p-adics in FLINT

Jean-Pierre Flori

ANSSI

September 4, 2013

FLINT: Fast Library for Number Theory

- C library on top of GMP/MPIR, MPFR (with support for NTL).
- FLINT 1 (2007/xx – 2010/12) originally developed by Hart, Harvey and Novocin.
- FLINT 2 (2011/01 –) is a completele rewrite by Hart, Johansson and Pancratz.
- About 130k lines of C code.
- Used by Sage since 2007.
- Used by Singular since 2011/12, code by Martin Lee; not used in Sage, see trac ticket 13331.

p-adics in FLINT

- `padic` module in FLINT 2 since version 2.2 (released 2011/06/04), mostly by Pancratz.
- `padic_poly`, `padic_matrix` and `qadic` modules on Pancratz's github since a few years, to be included into version 2.4.
- About 14k lines of C code.
- backward uncompatible changes between versions 2.3 and 2.4 (more on that later).

p-adics in Sage using FLINT

- Unramified p-adics implementation using the new template interface.
- See trac ticket 14304 and `https://github.com/saraedum/sage-renamed/tree/Zq`.
- This relies on the `fmpz_mod_poly` module.
- No implementation using the `padic`, `padic_poly` and `qadic` modules yet?

Other applications

- Point counting using deformation theory available on Pancratz's github.
- Point counting à la Satoh, ..., Harley using a custom `qadic_dense` module available on my github.
- Both of these are base on version 2.3, so have to be rebased.

Design decisions

Decision.

- Each p-adic operation treats the input as exact data and requires the desired output precision as a separate argument.

Rationale.

- A number is just a number.
- The intrinsic difficulty in p-adic arithmetic stems from the precision loss, which depends on the particular operation.
- Note that it would be straightforward to implement various precision models on top of this.

Design decisions

An element $x \neq 0$ is typically stored as $x = pu$ with $v = \mathrm{ord}_p(x) \in \mathbb{Z}$ and $u \in \mathbb{Z}$ with $p \nmid u$.

In 2.3 and before.

```
typedef struct {
    fmpz u ;
    long v ;
} padic_struct ;
```

After 2.3.

```
typedef struct {
    fmpz u;
    slong v;
    slong N;
} padic_struct;
```

Design decisions

Additional information stored in a context object.
In 2.3 and before.

```
typedef struct {
    fmpz_t p;
    long N;

    double pinv;

    fmpz *pow;
    long min;
    long max;

    enum padic_print_mode mode;
} padic_ctx_struct;
```

After 2.3 the precision is not stored anymore.

Design decisions

Remarks.

- Improved maintainability by having one data type; no special case depending on the size of p or p^N;
- One could consider a different implementation performing basic arithmetic to base p^k with k s.t. such that p^k fits in a word. This would allow replacing mod p^N operations by mod p^k operations (with a precomputed word-sized inverse) in many algorithms.

Functions for $\mathbb{Q}_p$

- Addition, subtraction, negation
- Multiplication, powers
- Inversion
- Inversion (with precomputed lifting structure)
- Division
- Square root
- Exponential
- Logarithm
- Teichmueller lift

Benchmarks for $\mathbb{Q}_p$

We present some timings for arithmetic in $\mathbb{Q}_p \mod p^N$ where $p = 17$, $N = 2^i$, $i = 0, ..., 10$, comparing the three systems Magma (V2.19-2), Sage (current github, 5.12.beta4) and FLINT (current github) on a machine with Intel Core i7-2620M CPU running at 2.70GHz.

To avoid worrying about taking the same random sequences of elements, we instead fix elements $a = 3^{3N}$, $b = 5^{2N}$ (and variations thereof) modulo p^N .

We consider the following operations:

- Addition
- Multiplication
- Inversion
- Square root
- Teichmueller lift
- Exponential
- Logarithm

Addition

Signature

```
void padic_add(z, x, y, ctx)
```

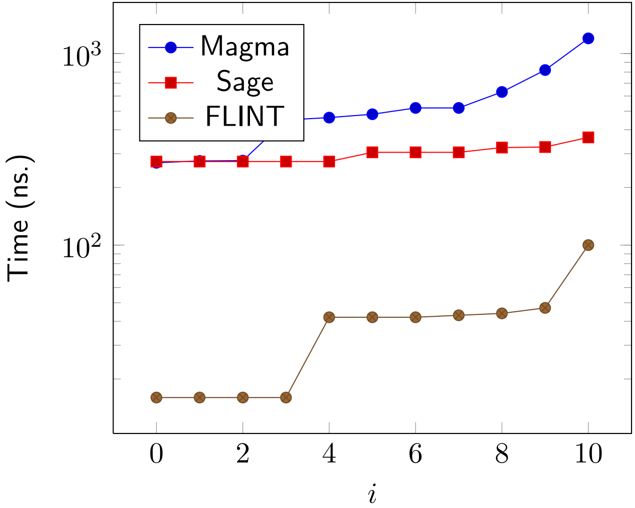
Contract

Assumes that x and y are reduced modulo p^N and returns z in reduced form, too.

Algorithm

Avoids expensive modulo operation, replacing this by one comparison and at most one subtraction.

Addition



Multiplication

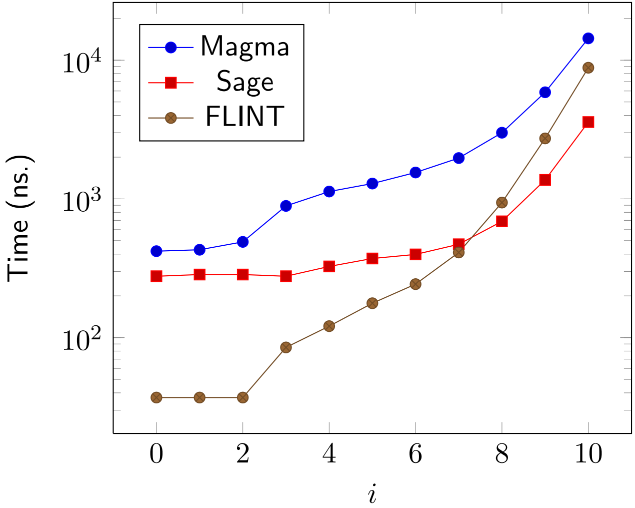
Signature

```
void padic_mul(z, x, y, ctx)
```

Contract

Makes no assumptions on x and y, returns z reduced modulo p^N.

Multiplication



Inversion

Signature

```
void padic_inv(z, x, ctx)
```

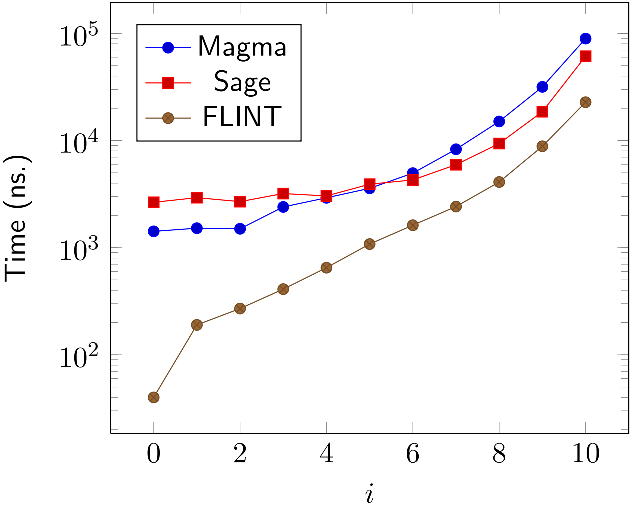
Contract

Makes no assumptions on x.

Algorithm

Hensel lifting on $g(X) = xX - 1$, starting from an inverse in $\mathbb{F}_p$ and using the update formula $z = z + z(1 - xz)$.

Inversion



Square root

Signature

```
int padic_sqrt(z, x, ctx)
```

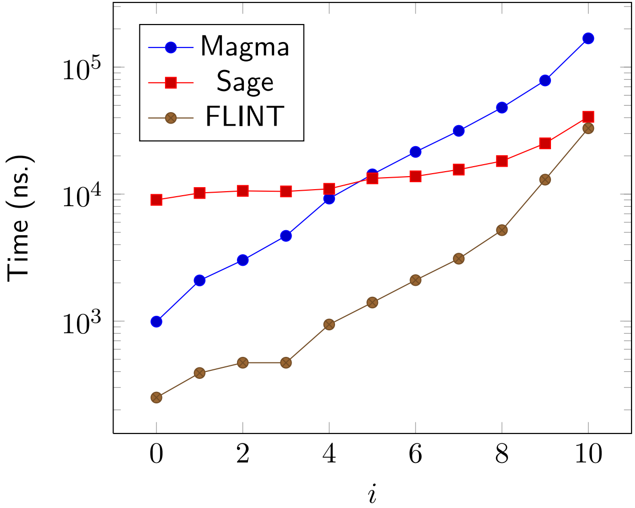
Contract

Makes no assumptions on x. Returns whether x is actually a square and if so computes its square root.

Algorithm

- Hensel lifting to compute an inverse square root to half precision.
- The final step performs the needed inversion as well.

Square root



Teichmueller lift

Signature

```
void padic_teichmuller(z, x, ctx)
```

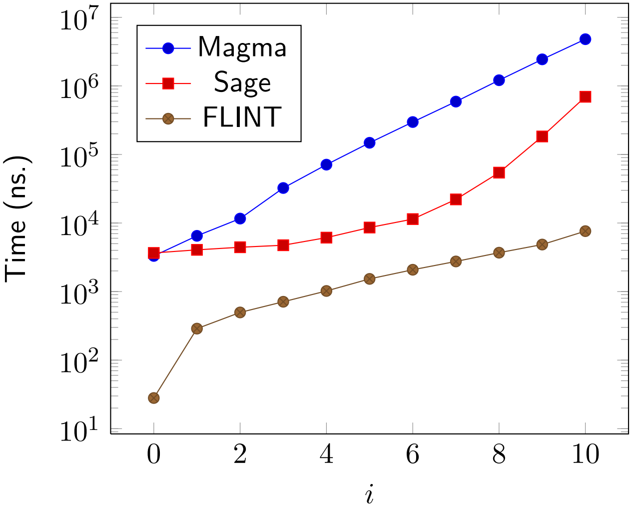
Contract

Assumes only that $\mathrm{ord}_p(x) = 0$.

Algorithm

Hensel lifting, avoiding inversions.

Teichmueller lift



Exponentiation

Signature

```
int padic_exp(z, x, ctx)
```

Contract

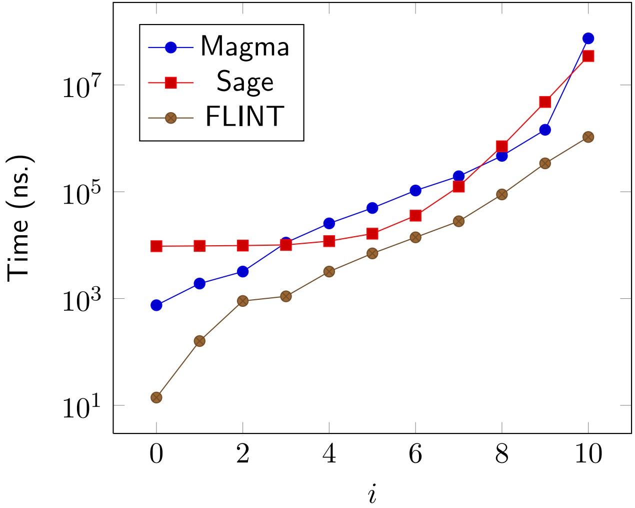
Return whether the series converges, and if so computes the exponential.

Algorithm

Evaluate the truncated series, multiplying by the common factorial in denominators, hence requiring only one inversion.

- Rectangular splitting.
- Balanced splitting.

Exponentiation



Logarithm

Signature

```
int padic_log(z, x, ctx)
```

Contract

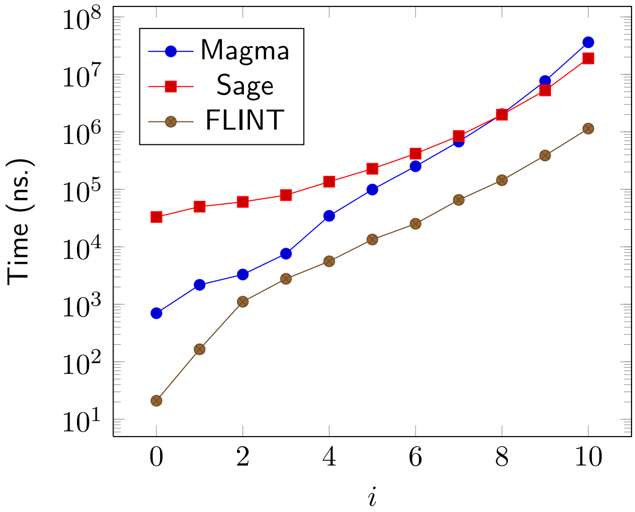
Return whether the series converges, and if so computes the logarithm.

Algorithm

Evaluate the truncated series, performing an inversion for each summand.

- Rectangular splitting.
- Balanced splitting (quasi-linear in N when p is fixed).
- à la SST.

Logarithm



Magma
Sage
FLINT
Time (ns.)
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
0
2
4
6
8
10
i

Polynomials over $\mathbb{Q}_p$

We represent a non-zero polynomial $f(X) \in \mathbb{Q}_p[X]$ as

$$f(X) = p^v(a_0 + a_1 X + \cdots + a_n X^n)$$

where $a_0, \ldots, a_n \in \mathbb{Z}$ and, for at least one i , p does not divide a_i.

Functions for $\mathbb{Q}_p[X]$

- Conversions to polynomials over $\mathbb{Z}$ and $\mathbb{Q}$
- Coefficient manipulation
- Addition, subtraction, negation
- Scalar multiplication
- Multiplication
- Powers
- Series inversion
- Derivative
- Evaluation
- Composition

Unramified extensions $\mathbb{Q}_q$

We represent an unramified extension of $\mathbb{Q}_p$ as

$$\mathbb{Q}_q = \mathbb{Q}_p[X]/(f(X))$$

where $f(X) \mod p$ is separable, storing $f(X)$ in a data structure for sparse polynomials.
This allows for the reduction of a degree n polynomial modulo $f(X)$ in linear time $O(n)$ (but slow Frobenius substitutions...).

Functions for $\mathbb{Q}_q$

- Addition, subtraction, negation
- Multiplication
- Powers
- Inversion
- Exponential
- Logarithm
- Frobenius
- Teichmueller lift
- Trace
- Norm

Benchmarks for $\mathbb{Q}_q$

We present some timings for arithmetic in $\mathbb{Q}_q \mod p^N$ where $p = 17$, $N = 2^i$, $i = 0, ..., 10$, comparing the three systems Magma (V2.19-2), Sage (current github, 5.12.beta4) and FLINT (current github) on a machine with Intel Core i7-2620M CPU running at 2.70GHz.
To avoid worrying about taking the same random sequences of elements, we instead fix elements as before.
We consider the following operations:

- Exponential
- Logarithm
- Frobenius
- Trace
- Norm

Exponential

Signature

```
int qadic_exp(z, x, ctx)
```

Contract

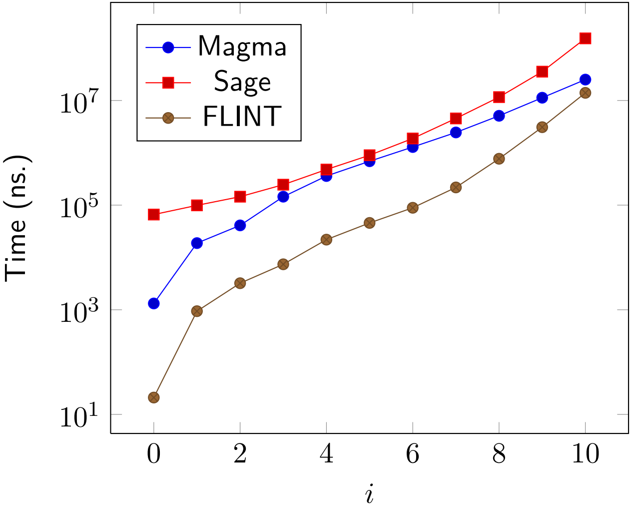
Return whether the series converges, and if so computes the exponential.

Algorithm

Evaluate the truncated series, performing an inversion at each step.

- Rectangular splitting.
- Balanced splitting.

Exponential



Addition

Signature

```
int qadic_log(z, x, ctx)
```

Contract

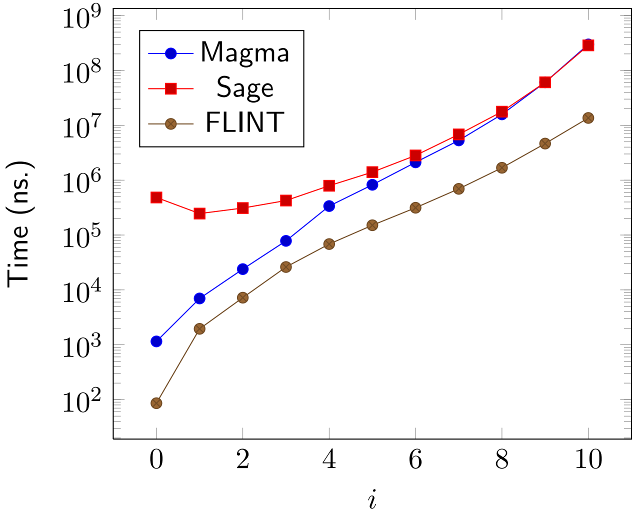
Return whether the series converges, and if so computes the logarithm.

Algorithm

Evaluate the truncated series, performing an inversion for each summand.

- Rectangular splitting.
- Balanced splitting.

Logarithm


Magma
Sage
FLINT
Time (ns.)
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9
0
2
4
6
8
10
i

Frobenius

Signature

```
void qadic_frobenius(z, x, k, ctx)
```

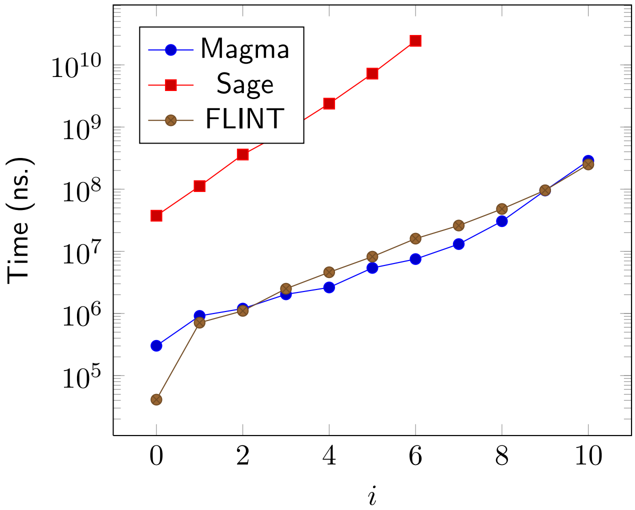
Contract

Computes $z = \Sigma^k(x)$.

Algorithm

- Compute $\Sigma^k(X)$ using Hensel lifting.
- Perform polynomal composition modulo p^N and $f(X)$.
- Generalize to use rectangular splitting.

Frobenius



Trace

Signature

```
void qadic_trace(z, x, ctx)
```

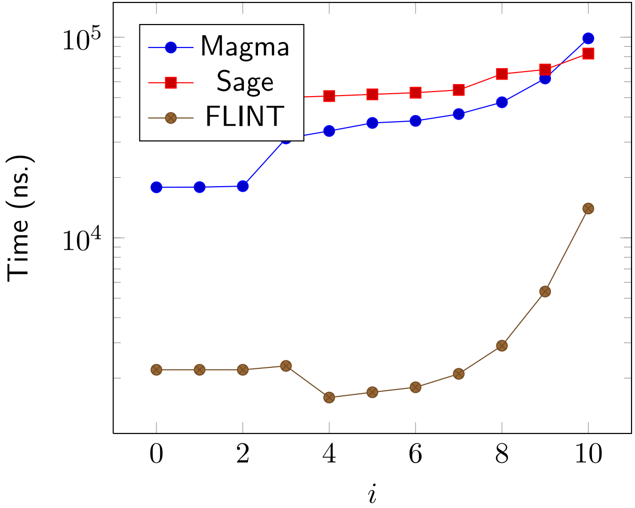
Contract

No assumptions are made on x.

Algorithm

- Compute the traces of X^i iteratively.
- Compute the trace of x.

Trace



Norm

Signature

```
void qadic_norm(z, x, ctx)
```

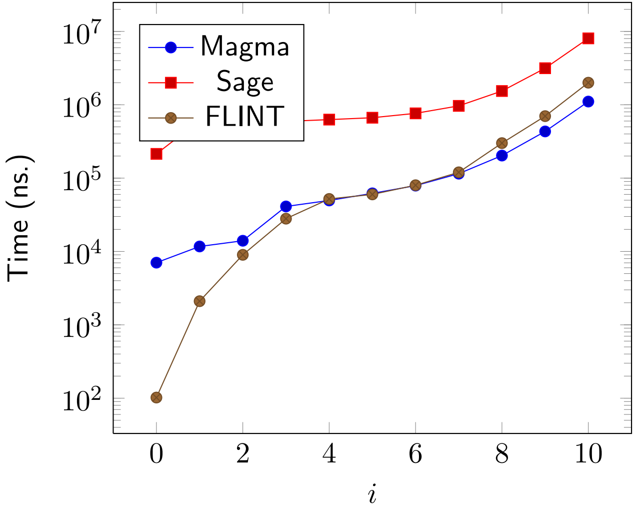
Contract

No assumptions are made on x.

Algorithm

- Using an analytical formula.
- Using resultants.

Norm



Future features?

- Specialize code for finite fields.
- Modular reduction for non-sparse modulus.
- Other types of extensions.
- Specific implementations for $p = 2$.