The practical complexity of arbitrary-precision functions

Fredrik Johansson *

* Inria Bordeaux

Fundamental Algorithms and Algorithmic Complexity
RTCA 2023

Institut Henri Poincaré
September 28, 2023

Introduction

Question

How quickly can we compute functions like $\exp(x)$ to n digits?

(Bit) complexity bounds quasilinear in n are classical.[1] But what happens in practice?

Much more generally, how should we analyze algorithms which use n-digit numbers? Estimates like

$$k \quad \text{arithmetic operations} \quad \to \quad k\, n^{1+o(1)} \text{ bit operations}$$

hide a lot of details!

[1] e.g. Brent, 1970s. With certain refinements in recent years.

An important unit of measurement

$M(n)$ = the time to multiply two n-digit integers.

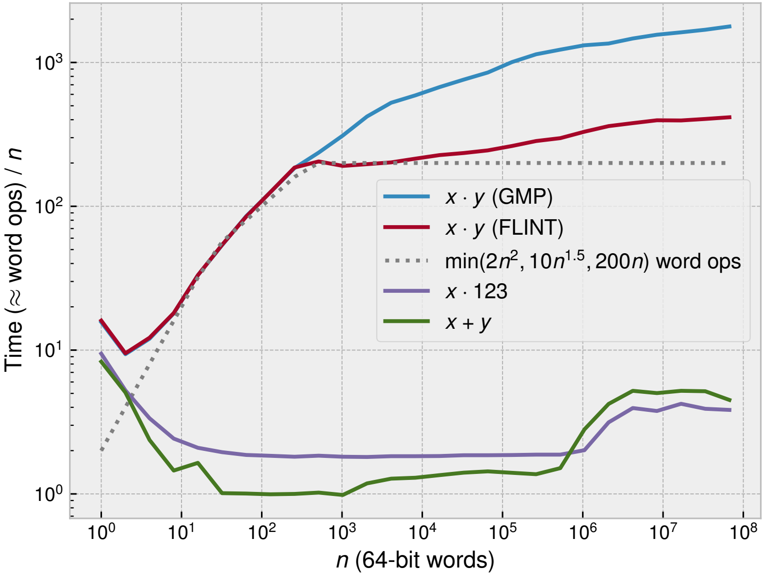
Notable algorithms:

- Basecase: $M(n) = O(n^2)$
- Toom-Cook: $M(n) = O(n^c)$ for $1 < c < 2$
- FFT: $M(n) = O(n \log n \ldots)$
  - Schönhage-Strassen (used in GMP)
  - Complex floating-point FFT
  - Number-theoretic transform (NTT) ($\mathbb{Z}/p\mathbb{Z}$, word-size p)
- Harvey - van der Hoeven: $M(n) = O(n \log n)$, not yet used

Remarks:

- In typical implementations, "digit" = "64-bit word".
- Some bounds stated in this talk make assumptions about $M(n)$.

Timings[2] for n-word integer arithmetic



[2] AMD Ryzen 7 PRO 5850U, GMP 6.2, FLINT 3.0.

Things we might be able to do in the span of $M(n)$

- 3 FFTs + pointwise multiplications
- 1.5 to 2 squarings
- 1 to 2 short products (top or bottom half of the full $2n$ words)
- 2 to 4 half-length multiplications with cost $M(n/2)$
- $\min(n^2, 100n)$ single-word operations
- $\min(n, 100)$ scalar operations ($x + y \cdot c$ with single-word c)
- Table lookups
- $\ldots$

Mantra

Reduce everything to multiplication.
But if possible, reduce further!

Arithmetic operations and Newton iteration

Newton iteration allows approximating x/y or $\sqrt{x}$ to n digits in time

$$O(M(n) + M(n/2) + M(n/4) + \ldots) = O(M(n)).$$

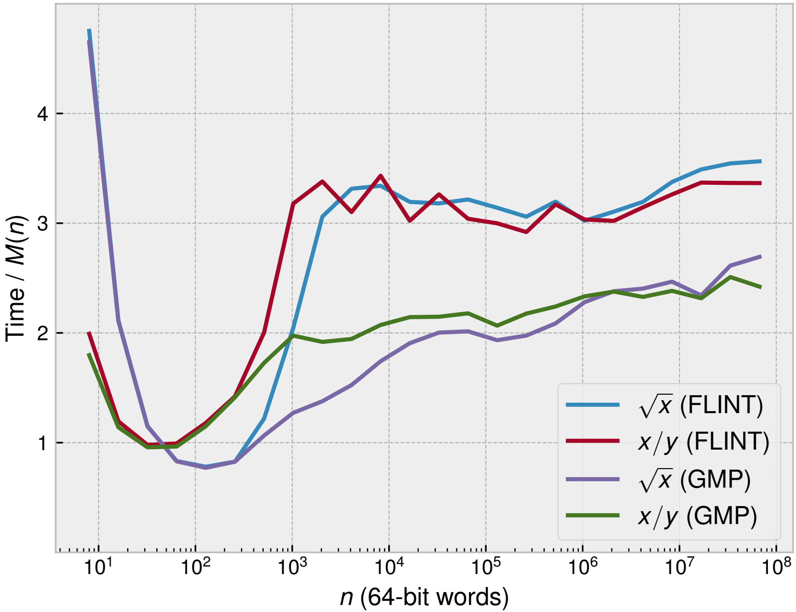
Some theoretical complexity bounds in the FFT model are:[3]

- Reciprocal : $\sim 1.444\ldots M(n)$
- Division: $\sim 1.666\ldots M(n)$
- Square root: $\sim 1.333\ldots M(n)$

These bounds rely on FFT tricks which so far are not widely used. In practice one may see $\approx M(n)$ near the basecase range and $\approx 2M(n)$ to $3M(n)$ in the FFT range.

[3] Table 4.1 in Brent and Zimmermann, *Modern Computer Arithmetic*.

Timings for division and square root



Binary splitting

Important tool used to compute, for example:

- Products of small integers like $N!$
- Hypergeometric series like $\sum_{k=0}^{N} x^k/k!$

Example

The cost to compute the NB-digit product of B-digit integers $c_1, c_2, \ldots, c_N$ is bounded by

$$M(NB/2) + 2M(NB/4) + 4M(NB/8) + \ldots \approx \frac{1}{2}M(NB)\log_2 N.$$

In practice, binary splitting often beats such estimates. Why?

- The nonlinearity of $M(n)$ (in reality, $2^k M(n/2^k) < M(n)$)
- Possibility of truncation when we want $n < NB$ digits
- Additional structure that can be exploited

Pi and the AGM

Most world records for π in the last 30 years (currently 10^{14} decimal digits) have used the Chudnovsky series

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k \,(6k)!\,(13591409 + 545140134k)}{(3k)!(k!)^3 \cdot 640320^{3k+3/2}}$$

which costs $O(M(n)\log^2 n)$ using binary splitting.

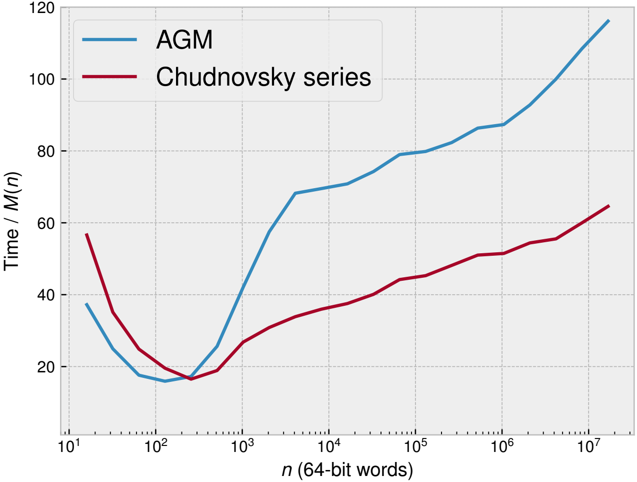
Why not use the arithmetic-geometric mean (AGM) method

$$\pi = \lim_{k\to\infty} \frac{(a_k + b_k)^2}{1 - \sum_{j=0}^{k} 2^j (a_j - b_j)^2},$$

$$a_0 = 1, \quad b_k = \frac{1}{\sqrt{2}}, \quad a_{k+1} = \frac{a_k + b_k}{2}, \quad b_{k+1} = \sqrt{a_k b_k}.$$

which achieves $O(M(n)\log n)$?

Time to compute π



Remark: Paul Zimmermann made a similar comparison in a 2006 talk. He observed a $5\times$ difference between the algorithms.

Elementary functions

The elementary functions have mostly analogous direct methods:

	exp	sin, cos	log	atan
Taylor series, $x \in \overline{\mathbb{Q}}$	$O(M(n)\log n)$		$O(M(n)\log^2 n)$	
Taylor series, $x^N = \varepsilon$	$O(\sqrt{N}M(n) + N^{1+o(1)}n)$			
Bit-burst	$O(M(n)\log^2 n)$		$O(M(n)\log^3 n)$	
AGM			$O(M(n)\log n)$	

Constant-factor conversions:

- $C = 1 + o(1)$ for $f \to f^{-1}$ via Newton iteration
- $C \approx 2 - 4$ for $\mathbb{R} \to \mathbb{C}$

$$\begin{tikzcd}
\exp(x) \arrow[r, leftrightarrow, "\text{Newton}"] \arrow[d, leftrightarrow, "\mathbb{C}"'] & \log(x) \arrow[d, leftrightarrow, "\mathbb{C}"'] \\
\sin(x), \cos(x) \arrow[r, leftrightarrow, "\text{Newton}"] & \operatorname{atan}(x)
\end{tikzcd}$$

The AGM for elementary functions

The logarithm can be computed as

$$\log(x) \approx \frac{\pi}{2\,\mathrm{agm}(1, 4/s)} - m\log(2), \quad s = x \cdot 2^m > 2^{\mathrm{bits}/2},$$

$$\mathrm{agm}(x_0, y_0) = \lim_{n\to\infty} x_n, \quad x_{n+1} = (x_n + y_n)/2,\; y_{n+1} = \sqrt{x_n y_n}$$

where π and $\log(2)$ are precomputed.

- The number of AGM iterations is $\sim 2\log_2(n)$.
- We can save $O(1)$ iterations using series expansions.
- In the FFT model, an upper bound for the complexity is $\sim 4\log_2(n)M(n)$ with real arithmetic (computing exp, log).
- We need complex arithmetic for trigonometric functions.

The bit-burst algorithm

Write $\exp(x) = \exp(x_1) \cdot \exp(x_2) \cdots$ where x_i extracts 2^i bits in the binary expansion of x. Use binary splitting to evaluate

$$\exp(x_i) \approx \sum_{k=0}^{N_i} \frac{x_i^k}{k!}.$$

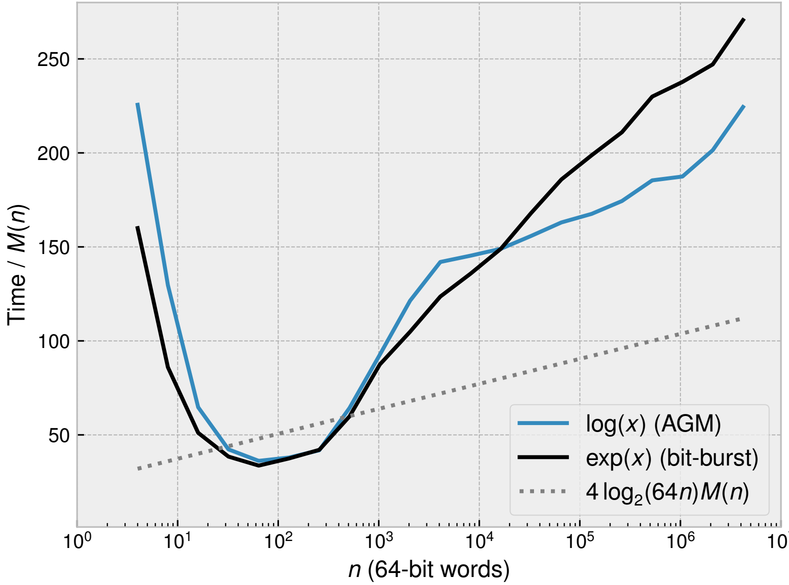
Important optimization 1: do an initial argument reduction

$$\exp(x) \to \exp(x/2^r)^{2^r}, \quad r = o(\log^2(n)).$$

This trades the first and most expensive Taylor series for cheaper squarings. In practice, $r \approx 10 - 100$ varying with n.

Important optimization 2: for smallish n, just use one Taylor series, with rectangular splitting ($O(\sqrt{N}M(n) + N^{1+o(1)}n)$ for N terms). One can make $N \to N/2$ using $\exp(t) = s + \sqrt{s^2+1}$, $s = \sinh(t)$.

AGM vs bit-burst vs theory



Argument reduction using precomputation

There are faster ways to reduce r by a factor 2^r if we allow precomputations. Different tradeoffs are possible. For simplicity, we limit the reduction time to $O(M(n))$. Two example designs:

Method A: $O(n2^r)$ table size, any r

Precompute $\{\exp(i/2^r)\}_{i=0}^{r-1}$. Use $\exp(x) = \exp(i/2^r)\exp(x - i/2^r)$.

Method B:[4] $O(nr)$ table size, $r = O(\log n)$

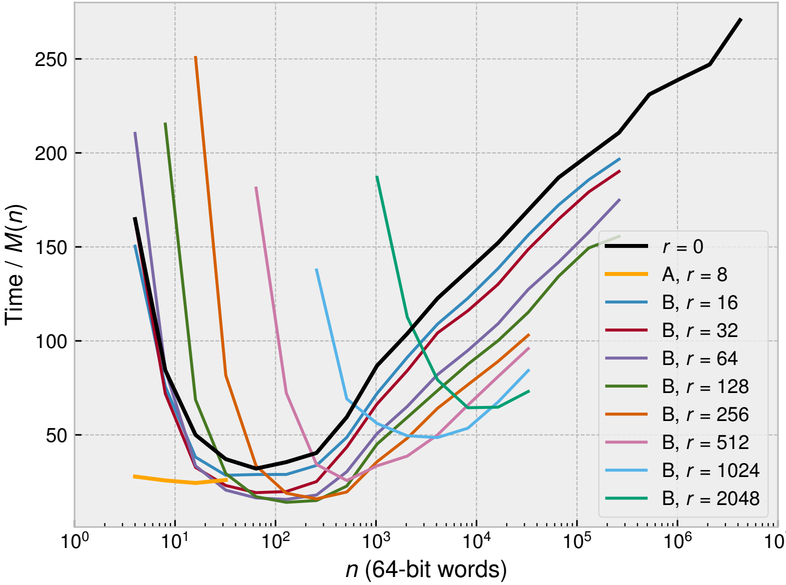
Pick rationals $|q_i - \exp(2^{-i})| < 2^{-r}/r$. Precompute $\{\log(q_i)\}_{i=1}^{r}$.

$$\exp(x) = \underbrace{\left(q_1^{b_1} \cdots q_r^{b_r}\right)}_{\text{Binary splitting}} \exp(x - \underbrace{(b_1 \log(q_1) + \cdots + b_r \log(q_r))}_{\text{Scalar operations}})$$

Reduction costs $O(M(r^2)\log r + nr)$, so we can have $r = O(\log n)$. In practice, the optimal r initially grows more like $\sqrt{n}$.

[4] Thanks to Joris van der Hoeven for suggesting this version.

Time to compute exp(x), with table reduction to 2^{-r}



Avoiding large tables: Schönhage's method

Method C: $O(n)$ table size, $r = O(\log n)$

Precompute $\log(2)$ and $\log(3)$. Given x, compute r-bit integers c, d such that $2^c 3^d \approx \exp(x)$ within 2^{-r}.

$$\exp(x) = 2^c 3^d \exp(x - c\log(2) - d\log(3))$$

Example: $x = \log(\pi)$

$$2^{8} \cdot 3^{-4} = 3.16\ldots$$
$$2^{1931643} \cdot 3^{-1218730} = 3.141592601\ldots$$
$$2^{-3824416943916269} \cdot 3^{2412938439979599} = 3.14159265358979360\ldots$$

- If $r \leq \log_2(n)$, computing 3^d costs $O(M(2^r)) = O(M(n))$.
- If $r > \log_2(n)$, continued powering degenerates to full n-digit powering; we don't save anything over simply doing $x \to x/2^r$.
- For trigonometric functions, use two Gaussian primes.

Multi-prime method[5]

Method C with ℓ primes: $O(\ell n)$ table size

Precompute $\log(2), \ldots, \log(p_\ell)$. Given x, compute integers $c_1, \ldots, c_\ell$ such that $2^{c_1} \cdots {p_\ell}^{c_\ell} \approx \exp(x)$ within 2^{-r}.

$$\exp(x) = 2^{c_1} \cdots {p_\ell}^{c_\ell} \, \exp(x - (c_1 \log(2) + \ldots + c_\ell \log(p_\ell)))$$

Example: $\ell = 5$ and $x = \log(\pi)$

$$\begin{aligned}
2^6 \cdot 3^4 \cdot 5^{-10} \cdot 7^2 \cdot 11^2 \quad &= 3.1473\ldots \\
2^{-31} \cdot 3^{-57} \cdot 5^{136} \cdot 7^{41} \cdot 11^{-89} \quad &= 3.141592609\ldots \\
2^{-583} \cdot 3^{3227} \cdot 5^{7718} \cdot 7^{-8681} \cdot 11^{555} \quad &= 3.14159265358979346\ldots
\end{aligned}$$

Heuristically, the exponents now only have around r/ℓ bits and computing the power product costs $O(M(2^{r/\ell} \cdot \ell^{O(1)}))$.

Heuristically, we can choose $r \propto \ell^2$ with $\ell \propto \log(n)$.

[5] J. *Computing elementary functions using multi-prime argument reduction*, 2022

Computing smooth rational approximations

To quickly solve the inhomogeneous integer relation problem

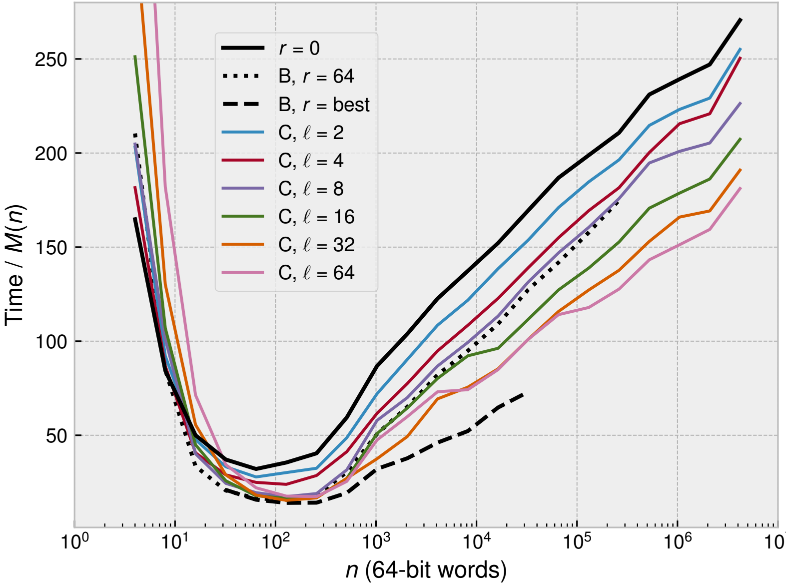
$$x \approx c_1\alpha_1 + \dots c_\ell\alpha_\ell,$$

precompute (using LLL, say) solutions $\varepsilon_1 > \varepsilon_2 > \dots$ to the homogeneous problem $0 \approx d_1\alpha_1 + \dots d_\ell\alpha_\ell$:

$$\begin{pmatrix} 1 & 1 & -1 & 0 & 0 \\ 0 & -1 & -2 & 1 & 1 \\ 1 & 2 & -3 & 1 & 0 \\ -3 & 4 & -2 & -2 & 2 \\ -2 & 2 & 2 & -7 & 4 \\ -18 & -3 & 22 & 1 & -9 \\ 19 & -23 & -22 & 1 & 19 \\ 23 & -12 & 47 & 9 & -40 \end{pmatrix} \begin{pmatrix} \log(2) \\ \log(3) \\ \log(5) \\ \log(7) \\ \log(11) \end{pmatrix} = \begin{pmatrix} 0.182 \\ 0.0263 \\ 0.00797 \\ 0.000102 \\ 1.61 \cdot 10^{-5} \\ 6.51 \cdot 10^{-7} \\ 4.99 \cdot 10^{-8} \\ 2.83 \cdot 10^{-9} \end{pmatrix}$$

We can then build $c_1, \dots, c_n$ by removing $\varepsilon_1, \varepsilon_2, \dots$ in turn from x.

Time to compute $\exp(x)$, different number of primes ℓ



Simultaneous logarithm precomputations

We can compute $\{\log(p_1), \ldots, \log(p_\ell)\}$ simultaneously using ℓ-term Machin-like formulas.[6] Example for $\ell = 2$:

$$\begin{pmatrix} \log(2) \\ \log(3) \end{pmatrix} = \begin{pmatrix} 4 & 2 \\ 6 & 4 \end{pmatrix} \begin{pmatrix} \operatorname{acoth}(7) \\ \operatorname{acoth}(17) \end{pmatrix}$$

where we use binary splitting to compute

$$\operatorname{acoth}(x) = \sum_{k=0}^{\infty} \frac{1}{(2k+1)} \frac{1}{x^{2k+1}}.$$

For each ℓ, all such formulas can be found using a method of Gauss.

The (conjecturally) best ℓ-term formulas up to $\ell = 25$ (and $\ell = 22$ for Gaussian primes) are tabulated in (J. 2022).

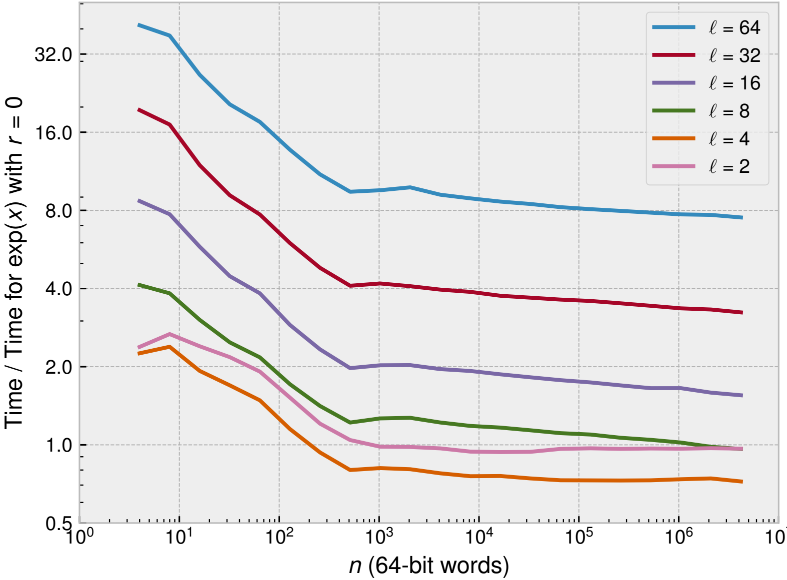
[6] So named after Machin's formula $\pi/4 = 4\operatorname{acot}(5) - \operatorname{acot}(239)$.

Best ℓ-term formulas for the first ℓ primes

ℓ	$p_1, \ldots, p_\ell$	X	$\mu(X)$
1	2	3	2.09590
2	2, 3	7, 17	1.99601
3	2, 3, 5	31, 49, 161	1.71531
4	2, ..., 7	251, 449, 4801, 8749	1.31908
5	2, ..., 11	351, 1079, 4801, 8749, 19601	1.48088
6	2, ..., 13	1574, 4801, 8749, 13311, 21295, 246401	1.49710
7	2, ..., 17	8749, 21295, 24751, 28799, 74359, 388961, 672281	1.49235
8	2, ..., 19	57799, 74359, 87361, 388961, 672281, 1419263, 11819521, 23718421	1.40768
13	2, ..., 41	51744295, 170918749, 265326335, 287080366, 362074049, 587270881, 831409151, 2470954914, 3222617399, 6926399999, 9447152318, 90211378321, 127855050751	1.42585
25	2, ..., 97	373632043520429, 386624124661501, 473599589105798, 478877529936961, 523367485875499, 543267330048757, 666173153712219, 1433006524150291, 1447605165402271, 1744315135589377, 1796745215731101, 1814660314218751, 2236100361188849, 2767427997467797, 2838712971108351, 3729784979457601, 4573663454608289, 9747977591754401, 11305332448031249, 17431549081705001, 21866103101518721, 34903240221563713, 99913980938200001, 332110803172167361, 1918293747470381875l	1.60385

The *Lehmer measure* $\mu(X) = \sum_{x \in X} \frac{1}{\log_{10}(|x|)}$ is an estimate of efficiency of a Machin-like formula (lower is better).

Precomputation time, different number of primes ℓ



Assorted transcendental functions

Functions	Restriction	$O(M(n))$ complexity	Notes
Elementary		$\log n$	
Holonomic (e.g. $\mathrm{erf}(z)$, $J_\nu(z)$)	$\nu \in \mathbb{C}$	$n^{0.5+o(1)}$	7
	$\nu \in \overline{\mathbb{Q}}$	$\log^c n$	
$\Gamma(z)$, $\psi(z)$	$z \in \mathbb{C}$	$n^{0.5+o(1)}$	8
	$z \in \overline{\mathbb{Q}}$	$\log^c n$	
$\zeta(s)$, $L(s,\chi)$	$s \in \mathbb{C}$	$n^{1+o(1)}$	9
	$s \in \overline{\mathbb{Q}}$	$n^{0.5+o(1)}$	
	$s \in \mathbb{Z}$	$\log^c n$	
$\theta(z \mid \tau)$		$\log n$	10

Fun fact: all results rely on holonomic functions or the AGM.

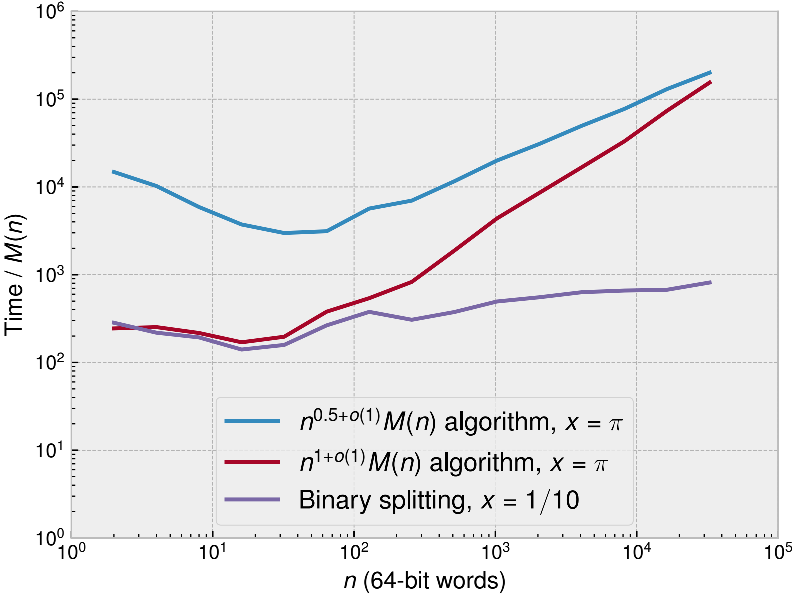
[7] Chudnovsky^2; van der Hoeven; Mezzarobba

[8] See survey: J., *Arbitrary-precision computation of the gamma function*, 2023

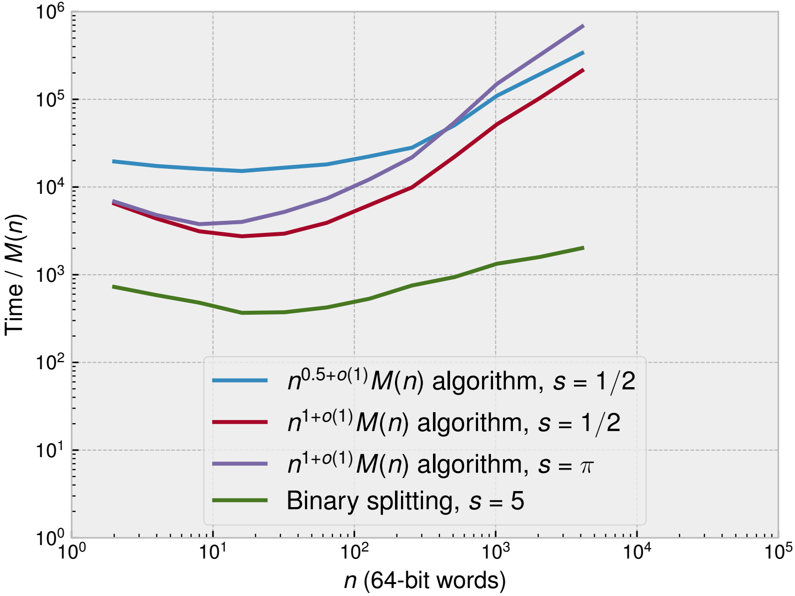
[9] J., *Rapid computation of special values of Dirichlet L-functions*, 2022

[10] Dupont; Labrande & Thomé; Kieffer; Kieffer & Elkies (unpublished)

Time to compute $\Gamma(x)$



Time to compute $\zeta(s)$



Special points

Principle: transcendental functions can often be evaluated faster at "special" points than at "generic" points.

- Points in $\mathbb{Z}$, $\mathbb{Z}[\frac{1}{2}]$, $\mathbb{Q}$, or even $\overline{\mathbb{Q}}$ (with bit size $\ll$ precision)
  - We may have special formulas, e.g. $\zeta(2) = \pi^2/6$
  - Binary splitting, scalar arithmetic, ...
- Points close to singularities and special points
  - Series expansions converge faster

Question

We have already seen how special points are useful in argument reduction for elementary functions. In what other situtations can we exploit special points?

Example: polynomial interpolation

Let's compute 1000 digits of

$$\int_0^1 \Gamma(1+x)dx \approx \sum_{k=1}^{N} w_k \Gamma(1+x_k)$$

using polynomial (Lagrange) interpolation. How should we choose the N sample points to minimize the Γ-function evaluation time?

Newton-Cotes
$x_k = k/N$
$N = 1667$
2500 digits working precision
Time: 3.02 s

Gauss-Legendre
$x_k =$ root of P_N
$N = 654$
Time: 0.14 s

Perturbed Gauss
x_k = root of P_N, rounded to 53 bits
$N = 1294$
Time: 0.075 s

Thank you!