Relaxed $\mathfrak{p}$-adics in Mathemagix

Jérémy Berthomieu

Laboratoire d'Informatique de Paris 6
UPMC, Université Pierre-et-Marie-Curie
INRIA, Paris – Rocquencourt, PolSys
CNRS, UMR 7606

Romain Lebreton

Laboratoire d'Informatique Robotique
et de Microélectronique de Montpellier
Université Montpellier 2
CNRS, UMR 5506

Grégoire Lecerf

Laboratoire d'Informatique de l'X
École polytechnique
CNRS, UMR 7161

Joris van der Hoeven

Laboratoire d'Informatique de l'X
École polytechnique
CNRS, UMR 7161

Sage Days p-adics – Rennes
Tuesday 3rd September 2013

What are $\mathfrak{p}$-adics?

Definition.

- R an integral ring.
- $(\mathfrak{p})$ a prime ideal.
- $R_{\mathfrak{p}} = \{\sum_{n=0}^{\infty} a_n, \ a_n \in (\mathfrak{p}^n)\}$.

Examples.

1. $R = \mathbb{K}[x]$, $\mathfrak{p} = (x)$, $R_{\mathfrak{p}} = \mathbb{K}[[x]] = \{\sum_{n=0}^{\infty} a_n x^n, \ a_n \in \mathbb{K}\}$.
2. $R = \mathbb{Z}$, $\mathfrak{p} = (p)$, $R_{\mathfrak{p}} = \mathbb{Z}_p = \{\sum_{n=0}^{\infty} a_n p^n, \ a_n \in \{0, \ldots, p-1\}\}$.
3. $R = \mathbb{R}[x]$, $\mathfrak{p} = (x^2+1)$,

$$R_{\mathfrak{p}} = \mathbb{R}[x]_{(x^2+1)} = \left\{ \sum_{n=0}^{\infty} (a_n x + b_n)(x^2+1)^n, \ a_n, b_n \in \mathbb{R} \right\} \simeq \mathbb{C}[[t]].$$

Representations of $\mathfrak{p}$-adics

Problem.

$\mathfrak{p}$-adics have infinitely many coefficients.

N computed coefficients $(a_0, \ldots, a_{N-1}) \rightsquigarrow$ precision N.

Two paradigms.

1. Zealous model: Double precision at each step.
2. Lazy model: Increase precision 1 by 1.

Zealous model

Definition.

Fixed precision N.

$\rightsquigarrow$ Computation $\bmod \mathfrak{p}^N$.

If the precision is too low, then double it.

$\rightsquigarrow$ Computation $\bmod \mathfrak{p}^{2N}$ from the beginning.

Pros.

- Precision known $\rightsquigarrow$ No useless computations.
- Better multiplication complexity $O(\mathsf{M}(N))$
  $\rightsquigarrow$ Same as multiplying polynomials of degree less than N.

Cons.

- Pessimistic precision bounds.
- Computation of the inverse Jacobian at precision $N/2$.

Zealous $\mathfrak{p}$-adics in Pari/GP

Definition.

A zealous $\mathfrak{p}$-adic at precision N is an element of $R/(\mathfrak{p}^N)$.

```
Pari] a= 1234 + O(5^4)
```

%1 = $4+5+4\cdot5^2+4\cdot5^3+O(5^4)$

```
Pari] 2*a
```

%2 = $3+3\cdot5+3\cdot5^2+4\cdot5^3+O(5^4)$

```
Pari] 5*a
```

%3 = $4\cdot5+5^2+4\cdot5^3+4\cdot5^4+O(5^5)$

```
Pari] b= 1/(8 + O(5^15))
```

%4 = $2+4\cdot5+5^2+4\cdot5^3+5^4+4\cdot5^5+5^6+4\cdot5^7+5^8+4\cdot5^9+5^{10}+4\cdot5^{11}+5^{12}+4\cdot5^{13}+5^{14}+O(5^{15})$

```
Pari] a*b
```

%5 = $3+4\cdot5+4\cdot5^2+4\cdot5^3+O(5^4)$

```
Pari] a= 1234 + O(5^8); a*b
```

%6 = $3+4\cdot 5+4\cdot 5^2+4\cdot 5^3+3\cdot 5^4+3\cdot 5^5+3\cdot 5^6+3\cdot 5^7+O(5^8)$

Lazy model

Definition.

Flow of coefficients. Computations should require the minimum knowledge on the input.

$\rightsquigarrow$ Table of computed coefficients and a method for computing the next one.

$\rightsquigarrow$ If the precision is too low, call the next() method.

Pros.

- User friendly.
- No useless computations, whether the suitable precision is known or not.
- Computation of the inverse Jacobian at precision 0.

Cons.

- Overhead for the multiplication complexity: $\mathsf{R}(N) = O(\mathsf{M}(N) \log N)$.

Lazy multiplication $c = a \times b$ - step 0

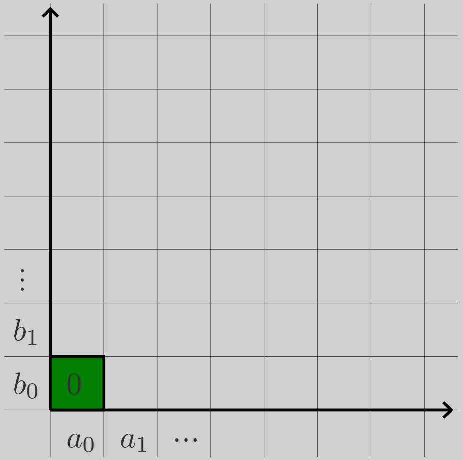

Figure. Naive multiplication.

step 0: $c = a_0 b_0$

Lazy multiplication $c = a \times b$ - step 1

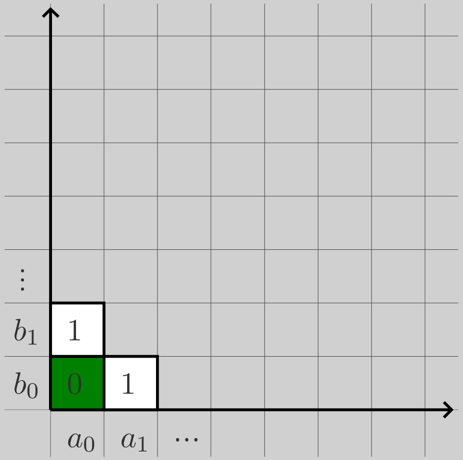

Figure. Naive multiplication.

step 0: $c = a_0\, b_0$
step 1: $c \mathrel{+}= \mathfrak{p}\,(a_0\, b_1 + a_1\, b_0)$

LAZY MULTIPLICATION $c = a \times b$ - STEP 2

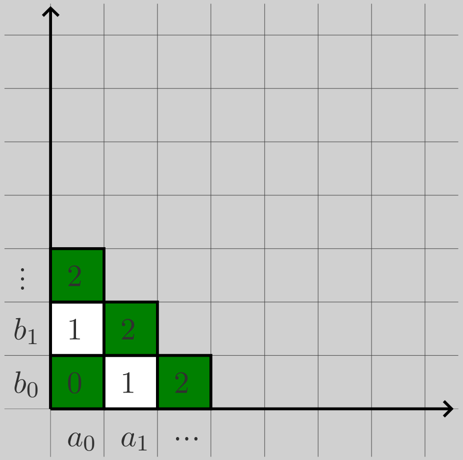

Figure. Naive multiplication.

step 0: $c = a_0\, b_0$
step 1: $c \mathrel{+}= \mathfrak{p}\,(a_0\, b_1 + a_1\, b_0)$
step 2: $c \mathrel{+}= \mathfrak{p}^2\,(a_0\, b_2 + a_2\, b_0 + a_1\, b_1)$

LAZY MULTIPLICATION $c = a \times b$ - STEP 3

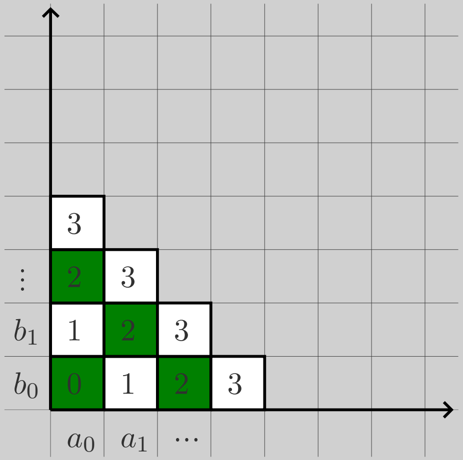

Figure. Naive multiplication.

step 0: $c = a_0\, b_0$
step 1: $c += \mathfrak{p}\,(a_0\, b_1 + a_1\, b_0)$
step 2: $c += \mathfrak{p}^2\,(a_0\, b_2 + a_2\, b_0 + a_1\, b_1)$
step 3: $c += \mathfrak{p}^3\,(a_0\, b_3 + a_3\, b_0 + a_1\, b_2 + a_2\, b_1)$

Lazy multiplication $c = a \times b$ - step 4

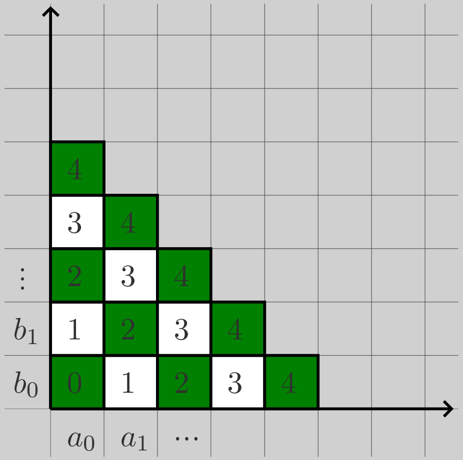

Figure. Naive multiplication.

step 0: $c = a_0\, b_0$

step 1: $c \mathrel{+}= \mathfrak{p}\,(a_0\, b_1 + a_1\, b_0)$

step 2: $c \mathrel{+}= \mathfrak{p}^2\,(a_0\, b_2 + a_2\, b_0 + a_1\, b_1)$

step 3: $c \mathrel{+}= \mathfrak{p}^3\,(a_0\, b_3 + a_3\, b_0 + a_1\, b_2 + a_2\, b_1)$

step 4: $c \mathrel{+}= \mathfrak{p}^4\,(a_0\, b_4 + a_4\, b_0 + a_1\, b_3 + a_3\, b_1 + a_2\, b_2)$

LAZY MULTIPLICATION $c = a \times b$ - STEP 5

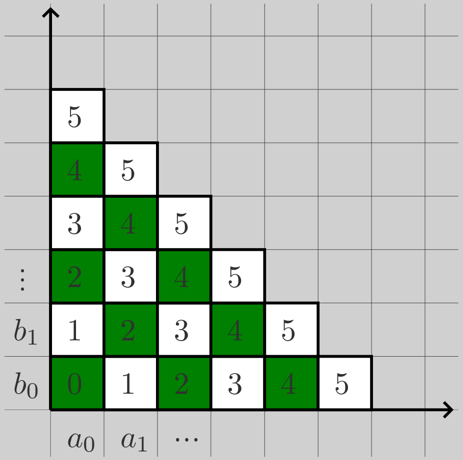

Figure. Naive multiplication.

step 0: $c = a_0\, b_0$

step 1: $c \mathrel{+}= \mathfrak{p}\,(a_0\, b_1 + a_1\, b_0)$

step 2: $c \mathrel{+}= \mathfrak{p}^2\,(a_0\, b_2 + a_2\, b_0 + a_1\, b_1)$

step 3: $c \mathrel{+}= \mathfrak{p}^3\,(a_0\, b_3 + a_3\, b_0 + a_1\, b_2 + a_2\, b_1)$

step 4: $c \mathrel{+}= \mathfrak{p}^4\,(a_0\, b_4 + a_4\, b_0 + a_1\, b_3 + a_3\, b_1 + a_2\, b_2)$

step 5: $c \mathrel{+}= \mathfrak{p}^5\,(a_0\, b_5 + a_5\, b_0 + a_1\, b_4 + a_4\, b_1 + a_2\, b_3 + a_3\, b_2)$

Lazy multiplication $c = a \times b$ - step 6

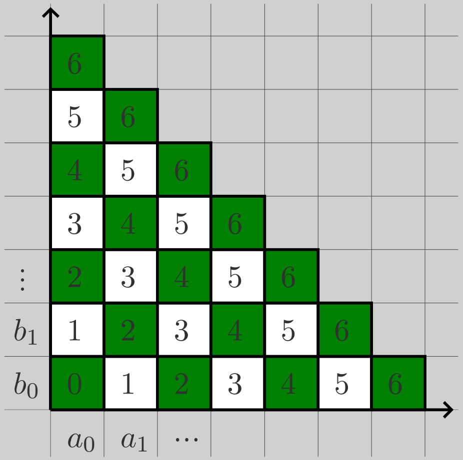

Figure. Naive multiplication.

step 0: $c = a_0\, b_0$
step 1: $c \mathrel{+}= \mathfrak{p}\,(a_0\, b_1 + a_1\, b_0)$
step 2: $c \mathrel{+}= \mathfrak{p}^2\,(a_0\, b_2 + a_2\, b_0 + a_1\, b_1)$
step 3: $c \mathrel{+}= \mathfrak{p}^3\,(a_0\, b_3 + a_3\, b_0 + a_1\, b_2 + a_2\, b_1)$
step 4: $c \mathrel{+}= \mathfrak{p}^4\,(a_0\, b_4 + a_4\, b_0 + a_1\, b_3 + a_3\, b_1 + a_2\, b_2)$
step 5: $c \mathrel{+}= \mathfrak{p}^5\,(a_0\, b_5 + a_5\, b_0 + a_1\, b_4 + a_4\, b_1 + a_2\, b_3 + a_3\, b_2)$
step 6: $c \mathrel{+}= \mathfrak{p}^6\,(a_0\, b_6 + a_6\, b_0 + a_1\, b_5 + a_5\, b_1 + a_2\, b_4 + a_4\, b_2 + a_3\, b_3)$

Problem.

The complexity of the naive multiplication of lazy $\mathfrak{p}$-adics is $O(N^2)$.

Definition.

A lazy $\mathfrak{p}$-adic at precision N is a table of N coefficients and a method to compute the $(N+1)$th coefficient.

```
Mmx] use "algebramix"; p_adic (a, p) == p_adic (@p_expansion (a, modulus p));
     x == p_adic (1234, 5)
```

$4+p+4\,p^2+4\,p^3+p^4+O(p^{10})$

```
Mmx] x == p_adic (1, 13) / p_adic (9876543210^1000, 13)
```

$1+p+p^2+4\,p^3+7\,p^4+p^5+12\,p^6+9\,p^7+4\,p^8+11\,p^9+O(p^{10})$

```
Mmx] [x[10], x[20], x[50], x[100], x[200], x[500], x[1000], x[2000], x[5000]]
```

$[12,0,11,4,4,11,0,7,2]$

```
Mmx] set_output_order (x, 50); x
```

$1+p+p^2+4\,p^3+7\,p^4+p^5+12\,p^6+9\,p^7+4\,p^8+11\,p^9+12\,p^{10}+8\,p^{11}+3\,p^{12}+6\,p^{15}+3\,p^{16}+6\,p^{17}+p^{18}+12\,p^{19}+2\,p^{22}+p^{23}+8\,p^{24}+2\,p^{25}+6\,p^{26}+2\,p^{27}+4\,p^{28}+5\,p^{29}+6\,p^{30}+7\,p^{31}+9\,p^{32}+6\,p^{33}+9\,p^{34}+6\,p^{35}+3\,p^{36}+3\,p^{37}+4\,p^{38}+p^{39}+9\,p^{40}+7\,p^{41}+11\,p^{42}+12\,p^{43}+11\,p^{44}+4\,p^{45}+11\,p^{46}+11\,p^{47}+3\,p^{48}+9\,p^{49}+O(p^{50})$

Implementation in Mathemagix

- Code in C++.
- Lazy representation of rings of formal powers series $\mathbb{K}[[t]]$ as a templated class: `series<C,V>`
  - Template `C` for the coefficient ring $\mathbb{K}$: `int`, `double`, GMP `mpz_t`, another class...
  - Template `V` for the variant of the operations: naive (`series_naive`), relaxed (`series_relaxed`)...

    , naive with a carry (`series_carry_naive`), relaxed with a carry (`series_carry`
- Code can be adapted to any $\mathfrak{p}$-adic ring $R_{\mathfrak{p}}$.

The series class

```
template<typename C, typename V>
class series_rep REP_STRUCT_1(C) {
public:
  C*  a;  // coefficients
  nat n;  // number of computed coefficients
  nat l;  // number of allocated coefficients
          // a[n],...,a[l-1] may be used by relaxed computations
  inline series_rep (const Format& fm):
    Format (fm), a (mmx_new<C> (0)), n (0), l (0) {}
  inline virtual ~series_rep () { mmx_delete<C> (a, l); }
  inline C zero () { return promote (0, this->tfm ()); }
  inline C one () { return promote (1, this->tfm ()); }
  inline Series me () const;
  virtual C next () = 0;
  virtual syntactic expression (const syntactic& z) const = 0;

public:
  virtual void Set_order (nat l2);
  virtual void Increase_order (nat l2=0);
  virtual inline bool test_zero () const { return false; }
  friend class Series;
};
```

```
template<typename C, typename V>
class series {
INDIRECT_PROTO_2 (series, series_rep, C, V)
  typedef implementation<series_defaults,V> Ser;
  typedef typename Ser::template global_variables<Series> S;

public:
  static inline generic get_variable_name () {
    return S::get_variable_name (); }
  static inline void set_variable_name (const generic& x) {
    S::set_variable_name (x); }

  static inline nat get_output_order () {
    return S::get_output_order (); }
  static inline void set_output_order (const nat& x) {
    S::set_output_order (x); }

  static inline nat get_cancel_order () {
    return S::get_cancel_order (); }
  static inline void set_cancel_order (const nat& x) {
    S::set_cancel_order (x); }

  static inline bool get_formula_output () {
    return S::get_formula_output (); }
  static inline void set_formula_output (const bool& x) {
    S::set_formula_output (x); }
```

```
public:
  series ();
  series (const Format& fm);
  series (const C& c);
  template<typename T> series (const T& c);
  template<typename T> series (const T& c, const Format& fm);
  template<typename T> series (const T& c, nat deg);
  template<typename W> series (const polynomial<C,W>& P);
  template<typename W> series (const polynomial<C,W>& P, const Format& fm);
  template<typename T, typename W>
  series (const series<T,W>& f);
  template<typename T, typename W>
  series (const series<T,W>& f, const Format& fm);
  series (const vector<C>& coeffs);
  series (const iterator<C>& it, const string& name= "explicit");
  series (C (*coeffs) (nat), const string& name= "explicit");
  const C& operator [] (nat n) const;
  const C* operator () (nat start, nat end) const;
};
```

The zero series

```
template<typename C, typename V>
class zero_series_rep: public series_rep<C,V> {
public:
  zero_series_rep (const Format& fm):
    series_rep<C,V> (fm) {}
  syntactic expression (const syntactic&) const {
    return flatten (0); }
  bool test_zero () const {
    return true; }
  C next () { return this->zero (); }
};
```

The sum or the difference of two series

```
#define Series series<C, V>
#define Series_rep series_rep<C, V>
typedef unsigned int nat;

template<typename Op, typename C, typename V>
class binary_series_rep: public Series_rep {
  protected:
    const Series f, g;
  public:
    inline binary_series_rep (const Series& f2, const Series& g2):
      f (f2), g (g2) {}
    virtual void Increase_order (nat l) {
      Series_rep::Increase_order (l);
      increase_order (f, l);
      increase_order (g, l);
    }
    virtual C next () {
      return Op::op (f[this->n], g[this->n]);
    }
};
```

The sum or the difference of two $\mathfrak{p}$-adics

```
#define Series series<M, V>
#define Series_rep series_rep<M, V>
typedef unsigned int nat;

template<typename Op, typename M, typename V>
class binary_series_rep: public Series_rep {
  protected:
    const Series f, g;
    C carry;
  public:
    inline binary_series_rep (const Series& f2, const Series& g2):
      f (f2), g (g2), carry (0) {}
    virtual void Increase_order (nat l) {
      Series_rep::Increase_order (l);
      increase_order (f, l);
      increase_order (g, l);
    }
    virtual M next () {
      return Op::op_mod (f[this->n].rep, g[this->n].rep, M::get_modulus (),
                         carry);
    }
};
```

Relaxed multiplication $c = a \times b$ - step 0

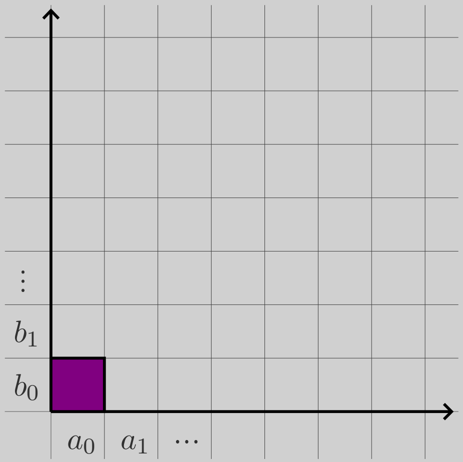

Figure. Minimum knowledge on the input.

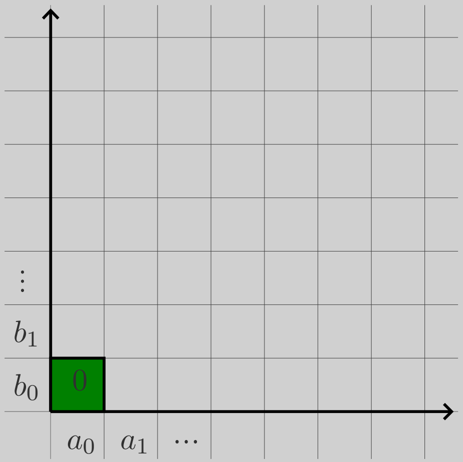

Figure. What we compute.

step 0: $c = a_0 \, b_0$

Relaxed multiplication $c = a \times b$ - step 1

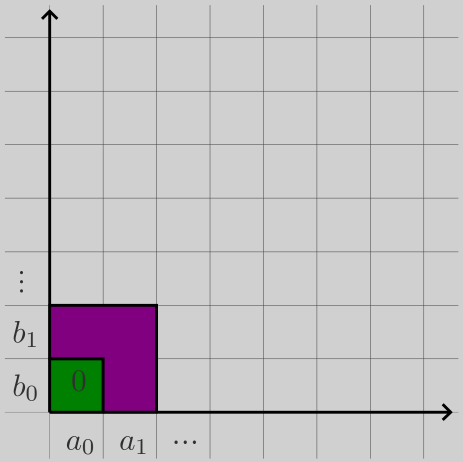

Figure. Minimum knowledge on the input.

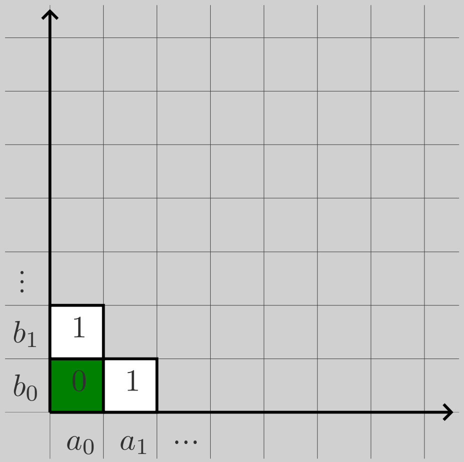

Figure. What we compute.

step 0: $c = a_0 \, b_0$

step 1: $c \mathrel{+}= \mathfrak{p} \, (a_0 \, b_1 + a_1 \, b_0)$

Relaxed multiplication $c = a \times b$ - step 2

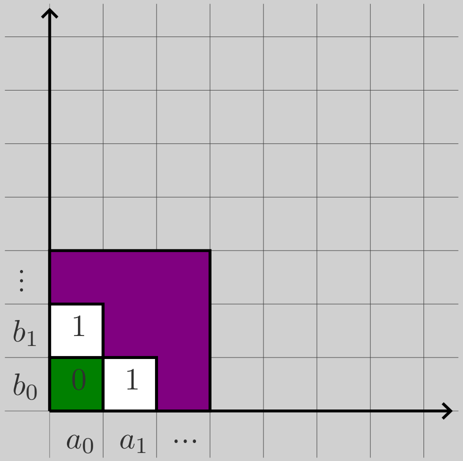

Figure. Minimum knowledge on the input.

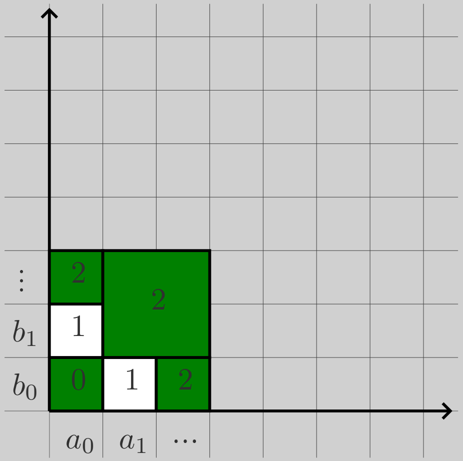

Figure. What we compute.

step 0: $c = a_0\, b_0$

step 1: $c \mathrel{+}= \mathfrak{p}\,(a_0\, b_1 + a_1\, b_0)$

step 2: $c \mathrel{+}= \mathfrak{p}^2\,(a_0\, b_2 + a_2\, b_0 + (a_1 + a_2\, \mathfrak{p})\,(b_1 + b_2\, \mathfrak{p}))$

Relaxed multiplication $c = a \times b$ - step 3

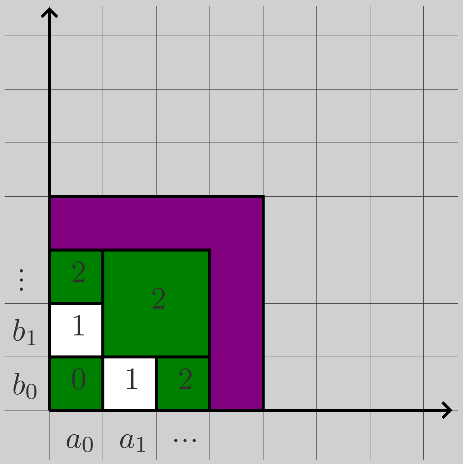

Figure. Minimum knowledge on the input.

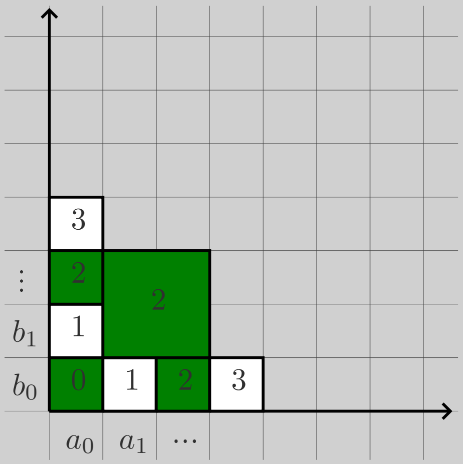

Figure. What we compute.

step 0: $c = a_0\, b_0$

step 1: $c \mathrel{+}= \mathfrak{p}\, (a_0\, b_1 + a_1\, b_0)$

step 2: $c \mathrel{+}= \mathfrak{p}^2\, (a_0\, b_2 + a_2\, b_0 + (a_1 + a_2\, \mathfrak{p})\, (b_1 + b_2\, \mathfrak{p}))$

step 3: $c \mathrel{+}= \mathfrak{p}^3\, (a_0\, b_3 + a_3\, b_0)$

Relaxed multiplication $c = a \times b$ - step 4

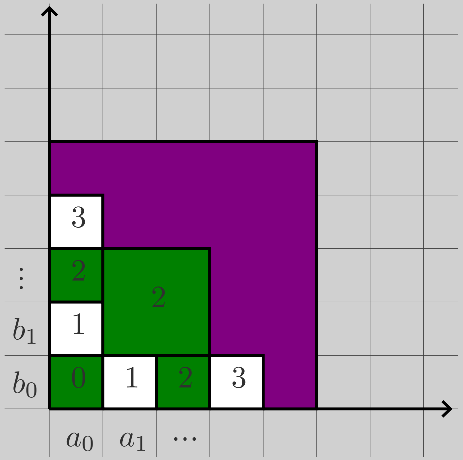

Figure. Minimum knowledge on the input.

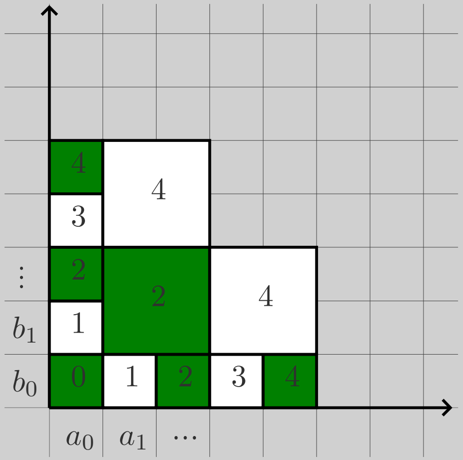

Figure. What we compute.

step 0: $c = a_0\, b_0$

step 1: $c \mathrel{+}= \mathfrak{p}\,(a_0\, b_1 + a_1\, b_0)$

step 2: $c \mathrel{+}= \mathfrak{p}^2\,(a_0\, b_2 + a_2\, b_0 + (a_1 + a_2\,\mathfrak{p})\,(b_1 + b_2\,\mathfrak{p}))$

step 3: $c \mathrel{+}= \mathfrak{p}^3\,(a_0\, b_3 + a_3\, b_0)$

step 4: $c \mathrel{+}= \mathfrak{p}^4\,(a_0\, b_4 + a_4\, b_0 + (a_1 + a_2\,\mathfrak{p})\,(b_3 + b_4\,\mathfrak{p}) + (a_3 + a_4\,\mathfrak{p})\,(b_1 + b_2\,\mathfrak{p}))$

RELAXED MULTIPLICATION $c = a \times b$ - STEP 5

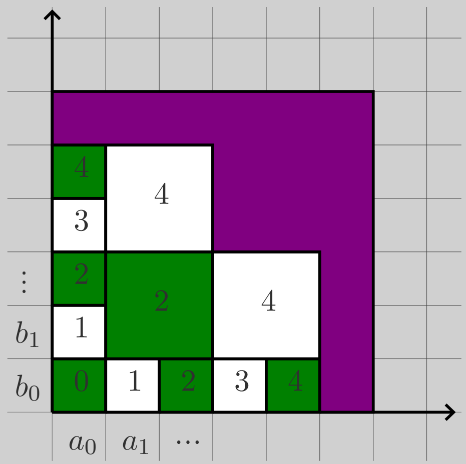

Figure. Minimum knowledge on the input.

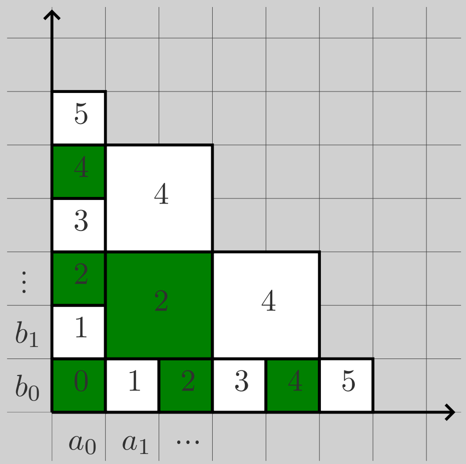

Figure. What we compute.

step 0: $c = a_0\, b_0$

step 1: $c \mathrel{+}= \mathfrak{p}\, (a_0\, b_1 + a_1\, b_0)$

step 2: $c \mathrel{+}= \mathfrak{p}^2\, (a_0\, b_2 + a_2\, b_0 + (a_1 + a_2\, \mathfrak{p})\, (b_1 + b_2\, \mathfrak{p}))$

step 3: $c \mathrel{+}= \mathfrak{p}^3\, (a_0\, b_3 + a_3\, b_0)$

step 4: $c \mathrel{+}= \mathfrak{p}^4\, (a_0\, b_4 + a_4\, b_0 + (a_1 + a_2\, \mathfrak{p})\, (b_3 + b_4\, \mathfrak{p}) + (a_3 + a_4\, \mathfrak{p})\, (b_1 + b_2\, \mathfrak{p}))$

step 5: $c \mathrel{+}= \mathfrak{p}^5\, (a_0\, b_5 + a_5\, b_0)$

Relaxed multiplication $c = a \times b$ - step 6

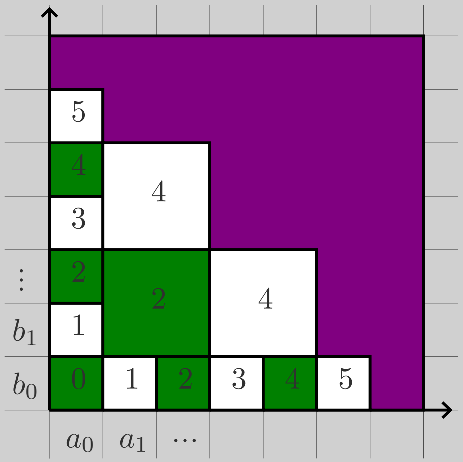

Figure. Minimum knowledge on the input.

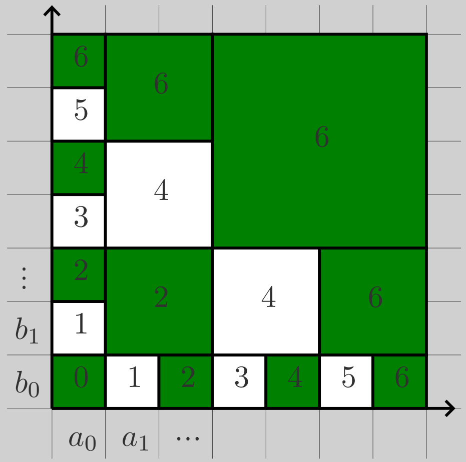

Figure. What we compute.

step 0: $c = a_0 b_0$

step 1: $c \mathrel{+}= \mathfrak{p}\,(a_0 b_1 + a_1 b_0)$

step 2: $c \mathrel{+}= \mathfrak{p}^2\,(a_0 b_2 + a_2 b_0 + (a_1 + a_2 \mathfrak{p})\,(b_1 + b_2 \mathfrak{p}))$

step 3: $c \mathrel{+}= \mathfrak{p}^3\,(a_0 b_3 + a_3 b_0)$

step 4: $c \mathrel{+}= \mathfrak{p}^4\,(a_0 b_4 + a_4 b_0 + (a_1 + a_2 \mathfrak{p})\,(b_3 + b_4 \mathfrak{p}) + (a_3 + a_4 \mathfrak{p})\,(b_1 + b_2 \mathfrak{p}))$

step 5: $c \mathrel{+}= \mathfrak{p}^5\,(a_0 b_5 + a_5 b_0)$

step 6: $c \mathrel{+}= \mathfrak{p}^6\,(a_0 b_6 + a_6 b_0 + (a_1 + a_2 \mathfrak{p})\,(b_5 + b_6 \mathfrak{p}) + (a_5 + a_6 \mathfrak{p})\,(b_1 + b_2 \mathfrak{p})$
$+ (a_3 + \cdots + a_6 \mathfrak{p}^3)\,(b_3 + \cdots + b_6 \mathfrak{p}^3))$

Relaxed multiplication

Theorem [Fischer, Stockmeyer 1974], [van der Hoeven 1997], [Berthomieu, van der Hoeven, Lecerf 2011].

Let a and b be two relaxed $\mathfrak{p}$-adics known up to precision N, then $(a\,b)_0, \ldots, (a\,b)_{N-1}$ can be computed in $\mathsf{R}(N) = O(\mathsf{M}(N) \log N)$ operations.

Definition.

A $\mathfrak{p}$-adic a is recursive of order n_0 if it is solution of an equation $a = \Phi(a)$, where the expression $\Phi(a)_n$ only depends on $a_0, \ldots, a_{n-1}$ for all $n \geq n_0$.

EXAMPLE.

Quotient of two $\mathfrak{p}$-adics is recursive of order 1:

$$c = \frac{a}{b} = \frac{a - (b - b_0)\, c}{b_0} \quad , \quad c_0 = a_0 / b_0 \bmod \mathfrak{p}.$$

$$\frac{a - \mathfrak{p}\left(\frac{b - b_0}{\mathfrak{p}}\right) c}{b_0}$$

$\gamma := \frac{(b - b_0)}{\mathfrak{p}}\, c = (b_1 + b_2\, \mathfrak{p} + b_3\, \mathfrak{p}^2 + \cdots)\, c.$

step 0: $\gamma = b_1\, c_0$ $\rightsquigarrow c_1 = (a_1 - \gamma_0) / b_0$

$\gamma := \frac{(b - b_0)}{\mathfrak{p}}\, c = (b_1 + b_2\, \mathfrak{p} + b_3\, \mathfrak{p}^2 + \cdots)\, c.$

step 0: $\gamma = b_1\, c_0$ $\rightsquigarrow c_1 = (a_1 - \gamma_0)/b_0$

step 1: $\gamma \mathrel{+}= \mathfrak{p}\,(b_1\, c_1 + b_2\, c_0)$ $\rightsquigarrow c_2 = (a_2 - \gamma_1)/b_0$

$\gamma := \frac{(b - b_0)}{\mathfrak{p}}\, c = (b_1 + b_2\, \mathfrak{p} + b_3\, \mathfrak{p}^2 + \cdots)\, c.$

step 0: $\gamma = b_1\, c_0$ $\rightsquigarrow c_1 = (a_1 - d_0)/b_0$

step 1: $\gamma \mathrel{+}= \mathfrak{p}\,(b_1\, c_1 + b_2\, c_0)$ $\rightsquigarrow c_2 = (a_2 - \gamma_1)/b_0$

step 2: $\gamma \mathrel{+}= \mathfrak{p}^2\,(b_1\, c_2 + b_3\, c_0 + (b_2 + b_3\, \mathfrak{p})\,(c_1 + c_2\, \mathfrak{p}))$ $\rightsquigarrow c_3 = (a_3 - \gamma_2)/b_0$

$\gamma := \frac{(b-b_0)}{\mathfrak{p}}\, c = (b_1 + b_2\,\mathfrak{p} + b_3\,\mathfrak{p}^2 + \cdots)\, c.$

step 0: $\gamma = b_1\, c_0$ $\rightsquigarrow\ c_1 = (a_1 - d_0)/b_0$

step 1: $\gamma \mathrel{+}= \mathfrak{p}\,(b_1\, c_1 + b_2\, c_0)$ $\rightsquigarrow\ c_2 = (a_2 - \gamma_1)/b_0$

step 2: $\gamma \mathrel{+}= \mathfrak{p}^2\,(b_1\, c_2 + b_3\, c_0 + (b_2 + b_3\,\mathfrak{p})\,(c_1 + c_2\,\mathfrak{p}))$ $\rightsquigarrow\ c_3 = (a_3 - \gamma_2)/b_0$

step 3: $\gamma \mathrel{+}= \mathfrak{p}^3\,(b_1\, c_3 + b_4\, c_0)$ $\rightsquigarrow\ c_4 = (a_4 - \gamma_3)/b_0$

$\gamma := \frac{(b-b_0)}{\mathfrak{p}}\, c = (b_1 + b_2\,\mathfrak{p} + b_3\,\mathfrak{p}^2 + \cdots)\, c.$

step 0: $\gamma = b_1\, c_0$ $\rightsquigarrow\ c_1 = (a_1 - d_0)/b_0$

step 1: $\gamma \mathrel{+}= \mathfrak{p}\,(b_1\, c_1 + b_2\, c_0)$ $\rightsquigarrow\ c_2 = (a_2 - \gamma_1)/b_0$

step 2: $\gamma \mathrel{+}= \mathfrak{p}^2\,(b_1\, c_2 + b_3\, c_0 + (b_2 + b_3\,\mathfrak{p})\,(c_1 + c_2\,\mathfrak{p}))$ $\rightsquigarrow\ c_3 = (a_3 - \gamma_2)/b_0$

step 3: $\gamma \mathrel{+}= \mathfrak{p}^3\,(b_1\, c_3 + b_4\, c_0)$ $\rightsquigarrow\ c_4 = (a_4 - \gamma_3)/b_0$

step 4: $\gamma \mathrel{+}= \mathfrak{p}^4\,(b_1\, c_4 + b_5\, c_0 + (b_2 + b_3\,\mathfrak{p})\,(c_3 + c_4\,\mathfrak{p}) + (b_4 + b_5\,\mathfrak{p})\,(c_1 + c_2\,\mathfrak{p}))$ $\rightsquigarrow\ c_5 = (a_5 - \gamma_4)/b_0$

Result:

Relaxed division $c = a/b$ in time: $\mathrm{R}(N) + O(N)$.

Recursive series

```
#define TMPL template<typename C, typename V>
TMPL
class recursive_series_rep: public Series_rep {
public:
  Series eq;
public:
  inline recursive_series_rep (const Format& fm):
    Series_rep (fm) {}
  virtual Series initialize () = 0;
  C& initial (nat n2) {
    if (n2>=this->n) {
      this->n = n2+1;
      this->Set_order (this->n); }
    return this->a[n2]; }
  virtual void Increase_order (nat l) {
    Series_rep::Increase_order (l);
    increase_order (eq, l); }
  inline C next () { return eq[this->n]; }
};

TMPL
class recursive_container_series_rep: public Series_rep {
  Series f;
public:
```

```
  recursive_container_series_rep (const Series& f2):
    Series_rep (CF(f2)), f (f2) {
    Recursive_series_rep* rep=
      (Recursive_series_rep*) f.operator -> ();
    rep->eq= rep->initialize (); }
  ~recursive_container_series_rep () {
    Recursive_series_rep* rep=
      (Recursive_series_rep*) f.operator -> ();
    rep->eq= Series (); }
  syntactic expression (const syntactic& z) const {
    return flatten (f, z); }
  virtual void Increase_order (nat l) {
    Series_rep::Increase_order (l);
    increase_order (f, l); }
  C next () { return f[this->n]; }
};
```

Exponential of a series

```
template<typename Op, typename C, typename V>
class unary_recursive_series_rep: public Recursive_series_rep {
protected:
  Series f;
  bool with_init;
  C c;
public:
  unary_recursive_series_rep (const Series& f2):
    Recursive_series_rep (CF(f2)), f (f2), with_init (false) {}
  unary_recursive_series_rep (const Series& f2, const C& c2):
    Recursive_series_rep (CF(f2)), f (f2), with_init (true), c (c2) {}
  syntactic expression (const syntactic& z) const {
    return Op::op (flatten (f, z)); }
  virtual void Increase_order (nat l) {
    Recursive_series_rep::Increase_order (l);
    increase_order (f, l); }
  Series initialize () {
    if (with_init) this->initial (0)= c;
    else {
      ASSERT (Op::nr_init () <= 1, "wrong number of initial conditions");
      if (Op::nr_init () == 1)
        this->initial (0)= Op::op (f[0]);
    }
```

```
    return Op::def (Series (this->me ()), f); }
};
```

Block representation

Remark.

A $\mathfrak{p}$-adic $a = \sum_{n=0}^{\infty} a_n \mathfrak{p}^n$ can be seen as a $\mathfrak{p}^k$-adic $A = \sum_{n=0}^{\infty} A_n \mathfrak{p}^{kn}$ with

$$A_n = \sum_{i=0}^{k-1} a_{kn+i} \mathfrak{p}^i.$$

Example.

$y = 1 + 2^2 + 2^3 + 2^5 + 2^8 + O(2^{10}) = 1 + 3 \times 4 + 2 \times 4^2 + 4^4 + O(4^5)$.

The bigger the integers, the faster their product is computed.

Timings for $\mathbb{Z}_p$

n	8	16	32	64	128	256	512	1024	2048
Naive multiplication	4	7	15	35	95	300	1000	3700	14000
Relaxed multiplication	9	21	44	93	200	420	920	2000	4800
Relaxed mult. with blocks of size 32				65	180	410	900	2000	4500
Gmp's extended g.c.d.	3	6	14	35	92	250	730	2200	5600
Maple 13	240	320	520	1200	3500	11000	38000	160000	∞
Pari/Gp	0.68	1.1	2.8	8.5	28	99	360	1300	4800

Table 1. Divisions for $p = 536870923$, in microseconds.

Recursive $\mathfrak{p}$-adics

Example.

$a \in \mathbb{Z}_p$ with $a_0 \neq 0$, $b_0^r = a_0 \bmod p$. The rth root b of a is recursive of order 1:

$$b = \frac{a - b^r + r\, b_0^{r-1}\, b}{r\, b_0^{r-1}}, \qquad b_0^r = a_0 \bmod p.$$

$\rightsquigarrow$ Improvement of formulae found in [van der Hoeven 2002] for $\mathbb{C}[[X]]$.

```
Mmx] set_output_order (x, 10); minus_one == -p_adic (1, 5)
```

$4 + 4\,p + 4\,p^2 + 4\,p^3 + 4\,p^4 + 4\,p^5 + 4\,p^6 + 4\,p^7 + 4\,p^8 + 4\,p^9 + O(p^{10})$

```
Mmx] i == separable_root (minus_one, 2)
```

$2 + p + 2\,p^2 + p^3 + 3\,p^4 + 4\,p^5 + 2\,p^6 + 3\,p^7 + 3\,p^9 + O(p^{10})$

```
Mmx] set_output_order (i, 1000); i^2-minus_one
```

$O(p^{1000})$

Proposition.

If p and r are coprime, then the rth root $b \in \mathbb{Z}_p$ of $a \in \mathbb{Z}_p$ can be computed at precision N in $O(\log r)\,\mathsf{R}(N)$ operations.

```
template<typename Op, typename M, typename V, typename X>
class binary_scalar_series_rep: public Series_rep {
protected:
  Series f;
  M x;
  C c;
public:
  inline binary_scalar_series_rep (const Series& f2, const X& x2):
    Series_rep (CF(f2)), f(f2), x (as<M> (x2)), c (0) {}
  syntactic expression (const syntactic& z) const {
    return Op::op (flatten (f, z), flatten (x)); }
  virtual void Increase_order (nat l) {
    Series_rep::Increase_order (l);
    increase_order (f, l); }
  virtual M next () {
    return Op::op_mod (f[this->n].rep, x.rep, M::get_modulus (), c); }
};
```

IMPORTANT LEMMA

HENSEL'S LEMMA (NEWTON – HENSEL OPERATOR).

Let $\boldsymbol{Y} = (Y_1, \ldots, Y_r)$. Let $\boldsymbol{P}(\boldsymbol{Y}) \in R[\boldsymbol{Y}]^s$ be such that $\boldsymbol{P}(\boldsymbol{y}_0) = \boldsymbol{0} \bmod \mathfrak{p}$.

If $\mathrm{d}\,\boldsymbol{P}_{\boldsymbol{y}_0}$ is invertible, then

$$\exists!\, \boldsymbol{y} \in R_{\mathfrak{p}}^r,\ (\boldsymbol{y})_0 = \boldsymbol{y}_0,\ \boldsymbol{P}(\boldsymbol{y}) = \boldsymbol{0}.$$

COROLLARY.

$\rightsquigarrow$ $r = s$.

$\rightsquigarrow$ Complexity to compute $\boldsymbol{y}$ depends on r and on the evaluation complexity of $\boldsymbol{P}$.

Shifted algorithms

Definition.

- An operator Φ is recursive if $\Phi(y)_n$ only depends on $y_0, \ldots, y_{n-1}$.
- A recursive operator Ψ is a shifted algorithm is the shift is made explicit.

Example.

Recursive operator Φ	Shifted algorithm Ψ
$\Phi(y) = \frac{a-(b-b_0)\,y}{b_0}, \quad y_0 = a_0/b_0$	$\Psi(y) = \frac{a - \mathfrak{p}\left(\left(\frac{b-b_0}{\mathfrak{p}}\right) y\right)}{b_0}$
$\Phi(y) = y^2 + \mathfrak{p}, \quad y_0 = 0$	$\Psi(y) = \mathfrak{p}^2 \left(\frac{y}{\mathfrak{p}}\right)^2 + \mathfrak{p}$

Polynomial to recursive equation

Let $P \in R_{\mathfrak{p}}[Y]$ and let y_0 be a simple root of P modulo $\mathfrak{p}$.

Hensel's Lemma.

There exists a unique $y \in R_{\mathfrak{p}}$ such that $P(y) = 0$ and $y = y_0 \bmod \mathfrak{p}$.

Idea of Proof.

Write $$P(Y) = P(y_0) + P'(y_0)(Y - y_0) + (Y - y_0)^2\, Q(Y).$$

Then $$0 = P(y) = P'(y_0)\, y + \underbrace{(P(y_0) - P'(y_0)\, y_0 + (y - y_0)^2\, Q(y))}_{\text{Coefficient in } \mathfrak{p}^n \text{ involves only } y_0, \ldots, y_{n-1}}.$$

Recursive equation: $$y = \frac{P(y_0) - P'(y_0)\, y_0 + (y - y_0)^2\, Q(y)}{-P'(y_0)}.$$

Write $$P(Y) = P(y_0) + P'(y_0)(Y - y_0) + (Y - y_0)^2\, Q(Y).$$

Then $$0 = P(y) = P'(y_0)\, y + \underbrace{(P(y_0) - P'(y_0)\, y_0 + (y - y_0)^2\, Q(y))}_{\text{Coefficient in } \mathfrak{p}^n \text{ involves only } y_0, \ldots, y_{n-1}}.$$

Recursive equation: $$y = \frac{P(y_0) - P'(y_0)\, y_0 + \mathfrak{p}^2 \left(\frac{y - y_0}{\mathfrak{p}}\right)^2 Q(y)}{-P'(y_0)}.$$

Simple root lifting of dense univariate polynomials in $R_{\mathfrak{p}}$

Theorem [Berthomieu, Lebreton 2012].

Let $P \in R_{\mathfrak{p}}[Y]$ of degree d and let y_0 be a simple root of P modulo $\mathfrak{p}$. Let $y \in R_{\mathfrak{p}}$ be the unique root lifted from y_0. Then, one can compute y at precision N in time

$$d\,\mathsf{R}(N) + O(N).$$

Proof.

Compute the fixed point of the recursive equation:

$$y = \Psi(y) := \frac{P(y_0) - P'(y_0)\,y_0 + \mathfrak{p}^2\left(\left(\frac{y - y_0}{\mathfrak{p}}\right)^2 Q(y)\right)}{-P'(y_0)}.$$

Comparison.

Zealous representation and Newton operator: $(3\,d + 4)\,\mathsf{M}(N) + O(N)$.

Timings for $\mathbb{Z}_p$

n	4	16	64	256	1024	2048	4096	2^{14}	2^{16}
Naive multiplication	0.0079	0.052	0.29	2.9	39	152	600	9500	150000
Naive mult. with blocks of size 32			0.32	0.60	2.9	8.9	31	440	6700
Naive mult. with blocks of size 1024						20	27	120	1300
Relaxed multiplication	0.21	0.13	0.65	2.9	14	31	71	400	2400
Relaxed mult. with blocks of size 32			0.33	0.73	4.1	11	32	240	1700
Relaxed mult. with blocks of size 1024						20	30	170	1400
Newton	0.0090	0.023	0.079	0.52	4.1	11	29	170	870

Table 2. Solving a polynomial of dense size 8 with $p = 536871001$, in milliseconds.

n	4	16	64	256	1024	2048	4096	2^{14}	2^{16}
Naive multiplication	0.086	0.71	4.4	46	640	2500	9800	160000	∞
Naive mult. with blocks of size 32			100	110	140	240	610	8000	120000
Naive mult. with blocks of size 1024						12000	13000	14000	35000
Relaxed multiplication	0.25	2.3	12	54	250	560	1300	7200	42000
Relaxed mult. with blocks of size 32			110	110	160	270	600	4200	30000
Relaxed mult. with blocks of size 1024						12000	13000	15000	34000
Newton	0.21	0.89	8.0	86	720	2000	5300	30000	140000

Table 3. Solving a polynomial of dense size 128 with $p = 536871001$, in milliseconds.

PROBLEM.

Let $B \in \mathrm{GL}_r(R_{\mathfrak{p}})$, $A \in \mathcal{M}_{r,1}(R_{\mathfrak{p}})$. Find $C \in \mathcal{M}_{r,1}(R_{\mathfrak{p}})$ such that

$$B \cdot C = A.$$

COMPARISON.

- Newton iteration: $O(r^{\omega}\,\mathsf{M}(N))$.
- Relaxed algorithm:
  - Compute $C = B^{-1} \cdot A \in \mathcal{M}_{r,1}(R_{\mathfrak{p}})$ with

$$C = B_0^{-1} \cdot \left(A - \mathfrak{p}\left(\frac{B - B_0}{\mathfrak{p}} \cdot C\right)\right), \quad C_0 = B_0^{-1}\, A_0 \bmod \mathfrak{p}.$$

  - Cost: $O(r^2\,\mathsf{R}(N) + r^{\omega})$.
  - Related to a divide-and-conquer approach on the precision of $\mathfrak{p}$-adics.

TIMINGS FOR $\mathbb{Z}_p$

n	4	16	64	256	1024	4096	2^{14}	2^{16}
Newton	0.097	0.22	0.89	6.8	59	490	3400	20000
MMX	0.15	0.61	3.1	8.1	38	335	1600	14000
Variant	Naive	Naive	Naive	Naive 32	Naive 32	Naive 32	Naive 1024	Naive 1024

Table 4. Solving a linear system of size $r=8$ with $p=536871001$, in milliseconds.

n	4	16	64	256	1024
Newton	930	2600	14000	140000	1300000
MMX	3600	18000	53000	150000	1000000
Variant	Naive	Naive	Naive	Naive 32	Naive 32

Table 5. Solving a linear system of size $r=128$ with $p=536871001$, in milliseconds.

Let $\boldsymbol{P} \in R_{\mathfrak{p}}[\boldsymbol{Y}]^r$ and let $\boldsymbol{y}_0$ be a regular root of P modulo $\mathfrak{p}$.

Definitions.

For j, k such that $1 \leq j \leq k \leq r$, let $\boldsymbol{Q}^{(j,k)} \in \mathcal{M}_{r,1}(R[\boldsymbol{Y}])$ such that

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

Evaluate $\boldsymbol{Y} \leftarrow \boldsymbol{y}$

$$\begin{aligned}\mathbf{0} &= \boldsymbol{P}(\boldsymbol{y})\\ \mathbf{0} &= \mathrm{d}\,\boldsymbol{P}(\boldsymbol{y}_0)\cdot\boldsymbol{y}\\ &\quad+\underbrace{\left(\boldsymbol{P}(\boldsymbol{y}_0)-\mathrm{d}\,\boldsymbol{P}(\boldsymbol{y}_0)\cdot\boldsymbol{y}_0+\sum_{1\le j\le k\le r}\boldsymbol{Q}^{(j,k)}(\boldsymbol{y})\,(y_j-y_{j,0})\,(y_k-y_{k,0})\right)}\end{aligned}$$

Coefficient in $\mathfrak{p}^n$ involves only $\boldsymbol{y}_0,\ldots,\boldsymbol{y}_{n-1}$.

Recursive equation:

$$\boldsymbol{y}=-\mathrm{d}\,\boldsymbol{P}(\boldsymbol{y}_0)^{-1}\left(\boldsymbol{P}(\boldsymbol{y}_0)-\mathrm{d}\,\boldsymbol{P}(\boldsymbol{y}_0)\cdot\boldsymbol{y}_0+\sum_{1\le j\le k\le r}\boldsymbol{Q}^{(j,k)}(\boldsymbol{y})\,(y_j-y_{j,0})\,(y_k-y_{k,0})\right).$$

Recursive equation:

$$\boldsymbol{y}=-\mathrm{d}\,\boldsymbol{P}(\boldsymbol{y}_0)^{-1}\left(\boldsymbol{P}(\boldsymbol{y}_0)-\mathrm{d}\,\boldsymbol{P}(\boldsymbol{y}_0)\cdot\boldsymbol{y}_0+\mathfrak{p}^2\left(\sum_{1\le j\le k\le r}\boldsymbol{Q}^{(j,k)}(\boldsymbol{y})\left(\frac{y_j-y_{j,0}}{\mathfrak{p}}\right)\left(\frac{y_k-y_{k,0}}{\mathfrak{p}}\right)\right)\right).$$

THEOREM [BERTHOMIEU, LEBRETON 2012].

Let $\boldsymbol{P}$ be an algebraic system in $R_{\mathfrak{p}}[\boldsymbol{Y}]^r$ such that $\deg_{Y_i} \boldsymbol{P} < d$. Let $\boldsymbol{y}_0$ be a regular root of $\boldsymbol{P}$ modulo $\mathfrak{p}$ and $\boldsymbol{y} \in R_{\mathfrak{p}}^r$ be the unique lifted root from $\boldsymbol{y}_0$.

Then one can compute $\boldsymbol{y}$ at precision N in time

$$r\, d^r\, \mathsf{R}(N) + O(r^2\, N + r^{\omega}).$$

Conclusion

Two general paradigms:

Newton operator	Relaxed algorithms
Solve implicit equations	Solve recursive equations
Faster for higher precision	
	Less relaxed arithmetic operations Can increase the precision without doubling it

Follow-up

- What can be done if the solution is not regular modulo $\mathfrak{p}$?

  $\rightsquigarrow$ Computation of the pth root in $\mathbb{Z}_p$.
  $\rightsquigarrow$ Extension to the more general case.

- How to use the structure of some linear systems?

```
Mmx] set_output_order (x, 10); a == p_adic (361, 2)
```

$$1+p^3+p^5+p^6+p^8+O(p^{10})$$

```
Mmx] b == pth_root (a)
```

$$1+p^2+p^3+p^5+p^6+p^7+p^8+p^9+O(p^{10})$$

```
Mmx] d == p_adic (6377+5*101^2+48*101^5, 101)
```

$$14+63\,p+5\,p^2+48\,p^5+O(p^{10})$$

```
Mmx] e == pth_root (d)
```

$$14+30\,p+72\,p^2+67\,p^3+64\,p^4+69\,p^5+27\,p^6+50\,p^7+p^8+15\,p^9+O(p^{10})$$

Series in Mathemagix language

```
class Series (R: Type) == {
  c: Vector R;
  f: Int -> R;
  constructor series (f2: Int -> R) == {
    c == [];
    f == f2;
  }
}

forall (R: Type) {
  convert (f: Int -> R): Series R == series f;
  postfix [] (f: Series R, i: Int): R == {
    while #f.c <= i do append (f.c, [ f.f (#f.c) ]);
    return f.c[i];
  }
  flatten (f: Series R): Syntactic == {
    r: Syntactic := flatten (0);
    for i: Int in 0..series_order do
      r := r + flatten (f[i]) * flatten ('x) ^ flatten (i);
    r := r + apply (flatten ('O), flatten ('x) ^ flatten (series_order));
    return r;
  }
  infix << (p: Port, f: Series R): Port == p << flatten f;
```

```
}

forall (C: Type, D: Type)
map (f: C -> D, s: Series C): Series D ==
  lambda (i: Int): D do f s[i];

forall (R: Ring) {
  series (p: Polynomial R): Series R ==
    lambda (i: Int): R do p[i];
  as_series (v: Vector R): Series R ==
    series as_polynomial v;
  series (t: Tuple R): Series R ==
    series polynomial t;
  upgrade (c: R): Series R == series (c);
}
```

```
forall (R: Type) {
  private_recursive (data: Pointer Series R, init: Vector R): Series R ==
    lambda (i: Int): R do
      if i < # init then init[i] else data[1][i];
  public_recursive (data: Vector Series R): Series R ==
    lambda (i: Int): R do data[0][i];
  recursive (Phi: Series R -> Series R, init: Vector R): Series R == {
    dummy: Series R;
    data : Vector Series R := [ dummy, dummy ];
    data[0] := private_recursive (coefficients data, init);
    data[1] := Phi data[0];
    return public_recursive data;
  }
  recursive (Phi: Series R -> Series R, init: R): Series R ==
    recursive (Phi, [ init ]);
}

forall (R: Type) {
  as_series (v: Vector Series R): Series Vector R ==
    lambda (i: Int): Vector R do [ v[k][i] | k: Int in 0..#v ];
  vector_access (f: Series Vector R, j: Int): Series R ==
    lambda (i: Int): R do f[i][j];
  as_vector (f: Series Vector R): Vector Series R ==
```

```
    [ vector_access (f, i) | i: Int in 0..#f[0] ];
  as_vector (f: Series Vector R, n: Int): Vector Series R ==
    [ vector_access (f, i) | i: Int in 0..n ];
  recursive (Phi: Vector Series R -> Vector Series R,
             init: Vector Vector R): Vector Series R == {
    Phi2 (f: Series Vector R): Series Vector R ==
      as_series Phi as_vector (f, #init);
    init2: Vector Vector R ==
      [ [ init[j][i] | j: Int in 0..#init ] | i: Int in 0..#init[0] ];
    return as_vector (recursive (Phi2, init2), #init);
  }
  recursive (Phi: Vector Series R -> Vector Series R,
             init: Vector R): Vector Series R ==
    recursive (Phi, [ [ c ] | c: R in init ]);
}
```

Thank you

for your attention!