ for $-M \leq v \leq M$ and $0 \leq u \leq p^N$ and define arithmetic operations to yield the closest representable number to the true answer.

Can sometimes mathematically analyze precision and show that the result is more precise than what interval arithmetic would predict. But without this work, floating point calculations have no guarantees.

Lazy and relaxed arithmetic

Can record elements' construction in addition to their precision.

Lazy elements

A lazy element includes

- an approximation
- a current precision
- a pointer back to its inputs.

This structure allows for the computation of additional digits at need.
Two approaches:

- Approximation using standard elements; double precision when asked for more.
- Approximation using polynomials with small coefficients; there's an asymptotically better algorithm for multiplication.

Lattices

Ball arithmetic is optimal if you consider precision to be an attribute of an individual number. But we can do better if we allow precision data to incorporate relationships between variables.

Given n variables, we will think about their values as a vector in a n-dimensional vector space E. A *lattice* $H \subset E$ is a bounded sub-$\mathbb{Z}_p$-module which generates E over $\mathbb{Q}_p$. In practice, it's just the $\mathbb{Z}_p$-span of n vectors. For example, the ball of radius $r > 0$ around the origin is a lattice, as is the diagonal lattice spanned by

$$\begin{gathered}(p^{m_1}, 0, \ldots, 0),\\ (0, p^{m_2}, \ldots, 0),\\ \vdots\\ (0, 0, \ldots, p^{m_n}).\end{gathered}$$

An *approximate element* of E takes the form $\mathrm{x} + H$ for some lattice H.

Lattice example

Suppose $x = 4 + O(3^6)$ and $y = 2 + O(3^4)$. Set $u = x + y$ and $v = x - y$. Then the precision of (x, y) is

$$(3^6, 0)\mathbb{Z}_3 + (0, 3^4)\mathbb{Z}_3,$$

and the precision of (u, v) is obtained by multiplying by $\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$:

$$L = (3^6, 3^6)\mathbb{Z}_3 + (3^4, -3^4)\mathbb{Z}_3 = (0, 3^6)\mathbb{Z}_3 + (3^4, -3^4)\mathbb{Z}_3.$$

This lattice is not diagonal: the smallest diagonal lattice containing it is $(3^4, 0)\mathbb{Z}_3 + (0, 3^4)\mathbb{Z}_3$, corresponding to the fact that u and v have absolute precision 4 in interval arithmetic. But if you compute $r = u + v = 2x$ and $s = u - v = 2y$ then (r, s) has precision lattice

$$(0, 2 \cdot 3^4)\mathbb{Z}_3 + (3^6, -3^6)\mathbb{Z}_3 = (3^6, 0)\mathbb{Z}_3 + (0, 3^4)\mathbb{Z}_3.$$

The precision lemma (Caruso-R-Vaccon)

Using lattices to track precision is optimal in the following sense:

Lemma

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. Suppose f is differentiable at $x \in U$ and df_x is surjective. Then for sufficiently small H,

$$f(x + H) = f(x) + df_x(H).$$

The required size on H depends on the magnitude of the higher derivatives, and is computable for concrete f. We also show variants for maps between p-adic manifolds and infinite dimensional Banach spaces.

Representing lattices

- We represent a precision lattice using an upper triangular matrix, whose rows give a basis for the lattice.
- We scale the diagonal entries to be powers of p.
- Note that lattices are *exact*, since we may use row operations to reduce each column modulo the power of p on the diagonal.

Adding and deleting variables

- Adding a variable via arithmetic: `w = f(v_1,...v_n)`. We add a new column with entries given by $\frac{\partial f}{\partial v_i}$.
- The resulting matrix is no longer square.
  - In one model (`ZpLF`) we allow such submodules (at the cost of working with inexact objects).
  - In the other model (`ZpLC`) we add a new row $(0, \ldots, 0, p^C)$ for some cap C.
- Introducing a constant or input: `w = R(value, prec)`. We add a column of zeros, possibly capped by a new row's p^C.
- Deleting a variable: delete the corresponding column, re-echelonize, then delete a row of zeros at the bottom.

Complexity

- Tracking precision this way requires storing a matrix with number of columns equal to the number of variables, and number of rows either the number of variables (`ZpLC`) or the number of input variables (`ZpLF`).
- The size of the entries is bounded by p^C, where C is the precision cap (`ZpLC`) or the precision of floating point arithmetic (`ZpLF`).
- Adding variables requires $O(nr)$ operations, where n is the arity of the operation (often 2) and r is the current number of rows.
- Deleting a variable requires $O(z^2)$ operations (`ZpLC`), where z is the number of columns to the right of the deleted column. In practice negligible due to temporal locality; `ZpLF` is even better.
- In an algorithm with complexity c, input/output size s and memory usage $m \approx s + \sqrt{c}$, tracking precision using with `ZpLC` takes $O(sc)$ operations; with `ZpFL` $O(c^{3/2} + sc)$.

Correctness and optimality

- Our current implementation does not check the smallness condition required to apply the precision lemma, so the results are not provably correct.
- The precision cap can reduce the precision of the result, but this is checkable a fortiori.
- Can quantify the amount of precision lost by checking precision on individual variables. If $H \subset E$ is a lattice and $\pi_i : E \to \mathbb{Q}_p e_i$ are projections onto the variables, the number of diffused digits of precision is the length of H_0/H, where $H_0 = \pi_1(H) \oplus \cdots \oplus \pi_n(H)$.

Demo

To Sage!

padic_demo

May 7, 2021

0.1 The SOMOS 4 sequence

The SOMOS 4 sequence is the sequence defined by the recurrence

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

[3]:
```
def somos(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
        print(d)
```

[4]:
```
Z2 = Zp(2,40,print_mode='digits')
u0 = u1 = u2 = Z2(1,15); u3 = Z2(3,15)
somos(u0,u1,u2,u3,18)
```

```
…000000000000100
…000000000001101
…000000000110111
…101010111010111
…1100111101111
…1110000010010
…0001000111001
…0000011111101
…1000000110101
…101101010011
…110000000000
…000101011101
…001001101011
…000011110011
…11
```

[5]:
```
Z2 = ZpLC(2,40,print_mode='digits')
u0 = u1 = u2 = Z2(1,15); u3 = Z2(3,15)
somos(u0,u1,u2,u3,18)
```

```
…000000000000100
…000000000001101
```

```
…000000000110111
…101010111010111
…101100111101111
…1111110000010010
…100001000111001
…100000011111101
…001000000110101
…010101101010011
…001110000000000
…111000101011101
…111001001101011
…111000011110011
…100000000000111
```

0.2 Basic arithmetic

[6]:
```
Z3 = ZpLC(3)
x = Z3(4, 6)
y = Z3(2, 4)
u = x + y
v = x - y
u, v
```

[6]:
```
(2*3 + O(3^4), 2 + O(3^4))
```

[7]:
```
(u+v, u-v)
```

[7]:
```
(2 + 2*3 + O(3^6), 1 + 3 + O(3^4))
```

[8]:
```
x+x+x
```

[8]:
```
3 + 3^2 + O(3^7)
```

[9]:
```
L = Z3.precision()
L.precision_lattice([u,v])
```

[9]:
```
[ 81 648]
[  0 729]
```

[11]:
```
L.diffused_digits([u,v])
```

[11]:
```
2
```

0.3 Computing with Matrices

Products of many matrices

[57]:
```
def random_element(R, prec):
    return R(ZZ.random_element(R.prime()^prec), prec)
```

[58]:
```
def random_matrix(MS, prec):
    return MS([random_element(MS.base_ring(), prec) for _ in range(MS.nrows() *␣
 ↪MS.ncols())])
```

[50]:
```
Z2 = Zp(2,40,print_mode='digits')
MS = MatrixSpace(Z2,2)
M = MS(1)
for _ in range(60):
    M *= random_matrix(MS, prec=8)
M
```

[50]:
```
[   …00000000000000000 …0000000000000000000]
[   …00000000000000000 …0000000000000000000]
```

[51]:
```
Z2 = ZpLC(2,40,print_mode='digits')
MS = MatrixSpace(Z2,2)
M = MS(1)
for _ in range(60):
    M *= random_matrix(MS, prec=8)
M
```

[51]:
```
[…10110100000000000000000 …00000100000000000000000]
[  …11011100000000000000    …11001100000000000000000]
```

Determinants and characteristic polynomials

[52]:
```
Z3 = Zp(3,40,print_mode='digits')
D = diagonal_matrix([ Z3(1,5), Z3(9,5), Z3(27,5), Z3(81,5) ])
D.determinant()
```

[52]:
```
…1000000000
```

[53]:
```
MS = D.parent()
P = random_matrix(MS, prec=5)
Q = random_matrix(MS, prec=5)
M = P*D*Q
M.determinant()
```

[53]:
```
…00000000000
```

[54]:
```
M.characteristic_polynomial()
```

[54]:
```
…000000000000000000000000000000000000000001*x^4 + …122*x^3 + …000*x^2 +
…00000*x + …00000000
```

[55]:
```
Z3 = ZpLC(3,40,print_mode='digits')
D = diagonal_matrix([ Z3(1,5), Z3(9,5), Z3(27,5), Z3(81,5) ])
MS = D.parent()
P = random_matrix(MS, prec=5)
Q = random_matrix(MS, prec=5)
M = P*D*Q
M.determinant()
```

[55]:
```
…?.0
```

[56]:
```
M.characteristic_polynomial()
```

[56]:
```
…0000000000000000000000000000000000000001*x^4 + …01211*x^3 + …01200*x^2 +
…12000000*x + …20000000000000
```

0.4 Computing with polynomials

Euclidean algorithm

[61]:
```
def random_polynomial(S, degree, prec):
    return S([random_element(S.base_ring(), prec) for _ in range(degree+1)])
```

[30]:
```
def euclidean(A,B):
    while B != 0:
        A, B = B, A % B
    return A.monic()
```

[65]:
```
Q2 = Qp(2,40,print_mode='digits')
S.<x> = PolynomialRing(Q2)
P = random_polynomial(S, degree=10, prec=5)
Q = random_polynomial(S, degree=10, prec=5)
D = x^5 + random_polynomial(S, degree=4, prec=8)
D
```

[65]:
```
…0000000000000000000000000000000000000001*x^5 + …11001000*x^4 +
…10010111*x^3 + …11110111*x^2 + …01101101*x + …01110101
```

With high probability, P and Q are coprime, implying that the gcd of DP is DQ is D. However, we observe that we do not always get this expected result:

[66]:
```
euclidean(D*P, D*Q)
```

[66]:
```
…1*x^12 + …?.?0*x^11 + …?.1*x^10 + …?.?0*x^9 + …?.?0*x^8 + …?.?0*x^7
+ …?.?0*x^6 + …?.?0*x^5 + …?.1*x^4 + …?.?0*x^3 + …?.0*x^2 + …?.0*x +
…?.?0
```

Here, we stopped too early since the remainder lost its precision. If we use floating point arithmetic, the opposite problem occurs, since the remainder is not zero soon enough.

[67]:
```
Q2 = QpFP(2,15,print_mode='digits')
S.<x> = PolynomialRing(Q2)
P = random_polynomial(S, degree=10, prec=5)
Q = random_polynomial(S, degree=10, prec=5)
D = x^5 + random_polynomial(S, degree=4, prec=8)
D
```

[67]:
```
x^5 + 11100101*x^4 + 10110000*x^3 + 11001100*x^2 + 100101*x + 10101110
```

[68]:
```
euclidean(D*P, D*Q)
```

[68]:
```
1
```

With lattice precision, we get the right answer.

[73]:
```
Q2 = QpLC(2,40,print_mode='digits')
S.<x> = PolynomialRing(Q2)
P = random_polynomial(S, degree=10, prec=5)
Q = random_polynomial(S, degree=10, prec=5)
D = x^5 + random_polynomial(S, degree=4, prec=8)
D
```

[73]:
```
…0000000000000000000000000000000000000001*x^5 + …00110100*x^4 +
…01010011*x^3 + …10011010*x^2 + …01111101*x + …11111100
```

[74]:
```
euclidean(D*P, D*Q)
```

[74]:
```
…0000000000000000000000000000000000000001*x^5 + …00110100*x^4 +
…01010011*x^3 + …10011010*x^2 + …01111101*x + …11111100
```

0.5 Lazy p-adics

[76]:
```
R = ZpER(5, print_mode='digits')
```

[79]:
```
a = R(6).sqrt()
a
```

[79]:
```
…42223233033132124031
```

[80]:
```
a[:50]
```

[80]:
```
…?20344201141421014133000210413342223233033132124031
```

[81]:
```
a[:150]
```

[81]:
```
…?2140112412204203103043313134242002400043312041344041012040014113232133044443
24101132344211242434114320344201141421014133000210413342223233033132124031
```

[82]:
```
u = R.unknown()
v = R.unknown()
w = R.unknown()
u.set(1 + 2*v + 3*w^2 + 5*u*v*w)
v.set(2 + 4*w + sqrt(1 + 5*u + 10*v + 15*w))
w.set(3 + 25*(u*v + v*w + u*w))
```

[82]:
```
True
```

[83]:
```
u
```

[83]:
```
…31203130103131131433
```

[84]:
```
v
```

[84]:
```
…33441043031103114240
```

[85]:
```
w
```

[85]:
```
…30212422041102444403
```

[86]:
```
u == 1 + 2*v + 3*w^2 + 5*u*v*w
```

[86]:
```
True
```

[87]:
```
v == 2 + 4*w + sqrt(1 + 5*u + 10*v + 15*w)
```

[87]:
```
True
```

[88]:
```
w == 3 + 25*(u*v + v*w + u*w)
```

[88]:
```
True
```

[ ]:
```

```