Fast multiple precision $\exp(x)$ with precomputations

Joris van der Hoeven
CNRS, LIX (UMR 7161)
Palaiseau, France
vdhoeven@lix.polytechnique.fr

Fredrik Johansson
Inria, IMB (UMR 5251)
Bordeaux, France
fredrik.johansson@gmail.com

***Abstract*—What is the most efficient way to compute the exponential function when allowing for the precomputation of lookup tables? In this paper we study this question as a function of the working precision and analyze both classical and asymptotically fast approaches. We present new complexity results, discuss efficient parameter choices and point out improvements that lead to speedups over existing implementations.**

***Index Terms*—Elementary functions, Multiple-precision arithmetic, Table-based methods, FFT**

I. Introduction

We are interested in efficiently computing the exponential function $\exp(x)$ to arbitrary n-bit precision where n may be much larger than the machine word size, for example $n \approx 10^4$. For general background and standard techniques that will be referenced, see [Mul16], [BZ11].

We denote by $\mathsf{E}(n)$ the cost of evaluating $\exp(x)$ given an input on the standard interval $0 \leqslant x < 1$. We further denote by $\mathsf{E}(n, r)$ the cost of evaluating $\exp(\varepsilon)$ on the reduced interval $0 \leqslant \varepsilon < 2^{-r}$, where a Taylor series converges more rapidly.

The classical argument reduction formula $\exp(x) = \exp(x/2^r)^{2^r}$ maps the standard interval to the reduced interval via r halvings and squarings. A similar r-bit reduction might be achieved more cheaply after precomputations, say in time $\mathsf{R}_T(n, r)$ given some lookup table T. Accordingly, we want to choose a table design and reduction parameter r minimizing

$$\mathsf{E}_T(n) := \mathsf{R}_T(n, r) + \mathsf{E}(n, r).$$

In practice, this minimization problem will be constrained by the precomputation time or the available space for tables. If we aim to perform a large number N of evaluations and generating the table costs $\mathsf{T}_T(n)$, then we want to minimize the amortized cost $\mathsf{E}_T(n) + \mathsf{T}_T(n)/N$.

In this work, we will study several table-based methods and the associated cost functions $\mathsf{R}_T(n, r)$ and $\mathsf{T}_T(n)$. We provide new complexity analyses, point out practical and theoretical improvements over previous designs, and make empirical observations about realistic parameter values and attainable speedups E/E_T.

II. Multiple precision arithmetic

Consider a machine with β-bit words. Typically, $\beta = 64$. We assume that n-bit positive integers are represented using $\lceil n/\beta \rceil$ words and that n-bit real numbers in $[0, 2^{-k})$ are represented in fixed-point format using $\lceil (n-k)/\beta \rceil$ words. We make frequent use of the following primitives:

$\mathsf{M}(n)$	Cost of multiplying two n-bit integers
$\mathsf{A}(n)$	Cost of adding two n-bit integers
$\mathsf{P}(n)$	Cost of an n-bit product $x_1 \cdots x_\ell \in \mathbb{Z}$

It is useful to distinguish between the "classical regime" with $\mathsf{M}(n) = O(n^2)$ and the "FFT regime" for huge n where we can assume that $\mathsf{M}(n) = O(n \log n)$ [HH21]. In between, one may also consider the "Karatsuba regime" with $\mathsf{M}(n) = O(n^{\log_2 3})$ or various "Toom-Cook regimes" with $\mathsf{M}(n) = O(n^\gamma)$ and $1 < \gamma \leqslant \log_2 3$. We will always assume that $\mathsf{M}(n)/(n \log n)$ is a non-decreasing function. We denote by n_{FFT} the threshold where we enter the FFT regime.

It is well known that various basic complexities can be expressed in terms of $\mathsf{M}(n)$. For instance, quotients and square roots can be computed in time $O(\mathsf{M}(n))$.

We clearly have $\mathsf{A}(n) \propto n$, but it is the ratio $\mathsf{M}(n)/\mathsf{A}(n)$ that is often interesting to keep in mind. For $n \leqslant \beta$, this ratio is close to one on modern computers. For $n \leqslant k\beta$ and small k, the ratio scales linearly with k. In the FFT regime, the ratio typically exceeds 100. Other linear time operations, such as multiplying or dividing x by a single-word integer or power of two, can also be done in time $c\mathsf{A}(n)$, for some constant close to one (e.g. $1/4 < c < 4$) that depends on the operation.

Another important operation is the computation of *smooth products*. Given $x_1, \ldots, x_\ell \in \mathbb{Z} \setminus \{-1, 0, 1\}$ such that the product $\pi := x_1 \cdots x_\ell$ has bit-size at most n, we denote by $\mathsf{P}(n)$ the cost to compute π. The hardest case is when $x_1, \ldots, x_\ell$ are all small, whence the name of the operation. It is classical that π can be computed in time

$$\mathsf{P}(n) \leqslant \mathsf{M}(n/2) + 2\mathsf{M}(n/4) + \cdots = O(\mathsf{M}(n) \log n)$$

using binary splitting. But here again, the precise ratio $\mathsf{P}(n)/\mathsf{M}(n)$ depends on the regime. In the naive regime, one typically has $\mathsf{P}(n) \lesssim {}^1\!/_2\, \mathsf{M}(n) + O(n)$. In the FFT regime, one has $\mathsf{P}(n)/\mathsf{M}(n) \sim {}^1\!/_2 \log(n/n_{\mathrm{FFT}})$. If $\mathsf{M}(n) \propto n^\gamma$, then $\mathsf{P}(n)/\mathsf{M}(n) \sim 1/(2^\gamma - 2)$.

We define $\mathsf{P}(m, n) := \mathsf{P}(m) + \frac{n}{m}\mathsf{M}(m)$: the cost of a smooth product of bitsize m and multiplying the result with a number of n bits. We shall assume that an n-bit hypergeometric sum $s := t_1 + \cdots + t_\ell \in \mathbb{Q}$, $t_{k+1}/t_k \in \mathbb{Q}(k)$, can be computed in time $O(\mathsf{P}(n))$ via a similar binary splitting process.

A. *The standard table-free algorithm for* exp

We recall the classical Taylor series-based algorithms by Brent and Smith [Bre76a], [Smi89], used in popular software like MPFR [FHL+07] and FLINT/Arb [Joh17].

If n/r is relatively small, then the best approach to compute $\exp(\varepsilon)$ uses rectangular splitting to evaluate the series

This work has been supported in parts by the ANR projects NODE (ANR-22-CE48-0016) and NuSCAP (ANR-20-CE48-0014).

$\sum_{k=0}^{N-1} \varepsilon^k/k!$ where $N \approx n/r$. Together with r squarings for argument reduction, this yields the overall complexity

$$\mathsf{E}(n) = c_1 r \mathsf{M}(n) + \underbrace{c_2 \sqrt{n/r}\mathsf{M}(n) + c_3 (n/r)\mathsf{A}(n)}_{\mathsf{E}(n,r)}$$

where c_1, c_2, c_3 are constants which depend on the multiplicative regime. In the classical and Karatsuba regimes, one should choose $r \propto n^{1/3}$ which gives $\mathsf{E}(n) = O(n^{1/3}\mathsf{M}(n))$. In the FFT regime, taking $r \propto n^{1/2}$ yields $\mathsf{E}(n) = O(n^{1/2}\mathsf{M}(n))$.

A variation of the same algorithm which saves a constant factor is to evaluate the hyperbolic sine series

$$\exp(\varepsilon) = s + \sqrt{1+s^2}, \quad s = \sum_{k=0}^{N-1} \frac{\varepsilon^{2k+1}}{(2k+1)!}, \; N \approx \frac{n}{2r}. \quad (1)$$

An asymptotically faster algorithm is the *bit-burst method* in which we write $\exp(\varepsilon) = \exp(\varepsilon_1)\exp(\varepsilon - \varepsilon_1)$ where $\varepsilon_1 = \lfloor \varepsilon 2^{2r} \rfloor / 2^{2r}$ and evaluate the Taylor series for $\exp(\varepsilon_1)$ using binary splitting. One observes that this series is hypergeometric with $O(n)$ bits. This "bit-burst step" can be viewed as an extra argument reduction $r \to 2r$, giving

$$\mathsf{E}(n,r) = O(\mathsf{P}(n)) + \mathsf{E}(n, 2r).$$

Iterated until $r \geqslant n$, this results in the complexity $\mathsf{E}(n) = O(\mathsf{P}(n)\log(n))$, or $O(\mathsf{M}(n)\log^2(n))$ in the FFT regime.

For the bit-burst method, doing r initial squarings does not reduce the asymptotic complexity in the FFT regime, but it does help in practice since the first few binary splitting sums are more expensive.

We mention here a hybrid method which seems to be new: we combine k bit-burst steps with rectangular splitting, where the sinh series now only requires $N \approx n/(2^{k+1}r)$ terms.

In summary, we will use one of the following series evaluation strategies to compute $\exp(\varepsilon)$ on $0 \leqslant \varepsilon < 2^{-r}$, where the optimal choice depends on n and r:

1) EXP: exp series with rectangular splitting (n/r terms)
2) SH: sinh series with rectangular splitting ($n/(2r)$ terms)
3) BSH: performing some initial bit-burst steps to improve the rate of convergence of method SH (e.g. one step giving $n/(4r)$ terms, or two steps giving $n/(8r)$ terms)
4) BB: full bit-burst algorithm ($O(\log n)$ iterations)

A completely different algorithm by Brent [Bre76b] and Salamin [Sal76] involves computing $\exp(x)$ via the arithmetic-geometric mean, achieving $\mathsf{E}(n) = O(\mathsf{M}(n)\log n)$. The crossover where the Brent-Salamin algorithm beats the bit-burst method is quite large, however, e.g. $n > 10^7$ (see Table I). Faster argument reduction will increase this crossover further since Brent-Salamin does not become faster as $x \to 0$.

B. Empirical comparison of basic operations

For our implementation experiments, we use low-level fixed-point and integer routines from GMP 6.3 and FLINT 3.1. Timings were obtained on an AMD Ryzen 7 PRO 5850U (Zen3 architecture). In this work, we test only single-threaded performance. We mention that FLINT uses the cutoff $n_{\text{FFT}} = 25600$ bits for FFT multiplication.

Table I shows relative timings for $\mathsf{A}(n)$ (calling GMP's `mpn_add_n` with $n/64$ words of input), $\mathsf{M}(n)$ (calling FLINT's `flint_mpn_mul_n`), $\mathsf{P}(n)$ (forming the n-bit product of $n/64$ words by calling `flint_mpn_mul_n` in a tree), and $\mathsf{E}(n)$ where the exponential function is implemented using an empirically determined near-optimal number of squarings r and series strategy EXP, SH, BSH or BB (see Table III for further details). The series evaluation is implemented using slightly modified code from FLINT.

TABLE I
MEASURED RELATIVE TIME FOR n-BIT ADDITION $\mathsf{A}(n)$, MULTIPLICATION $\mathsf{M}(n)$, SMOOTH PRODUCT $\mathsf{P}(n)$, AND EXPONENTIAL FUNCTION $\mathsf{E}(n)$. RELATIVE TIMINGS FOR THE BRENT-SALAMIN (AGM) ALGORITHM AND FOR THE EXPONENTIAL FUNCTION IN MPFR 4.2 ARE ALSO SHOWN.

n	$\frac{\mathsf{M}(n)}{\mathsf{A}(n)}$	$\frac{\mathsf{P}(n)}{\mathsf{M}(n)}$	$\frac{\mathsf{E}(n)}{\mathsf{P}(n)}$	$\frac{\mathsf{E}_{\text{AGM}}(n)}{\mathsf{E}(n)}$	$\frac{\mathsf{E}_{\text{MPFR}}(n)}{\mathsf{E}(n)}$
128	0.9	1.04	77.0	15.00	5.76
256	2.1	0.62	102.7	14.55	4.34
512	5.9	0.77	70.2	8.38	2.80
1024	19.5	0.49	62.1	4.91	2.04
2048	43.8	0.60	52.8	2.49	1.68
4096	72.0	0.70	46.8	1.61	1.72
8192	109.0	0.81	43.5	1.38	1.67
16384	159.9	0.91	43.6	1.34	1.62
32768	188.1	1.14	47.9	1.15	1.44
65536	187.9	1.74	48.9	1.12	1.46
131072	164.1	2.19	45.7	1.23	1.74
262144	145.1	2.60	44.9	1.32	1.99
524288	156.7	3.13	41.2	1.30	2.23
1048576	164.2	3.45	40.6	1.23	2.34
2097152	162.7	3.87	40.1	1.12	2.43
4194304	171.4	4.21	40.9	1.12	2.57
8388608	190.1	4.40	42.2	1.05	2.64
16777216	202.2	4.53	43.2	0.99	2.98
33554432	191.2	4.85	44.1	0.96	3.20

For comparison, Table I includes timings for two alternative implementations of $\exp(x)$: the Brent-Salamin algorithm (AGM) and `mpfr_exp` in MPFR 4.2. For FLINT's builtin exponential function, which uses precomputations, see Table III.

Several related phenomena can be observed roughly when we enter the FFT regime around $n \approx n_{\text{FFT}} \approx 25000$:

- The ratio $\mathsf{M}(n)/\mathsf{A}(n)$ becomes roughly constant, settling on a rather large order of magnitude $\approx 100 - 200$.
- The ratio $\mathsf{P}(n)/\mathsf{M}(n)$ becomes greater than 1 and subsequently grows slowly.
- The optimal value r stabilizes around 32 (see Table III).

The ratio $\mathsf{E}(n)/\mathsf{P}(n) \approx 50$ is remarkably constant considering that asymptotics predict $O(\log n)$.

III. REDUCTION USING TABLE LOOKUP

We will review several strategies for precomputation-based argument reduction, all of which share the same form (Algorithm 1).

Different choices of the rational parameters q_j lead to different specific algorithms, summarized in Table II. For each method, we indicate the expected cost $\mathsf{R}_T(n,r)$ to achieve $0 \leqslant \varepsilon < 2^{-r}$ and the size of precomputed tables (in bits), where n is the precision. The complexities will be justified further in the next sections.

Algorithm 1 Meta-algorithm to compute $\exp(x)$ to n-bit precision via table-based reduction to a number $0 \leqslant \varepsilon < 2^{-r}$

0) **Precomputation**: choose $q_1, \ldots, q_k \in \mathbb{Q}$, precompute $\log q_1, \ldots, \log q_k$ as n-bit fixed-point numbers and store them in a table.
1) **Argument reduction**: compute $c_1, \ldots, c_k \in \mathbb{Z}$ such that $0 \leqslant \varepsilon < 2^{-r}$ where $\varepsilon := x - L$,
$$L = c_1 \log(q_1) + \cdots + c_k \log(q_k). \tag{2}$$
2) **Taylor series**: compute $y := \exp(\varepsilon)$ as in section II-A.
3) **Reconstruction**: output $\exp(x) = y \cdot E$ where
$$E = q_1^{c_1} \cdots q_k^{c_k}. \tag{3}$$

To first order, the precomputation cost for each method can be be estimated as $\mathsf{T}_T(n) \approx k\mathsf{E}(n)$ for a table of size kn bits (see "Space" in the table). In practice, the numbers q_j in all methods have special form and the $\log(q_j)$ can therefore can be computed somewhat faster than general exponentials or logarithms, e.g. using binary splitting summation of appropriate Taylor series for $\log(x)$ or $\operatorname{arctanh}(x)$. We can also save time with batch evaluation schemes. For example, the constants $\{\log(2), \log(3), \log(5), \ldots\}$ used in the Diophantine method can be computed simultaneously using Machin-like formulas as discussed in [Joh22]. For the $q_i = 1 + 2^{-i}$ in the bitwise method, a useful method in the sub-FFT regime is to batch the evaluations with large i (say for $i \geqslant \sqrt{n}$), computing each reciprocal $1/3, 1/5, \ldots$ as a fixed-point number and adding or subtracting bit-shifted copies to sums for all the $\log(q_i)$.

We note here that for all algorithms, we can choose parameters so that a table valid for n also works efficiently as a table for any $n' < n$ by simply restricting to a subset of the table and reading only the most significant words of the entries.

IV. Bitwise reduction

Most traditional ways to perform argument reductions using table lookup can be expressed as the following specialization of Algorithm 1:

0) Precompute $L_i = \log q_i$ for $i = 1, \ldots, \varrho$, where the q_i are of the form $1 + k2^{-mj}$ with $1 \leqslant k < 2^m$.
1) Compute $i_1, \ldots, i_\kappa$ with $\varepsilon := x - \sum_{j=1}^{\kappa} L_{i_j} < 2^{-r}$.
2) Compute $y \approx \exp(\varepsilon)$ using another algorithm.
3) Output $y \prod_{j=1}^{\kappa} q_{i_j}$.

The precise ways how to choose the q_i and how to peform step 3 give rise to various variants that we shall discuss now.

A. Traditional Cordic-BKM style method

The most traditional variant is to take $\varrho := r$ and $q_i := 1 + 2^{-i}$. For step 1, we start with $\varepsilon := x$. For $j = 1, \ldots, \kappa$, we then take i_j maximal with $L_{i_j} < \varepsilon$ and update $\varepsilon \leftarrow \varepsilon - L_{i_j}$. The cost of this step is $\kappa\mathsf{A}(n)$, where the average value of κ is $r/2$. The required table of logarithms has size rn.

Letting $r = n$ gives the classical BKM method [BKM94], closely related to CORDIC [Vol59]. The hybrid method of combining partial BKM-style reduction with polynomial expansion (e.g. Taylor series) appears previously in [BEIR00].

B. Processing m-bits at a time, greedy variant

The computation time can be reduced by resorting to larger tables. For $m > 1$, we now take $q_{i(2^m-1)+k} := 1 + k2^{-mj}$, for $j = 1, \ldots, \lceil \frac{r}{2^m-1} \rceil$ and $k = 1, \ldots, 2^m - 1$. Hence $\varrho \approx (2^m - 1)r/m$ and the table size becomes $\frac{2^m-1}{m} rn$. Step 1 is done using the same method as above; its average cost drops to $\frac{1-2^{-m}}{m} r\mathsf{A}(n)$.

The most extreme version of this variant, when $m = r$, coincides with traditional table lookup. Taking $m = r/\tilde{m}$ corresponds to $\tilde{m}$-partite table lookup.

C. Processing m-bits at a time, sparse variant

The size of the table can be reduced by an approximate factor $m > 1$ by taking $\varrho := \lceil r/m \rceil$ and $q_i := 1 + 2^{-mi}$. For step 1, we again start with $\varepsilon := x$. For $k = 1, 2, \ldots$, let $i_{\max}$ be maximal with $L_{i_{\max}} < \varepsilon$. Then we add $\lfloor \varepsilon / L_{i_{\max}} \rfloor$ copies of $i_{\max}$ to the list of i_j and update $\varepsilon \leftarrow \varepsilon - \lfloor \varepsilon / L_{i_{\max}} \rfloor L_{i_{\max}}$. Assuming that every such update can be done in time $\mathsf{A}(n)$, the average cost of step 1 is $\frac{1-2^{-m}}{m} r\mathsf{A}(n)$, as for the greedy variant.

D. Terminate with shifts and adds

The traditional BKM algorithm computes $y \prod_{j=1}^{\kappa} q_{i_j}$ in step 3 by updating $y \leftarrow (1 + 2^{-i_j})y$ for $j = 1, \ldots, \kappa$, after which we return y. This also works for the other variants, except that multiplications with numbers of the form $1 + k2^{-jm}$ may also involve FMAs at machine precision instead of mere additions.

Assuming that FMAs are approximately as fast as additions, the overall cost now becomes $2\kappa\mathsf{A}(n)$. For the traditional and greedy variants, the average value of κ is $\frac{1-2^{-m}}{m} r$. For the sparse variant, the average value becomes $\frac{2^m-1}{2m} r$.

E. Terminate with binary splitting

Another idea is to compute $E = \prod_{j=1}^{\kappa} q_{i_j}$ using binary splitting in step 3 and then multiply with y. This is fastest when the bitsize p of E is at most n. Otherwise, the product can still be split into $\lceil p/n \rceil$ parts which are then multiplied out with full precision n. For the traditional and greedy variants, the expected bitsize p is $\kappa r/2 \approx \frac{1-2^{-m}}{2m} r^2$. For the sparse variant, we get $p \approx \frac{2^m-1}{4m} r^2$.

F. Discussion

It is instructive to analyze the cost of these reduction algorithms with respect to $\mathsf{P}(n)$. In order to be competitive with other algorithms (see section VII below), this ratio should remain reasonably small; ideally, it should remain below one. For the binary splitting variant, this implies $p \lessapprox n$; see the column $r_{\max}$ in Table II. For the greedy shift-and-add variant, the constraint translates into $\frac{r}{m}\mathsf{A}(n) \lessapprox \mathsf{M}(n)$; indeed, this variant is most useful when n is small or medium, so $\mathsf{P}(n) \propto \mathsf{M}(n)$.

TABLE II
ARGUMENT REDUCTION STRATEGIES BASED ON TABLE LOOKUP AND LINEAR COMBINATIONS OF LOGARITHMS.

Method	j	q_j	c_j	Space	$\mathsf{R}_T(n,r)$ (expected)	$r_{\max}$	$\mathsf{R}_T(n,r)$ (expected)
					Shift-add variants	Binary splitting variants	
Bitwise	$1 \leqslant j \leqslant r$	$1+2^{-j}$	$0,1$	rn	$\frac{3r}{2}\mathsf{A}(n)$	$2\sqrt{n}$	$\frac{r}{2}\mathsf{A}(n)+\mathsf{P}\left(\frac{r^2}{4},n\right)$
Greedy m-bitwise	$1 \leqslant a2^m+b \leqslant r$	$1+b2^{-am}$	$0,1$	$\frac{2^m}{m}rn$	$\frac{3r}{m}\mathsf{A}(n)$	$\sqrt{2mn}$	$\frac{r}{m}\mathsf{A}(n)+\mathsf{P}\left(\frac{r^2}{2m},n\right)$
Sparse m-bitwise	$1 \leqslant j < r/m$	$1+2^{-jm}$	$0,\ldots,2^m-1$	$\frac{1}{m}rn$	$\frac{2^m r}{m}\mathsf{A}(n)$	$2^{1-m/2}\sqrt{mn}$	$\frac{r}{m}\mathsf{A}(n)+\mathsf{P}\left(\frac{2^m r^2}{4m},n\right)$
Diophantine	$1 \leqslant j \leqslant 2$	$2,3$	$O(2^r)$	$2n$	$O(\beta^{-1}2^r)\mathsf{A}(n)$	$\log_2\left(\frac{n}{2\mathrm{e}}\right)$	$1.4r\mathsf{A}(n)+\mathsf{P}(\mathrm{e}2^{r+1},n)$
m-Diophantine	$1 \leqslant j \leqslant m$	$2,3,\ldots,p_m$	$O(m2^{r/(m-1)})$	mn	$O(\beta^{-1}2^{r/(m-1)}$ $m^2\log m)\mathsf{A}(n)$	$2.9m \leqslant \sqrt{\frac{0.41n}{\log_2 n}}$	$1.4r\mathsf{A}(n)+\mathsf{P}(\mathrm{e}^3m^2\log_2 m,n)$

G. SIMD acceleration

If n is large, then step 1 can benefit from from SIMD accelerations, for all three variants. Let us briefly sketch how in the case of the greedy m-bitwise algorithm. Instead of doing the updates $t \leftarrow t - L_i$ for one i at the time, we proceed by batches of, say β, indices i. This means that our updates are of the form $t \leftarrow t - \Sigma$, where $\Sigma \leftarrow \sum_{i\in\mathcal{I}} L_i$ for a batch $\mathcal{I}$ of new indices in S. Here we rely on the fact that $\mathcal{I}$ can essentially be determined from the β most significant digits of t. For the computation of Σ, we represent the L_i using a redundant SIMD representation that allows for the addition of β numbers without carry propagation. The normalization is done at the end.

V. DIOPHANTINE APPROXIMATION

The Diophantine approximation method [Sch06], [Joh22] is an instance of Algorithm 1 which achieves r-bit reduction using much smaller tables than the methods in the previous section. Take $q_1,\ldots,q_m$ to be the first $m \geqslant 2$ prime numbers. Then any x can be approximated arbitrarily well by $\mathbb{Z}$-linear combinations $c_1\log(q_1)+\cdots c_m\log(q_m)$. Equivalently, in exponential form, any $y>0$ can be approximated arbitrarily well by q_m-smooth rational numbers $q_1^{c_1}\cdots q_m^{c_m}$.

Heuristically, we can find such an r-bit accurate approximation with $|c_j| = O(2^{r/(m-1)})$. For example, using the first $m=2$ primes, some approximations of π are

$$2^8\cdot 3^{-4} = 3.16\ \ldots$$
$$2^{1931643}\cdot 3^{-1218730} = 3.141592601\ \ldots$$
$$2^{-3824416943916269}\cdot 3^{2412938439979599} = 3.1415926535897933\ \ldots$$

where the last approximation achieves 16-digit accuracy using 16-digit coefficients c_j. Using $m=5$ primes, we can achieve 16-digit accuracy with 4-digit c_j:

$$2^6\cdot 3^4\cdot 5^{-10}\cdot 7^2\cdot 11^2 = 3.1473\ \ldots$$
$$2^{-31}\cdot 3^{-57}\cdot 5^{136}\cdot 7^{41}\cdot 11^{-89} = 3.141592609\ \ldots$$
$$2^{-583}\cdot 3^{3227}\cdot 5^{7718}\cdot 7^{-8681}\cdot 11^{555} = 3.1415926535897934\ \ldots$$

Finding such c_j amounts to solving the approximate inhomogeneous integer relation problem $x \approx c_1\alpha_1+\cdots+c_m\alpha_m$. We solve this problem in two stages:

1. Precomputation (independent of both x and n, depending only on the algorithm parameters m and r): use LLL [LLL82] to find increasingly precise solutions to the homogeneous problem $d_1\alpha_1+\cdots+d_m\alpha_m \approx 0$, for example:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0\\ 1 & 1 & -1 & 0 & 0\\ 0 & -1 & -2 & 1 & 1\\ 1 & 2 & -3 & 1 & 0\\ -3 & 4 & -2 & -2 & 2\\ -2 & 2 & 2 & -7 & 4\\ -18 & -3 & 22 & 1 & -9\\ 19 & -23 & -22 & 1 & 19 \end{pmatrix}\begin{pmatrix}\log(2)\\ \log(3)\\ \log(5)\\ \log(7)\\ \log(11)\end{pmatrix} = \begin{pmatrix} 0.693\\ 0.182\\ 0.0263\\ 0.00797\\ 0.000102\\ 1.61\cdot 10^{-5}\\ 6.51\cdot 10^{-7}\\ 4.99\cdot 10^{-8}\end{pmatrix}$$

Let $(d_{i,1},\ldots,d_{i,m})$ be row vectors of such solutions and $\varepsilon_i = d_{i,1}\log q_1+\cdots+d_{i,m}\log q_m$, for $i=1,\ldots,\ell$. For a tuning parameter $\tau>0$, we retain the best solution that achieves $\varepsilon_i \approx 2^{-\tau i}$.

2. For a given x, we now compute $c_1,\ldots,c_n$ as follows. We start with $c_j \leftarrow 0$ for $j=1,\ldots,m$. For $i=1,\ldots,\ell$, we let $k_i \leftarrow \lfloor x/\varepsilon_i \rceil$ and we update $x \leftarrow x - k_i\varepsilon_i$ and $c_j \leftarrow c_j + k_i d_{i,j}$ (for $j=1,\ldots,m$).[1]

To analyze the parameters in this method, we will assume that the discovered integer relations satisfy heuristic bounds stronger than those strictly guaranteed by LLL.

In order to achieve an r-bit argument reduction, we should have $\varepsilon_\ell \approx 2^{-\tau\ell} \approx 2^{-r}$, so we take $\ell \approx r/\tau$. Heuristically, we have $|d_{i,j}| \lessapprox 2^{\tau i/(m-1)}$, so $|c_j| \lessapprox 2^{\tau+\ell\tau/(m-1)}/(1-2^{-\tau/(m-1)})$. In order to keep the $|c_j|$ small, it is therefore important to take τ small. When $\tau \leqslant m$, we may use the approximation $(1-2^{-\tau/(m-1)})^{-1} \approx (m-1)/(\tau\log 2)$, which is correct up to a factor two. Since $\log_2(q_1\cdots q_m) \approx m\log_2 m$ (by the prime number theorem), we then have

$$\log_2(q_1^{|c_1|}\cdots q_m^{|c_m|}) \quad \lessapprox \quad 2^{\tau+\frac{r}{m-1}}\frac{m-1}{\tau\log 2}m\log_2 m.$$

For similar reasons as in section IV, we wish to bound the right hand side by n. For fixed m and r, the minimum of $2^\tau/\tau$ is reached at $\tau \approx 1/\log 2 \approx 1.44$, after which the right hand side simplifies into $\varphi(m,r) := \mathrm{e}2^{r/(m-1)}(m-1)m\log_2 m$. When m is fixed, we have $\varphi(m,r) \leqslant n$ as long as

$$r \leqslant (m-1)\log_2\frac{n}{\mathrm{e}(m^2-m)\log_2 m}. \qquad (4)$$

[1]This algorithm can be viewed as a version of Babai's nearest plane algorithm for solving the closest vector problem (CVP) for lattices [Bab86]. The authors thank Léo Ducas for pointing this out.

In the case when $m = 2$, this yields $r \leqslant \log_2(n/2\mathrm{e})$. Let us now consider $\varphi(m, r) \approx \mathrm{e}2^{r/m} m^2 \log_2 m$ for larger values of m. If we fix the ratio $\lambda := r/m$, then $\varphi(m, r) \approx n$ yields

$$m \approx \sqrt{\frac{n}{\mathrm{e}2^{\lambda} \log_2 m}} \approx \sqrt{\frac{n}{\mathrm{e}2^{\lambda} \log_2 n}}$$

Hence $r = \lambda m$ is maximal if $\frac{\lambda}{2^{\lambda/2}}$ is maximal, which happens for $\lambda \approx \frac{2}{\log 2} \approx 2.89$. Taking λ this way, we then have $\varphi(m, r) \leqslant n$ when $m \lessapprox \sqrt{n/(\mathrm{e}^3 \log_2 n)}$.

The logarithms $\log q_1, \ldots, \log q_m$ and the matrix $(d_{i,j})$ are precomputed and respectively require nm and $\approx r\ell$ bits of space. Since $\varphi(m, r) \leqslant n$, the entries $d_{i,j}$ and the c_j fit into integers of size β. If they actually fit into integers of size $\beta/2, \beta/4, \ldots$, then we may use a packed representation for the matrix $(d_{i,j})$. Optionally (and in particular when $r = \lambda m$ with λ as above), we may also store $\varepsilon_1, \ldots, \varepsilon_\ell$ in a table, which requires $n\ell \approx nr/\tau \approx nr \log 2$ more bits of space. Such a table can be (re)computed in time $\ell m \mathsf{A}(n) \approx rm\mathsf{A}(n) \log 2$.

With these tables, the costs of the updates $x \leftarrow x - k_i \varepsilon_i$ and $c_i \leftarrow c_i + k_i d_{i,j}$ are bounded by $\ell\mathsf{A}(n)$ resp. $\ell m \mathsf{M}(\tau + r/m)$. Since $\tau + r/m \leqslant \beta$, we have $\mathsf{M}(\tau + r/m) \approx \mathsf{A}(\beta)$, the cost of an arithmetic operation with machine precision. If $\tau + r/m \leqslant \beta/2^\kappa$, then we may achieve $m\mathsf{M}(\tau + r/m) \approx m\mathsf{A}(\beta)/2^\kappa$ by storing the $d_{i,j}$ in packed format.

If m is small, then the cost of all updates becomes $r\mathsf{A}(n) \log 2 + rm\mathsf{M}(\beta) \log 2 \approx r\mathsf{A}(n) \log 2$, which is typically negligible with respect to $\mathsf{P}(n)$ due to the bound (4). If m is large and $r = \lambda m$ with $\lambda = 2/\log 2$ as above, then the cost becomes $2m\mathsf{A}(n) + 2m^2\mathsf{M}(\beta)$. Since $m^2 \approx n/(\mathrm{e}^3 \log_2 n)$, we have $2m^2\mathsf{M}(\beta) \lessapprox (2\beta/(\mathrm{e}^3 \log_2 n))\mathsf{M}(n)$, which is generally small compared to $2m\mathsf{A}(n)$. The cost therefore simplifies to $2m\mathsf{A}(n)$.

Under the constraints on m, the denominator and numerator of $E := q_1^{c_1} \cdots q_m^{c_m}$ are typically both of size $\lessapprox n/2$. The cost to evaluate this fraction with a precision of n bits is bounded by $2\mathsf{P}(n/2) + c\mathsf{M}(n) = \mathsf{P}(n) + (c - 1/2)\,\mathsf{M}(n)$, where $c \leqslant 5/3$ depends on the multiplicative regime.

See Table II for a summary of the complexities and memory requirements.

VI. Empirical comparison of table-based methods

We have implemented prototype code for computing $\exp(x)$ using the following table-based methods:

1) The m-Diophantine method
2) The 1-bitwise method (both the classical and binary splitting variants)
3) The greedy 8-bitwise method

Table III, presents the absolute running time for the table-free method as a baseline $\mathsf{E}(n)$, compared with the relative costs $\tilde{\mathsf{E}}_T = \mathsf{E}_T/\mathsf{E}$ and $\tilde{\mathsf{T}}_T = \mathsf{T}_T/\mathsf{E}$ of the other methods.

For each method, we report the series strategy (EXP, SH, BSH, BB), reduction parameter r, and number of primes m (Diophantine method only), which minimize E_T, by exhaustively timing all combinations. We have restricted the search space to r and m of the form 2^k or $3 \cdot 2^k$.

We also show the corresponding $\tilde{\mathsf{E}}_T$ for the current default exponential function `arb_exp` in FLINT. That implementation uses static tables for multiplicative reduction $\exp(x) = \exp(x - j/2^r)\exp(j/2^r)$ at low precision (5 KB tables for $r = 8$ up to $n < 512$ and 36 KB bipartite tables for $r = 10$ up to $n < 4608$) [Joh15] and dynamic m-Diophantine tables with $m = 13$ for $n < 4194304$.

Some of our observations follow.

A. Best methods for dynamic tables

For applications requiring N function evaluations, the measured $\tilde{\mathsf{E}}_T$ and $\tilde{\mathsf{T}}_T$ values roughly suggest generating

- 8-bitwise tables when we expect $N > 10^3$,
- 1-bitwise tables when we expect $N > 10^2$,
- m-Diophantine tables when we expect $N > 10^1$,

provided, in each case, that the memory consumption is acceptable for the given n.

Note that the parameters in Table III are chosen for $N \to \infty$. If the goal is to minimize the cost for a specific N, then the parameter values in the table are suggestive, but somewhat different parameters may be optimal: typically, it will be more efficient to trade a much (e.g. 2 or 4 times) smaller r and T_T for a slightly (e.g. 10%) larger E_T.

B. Static tables

The 1-bitwise method is up to twice as fast as the argument reduction with static tables currently used in FLINT's `arb_exp` while using comparably-sized tables. The superiority of the classical BKM-style reduction is explained by the large size of $\mathsf{M}(n)/\mathsf{A}(n)$ beyond machine precision, making addition of logarithms better than multiplication of exponentials. The 8-bitwise method is even more efficient, but uses significantly larger tables.

For software that can ship with $\approx 10^2$ KB of static tables for elementary functions, it appears reasonable to use 1-bitwise tables for n up to a few thousand and $(m \approx 8)$-bitwise tables for n up to a few hundred.

C. Huge tables

If $N \to \infty$, further speedups are possible using even larger m-bitwise tables. For example, 16-bitwise reduction (tested but not reported in Table III) achieves $\tilde{\mathsf{E}}_T = 0.29$ for $n = 128$ with a 3 MB table ($r = 48$) and $\tilde{\mathsf{E}}_T = 0.16$ for $n = 4096$ with a 2 GB table ($r = 1024$). In practice, there may be diminishing returns for such large tables due to cache size and memory bandwidth bottlenecks; we have not investigated these effects further.

D. Maximum speedup

The speedup $1/\tilde{\mathsf{E}}_T$ with each tested table-based method is maximized near $n \approx 2^{16}$ bits. Intuitively, this occurs when we enter the FFT regime where $\mathsf{M}(n)/\mathsf{A}(n)$ flattens out. Beyond this point, trading full-precision multiplications for additions gives little further improvement.

TABLE III

CALCULATING $\exp(x)$, $x = \sqrt{2} - 1$, TO n-BIT PRECISION, USING VARIOUS ALGORITHMS. FOR EACH METHOD AND n, WE SHOW THE EMPIRICALLY DETERMINED OPTIMAL PARAMETER r AND SERIES EVALUATION STRATEGY. FOR THE METHODS USING A TABLE T, WE SHOW THE RELATIVE EVALUATION TIME $\tilde{\mathsf{E}}_T = \mathsf{E}_T/\mathsf{E}$ AND PRECOMPUTATION TIME $\tilde{\mathsf{T}}_T = \mathsf{T}_T/\mathsf{E}$ (LOWER VALUES ARE BETTER).

	No tables			m-Diophantine						1-bitwise					Greedy 8-bitwise					FLINT
Bits n	r	Series	Time (E)	m	r	Series	$\tilde{\mathsf{E}}_T$	$\tilde{\mathsf{T}}_T$	Table	r	Series	$\tilde{\mathsf{E}}_T$	$\tilde{\mathsf{T}}_T$	Table	r	Series	$\tilde{\mathsf{E}}_T$	$\tilde{\mathsf{T}}_T$	Table	$\tilde{\mathsf{E}}_T$
128	4	EXP	0.18 µs	2	6	EXP	4.16	16.0	32 B	8	EXP	0.72	46	128 B	48	EXP	0.47	1995	24 KB	0.76
256	12	EXP	0.38 µs	4	16	EXP	2.53	12.4	128 B	24	EXP	0.56	56	768 B	32	EXP	0.43	1732	32 KB	0.63
512	8	EXP	1.06 µs	4	16	EXP	1.40	7.1	256 B	32	EXP	0.51	36	2.0 KB	64	EXP	0.30	1491	128 KB	0.69
1024	12	SH	3.28 µs	3	24	EXP	0.98	4.0	384 B	64	EXP	0.43	29	8.0 KB	128	EXP	0.25	1244	512 KB	0.80
2048	12	SH	11.5 µs	3	24	SH	0.72	2.3	768 B	128	EXP	0.34	23	32 KB	256	EXP	0.21	1017	2.0 MB	0.79
4096	24	SH	37.8 µs	12	48	SH	0.57	3.2	6.0 KB	128	SH	0.31	19	64 KB	512	EXP	0.21	984	8.0 MB	0.56
8192	24	SH	0.12 ms	16	64	SH	0.49	3.2	16 KB	256	SH	0.26	22	256 KB	512	SH	0.19	959	16 MB	0.45
16384	24	BSH	0.41 ms	32	128	SH	0.43	5.3	64 KB	384	SH	0.26	20	768 KB	768	SH	0.17	892	48 MB	0.45
32768	32	BSH	1.40 ms	32	192	BSH	0.35	4.2	128 KB	512	SH	0.22	17	2.0 MB	1024	SH	0.16	850	128 MB	0.48
65536	16	BB	4.11 ms	32	192	BSH	0.38	3.9	256 KB	768	SH	0.23	19	6.0 MB	1536	SH	0.16	948	384 MB	0.59
131072	32	BB	10.0 ms	32	192	BSH	0.41	4.0	512 KB	1024	SH	0.24	22	16 MB	2048	SH	0.17	1096	1.0 GB	0.62
262144	32	BB	24.0 ms	32	192	BSH	0.44	3.9	1.0 MB	768	BSH	0.29	23	24 MB	2048	SH	0.19	1271	2.0 GB	0.60
524288	32	BB	56.1 ms	64	384	BSH	0.45	8.8	4.0 MB	1536	SH	0.31	34	96 MB	2048	SH	0.22	1494	4.0 GB	0.62
1048576	32	BB	0.13 s	64	512	BSH	0.48	8.5	8.0 MB	1536	BSH	0.32	39	192 MB	4096	SH	0.23	2198	16 GB	0.64
2097152	32	BB	0.30 s	64	512	BSH	0.52	8.5	16 MB	2048	BSH	0.34	52	512 MB						0.67
4194304	32	BB	0.69 s	64	512	BB	0.61	8.3	32 MB	3072	BSH	0.35	72	1.5 GB						1.00
8388608	32	BB	1.60 s	96	768	BB	0.62	12.5	96 MB	4096	BSH	0.37	96	4.0 GB						1.00
16777216	32	BB	3.66 s	96	768	BB	0.62	12.3	192 MB	4096	BSH	0.42	105	8.0 GB						1.01
33554432	32	BB	8.38 s	96	1024	BB	0.62	12.1	384 MB	4096	BB	0.46	109	16 GB						1.01

E. *Binary splitting products*

The crossover where our binary splitting variant of the bitwise method wins over the classical BKM-like shift-and-add version is 262K bits (in the table, we report timings for the classical method below this point and for the binary splitting version above). For n in the millions, it gives a 10% to 20% speedup. For example, at 1M bits, we have $\tilde{\mathsf{E}}_T = 0.32$ with binary splitting and $\tilde{\mathsf{E}}_T = 0.37$ without.

F. *Low precision*

In the few-word regime, say for $n \leqslant 1024$, function call and loop overheads remain significant in our prototype code which handles generic n, and we should be able to achieve better performance by specializing code for each multiple of the word size. For example, a version of the 1-bitwise method for $n = 128$ using fully inlined double-word arithmetic runs in around 57 ns ($\tilde{\mathsf{E}}_T = 0.31$), versus 133 ns ($\tilde{\mathsf{E}}_T = 0.72$) for the generic code. We leave a closer study for future work.

VII. BINARY SPLITTING

The binary splitting technique has a double character: it can both be considered as one of the strategies for power series evaluation and as another strategy for argument reduction (from $|x| < 2^{-r}$ to $|x| < 2^{-2r}$). Let us recall the technique with more details and discuss a few variants.

For complexity analyses in the FFT regime, we assume that the reader is familiar with the fact that one multiplication requires three conversions into and from an FFT representation that is crafted for the bit-size of the result. For a result of bit-size n, the cost of one conversion is approximately $\mathsf{M}(n)/6$.

A. *Traditional binary splitting*

Assume that $0 \leqslant x < 2^{-r}$ and $n = 2^l r$. We decompose $x = L + t$ with $L \in 2^{-2r}\mathbb{N}$ and $0 \leqslant y < 2^{-r}$. Our aim is the efficient approximation $E \approx \mathrm{e}^L$ with a precision of n bits, after which we obtain the exponential of x as $\mathrm{e}^x \approx Ey$, where $y \approx \mathrm{e}^t$. In what follows, it will be suggestive to set $\varepsilon := L$. For $k \in \mathbb{N}$ and $\delta \in 2^{\mathbb{N}}$, we define

$$\Sigma_{k;\delta} := \sum_{0 \leqslant i < \delta} \Pi_{k+i;\delta-i} \varepsilon^i, \quad \Pi_{k;\delta} := \frac{(k+\delta)!}{k!},$$

so that

$$\Sigma_{k;2\delta} = \Pi_{k+\delta;\delta} \Sigma_{k;\delta} + \Sigma_{k+\delta;\delta} \varepsilon^{\delta} \quad (5)$$

$$\Pi_{k;2\delta} = \Pi_{k+\delta;\delta} \Pi_{k;\delta}. \quad (6)$$

We compute $\Sigma_{0;2^l}$ and $\Pi_{0;2^l}$ using these recursive relations, after which $\Sigma_{0;2^l}/\Pi_{0;2^l} \approx \mathrm{e}^{\varepsilon}$. (A minor technical improvement would be to factor out $(k+\delta)!$ from $\Sigma_{k;\delta}$.)

For $k + \delta \leqslant n$, the bit-size of $\Pi_{k;\delta}$ is bounded by $\delta \log_2 n$ and the bit-size of $\varepsilon^{\delta} 2^{-\delta r}$ is at most δr. Since binary splitting is usually applied after some of the other argument reductions, we may assume $r \gg \log n$. Then the bit-size of $\Sigma_{k;\delta}$ is approximately δr and the cost of computing $\Pi_{0;2^l}$ (and all intermediate $\Pi_{k;\delta}$) negligible with respect to the cost to compute $\Sigma_{0;2^l}$.

In the naive, Karatsuba, and Toom-Cook regimes with $\mathsf{M}(n) \propto n^\gamma$, the multiplications $\Pi_{k+\delta}\Sigma_{k;\delta}$ are also much cheaper than the multiplications $\Sigma_{k+\delta;\delta}\varepsilon^\delta$. Consequently, the cost of the full algorithm is approximately the same as the cost $\mathsf{P}(n) \approx \mathsf{M}(n)/(2^\gamma - 2)$ of multiplying 2^l integers of bit-size $\leqslant r$ using binary splitting plus the cost $\leqslant 2/3\mathsf{M}(n)/(2^\gamma - 1)$ of the repeated squarings $\varepsilon^2, \varepsilon^4, \ldots, \varepsilon^{2^{l-1}}$. The cost of the final division is again negligible.

In the FFT regime, the cost of the multiplications $\Pi_{k+\delta}\Sigma_{k;\delta}$ and the final division cannot necessarily be neglected, so the total cost of the algorithm may *a priori* become as large as $2\mathsf{P}(n) + 4/3\mathsf{M}(n)$. But $\Sigma_{k;2\delta}$ can be computed in the FFT model and the FFT transform of ε^δ can also be cached. The complexity accordingly drops to $4/3\mathsf{P}(n) + 4/3\mathsf{M}(n)$. In the most favorable case when $n/r \ll n_{\text{fft}}$, the multiplications and divisions by the $\Pi_{k;\delta}$ actually do become negligible. In that case, the cost of the algorithm further drops to $2/3\mathsf{P}(n)$.

B. Optimizations when r approaches n

Below the FFT regime, the cost of binary splitting is only a constant times larger than the cost of multiplication. In the FFT regime, we recall that $\mathsf{P}(n) \approx \mathsf{M}(n)\log(n/n_{\text{FFT}})$. A particularly interesting case for us is when $r \geqslant n_{\text{FFT}}$ and n/r is moderately large (e.g. $n/r \approx 2^8$). Can we reduce the $\log(n/n_{\text{FFT}})$ overhead with respect to multiplication in this case?

Two things that we wish to exploit are the fact that multiplications with the $\Pi_{k;\delta}$ are cheap in this regime and that we may precompute some powers of ε and replace (5) a more efficient formula. More precisely, let $R = \Delta r$ with $\Delta = 2^\kappa$ be such that $r \leqslant R \leqslant n$. For any $k \in \Delta\mathbb{N}$ with $k < n$, we have

$$\Sigma_{k;\Delta} = \Pi_{k;\Delta} + \Pi_{k+1;\Delta-1}\varepsilon + \cdots + \Pi_{k+\Delta-1;1}\varepsilon^{\Delta-1}. \quad (7)$$

Assuming that $\varepsilon, \ldots, \varepsilon^{\Delta-1}$ are known, the computation of $\Sigma_{k;\Delta}$ is cheap, under our assumptions. From $\Sigma_{0;\Delta}, \Sigma_{\Delta;\Delta}, \ldots, \Sigma_{2^l-\Delta;\Delta}$, we may complete the computation of $\Sigma_{0;2^l}$ in time $(2/3(l-\kappa) + 1/3)\,\mathsf{M}(n)$, using binary splitting.

In the FFT regime, we can compute $\varepsilon^2, \ldots, \varepsilon^{\Delta-1}$ efficiently, by writing $\Delta = \Delta_1\Delta_2$ with $\Delta_1 \approx \Delta_2$, by precomputing $\varepsilon^2, \varepsilon^3, \ldots, \varepsilon^{\Delta_1-1}$ (of negligible cost) and $\varepsilon^{\Delta_1}, \varepsilon^{2\Delta_1}, \ldots, \varepsilon^{\Delta-\Delta_1}$, and then compute all products $\varepsilon^i\varepsilon^{j\Delta_1}$ jointly using an FFT representation that can contain an R-bit result. This can be done in time $\mathsf{S}(\Delta) \leqslant \mathsf{S}(\Delta_2) + (\Delta_1 + \Delta_2)\mathsf{M}(R)/6 + (\Delta_1 - 1)(\Delta_2 - 1)\mathsf{M}(R)/6 \leqslant (\Delta + \Delta_2 + \cdots)\mathsf{M}(R)/6 \approx (2^{2\kappa-l}/6)\mathsf{M}(n)$. For instance, if $n = 2^8 r$, then taking $\Delta = 32$, the total cost becomes $(2 + 1/3 + 4/6)\,\mathsf{M}(n) = 3\mathsf{M}(n)$, which is better than $17/3\mathsf{M}(n)$. In general, taking $\kappa = \lceil l/2 \rceil$, we achieve an approximate speed-up of two.

C. An asymptotic optimization

The above optimization accelerates the work for the nodes of the binary splitting trees that are close to the leafs. Can we do something similar for the inner nodes?

More precisely, assuming that $\Sigma_{0;\Delta}, \Sigma_{\Delta;\Delta}, \ldots, \Sigma_{2^l-\Delta;\Delta}$ are known, can we effciently compute $\Sigma_{0;\Delta'}, \Sigma_{\Delta';\Delta'}, \ldots, \Sigma_{2^l-\Delta';\Delta'}$ for some larger Δ' with $\Delta \mid \Delta' \mid 2^l$?

For any $k \in \Delta'\mathbb{N}$ with $k < n$, formula (7) generalizes to

$$\Sigma_{k;\Delta'} = \sum_{0 \leqslant i < \Delta'/\Delta} \Pi_{k+\Delta i;\Delta'-\Delta i}\Sigma_{k+\Delta i;\Delta}(\varepsilon^\Delta)^i. \quad (8)$$

In the most favorable case, the products $\Pi_{k+\Delta i;\Delta'-\Delta i}\Sigma_{k+\Delta i;\Delta}$ are cheap. *A minima*, our assumption $r \gg \log n$ ensures that their bit-sizes are approximately bounded by $r\Delta$.

Assuming that the products $\Pi_{k+\Delta i;\Delta'-\Delta i}\Sigma_{k+\Delta i;\Delta}$ are known, we wish to evaluate (8) in the FFT model. For this, we first precompute $\varepsilon^\Delta, \varepsilon^{2\Delta}, \ldots, \varepsilon^{\Delta'-\Delta}$. We next cut these powers into chunks of R bits and transform them into an FFT model capable of holding products of (a bit more than) $2R$ bits. We next transform the products $\Pi_{k+\Delta i;\Delta'-\Delta i}\Sigma_{k+\Delta i;\Delta}$. We can now evaluate (8) in the FFT model and finally transform back in order to obtain the desired result.

Not counting the precomputations, this method allows us to compute $\Sigma_{k;\Delta'}$ using Δ'/Δ forward and backward transforms and $\binom{\Delta'/\Delta}{2}$ products in the FFT model. Altogether, this can be done in time $\frac{2}{3}(\Delta'/\Delta)\mathsf{M}(R) + O((\Delta'/\Delta)^2 R)$. The $O((\Delta'/\Delta)R)$ term is subdominant as long as $\Delta'/\Delta = O(\log R)$. In unfavorable cases when the bit-size of $\Pi_{k+\Delta i;\Delta'-\Delta i}$ exceeds n_{FFT}, one may compute the products $\Pi_{k+\Delta i;\Delta'-\Delta i}\Sigma_{k+\Delta i;\Delta}(\varepsilon^\Delta)^i$ in an FFT representation for products of three numbers. The $\frac{2}{3}(\Delta'/\Delta)\mathsf{M}(R)$ term should then be replaced by $(\Delta'/\Delta)\mathsf{M}(R)$.

The total cost to compute $\Sigma_{0;\Delta'}, \ldots, \Sigma_{2^l-\Delta'}$ thus lies between $2/3\mathsf{M}(n)$ and $\mathsf{M}(n)$, still not counting precomputations. In other words, we are able to do $\log_2(\Delta'/\Delta) = O(\log R)$ recursive levels for about the price of a single one without the FFT optimizations. This acceleration is an example of FFT trading [Hoe10], [Hoe16].

The precomputations take time $\approx (\Delta'/\Delta)^2\mathsf{M}(R)/3$, by first computing $\varepsilon^\Delta, \varepsilon^{2\Delta}, \ldots, \varepsilon^{\Delta'-\Delta}$ as above and then converting into the chunked FFT model. As long as $(\Delta'/\Delta)^2 R \leqslant n$, the cost of the precomputations remains small with respect to $2/3\mathsf{M}(n)$.

Now consider the application of the above technique to compute all $\Sigma_{k;\Delta}$ for $\Delta = \Lambda, \Lambda^2, \ldots, \Lambda^p$ and $k = 0, \Delta, \ldots, (\Lambda - 1)\Delta$, where $\Lambda = 2^{\lfloor \log\log_2 n \rfloor}$ and p is maximal with $\Lambda^{p+1} \leqslant 2^l$. By what precedes, this can be done in time $O(p\mathsf{M}(n)) = O(l\mathsf{M}(n)/\log\log n)$. We may deduce $\Sigma_{0;2^l}$ and $\Pi_{0;2^l}$ using $l - p\log_2\Lambda \leqslant 2\log_2\Lambda$ conventional binary splitting steps of cost $O(\mathsf{M}(n)\log\log n)$.

As a conclusion, we have reduced the overall price of the argument reduction by an asymptotic factor $\log\log n$. This actually leads to a general exponentiation algorithm of cost $\mathsf{E}(n) = O(\mathsf{M}(n)\log^2 n/\log\log n)$, based on binary splitting only. This is worse than the Brent-Salamin method (of complexity $O(\mathsf{M}(n)\log n)$, but it remains remarkable that the asymptotic complexity of the binary splitting method can be reduced at all in this case.

VIII. Evaluating the Taylor series

Let us now investigate methods for evaluating the final Taylor series for exp after completion of all argument reductions. At this point, we assume that $0 \leqslant \varepsilon < 2^{-r}$ and that we want to compute $1 + \cdots + \frac{1}{(N-1)!}\varepsilon^{N-1}$ for $N \approx n/r$.

A. Rectangular splitting

For $p, q \in \mathbb{N}$ with $pq \approx N$, we may write

$$\mathrm{e}^{\varepsilon} \approx \frac{1}{(N-1)!} \sum_{0 \leqslant i < p} \left[\sum_{0 \leqslant j < q} \frac{(N-1)!}{(pj+i)!} (\varepsilon^p)^j \right] \varepsilon^i.$$

Let $y_{i,j} := \frac{(N-1)!}{(pj+i)!}(\varepsilon^p)^j$ be the innermost summand and $Y_i := y_{i,0} + \cdots + y_{i,q-1}$. The advantage of organizing the double sum in this way is that $y_{i,j}$ can be deduced from $y_{i+1,j}$ using $y_{i,j} = (pj+i+1)y_{i+1,j}$, where $pj+i+1$ always fits in a single word. Starting with the values $y_{p-1,0}, \ldots, y_{p-1,q-1}$ and Y_{p-1}, we use this to compute $y_{i,0}, \ldots, y_{i,q-1}$ and Y_i for $i = p-2, \ldots, 0$. We finally compute $Y_0 + Y_1\varepsilon + \cdots + Y_{p-1}\varepsilon^{p-1}$.

This time, ε and ε^p essentially have full n-bit precision. In the naive regime, $\varepsilon^2, \ldots, \varepsilon^{p-1}$ are most efficiently computed using squaring for all even powers. Since one square can be done in time $1/2\mathsf{M}(n)$, the p first powers can be computed in time $3/4p\mathsf{M}(n)$. In the Karatsuba and Toom-Cook regimes, squaring takes time $2/3\mathsf{M}(n)$, and the complexity becomes $5/6p\mathsf{M}(n)$. In the FFT regime, assuming for simplicity that $p = s^2$, we first compute transforms of $\varepsilon, \ldots, \varepsilon^{s-1}$ and $\varepsilon^s, \ldots, (\varepsilon^s)^{s-1}$, and then retrieve the ε^{is+j} from products of these transforms. The cost is $(p/3 + 2s/3 + O(\sqrt{s}))\,\mathsf{M}(n)$.

Altogether, the cost of rectangular splitting is $c(p+q)\mathsf{M}(n) + 2pq\mathsf{A}(n)$, where $c \in (1/3, 5/6)$ depends on the multiplicative regime. The second term is typically negligible. Taking $p \approx q$, the complexity thus becomes $2c\sqrt{N}\mathsf{M}(n)$.

For which r should we use this method? When taking r twice as large, we save $c'\sqrt{N}\mathsf{M}(n)$ operations where $c' = 2c\left(1 - \sqrt{1/2}\right)$. This should be compared with the cost of using one step of binary splitting. In the Karatsuba regime, we have $c' \approx 0.5$ and the cost of one step of binary splitting is $4/3\mathsf{M}(n)$. The threshold for N therefore lies around 7. In the FFT regime, c' may approach 0.2 and one binary splitting step costs $1/3\mathsf{M}(n)\log(n/n_{\mathrm{fft}})$, so the threshold becomes $N \approx 2.8\log_2^2(n/n_{\mathrm{fft}})$. When using the optimized version of binary splitting, the threshold becomes $N \approx 0.7\log_2^2(n/n_{\mathrm{fft}})$.

B. Hyperbolic optimizations

An interesting question concerns the existence of "higher order" generalizations of the hyperbolic formula (1). It is remarkable that an order three generalization (mainly of theoretical interest) indeed exists: with $f : x \mapsto 1 + \frac{1}{6!}x^6 + \frac{1}{12!}x^{12} + \cdots$, we claim that e^{ε} can be recovered from $f(\varepsilon)$ and $f(2\varepsilon)$. Indeed, setting $\omega = \mathrm{e}^{2\pi\mathrm{i}/6}$ and $y_i := \exp(\omega^i\varepsilon)$ for $i = 0, \ldots, 5$, we have $\mathrm{e}^{\varepsilon} = y_0$, $f(\varepsilon) = y_0 + \cdots + y_5$, and $f(2\varepsilon) = y_0^2 + \cdots + y_5^2$. The remarkable relations $y_3 = -y_0$, $y_4 = -y_0$, $y_5 = -y_0$, and $y_1 = y_0y_2$ allow us to express $f(\varepsilon)$ and $f(2\varepsilon)$ as polynomials in terms of y_0 and y_2 only. We regard this as a system of two equations in y_0 and y_2, which can be solved using Newton's method in time $O(\mathsf{M}(n))$.

IX. Discussion

It remains to develop production-ready implementations of elementary functions using the techniques we have discussed. In particular, we have not yet implemented all tricks from sections VII and VIII, nor SIMD acceleration. An interesting question for future study is whether FFT techniques can yield practical improvements even at reasonably low precision, e.g. thousands of bits as opposed to millions of bits.

We note that all algorithms presented for the exponential function have direct analogs for other elementary functions: for example, we can compute trigonometric functions by writing $\exp(ix) = \cos(x) + i\sin(x)$ and using arctangents instead of logarithms and Gaussian integers instead of integers. Generalizing further, an interesting question is whether similar techniques work for holonomic functions.

References

[Bab86] László Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6:1–13, 1986.

[BEIR00] JC Bajard, M Ercegovac, L Imbert, and F Rico. Fast evaluation of elementary functions with combined shift-and-add and polynomial methods. In *4th Conference on Real Numbers and Computers*, pages 75–87, 2000.

[BKM94] J.-C. Bajard, S. Kla, and J.-M. Muller. BKM: a new hardware algorithm for complex elementary functions. *IEEE Transactions on Computers*, 43(8):955–963, 1994.

[Bre76a] R. P. Brent. The complexity of multiple-precision arithmetic. *The Complexity of Comp. Problem Solving*, pages 126–165, 1976.

[Bre76b] Richard P. Brent. Fast multiple-precision evaluation of elementary functions. *Journal of the ACM*, 23(2):242–251, April 1976.

[BZ11] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2011.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007.

[HH21] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics*, 193(2):563, 2021.

[Hoe10] Joris van der Hoeven. Newton's method and FFT trading. *Journal of Symbolic Computation*, 45(8):857–878, 2010.

[Hoe16] Joris van der Hoeven. Faster Chinese remaindering. https://hal.archives-ouvertes.fr/hal-01403810, 2016.

[Joh15] F. Johansson. Efficient implementation of elementary functions in the medium-precision range. In *22nd IEEE Symposium on Computer Arithmetic*, ARITH22, pages 83–89, 2015.

[Joh17] Fredrik Johansson. Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, 66(8):1281–1292, August 2017.

[Joh22] Fredrik Johansson. Computing elementary functions using multi-prime argument reduction. *arXiv:2207.02501*, 2022.

[LLL82] Arjen K Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische annalen*, 261:515–534, 1982.

[Mul16] Jean-Michel Muller. *Elementary Functions: Algorithms and Implementation*. Birkhäuser, 3rd edition, 2016.

[Sal76] Eugene Salamin. Computation of π using arithmetic-geometric mean. *Mathematics of Computation*, 30(135):565, July 1976.

[Sch06] Arnold Schönhage. Fast algorithms for computing exp, ln, sin, cos at medium precision. In Thomas Lickteig, Klaus Meer, and Luis Miguel Pardo, editors, *04061 Abstracts Collection - Real Computation and Complexity*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2006. https://drops.dagstuhl.de/opus/volltexte/2006/458/.

[Smi89] D. M. Smith. Efficient multiple-precision evaluation of elementary functions. *Mathematics of Computation*, 52:131–134, 1989.

[Vol59] Jack E Volder. The CORDIC trigonometric computing technique. *IRE Transactions on electronic computers*, (3):330–334, 1959.