Fast, rigorous arbitrary-precision numerics with ball arithmetic

Fredrik Johansson

Research Institute for Symbolic Computation, Johannes Kepler University Linz
Supported by Austrian Science Fund (FWF) grant Y464-N18

RISC Algorithmic Combinatorics Seminar, March 2013

Numerics in computer algebra

Typical numerical computation

$\tilde{y} = (y + \varepsilon) = (y + \varepsilon_{\text{round}} + \varepsilon_{\text{trunc}})$

$y \in \mathbb{R}$	Exact mathematical quantity
$\tilde{y} \in \mathbb{Q}$	Computable approximation
ε	Absolute error
$\varepsilon_{\text{round}}$	Rounding error from using finite-precision arithmetic
$\varepsilon_{\text{trunc}}$	Error from truncation of limits (e.g. $\sum_{n=0}^{\infty} \to \sum_{n=0}^{N}$)

Typical end goal

We know that $y \in \mathbb{Z}$. It can be determined from $\tilde{y}$ if we can prove that $|\varepsilon| < 1/2$.

Example: fast computation of the partition function

The Hardy-Ramanujan-Rademacher formula:

$$p(n) = \sum_{k=1}^{N} \frac{\sqrt{k}\, A_k(n)}{\pi\sqrt{2}} \frac{d}{dn} \left(\frac{\sinh \frac{\pi}{k} \sqrt{\frac{2}{3}\left(n - \frac{1}{24}\right)}}{\sqrt{n - \frac{1}{24}}} \right) + \varepsilon_{\text{trunc}}(n, N)$$

where $A_k(n)$ is a certain sum over complex $2k$-th roots of unity

Numerical part of the algorithm

- Determine $N = O(\sqrt{n})$ so that $|\varepsilon_{\text{trunc}}| < 1/4$
- Determine $\text{prec}(n, N, k) = O(\log |\,\text{term}\, k|) + O(\log n)$ so that $\sum |\varepsilon_{\text{round}}| < 1/4$
- Evaluate the terms using asymptotically fast arithmetic

Floating-point arithmetic

Numbers are represented as $m \times 2^e$, $m, e \in \mathbb{Z}$, where m is rounded to at most b bits.

Pros

- ✓ Efficient, well-understood numerical model

Cons

- ✗ Analyzing the error propagation is tedious and difficult, even for simple algorithms
- ✗ If we change the algorithm (for example to improve performance), we must redo the proof
- ✗ Provable *a priori* bounds are sometimes much worse than the actual error, giving poor efficiency

Interval arithmetic

Instead of just computing $\tilde{y} \approx y$, compute an interval I such that $y \in I$ is guaranteed.

Pros

- ✓ Error analysis is mostly needed for "atomic" operations
- ✓ Errors automatically propagate globally
- ✓ Can use simple and fast heuristics instead of *a priori* bounds, and check *a posteriori* that the result is correct

Cons

- ✗ More expensive than floating-point arithmetic
- ✗ Can give too pessimistic error bounds
- ✗ Some algorithms that converge in floating-point arithmetic do not converge in interval arithmetic, and require special care

Two versions of interval arithmetic

Interval (inf-sup) model

$I = [a, b]$, e.g. $[3.141592653, 3.141592654]$

Ball (mid-rad) model

$I = [m - r, m + r] \equiv m \pm r$, e.g. $3.1415926534 \pm 2 \times 10^{-10}$.

The ball model is slightly less accurate and less general. But it is more efficient at high precision since only m needs full precision.

Already used in: iRRAM (N. Müller), Mathemagix (J. van der Hoeven)

Goals for developing a new numerical library

Correctness

- Use ball model to compute with provably correct error bounds

Features

- Complex numbers, special functions
- Polynomials, power series, matrices

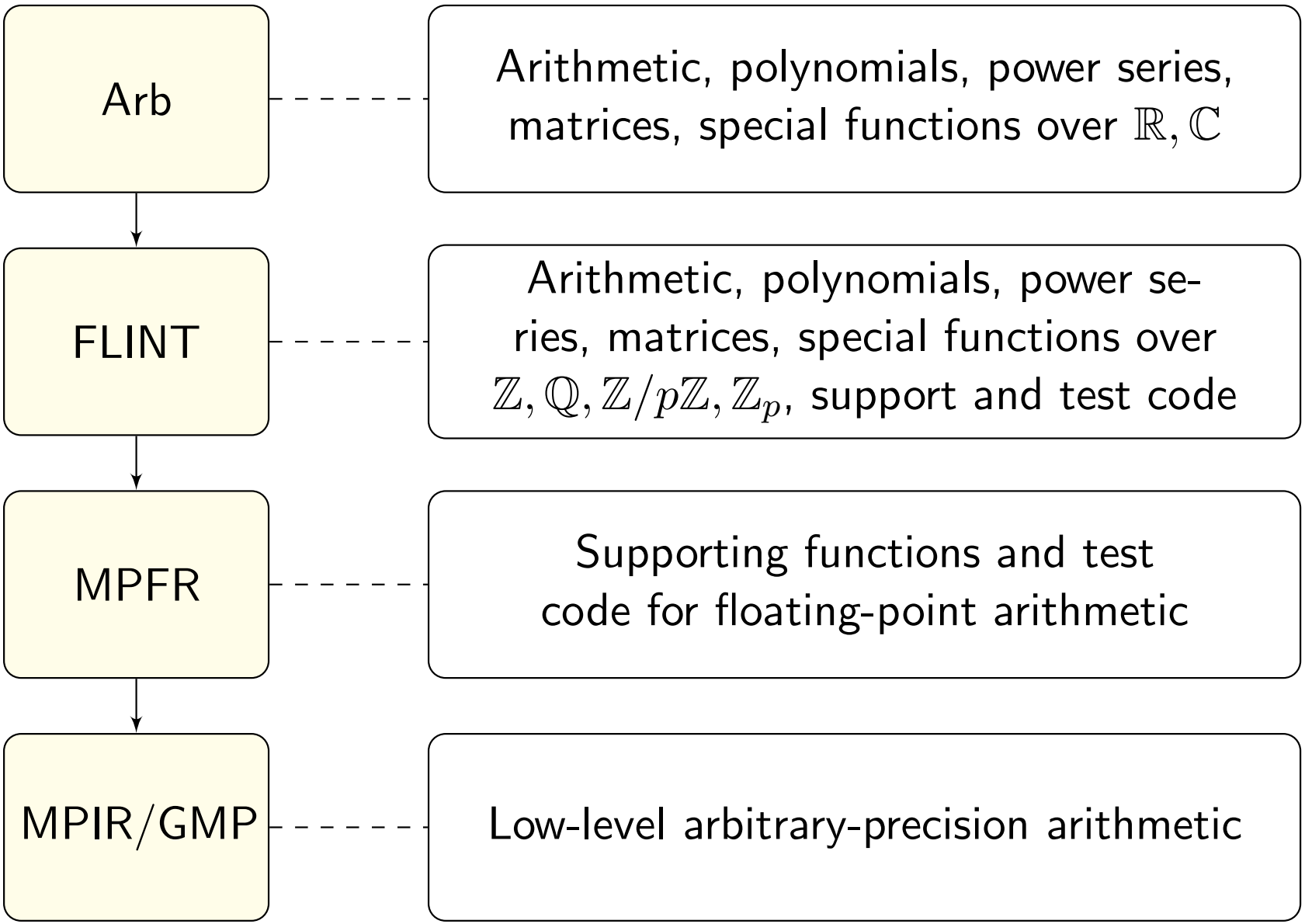
Performance

- Use asymptotically fast algorithms
- Minimize overhead (want $(1 + \varepsilon)$ of optimal speed)
- Support experimenting with different algorithms

Arb

- Library for ball interval arithmetic (ARB = Arbitrary-precision Real Balls)
- Code: `https://github.com/fredrik-johansson/arb/`
- Documentation: `http://fredrikj.net/arb/`
- Licensed GPL v2
- GMP / FLINT style API
- About 30,000 lines of C code ($\sim$ 50% test code)
- Started with some proof-of-concept code in April 2012
- Current codebase started with a full rewrite in August 2012
- Extension of my work on FLINT (and further back, mpmath)

Dependencies



Feature overview: types

`fmpr`: floating-point real numbers
$\mathbb{R}_D = \mathbb{Z} \times 2^{\mathbb{Z}} \cup \{-\infty, +\infty, \mathrm{NaN}\}$. The components are implemented as FLINT integers, and can grow dynamically.

`fmprb`: real numbers implemented as balls
$\mathbb{R}_B = \{[m - r, m + r] : m, r \in \mathbb{R}_D, r \geq 0\}$

`fmpcb`: complex numbers implemented as rectangular balls
$\mathbb{C}_B = \mathbb{R}_B[i]$

`fmprb_poly`, `fmpcb_poly`: polynomials over $\mathbb{R}_B$, $\mathbb{C}_B$

`fmprb_mat`, `fmpcb_mat`: matrices over $\mathbb{R}_B$, $\mathbb{C}_B$

Feature overview: functionality

Special functions

- Elementary functions, $\Gamma(z)$, $\zeta(s, a)$, hypergeometric series

Polynomials

- Fast multiplication, division, composition
- Fast power series arithmetic, composition, special functions
- Fast multipoint evaluation and interpolation
- Root isolation

Matrices

- Arithmetic, nonsingular solving, determinant, inverse

Algorithms for arithmetic

Floating-point operations by default use correct rounding, implemented mostly using arithmetic on FLINT integers.

Ball operations are implemented using floating-point arithmetic:
$(m_1 \pm r_1) \times (m_2 \pm r_2) =$
$\operatorname{round}(m_1 m_2) \pm (|m_1| r_2 + |m_2| r_1 + r_1 r_2 + \varepsilon_{\operatorname{round}})$

All radius operations are done using a fixed, small precision (30 bits). In many places, we save time by not rounding radii to the smallest possible bound (but of course always rounding up to guarantee that the bounds are valid).

Arithmetic performance

Cost of a real multiplication compared to MPFR.

Bits	`fmpr_mul`	`fmprb_mul`
32	0.6	2.3
128	1.4	2.7
512	0.9	1.4
2048	1.1	1.2
8192	1.2	1.2
32768	1.2	1.2
131072	1.1	1.1
524288	1.0	1.0

MPFR is 20% faster around 10^4 bits thanks to mulhigh. Most of the overhead below 1000 bits should be eliminated by a future rewrite of the `fmpr` type.

Algorithms for polynomial arithmetic

Multiplication in $\mathbb{R}_B[x]$ is performed by translating to $\mathbb{Z}[x]$, e.g. $1.2345x + 567.89x^2 \to 123x + 56789x^2$, with error bounds calculated separately.

This allows taking advantage of the fast polynomial multiplication code in FLINT, and appears superior to floating-point FFT.

If the coefficients vary in magnitude, small terms in the output have poor accuracy when compared to standard $O(n^2)$ multiplication. (A recent algorithm by J. van der Hoeven appears to solve this problem.)

Other polynomial operations are transformed to multiplication using various strategies (divide and conquer, Newton iteration, ...).

Algorithms for elementary functions

Mostly implemented by calling MPFR functions with the midpoint as input, performing separate error bounding.

Example bounds for error propagation, $x = m \pm r$

$\exp(x)$	$\exp(m+r) - \exp(m) = \exp(m)(\exp(r) - 1)$
$\log(x)$	$\log(m) - \log(m-r) = \log(1 + r/(m-r))$
$\sin(x), \cos(x)$	$\min(r, 2)$
$f(x)$	$r \sup_{t \in x} f'(t)$

Longer-term goal: faster algorithms, especially for low to mid precision ($<$ 10000 digits).

Computing roots of unity

In some applications (e.g. the Rademacher formula), we need high-precision values of $2q$-th roots of unity (or their real or imaginary parts)

$$\exp\left(\frac{p\pi i}{q}\right) = \cos\left(\frac{p\pi}{q}\right) + i\sin\left(\frac{p\pi}{q}\right)$$

Algorithm 1: evaluate the cosine function (Taylor series + argument reduction techniques, e.g. by calling MPFR)

Algorithm 2: Newton iteration (algebraic numbers). Asymptotically faster if q is small.

Rigorous root polishing using Newton iteration

We are given a polynomial f and an interval I known to contain a single root. We can bound $C = \sup_{t,u \in I} \frac{1}{2}|f''(t)|/|f'(u)|$.

Input: $x = [m - r, m + r] \subseteq I$ known to contain the root.

Output: $x' = [m' - r', m' + r']$ where $m' = m - f(m)/f'(m)$ and $r' = Cr^2$. If $x' \subseteq I$, it is guaranteed to contain the root, and if $r' < r$, we have made progress.

Each iteration roughly doubles the accuracy. Optimal precision steps: $\ldots, b/8, b/4, b/2, b$ bits.

(A version of this also works in complex arithmetic.)

Complex Newton iteration for roots of unity

Algorithm 2A

Evaluate $\exp(p\pi i/q)$ as a root of the polynomial $z^q + 1$.

The power z^q can be evaluated in $O(\log q)$ steps using binary exponentiation, so the cost is $O(M(b) \log q)$.

The drawback is that we have to use complex arithmetic.

We could also use a cyclotomic polynomial, possibly giving a lower degree, but this does not appear to give any advantage since we lose sparsity.

Real Newton iteration for roots of unity

Algorithm 2B

Evaluate $\cos(p\pi/q)$ as a root of its minimal polynomial.

The minimal polynomial is dense and has degree $O(q)$. Using Horner's rule, the cost is $O(M(b)q)$.

This is asymptotically worse than the complex iteration when q grows, but we avoid the overhead of complex arithmetic. In my implementation of the partition function, I found it to be faster.

Faster polynomial evaluation using rectangular splitting

Choose $m \approx \sqrt{n}$ and write the polynomial as an array with m columns.

Precompute the table of powers $[x^2, x^3, \ldots, x^m]$.

Evaluate inner polynomials (rows) using "scalar" multiplications, evaluate outer polynomial using Horner's rule.

$$\begin{array}{rccccc} & 1 & + & 1x & + & 2x^2 \\ P(x) = + & 3x^3 & + & 4x^4 & + & 5x^5 \\ + & 6x^6 & + & 7x^7 & + & 8x^8 \end{array}$$

$$P(x) = ((8x^2 + 7x + 6)x^3 + (5x^2 + 4x + 3))x^3 + (2x^2 + 1x + 1)$$

Analysis of rectangular splitting evaluation

Precision b bits, degree n, coefficients $c \leq b$ bits

Horner's rule

$O(nM(b)) = O(nb^{1+\varepsilon})$

Rectangular splitting

Nonscalar multiplications: $O(n^{1/2}M(b))$
Scalar multiplications: $O(nM(b,c)) \approx O(nb)$

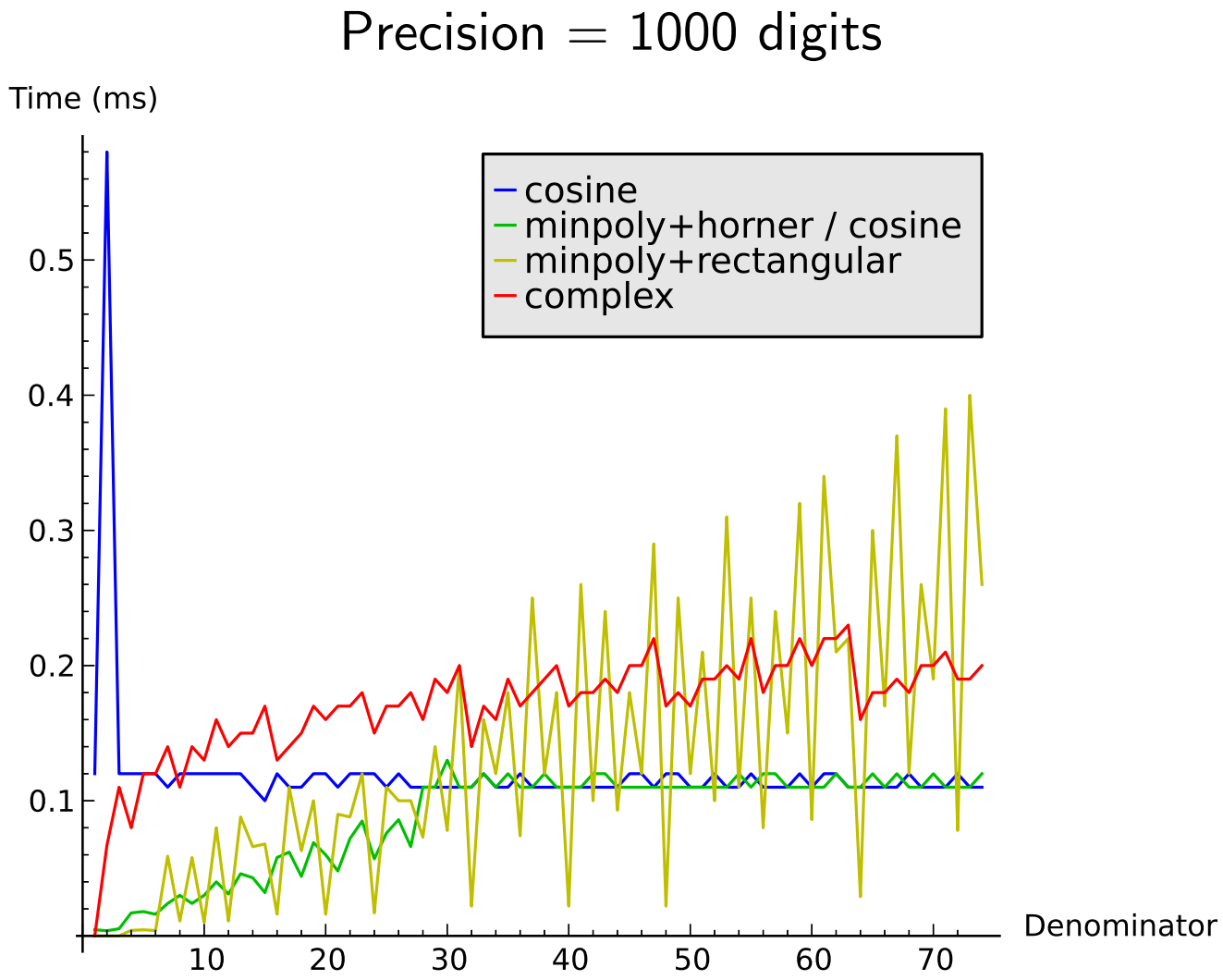
The improvement is theoretically at most b^{ε}, but in practice the nonscalar multiplications dominate if c is small.

Note: we can reuse the table of powers to compute $P(x), P'(x)$ faster simultaneously (useful for Newton iteration).

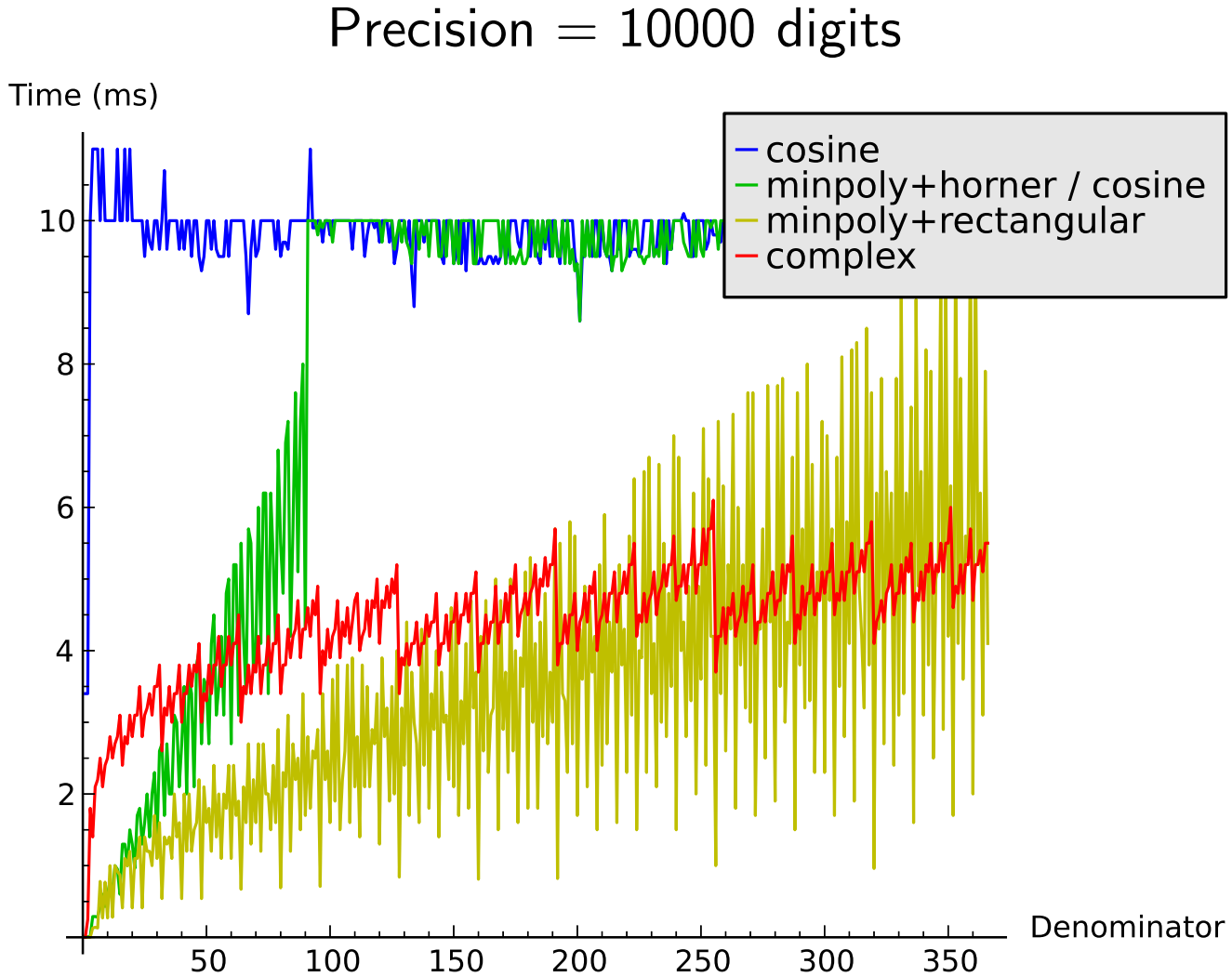
Rectangular splitting, historical notes

- First described by Paterson-Stockmeyer (1973)
- Transposed version for hypergeometric series (where c_{n+1}/c_n is small is small) given by D. M. Smith (1989) ("concurrent summation")
- The same idea is used in the Brent-Kung power series composition algorithm (1978)
- Further analyzed in Brent and Zimmermann, *Modern Computer Arithmetic* (2011)

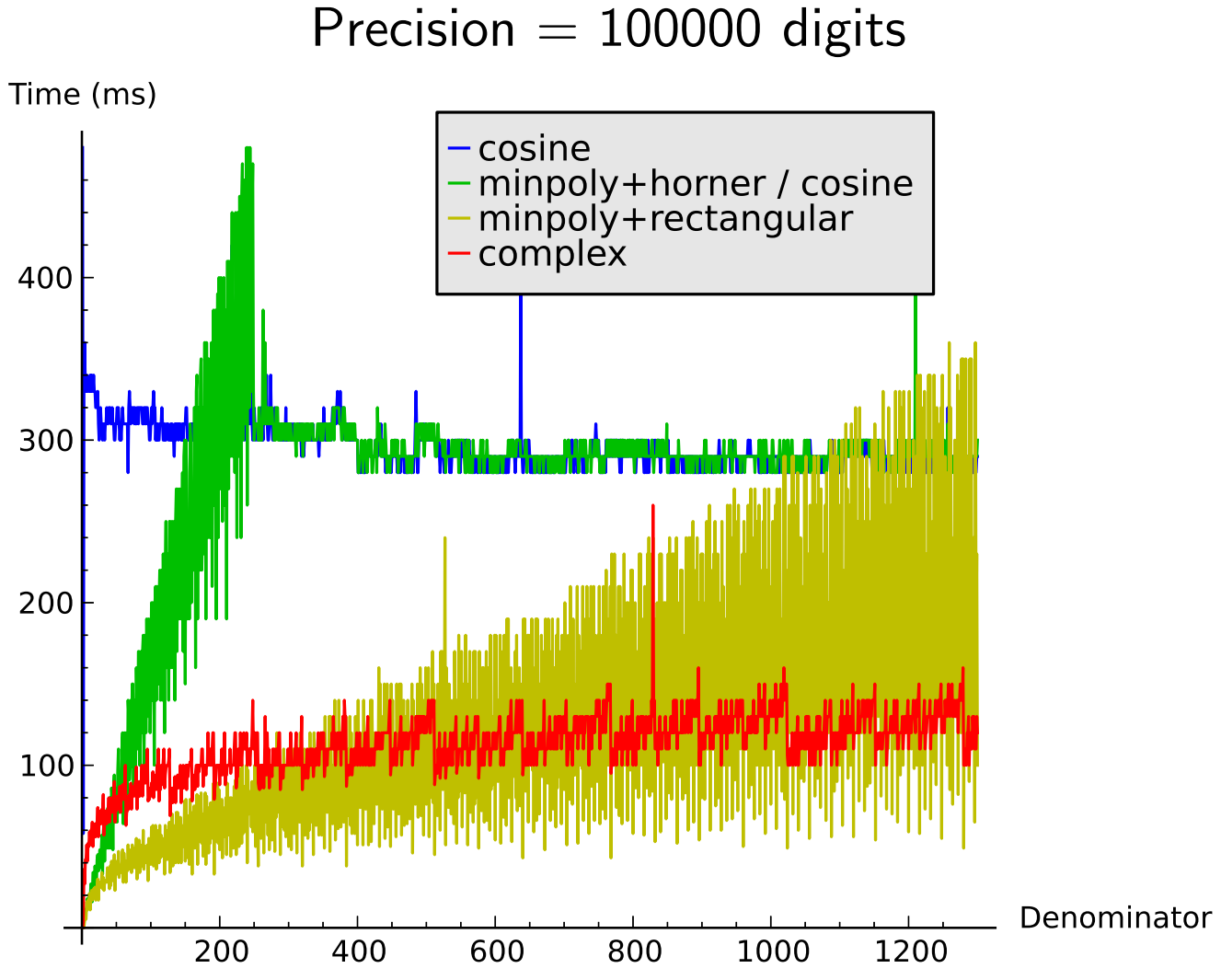
Comparison of algorithms for roots of unity



Comparison of algorithms for roots of unity



Comparison of algorithms for roots of unity



Special functions: Hurwitz zeta function

Arb provides the Hurwitz zeta function $\zeta(s,a) = \sum_{k=0}^{\infty} \frac{1}{(a+k)^s}$ and derivatives with respect to s.

Supports any $s, a \in \mathbb{C}$ (analytic continuation), fast simultaneous computation of $\zeta(s,a), \zeta'(s,a), \ldots \zeta^{(n)}(s,a)$, with rigorous error bounds.

Special cases

Riemann zeta function $\zeta(s) = \zeta(s,1)$, Dirichlet L-series, polygamma functions $\psi^{(m)}(z)$, polylogarithm $\mathrm{Li}_s(z)$, Bernoulli polynomials (not necessarily the best way to compute these functions).

Euler-Maclaurin summation

$$\sum_{k=0}^{U} f(k) = \sum_{k=0}^{N-1} f(k) + \int_{N}^{U} f(t)dt + \frac{1}{2}\left(f(N) + f(U)\right)$$

$$+ \sum_{k=1}^{M} \frac{B_{2k}}{(2k)!}\left(f^{(2k-1)}(U) - f^{(2k-1)}(N)\right)$$

$$- \int_{N}^{U} \frac{\tilde{B}_{2M}(t)}{(2M)!} f^{(2M)}(t)dt$$

- Hurwitz zeta: $f(k) = (a+k)^{-s}, U = \infty$
- Must have $\Re(a+N) > 0, \Re(s+2M-1) > 0$
- Rigorous error bounds from the tail integral

Derivatives

In the Euler-Maclaurin summation formula, put a formal power series $s + x \in \mathbb{C}[[x]]$ in place of $s \in \mathbb{C}$.

This is conceptually much simpler than deriving explicit recursion formulas or nested sums for the derivatives (some work is still needed to bound errors).

We also see where fast polynomial arithmetic can be exploited.

Computing Stieltjes (generalized Euler) constants

$$\zeta(s,a) = \frac{1}{s-1} + \sum_{n=0}^{\infty} \frac{(-1)^n}{n!} \gamma_n(a)\ (s-1)^n$$

$\gamma_n = \gamma_n(1), \gamma_0 = \gamma \approx 0.577216$

Computing $(\gamma_0, \ldots \gamma_{1000})$, 1000 significant digits: 14 seconds
Computing $(\gamma_0, \ldots \gamma_{5000})$, 5000 significant digits: 40 minutes

$\approx$ 1000 times faster than Mathematica's `StieltjesGamma[]`, mpmath's `stieltjes()`

Previous work by R. Kreminski: $\gamma_0, \ldots \gamma_{10000}$, isolated larger values (numerical integration)

Fast evaluation of holonomic sequences

Suppose $(c_k)_{k=0}^{\infty}$ is a holonomic sequence annihilated by an operator $L \in R[k][S_k]$. How fast can we compute c_n as n grows?

Naively: $O(n)$ arithmetic operations.

Better methods: based on the matrix form

$$\begin{bmatrix} c_{i+1} \\ c_{i+2} \\ \cdots \\ c_{i+r} \end{bmatrix} = M(i) \begin{bmatrix} c_i \\ c_{i+1} \\ \cdots \\ c_{i+r-1} \end{bmatrix}$$

where $M \in R[k]^{r \times r}$ is the companion matrix of L (possibly after factoring out a denominator polynomial).

C-finite case: matrix exponentiation

If M is constant, we have

$$\prod_{k=1}^{n} M(k) = M^n$$

This can be evaluated using $O(\log n)$ arithmetic operations, which is quasi-optimal.

Example: Fibonacci numbers.

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

Small coefficients: binary splitting

If $R = \mathbb{Z}$ (for example), the entries in $M(k)$ have size $O(\log k)$ bits.

Evaluating $\prod_k M(k)$ using divide-and-conquer gives a quasi-optimal time complexity of $O(n^{1+\varepsilon})$.

Example: $n! = \prod_{k=1}^{n} k$

$8! = ((1 \times 2) \times (3 \times 4)) \times ((5 \times 6) \times (7 \times 8))$

Binary splitting support

Common case: rational hypergeometric series

$$\sum_{k=0}^{\infty} T(k), \quad T(k) = \frac{P(k)}{Q(k)} T(k-1)$$

where $P, Q \in \mathbb{Z}[k]$.

Given P, Q and a bound 2^{-b} for the truncation error, Arb can evaluate $\sum_{k=0}^{\infty} T(k)$ with automatic error bounding. This is used for computing constants: $\pi, e, \log 2, \log 10, \zeta(3), K, \ldots$

Also some code for binary splitting evaluation of holonomic sequences (no truncation bounds).

Binary splitting with rounding

To get a precision of b bits, we often need $O(b)$ terms, which means that the final result of binary splitting with exact arithmetic has $O(b \log b)$ bits.

We can improve performance by using exact arithmetic until the numbers grow to b bits, and then keep rounding them to b bits.

In Arb, this can be accomplished by simply using Arb numbers throughout.

Binary splitting for power series

Example: fast simultaneous computation of $\zeta(3), \zeta(5), \zeta(7), \ldots$

$$\sum_{i=1}^{\infty} \zeta(2i+1)x^i = \sum_{k=1}^{\infty} \frac{1}{k^3(1-x/k^2)}$$

$$= \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k^3\binom{2k}{k}} \left(\frac{1}{2} + \frac{2}{1-x/k^2}\right) \prod_{j=1}^{k-1}(1-x/j^2).$$

Binary splitting over $\mathbb{Z}[x]$ balances both the polynomial degrees and the bit sizes (in the end we perform one power series division)

We can both truncate the polynomials and round the coefficients. Rounding improves speed by $\approx$ 2x in practice.

General case: fast multipoint evaluation

Suppose the coefficients of $L \in R[k][S_k]$ do not have small bit sizes (e.g. they are real numbers with the same precision as the final result). Assume $n = m^2$.

1. Use binary splitting to generate the polynomial matrix

$$P_m = M(k+m-1) \cdots M(k+1)M(k) \in R[k]^{r \times r}$$

2. Evaluate $P_m(k)$ for $k = 0, m, 2m, 3m, \ldots, (m-1)m$
3. Compute $P_m((m-1)m) \cdots P_m(2m)P_m(m)P_m(0)$.

Step 2 can be done using fast multipoint evaluation. The cost is $O(n^{1/2} \log^2 n)$ arithmetic operations (we must also store $O(n^{1/2} \log n)$ coefficients).

Fast multipoint evaluation in modular arithmetic

Evaluating $(n-1)! \bmod n$ using fast multipoint evaluation in FLINT.

n	Naive	Fast
10^5	0.89 ms	0.52 ms
10^6	9.8 ms	3.1 ms
10^7	110 ms	18 ms
10^8	1.2 s	0.12 s
10^9	12 s	0.71 s
10^{10}	151 s	3.5 s
10^{11}	1709 s	15 s
10^{12}	5 h (est.)	70 s
10^{13}	50 h (est.)	307 s
10^{14}	500 h (est.)	1282 s

Numerical fast multipoint evaluation

We consider evaluating the rising factorial

$$x^{\overline{n}} = x(x+1)(x+2)\cdots(x+n-1)$$

where x is a b-bit real or complex number and the final precision also is b bits.

Naive algorithm

$O(nM(b))$

Fast multipoint evaluation

$O(n^{1/2}\log^2 n\, M(b))$

Numerical disaster

Relative error when evaluating $x^{\overline{n}}$ at precision $b = 10000$:

n	Naive	Fast
10^1	$2^{-10000+3}$	$2^{-10000+6}$
10^2	$2^{-10000+5}$	$2^{-10000+73}$
10^3	$2^{-10000+5}$	$2^{-10000+738}$
10^4	$2^{-10000+6}$	$2^{-10000+3870}$
10^5	$2^{-10000+5}$	$2^{-10000+1238528}$

Empirically, we lose $\approx O(n)$ digits of accuracy when evaluating $x^{\overline{n}}$. This happens even if the slow $O(n^2)$ polynomial multiplication algorithm is used.

In the numerical case, fast multipoint evaluation only seems useful when b grows at least as fast as n.

Also noted in earlier work

S. Köhler and M. Ziegler, "On the Stability of Fast Polynomial Arithmetic" (2008):

> *We thus have shown that, in spite of its name, fast polynomial arithmetic is as slow as the naive algorithms: for practically relevant instances over real numbers, due to the increased intermediate precision required for reasonable output accuracy. Surprisingly (at least to us), these instabilities are not so much due to polynomial division with remainder alone; it is the multiplication of several real polynomials which leads to ill-conditioned input to the polynomial division.*
>
> *The big open challenge thus consists in devising some variant of fast polynomial arithmetic which is numerically sufficiently mild to yield a net benefit in running time over the naive $O(n^2)$ approach.*

Eight-point algorithm for rising factorials

Process eight factors at a time using the following formula (Crandall and Pomerance, 2005)

$$x(x+1)\cdots(x+7) = (28+98x+63x^2+14x^3+x^4)^2 - 16(7+2x)^2$$

Instead of 7 full-precision multiplications, we only need 3 squarings, 1 multiplication, and several "scalar" operations.

At high precision, this gives a constant factor speedup ($\approx$ 2x)

Rising factorials using rectangular splitting

We can process m factors at a time: use binary splitting over $\mathbb{Z}[t]$ to expand (the unsigned Stirling numbers of first kind)

$$P(t) = t(t+1)(t+2)\cdots(t+m-1) = \sum_{k=0}^{m} \begin{bmatrix} m \\ k \end{bmatrix} t^k$$

and use rectangular splitting for each polynomial
$x^{\overline{n}} = P(x)P(x+m)P(x+2m)\cdots$

Nonscalar multiplications: $O((n/m)m^{1/2} + n/m)$
If $m \sim n^{\alpha}$ this goes to $O(n^{1/2})$ as $\alpha \to 1$. However, larger m must be balanced against the larger coefficients.

Smith's algorithm

D. M. Smith (2001)

One group: $A = (x+k)(x+k+1)(x+k+2)(x+k+3)$

Next group $B = (x+k+4)(x+k+5)(x+k+6)(x+k+7)$

Then $B - A =$
$16x^3 + 24(2k+7)x^2 + 8(6k^2+42k+79)x + 8(2k+7)(k^2+7k+15)$

If we precompute x^2 and x^3, $B = A + (B - A)$ can be evaluated using only scalar operations

Constant factor improvement: $n \to n/4$ nonscalar multiplications

Generalizing Smith's method

For a fixed parameter m, write

$$\Delta_m(x,k) = (x+k+m)^{\overline{m}} - (x+k)^{\overline{m}} \in \mathbb{Z}[k][x]$$

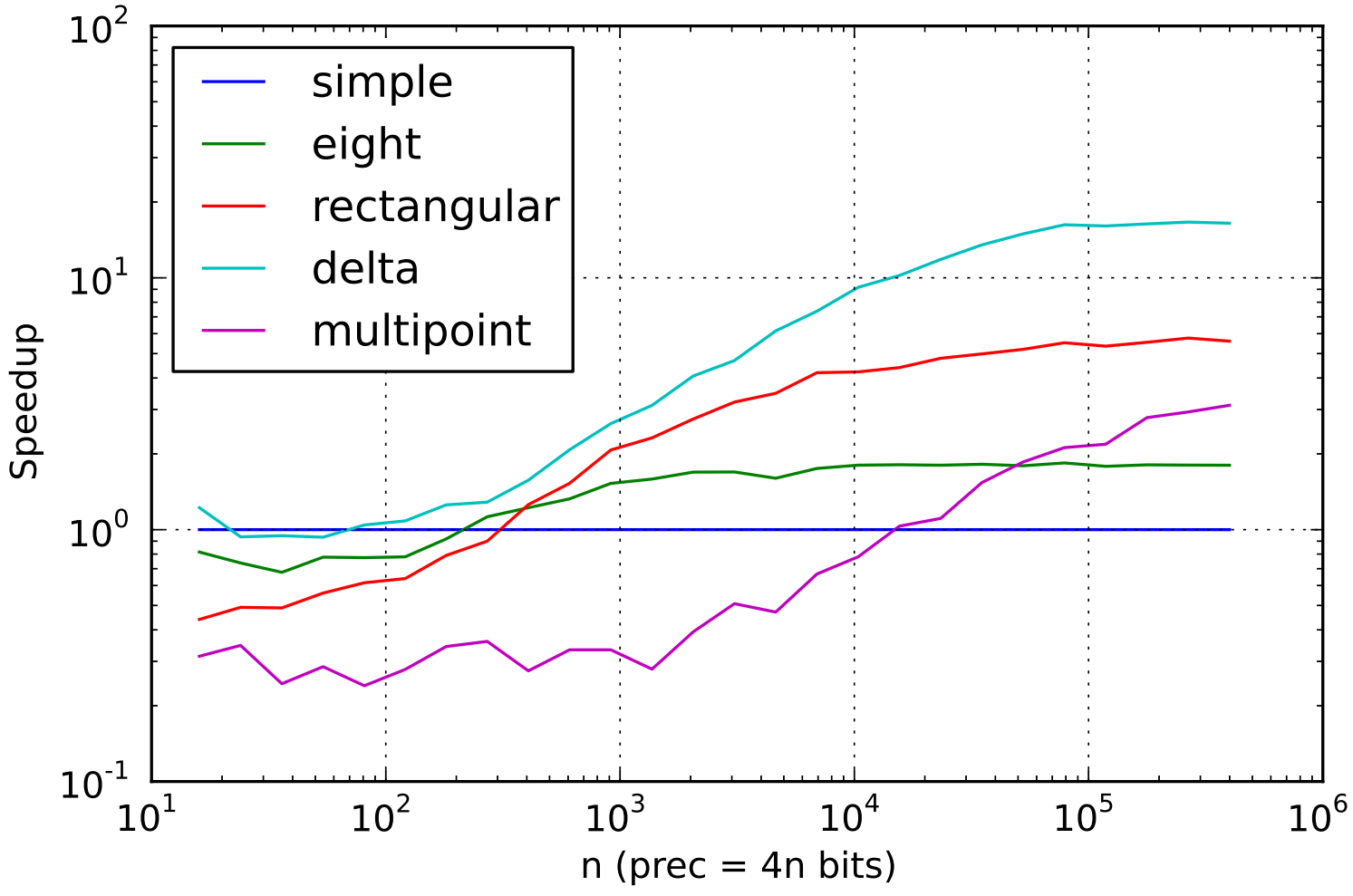
We choose m according to the input and generate this polynomial dynamically. Taking $m \sim n^{1/2}$ allows a reduction to $O(n^{1/2})$ nonscalar multiplications. (There is also a cost due to the growth of the scalars.)

The coefficients can be written in closed form:

$$\Delta_m(x,k) = \sum_{v=0}^{m-1} x^v \sum_{i=0}^{m-v-1} k^i \, C_m(v,i)$$

$$C_m(v,i) = \sum_{j=i+1}^{m-v} m^{j-i} \begin{bmatrix} m \\ v+j \end{bmatrix} \binom{v+j}{v} \binom{j}{i}$$

Comparison of rising factorial algorithms



Analysis of rising factorial algorithms

- The rectangular splitting algorithms ("delta", "rectangular") are best in practice
- Fast multipoint evaluation can beat the naive algorithm at very high precision, but in practice struggles to compete with the rectangular splitting algorithms

Evaluating the gamma function

The gamma function is computed using the asymptotic Stirling series

$$\log \Gamma(z) = \left(z - \frac{1}{2}\right) \log z - z + \frac{\log 2\pi}{2} + \sum_{k=1}^{n-1} \frac{B_{2k}}{2k(2k-1)z^{2k-1}} + R(n, z)$$

We can make $R(n, z)$ arbitrarily small by setting z large enough, using $\Gamma(x) = \Gamma(x + r)/(x(x + 1)(x + 2) \cdots (x + r - 1))$.

Optimization opportunity

A larger r allows us to use a smaller n, and a fast rising factorial makes a larger r cheap.

Timings, real gamma function

Evaluating $\Gamma(x)$, $x = \sqrt{2}$

Digits	Pari/GP	MPFR	Mathematica	Arb
30	0.000024	0.000090	0.000070	0.000043
100	0.000068	0.00036	0.00020	0.00011
300	0.00039	0.0028	0.00080	0.00032
1000	0.0046	0.046	0.058	0.0021
3000	0.12 (6.5)	1.2	0.76	0.017 (0.080)
10000	1.9 (233)	60	13	0.21 (1.3)
30000	13 (6154)	2680	186	2.4 (19)

Timings in seconds for repeated evaluation (first evaluation)

Timings, complex gamma function

Evaluating $\Gamma(z)$, $z = \sqrt{2} + i\sqrt{3}$

Digits	Pari/GP	mpmath	Mathematica	Arb
30	0.000064	0.00017	0.00021	0.00012
100	0.00018	0.00043	0.00052	0.00027
300	0.0010	0.0023	0.0022	0.00081
1000	0.013	0.026	0.020	0.0055
3000	0.19 (6.5)	0.36 (3.6)	0.31 (1.7)	0.049 (0.11)
10000	2.9 (235)	6.4 (95)	5.8 (44)	0.67 (1.7)
30000	38 (6030)	92 (3030)	80 (887)	8.5 (25)

Timings in seconds for repeated evaluation (first evaluation)

Generalizing to arbitrary holonomic sequences

Let $c_k(x)$ be a holonomic sequence with annihilator $L \in R[x][k][S_k]$ and companion matrix $M(x,k) \in R[x][k]^{r \times r}$.

Let a be an element of some R-algebra A. Then we can evaluate $c_n(a)$ using $O(n^{1/2})$ nonscalar multiplications $A \times A \to A$ and $O(n)$ scalar multiplications $A \times R \to A$.

$$\Delta_m = \prod_{i=0}^{m-1} M(x, k+m+i) - \prod_{i=0}^{m-1} M(x, k+i) \in R[x][k]^{r \times r}$$

Choose $m \sim n^{1/2}$, precompute $a, a^2, \ldots, a^{md}$, $d = \deg_x L$.

Sequences with multiple parameters

Let $c_k(x_1, \ldots, x_h)$, $L \in R[x_1, \ldots, x_h][k][S_k]$, $d_i = \deg_{x_i} L \leq d$. Then the entries of Δ_m are $R[k]$-linear combinations of $x_1^{e_{1,j}} \cdots x_h^{e_{h,j}}$, $0 \leq e_{i,j} \leq md_i \leq md$.

h	m	Nonscalar: $O(m^h + n/m)$	Scalar: $O((n/m)m^h)$
2	$n^{1/3}$	$n^{0.666}$	$n^{1.333}$
3	$n^{1/4}$	$n^{0.75}$	$n^{1.5}$
4	$n^{1/5}$	$n^{0.8}$	$n^{1.6}$

For $h > 1$, we can still reduce the number of nonscalar multiplications, but have to do asymptotically more scalar multiplications than by naive evaluation.

Taking m constant might still give a constant factor speedup.

Possible application: hypergeometric functions

We can numerically evaluate ${}_pF_q(a_1 \ldots a_p, b_1 \ldots b_q, z)$ where $a_i, b_i, z \in \mathbb{Q}(s)$ and s is transcendental, with a reduced number of full-precision multiplications (Paterson-Stockmeyer, Smith: z transcendental).

P. Borwein (1987): noticed that fast multipoint evaluation is applicable (costing $O(b^{1/2} \log^2 b)$ arithmetic operations for $O(2^{-b})$ error).

Unlike Borwein's method, we can expect to see speedups in practice, even at relatively modest precision (e.g. 10000 digits), and we avoid the numerical instability.

Gamma without Bernoulli numbers

$$\Gamma(s) = N^s \sum_{k=0}^{\infty} \frac{(-1)^k N^k}{k!(s+k)} + O(Ne^{-N})$$

The partial sums are holonomic of order $r = 2$.

Digits	Naive	Δ_{32}	Stirling (first)	Stirling (cached)
30000	345 s	52 s	33 s	5.5 s
100000	7451 s	425 s	602 s	72 s

Note: preliminary Sage implementation, 32-bit computer.

Stirling's series is still the winner...

...at least with Bernoulli numbers cached. The hypergeometric series shows some promise at least as an alternative in some situations.

The end