# Fast multiprecision evaluation of series of rational numbers

Bruno Haible
ILOG
9, rue de Verdun
94253 Gentilly Cedex
haible@ilog.fr

Thomas Papanikolaou
TH Darmstadt / Universität des Saarlandes
Postach 151150
66041 Saarbrücken
papanik@cs.uni-sb.de

April 4, 1997

**Abstract**

We describe two techniques for fast multiple-precision evaluation of linearly convergent series, including power series and Ramanujan series. The computation time for $N$ bits is $O((\log N)^2 M(N))$, where $M(N)$ is the time needed to multiply two $N$-bit numbers. Applications include fast algorithms for elementary functions, $\pi$, hypergeometric functions at rational points, Euler's, Catalan's and Apéry's constant. The algorithms are suitable for parallel computation.

## 1 Introduction

Multiple-precision evaluation of real numbers has become efficiently possible since Schönhage and Strassen [11] have showed that the bit complexity of the multiplication of two $N$-bit numbers is $M(N) = O(N \log N \log\log N)$. This is not only a theoretical result; a C++ implementation [7] can exploit this already for $N = 40000$ bits. Algorithms for computing elementary functions (exp, log, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, arsinh, arcosh, artanh) have appeared in [4], and a remarkable algorithm for $\pi$ was found by Brent and Salamin [10].

However, all these algorithms suffer from the fact that calculated results are not reusable, since the computation is done using real arithmetic (using exact rational arithmetic would be extremely inefficient). Therefore functions or constants have to be recomputed from the scratch every time higher precision is required.

In this note, we present algorithms for fast computation of sums of the form

$$S = \sum_{n=0}^{\infty} R(n) F(0) \cdots F(n)$$

where $R(n)$ and $F(n)$ are rational functions in $n$ with rational coefficients, provided that this sum is linearly convergent, i.e. that the $n$-th term is $O(c^{-n})$ with $c > 1$. Examples include elementary and hypergeometric functions at rational points in the *interior* of the circle of convergence, as well as $\pi$ and Euler's, Catalan's and Apéry's constants.

The presented algorithms are *easy to implement* and *extremely efficient*, since they take advantage of pure integer arithmetic. The calculated results are *exact*, making *checkpointing* and *reuse* of computations possible. Finally, the computation of our algorithms *can be easily parallelised.*

## 2 Evaluation of linearly convergent series

The technique presented here applies to all linearly convergent sums of the form

$$S = \sum_{n=0}^{\infty} \frac{a(n)}{b(n)} \frac{p(0) \cdots p(n)}{q(0) \cdots q(n)}$$

where $a(n)$, $b(n)$, $p(n)$, $q(n)$ are integers with $O(\log n)$ bits. The most often used case is that $a(n)$, $b(n)$, $p(n)$, $q(n)$ are polynomials in $n$ with integer coefficients.

**Algorithm:**

Given two index bounds $n_1$ and $n_2$, consider the partial sum

$$S = \sum_{n_1 \le n < n_2} \frac{a(n)}{b(n)} \frac{p(n_1) \cdots p(n)}{q(n_1) \cdots q(n)}$$

It is not computed directly. Instead, we compute the integers $P = p(n_1) \cdots p(n_2 - 1)$, $Q = q(n_1) \cdots q(n_2 - 1)$, $B = b(n_1) \cdots b(n_2 - 1)$ and $T = BQS$. If $n_2 - n_1 < 5$, these are computed directly. If $n_2 - n_1 \ge 5$, they are computed using *binary splitting*: Choose an index $n_m$ in the middle of $n_1$ and $n_2$, compute the components $P_l$, $Q_l$, $B_l$, $T_l$ belonging to the interval $n_1 \le n < n_m$, compute the components $P_r$, $Q_r$, $B_r$, $T_r$ belonging to the interval $n_m \le n < n_2$, and set $P = P_l P_r$, $Q = Q_l Q_r$, $B = B_l B_r$ and $T = B_r Q_r T_l + B_l P_l T_r$.

Finally, this algorithm is applied to $n_1 = 0$ and $n_2 = n_{\max} = O(N)$, and a final floating-point division $S = \frac{T}{BQ}$ is performed.

**Complexity:**

The bit complexity of computing $S$ with $N$ bits of precision is $O((\log N)^2 M(N))$.

**Proof:**

Since we have assumed the series to be linearly convergent, the $n$-th term is $O(c^{-n})$ with $c > 1$. Hence choosing $n_{\max} = N \frac{\log 2}{\log c} + O(1)$ will ensure that the round-off error is $< 2^{-N}$. By our assumption that $a(n)$, $b(n)$, $p(n)$, $q(n)$ are integers with $O(\log n)$ bits, the integers $P$, $Q$, $B$, $T$ belonging to the interval $n_1 \le n < n_2$ all have $O((n_2 - n_1) \log n_2)$ bits.

The algorithm's recursion depth is $d = \frac{\log n_{\max}}{\log 2} + O(1)$. At recursion depth $k$ $(1 \le k \le d)$, integers having each $O(\frac{n_{\max}}{2^k} \log n_{\max})$ bits are multiplied. Thus, the entire computation time $t$ is

$$
\begin{aligned}
t &= \sum_{k=1}^{d} 2^{k-1} O\left(M\left(\frac{n_{\max}}{2^k}\log n_{\max}\right)\right) \\
&= \sum_{k=1}^{d} O\left(M\left(n_{\max}\log n_{\max}\right)\right) \\
&= O(\log n_{\max} M(n_{\max}\log n_{\max}))
\end{aligned}
$$

Because of $n_{\max} = O(N)$ and

$$M(N\log N) = O(N\,(\log N)^2\,\log\log N) = O(\log N\,M(N))$$

we have the desired result.

**Checkpointing/Parallelising:**

A checkpoint can be easily done by storing the (integer) values of $n_1$, $n_2$, $P$, $Q$, $B$ and $T$. Similarly, if $m$ processors are available, then the interval $[0, n_{max}]$ can be divided into $m$ pieces of length $l = \lfloor n_{max}/m \rfloor$. After each processor $i$ has computed the sum of its interval $[il, (i+1)l]$, the partial sums are combined to the final result using the rules described above.

**Note:**

For the special case $a(n) = b(n) = 1$, the binary splitting algorithm has already been documented in [3], section 6, and [2], section 10.2.3.

Explicit computation of $P$, $Q$, $B$, $T$ is only required as a recursion base, for $n_2 - n_1 < 2$, but avoiding recursions for $n_2 - n_1 < 5$ gains some percent of execution speed.

The binary splitting algorithm is asymptotically faster than step-by-step evaluation of the sum – which has binary complexity $O(N^2)$ – because it pushes as much multiplication work as possible to the region where multiplication becomes efficient. If the multiplication were implemented as an $M(N) = O(N^2)$ algorithm, the binary splitting algorithm would provide no speedup over step-by-step evaluation.

**Implementation:**

In the following we present a simplified C++ implementation of the above algorithm[1]. The initialisation is done by a structure `abpq_series` containing arrays `a`, `b`, `p` and `q` of multiprecision integers (`bigints`). The values of the arrays at the index $n$ correspond to the values of the functions $a$, $b$, $p$ and $q$ at the integer point $n$. The (partial) results of the algorithm are stored in the `abpq_series_result` structure.

[1] A complete implementation can be found in CLN [7]. The implementation of the binary-splitting method will be also available in LiDIA-1.4

```
// abpq_series is initialised by user
struct { bigint *a, *b, *p, *q;
       } abpq_series;

// abpq_series_result holds the partial results
struct { bigint P, Q, B, T;
       } abpq_series_result;

// binary splitting summation for abpq_series
void sum_abpq(abpq_series_result & r,
              int n1, int n2,
              const abpq_series & arg)
{
  // check the length of the summation interval
  switch (n2 - n1)
  {
    case 0:
      error_handler("summation device",
               "sum_abpq:: n2-n1 should be > 0.");
      break;

    case 1: // the result at the point n1
      r.P = arg.p[n1];
      r.Q = arg.q[n1];
      r.B = arg.b[n1];
      r.T = arg.a[n1] * arg.p[n1];
      break;

    // cases 2, 3, 4 left out for simplicity

    default: // the general case

      // the left and the right partial sum
      abpq_series_result L, R;

      // find the middle of the interval
      int nm = (n1 + n2) / 2;

      // sum left side
      sum_abpq(L, n1, nm, arg);

      // sum right side
      sum_abpq(R, nm, n2, arg);
```

```
      // put together
      r.P = L.P * R.P;
      r.Q = L.Q * R.Q;
      r.B = L.B * R.B;
      r.T = R.B * R.Q * L.T + L.B * L.P * R.T;
      break;
  }
}
```

Note that the multiprecision integers could be replaced here by integer polynomials, or by any other ring providing the operators = (assignment), + (addition) and $*$ (multiplication). For example, one could regard a bivariate polynomial over the integers as a series over the second variable, with polynomials over the first variable as its coefficients. This would result an accelerated algorithm for summing bivariate (and thus multivariate) polynomials.

## 2.1 Example: The factorial

This is the most classical example of the binary splitting algorithm and was probably known long before [2].

Computation of the factorial is best done using the binary splitting algorithm, combined with a reduction of the even factors into odd factors and multiplication with a power of 2, according to the formula

$$n! = 2^{n-\sigma_2(n)} \cdot \prod_{k\geq 1} \left( \prod_{\frac{n}{2^k} < 2m+1 \leq \frac{n}{2^{k-1}}} (2m+1) \right)^k$$

and where the products

$$P(n_1, n_2) = \prod_{n_1 < m \leq n_2} (2m+1)$$

are evaluated according to the binary splitting algorithm: $P(n_1, n_2) = P(n_1, n_m)P(n_m, n_2)$ with $n_m = \lfloor \frac{n_1+n_2}{2} \rfloor$ if $n_2 - n_1 \geq 5$.

## 2.2 Example: Elementary functions at rational points

The binary splitting algorithm can be applied to the fast computation of the elementary functions at rational points $x = \frac{u}{v}$, simply by using the power series. We present how this can be done for $\exp(x)$, $\ln(x)$, $\sin(x)$, $\cos(x)$, $\arctan(x)$, $\sinh(x)$ and $\cosh(x)$. The calculation of other elementary functions is similar (or it can be reduced to the calculation of these functions).

### 2.2.1 $\exp(x)$ for rational $x$

This is a direct application of the above algorithm with $a(n) = 1$, $b(n) = 1$, $p(0) = q(0) = 1$, and $p(n) = u$, $q(n) = nv$ for $n > 0$. Because the series is not only linearly convergent – $\exp(x)$ is an entire function –, $n_{\max} = O(\frac{N}{\log N})$, hence the bit complexity is $O(\log N\, M(N))$.

### 2.2.2 $\exp(x)$ for real $x$

This can be computed using the addition theorem for exp, by a trick due to Brent [3] (see also [2], section 10.2, exercise 8). Write

$$x = x_0 + \sum_{k=0}^{\infty} \frac{u_k}{v_k}$$

with $x_0$ integer, $v_k = 2^{2^k}$ and $|u_k| < 2^{2^{k-1}}$, and compute

$$\exp(x) = \exp(x_0) \cdot \prod_{k \geq 0} \exp\left(\frac{u_k}{v_k}\right)$$

This algorithm has bit complexity $O((\log N)^2 M(N))$.

### 2.2.3 $\ln(x)$ for rational $x$

For rational $|x-1| < 1$, the binary splitting algorithm can also be applied directly to the power series for $\ln(x)$. Write $x - 1 = \frac{u}{v}$ and compute the series with $a(n) = 1$, $b(n) = n+1$, $q(n) = v$, $p(0) = u$, and $p(n) = -u$ for $n > 0$.

This algorithm has bit complexity $O((\log N)^2 M(N))$.

### 2.2.4 $\ln(x)$ for real $x$

This can be computed using the “inverse” Brent trick:

Start with $y := 0$.

As long as $x \neq 1$ within the actual precision, choose $k$ maximal with $|x-1| < 2^{-k}$. Put $z = 2^{-2k}\left[2^{2k}(x-1)\right]$, i.e. let $z$ contain the first $k$ significant bits of $x-1$. $z$ is a good approximation for $\ln(x)$. Set $y := y + z$ and $x := x \cdot \exp(-z)$.

Since $x \cdot \exp(y)$ is an invariant of the algorithm, the final $y$ is the desired value $\ln(x)$.

This algorithm has bit complexity $O((\log N)^2 M(N))$.

### 2.2.5 $\sin(x)$, $\cos(x)$ for rational $x$

These are direct applications of the binary splitting algorithm: For $\sin(x)$, put $a(n) = 1$, $b(n) = 1$, $p(0) = u$, $q(0) = v$, and $p(n) = -u^2$, $q(n) = (2n)(2n+1)v^2$ for $n > 0$. For $\cos(x)$, put $a(n) = 1$, $b(n) = 1$, $p(0) = 1$, $q(0) = 1$, and $p(n) = -u^2$, $q(n) = (2n-1)(2n)v^2$ for $n > 0$. Of course, when both $\sin(x)$ and $\cos(x)$ are needed, one should only compute $\sin(x)$ this way, and then set $\cos(x) = \pm\sqrt{1 - \sin(x)^2}$. This is a 20% speedup at least.

The bit complexity of these algorithms is $O(\log N \; M(N))$.

### 2.2.6 $\sin(x)$, $\cos(x)$ for real $x$

To compute $\cos(x) + i\sin(x) = \exp(ix)$ for real $x$, again the addition theorems and Brent's trick can be used. The resulting algorithm has bit complexity $O((\log N)^2 M(N))$.

#### 2.2.7 $\arctan(x)$ for rational $x$

For rational $|x| < 1$, the fastest way to compute $\arctan(x)$ with bit complexity $O((\log N)^2 M(N))$ is to apply the binary splitting algorithm directly to the power series for $\arctan(x)$. Put $a(n) = 1$, $b(n) = 2n + 1$, $q(n) = 1$, $p(0) = x$ and $p(n) = -x^2$ for $n > 0$.

#### 2.2.8 $\arctan(x)$ for real $x$

This again can be computed using the "inverse" Brent trick:

Start out with $z := \frac{1}{\sqrt{1+x^2}} + i\frac{x}{\sqrt{1+x^2}}$ and $\varphi := 0$. During the algorithm $z$ will be a complex number with $|z| = 1$ and $\mathrm{Re}(z) > 0$.

As long as $\mathrm{Im}(z) \neq 0$ within the actual precision, choose $k$ maximal with $|\mathrm{Im}(z)| < 2^{-k}$. Put $\alpha = 2^{-2k} \left[2^{2k} \mathrm{Im}(z)\right]$, i.e. let $\alpha$ contain the first $k$ significant bits of $\mathrm{Im}(z)$. $\alpha$ is a good approximation for $\arcsin(\mathrm{Im}(z))$. Set $\varphi := \varphi + \alpha$ and $z := z \cdot \exp(-i\alpha)$.

Since $z \cdot \exp(i\varphi)$ is an invariant of the algorithm, the final $\varphi$ is the desired value $\arcsin \frac{x}{\sqrt{1+x^2}}$.

This algorithm has bit complexity $O((\log N)^2 M(N))$.

#### 2.2.9 $\sinh(x)$, $\cosh(x)$ for rational and real $x$

These can be computed by similar algorithms as $\sin(x)$ and $\cos(x)$ above, with the same asymptotic bit complexity. The standard computation, using $\exp(x)$ and its reciprocal (calculated by the Newton method) results also to the same complexity and works equally well in practice.

The bit complexity of these algorithms is $O(\log N\, M(N))$ for rational $x$ and $O((\log N)^2 M(N))$ for real $x$.

### 2.3 Example: Hypergeometric functions at rational points

The binary splitting algorithm is well suited for the evaluation of a hypergeometric series

$$F\left(\begin{array}{ccc} a_1, & \ldots, & a_r \\ b_1, & \ldots, & b_s \end{array} \middle| x\right) = \sum_{n=0}^{\infty} \frac{a_1^{\overline{n}} \cdots a_r^{\overline{n}}}{b_1^{\overline{n}} \cdots b_s^{\overline{n}}} x^n$$

with rational coefficients $a_1$, ..., $a_r$, $b_1$, ..., $b_s$ at a rational point $x$ in the interior of the circle of convergence. Just put $a(n) = 1$, $b(n) = 1$, $p(0) = q(0) = 1$, and $\frac{p(n)}{q(n)} = \frac{(a_1+n-1)\cdots(a_r+n-1)x}{(b_1+n-1)\cdots(b_s+n-1)}$ for $n > 0$. The evaluation can thus be done with bit complexity $O((\log N)^2 M(N))$ for $r = s$ and $O(\log N\, M(N))$ for $r < s$.

### 2.4 Example: $\pi$

The Ramanujan series for $\pi$

$$\frac{1}{\pi} = \frac{12}{C^{3/2}} \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!(A + nB)}{(3n)! n!^3 C^{3n}}$$

with $A = 13591409$, $B = 545140134$, $C = 640320$ found by the Chudnovsky's [2] and which is used by the LiDIA [8] and the Pari [6] system to compute $\pi$, is usually written as an algorithm of bit complexity $O(N^2)$. It is, however, possible to apply binary splitting to the sum. Put $a(n) = A+nB$, $b(n) = 1$, $p(0) = 1$, $q(0) = 1$, and $p(n) = -(6n-5)(2n-1)(6n-1)$, $q(n) = n^3C^3/24$ for $n > 0$. This reduces the complexity to $O((\log N)^2 M(N))$. Although this is theoretically slower than Brent-Salamin's quadratically convergent iteration, which has a bit complexity of $O(\log N\, M(N))$, in practice the binary splitted Ramanujan sum is three times faster than Brent-Salamin, at least in the range from $N = 1000$ bits to $N = 1000000$ bits.

### 2.5 Example: Catalan's constant $G$

A linearly convergent sum for Catalan's constant

$$G := \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)^2}$$

is given in [2], p. 386:

$$G = \frac{3}{8} \sum_{n=0}^{\infty} \frac{1}{\binom{2n}{n}(2n+1)^2} + \frac{\pi}{8}\log(2+\sqrt{3})$$

The series is summed using binary splitting, putting $a(n) = 1$, $b(n) = 2n+1$, $p(0) = 1$, $q(0) = 1$, and $p(n) = n$, $q(n) = 2(2n+1)$ for $n > 0$. Thus $G$ can be computed with bit complexity $O((\log N)^2 M(N))$.

### 2.6 Example: The Gamma function at rational points

For evaluating $\Gamma(s)$ for rational $s$, we first reduce $s$ to the range $1 \le s \le 2$ by the formula $\Gamma(s+1) = s\Gamma(s)$. To compute $\Gamma(s)$ with a precision of $N$ bits, choose a positive integer $x$ with $xe^{-x} < 2^{-N}$. Partial integration lets us write

$$\begin{aligned}
\Gamma(s) &= \int_0^{\infty} e^{-t} t^{s-1} dt \\
&= x^s e^{-x} \sum_{n=0}^{\infty} \frac{x^n}{s(s+1)\cdots(s+n)} + \int_x^{\infty} e^{-t} t^{s-1} dt
\end{aligned}$$

The last integral is $< xe^{-x} < 2^{-N}$. The series is evaluated as a hypergeometric function (see above); the number of terms to be summed up is $O(N)$, since $x = O(N)$. Thus the entire computation can be done with bit complexity $O((\log N)^2 M(N))$.

Note: This result is already mentioned in [4].

[2] A special case of [2], formula (5.5.18), with N=163.

## 2.7 Example: The Riemann Zeta value $\zeta(3)$

Recently, Doron Zeilberger's method of "creative telescoping" has been applied to Riemann's zeta function at $s = 3$ (see [1]), which is also known as *Apéry's constant*:

$$\zeta(3) = \frac{1}{2} \sum_{n=1}^{\infty} \frac{(-1)^{n-1}(205n^2 - 160n + 32)}{n^5 \binom{2n}{n}^5}$$

This sum consists of three hypergeometric series. Binary splitting can also be applied directly, by putting $a(0) = 77$, $a(n) = 205n^2 - 160n + 32$ for $n > 0$, $b(n) = 1$, $p(0) = 1$, $p(n) = -n^5$ for $n > 0$, and $q(n) = 32(2n+1)^5$. Thus the bit complexity of computing $\zeta(3)$ is $O((\log N)^2 M(N))$.

**Note:**

Using this the authors were able to establish a new record in the calculation of $\zeta(3)$ by computing 1,000,000 decimals [9]. The computation took 8 hours on a Hewlett Packard 9000/712 machine. After distributing on a cluster of 4 HP 9000/712 machines the same computation required only 2.5 hours. The half hour was necessary for reading the partial results from disk and for recombining them. Again, we have used binary-splitting for recombining: the 4 partial result produced 2 results which were combined to the final 1,000,000 decimals value of $\zeta(3)$.

This example shows the importance of checkpointing. Even if a machine crashes through the calculation, the results of the other machines are still usable. Additionally, being able to parallelise the computation reduced the computing time dramatically.

# 3 Evaluation of linearly convergent series of sums

The technique presented in the previous section also applies to all linearly convergent sums of the form

$$U = \sum_{n=0}^{\infty} \frac{a(n)}{b(n)} \left( \frac{c(0)}{d(0)} + \cdots + \frac{c(n)}{d(n)} \right) \frac{p(0) \cdots p(n)}{q(0) \cdots q(n)}$$

where $a(n)$, $b(n)$, $c(n)$, $d(n)$, $p(n)$, $q(n)$ are integers with $O(\log n)$ bits. The most often used case is again that $a(n)$, $b(n)$, $c(n)$, $d(n)$, $p(n)$, $q(n)$ are polynomials in $n$ with integer coefficients.

**Algorithm:**

Given two index bounds $n_1$and $n_2$, consider the partial sums

$$S = \sum_{n_1 \le n < n_2} \frac{a(n)}{b(n)} \frac{p(n_1) \cdots p(n)}{q(n_1) \cdots q(n)}$$

and

$$U = \sum_{n_1 \le n < n_2} \frac{a(n)}{b(n)} \left( \frac{c(n_1)}{d(n_1)} + \cdots + \frac{c(n)}{d(n)} \right) \frac{p(n_1) \cdots p(n)}{q(n_1) \cdots q(n)}$$

As above, we compute the integers $P = p(n_1)\cdots p(n_2-1)$, $Q = q(n_1)\cdots q(n_2-1)$, $B = b(n_1)\cdots b(n_2-1)$, $T = BQS$, $D = d(n_1)\cdots d(n_2-1)$, $C = D\left(\frac{c(n_1)}{d(n_1)} + \cdots + \frac{c(n_2-1)}{d(n_2-1)}\right)$ and $V = DBQU$. If $n_2 - n_1 < 4$, these are computed directly. If $n_2 - n_1 \geq 4$, they are computed using *binary splitting*: Choose an index $n_m$ in the middle of $n_1$and $n_2$, compute the components $P_l$, $Q_l$, $B_l$, $T_l$, $D_l$, $C_l$, $V_l$ belonging to the interval $n_1 \leq n < n_m$, compute the components $P_r$, $Q_r$, $B_r$, $T_r$, $D_r$, $C_r$, $V_r$ belonging to the interval $n_m \leq n < n_2$, and set $P = P_l P_r$, $Q = Q_l Q_r$, $B = B_l B_r$, $T = B_r Q_r T_l + B_l P_l T_r$, $D = D_l D_r$, $C = C_l D_r + C_r D_l$ and $V = D_r B_r Q_r V_l + D_r C_l B_l P_l T_r + D_l B_l P_l V_r$.

Finally, this algorithm is applied to $n_1 = 0$ and $n_2 = n_{\max} = O(N)$, and final floating-point divisions $S = \frac{T}{BQ}$ and $U = \frac{V}{DBQ}$ are performed.

**Complexity:**

The bit complexity of computing $S$ and $U$ with $N$ bits of precision is $O((\log N)^2 M(N))$.

**Proof:**

By our assumption that $a(n)$, $b(n)$, $c(n)$, $d(n)$, $p(n)$, $q(n)$ are integers with $O(\log n)$ bits, the integers $P$, $Q$, $B$, $T$, $D$, $C$, $V$ belonging to the interval $n_1 \leq n < n_2$ all have $O((n_2 - n_1)\log n_2)$ bits. The rest of the proof is as in the previous section.

**Checkpointing/Parallelising:**

A checkpoint can be easily done by storing the (integer) values of $n_1$, $n_2$, $P$, $Q$, $B$, $T$ and additionally $D$, $C$, $V$. Similarly, if $m$ processors are available, then the interval $[0, n_{max}]$ can be divided into $m$ pieces of length $l = \lfloor n_{max}/m \rfloor$. After each processor $i$ has computed the sum of its interval $[il, (i+1)l]$, the partial sums are combined to the final result using the rules described above.

**Implementation:**

The C++ implementation of the above algorithm is very similar to the previous one. The initialisation is done now by a structure `abpqcd_series` containing arrays `a`, `b`, `p`, `q`, `c` and `d` of multiprecision integers. The values of the arrays at the index $n$ correspond to the values of the functions $a$, $b$, $p$, $q$, $c$ and $d$ at the integer point $n$. The (partial) results of the algorithm are stored in the `abpqcd_series_result` structure, which now contains 3 new elements (`C`, `D` and `V`).

```
// abpq_series is initialised by user
struct { bigint *a, *b, *p, *q, *c, *d;
       } abpqcd_series;

// abpq_series_result holds the partial results
struct { bigint P, Q, B, T, C, D, V;
       } abpqcd_series_result;

void sum_abpqcd(abpqcd_series_result & r,
```

```
                int n1, int n2,
                const abpqcd_series & arg)
{
  switch (n2 - n1)
  {
    case 0:
      error_handler("summation device",
            "sum_abpqcd:: n2-n1 should be > 0.");
      break;

    case 1: // the result at the point n1
      r.P = arg.p[n1];
      r.Q = arg.q[n1];
      r.B = arg.b[n1];
      r.T = arg.a[n1] * arg.p[n1];
      r.D = arg.d[n1];
      r.C = arg.c[n1];
      r.V = arg.a[n1] * arg.c[n1] * arg.p[n1];
      break;

    // cases 2, 3, 4 left out for simplicity

    default: // general case

      // the left and the right partial sum
      abpqcd_series_result L, R;

      // find the middle of the interval
      int nm = (n1 + n2) / 2;

      // sum left side
      sum_abpqcd(L, n1, nm, arg);

      // sum right side
      sum_abpqcd(R, nm, n2, arg);

      // put together
      r.P = L.P * R.P;
      r.Q = R.Q * L.Q;
      r.B = L.B * R.B;
      bigint tmp = L.B * L.P * R.T;
      r.T = R.B * R.Q * L.T + tmp;
      r.D = L.D * R.D;
      r.C = L.C * R.D + R.C * L.D;
```

```
      r.V = R.D * (R.B * R.Q * L.V + L.C * tmp)
            + L.D * L.B * L.P * R.V;
      break;
  }
}
```

### 3.1 Example: Euler's constant $C$

**Theorem:**

Let $f(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!^2}$ and $g(x) = \sum_{n=0}^{\infty} H_n \frac{x^n}{n!^2}$. Then for $x \to \infty$, $\frac{g(x)}{f(x)} = \frac{1}{2}\log x + C + O\left(e^{-4\sqrt{x}}\right)$.

**Proof:**

The Laplace method for asymptotic evaluation of exponentially growing sums and integrals yields

$$f(x) = e^{2\sqrt{x}} x^{-\frac{1}{4}} \frac{1}{2\sqrt{\pi}}(1 + O(x^{-\frac{1}{4}}))$$

and

$$g(x) = e^{2\sqrt{x}} x^{-\frac{1}{4}} \frac{1}{2\sqrt{\pi}} \left(\frac{1}{2}\log x + C + O(\log x \cdot x^{-\frac{1}{4}})\right)$$

On the other hand, $h(x) := \frac{g(x)}{f(x)}$ satisfies the differential equation

$$xf(x) \cdot h''(x) + (2xf'(x) + f(x)) \cdot h'(x) = f'(x)$$

hence

$$h(x) = \frac{1}{2}\log x + C + c_2 \int_x^{\infty} \frac{1}{tf(t)^2} dt = \frac{1}{2}\log x + C + O(e^{-4\sqrt{x}})$$

**Algorithm:**

To compute $C$ with a precision of $N$ bits, set $x = \left\lceil N \frac{\log 2}{4} \right\rceil^2$, and evaluate the series for $g(x)$ and $f(x)$ simultaneously, using the binary-splitting algorithm, with $a(n) = 1$, $b(n) = 1$, $c(n) = 1$, $d(n) = n + 1$, $p(n) = x$, $q(n) = (n + 1)^2$. Let $\alpha = 3.591121477\ldots$ be the solution of the equation $-\alpha \log \alpha + \alpha + 1 = 0$. Then $\alpha\sqrt{x} - \frac{1}{4 \log \alpha} \log \sqrt{x} + O(1)$ terms of the series suffice for the relative error to be bounded by $2^{-N}$.

**Complexity:**

The bit complexity of this algorithm is $O((\log N)^2 M(N))$.

**Note:**

This algorithm was first mentioned in [5]. It is by far the fastest known algorithm for computing Euler's constant.

# 4 Computational results

In this section we present some computational results of our CLN and LiDIA implementation of the algorithms presented in this note. We use the official version (1.2.1) and an experimental version (1.4a) of LiDIA. We have taken advantage of LiDIA's ability to replace its kernel (multiprecision arithmetic and memory management) [8], so we were able to use in both cases CLN's fast integer arithmetic routines.

The table in Figure 1 shows the running times for the calculation of $\exp(1)$, $\log(2)$, $\pi$, $C$, $G$ and $\zeta(3)$ to precision 100, 1000, 10000 and 100000 decimal digits. The timings are given in seconds and they denote the *real* time needed, i.e. system and user time. The computation was done on an Intel Pentium with 133Hz and 32MB of RAM.

| D | exp(1) | log(2) | $\pi$ | $C$ | $G$ | $\zeta(3)$ |
|---|---|---|---|---|---|---|
| $10^2$ | 0.0005 | 0.0020 | 0.0014 | 0.0309 | 0.0179 | 0.0027 |
| $10^3$ | 0.0069 | 0.0474 | 0.0141 | 0.8110 | 0.3580 | 0.0696 |
| $10^4$ | 0.2566 | 1.9100 | 0.6750 | 33.190 | 13.370 | 2.5600 |
| $10^5$ | 5.5549 | 45.640 | 17.430 | 784.93 | 340.33 | 72.970 |

Figure 1: LiDIA-1.4a timings of computation of constants using binary-splitting

The second table (Figure 2) summarizes the performance of $exp(x)$ in various Computer Algebra systems[3]. For a fair comparison of the algorithms, both argument and precision are chosen in such a way, that system–specific optimizations (BCD arithmetic in Maple, FFT multiplication in CLN, special exact argument handling in LiDIA) do not work. We use $x = -\sqrt{2}$ and precision $10^{(i/3)}$, with $i$ running from 4 to 15.

MapleV R3 is the slowest system in this comparison. This is probably due to the BCD arithmetic it uses. However, Maple seems to have an asymptotically better multiplication algorithm for numbers having more than 10000 decimals. In this range it outperforms Pari-1.39.03, which is the fastest system in the 0–200 decimals range.

The comparison indicating the strength of binary-splitting is between LiDIA-1.2.1 and CLN itself. Having the same kernel, the only difference is here that LiDIA-1.2.1 uses Brent's $O(\sqrt{n}M(n))$ for $\exp(x)$, whereas CLN changes from Brent's method to a binary-splitting version for large numbers.

As expected in the range of 1000–100000 decimals CLN outperforms LiDIA-1.2.1 by far. The fact that LiDIA-1.2.1 is faster in the range of 200–1000 decimals (also in some trig. functions) is probably due to a better optimized $O(\sqrt{n}M(n))$ method for $\exp(x)$.

[3] We do not list the timings of LiDIA-1.4a since these are comparable to those of CLN.

| D | Maple | Pari | LiDIA-1.2.1 | CLN |
|---|---|---|---|---|
| 21 | 0.00090 | 0.00047 | 0.00191 | 0.00075 |
| 46 | 0.00250 | 0.00065 | 0.00239 | 0.00109 |
| 100 | 0.01000 | 0.00160 | 0.00389 | 0.00239 |
| 215 | 0.03100 | 0.00530 | 0.00750 | 0.00690 |
| 464 | 0.11000 | 0.02500 | 0.02050 | 0.02991 |
| 1000 | 0.4000 | 0.2940 | 0.0704 | 0.0861 |
| 2154 | 1.7190 | 0.8980 | 0.2990 | 0.2527 |
| 4641 | 8.121 | 5.941 | 1.510 | 0.906 |
| 10000 | 39.340 | 39.776 | 7.360 | 4.059 |
| 21544 | 172.499 | 280.207 | 39.900 | 15.010 |
| 46415 | 868.841 | 1972.184 | 129.000 | 39.848 |
| 100000 | 4873.829 | 21369.197 | 437.000 | 106.990 |

Figure 2: Timings of computation of $\exp(-\sqrt{2})$

# 5 Conclusions

Although powerful, the binary splitting method has not been widely used. Especially, no information existed on the applicability of this method.

In this note we presented a generic binary-splitting summation device for evaluating two types of linearly convergent series. From this we derived simple and computationally efficient algorithms for the evaluation of elementary functions and constants. These algorithms work with *exact* objects, making them suitable for use within Computer Algebra systems.

We have shown that the practical performance of our algorithms is superior to current system implementations. In addition to existing methods, our algorithms provide the possibility of checkpointing and parallelising. These features can be useful for huge calculations, such as those done in analytic number theory research.

# 6 Thanks

The authors would like to thank Jörg Arndt, for pointing us to chapter 10 in [2].

# References

[1] Theodor Amdeberhan, and Doron Zeilberger. Acceleration of hypergeometric series via the wz method. To appear in: Electronic Journal of Combinatorics, Wilf Festschrift Volume.

[2] Jonathan M. Borwein, and Peter B. Borwein. *Pi and the AGM.* Wiley, 1987.

[3] Richard P. Brent. The complexity of multiple-precision arithmetic. *Complexity of Computational Problem Solving* (1976).

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

[5] Richard P. Brent, and Edwin M. McMillan. Some new algorithms for high-precision computation of euler's constant. *Mathematics of Computation* **34** (1980), 305–312.

[6] Henri Cohen, C. Batut, D. Bernardi, and M. Olivier. GP/PARI calculator – version 1.39.03. Available from ftp://megrez.math.u-bordeaux.fr, 1995.

[7] Bruno Haible. CLN, a class library for numbers. Available from ftp://ma2s2.mathematik.uni-karlsruhe.de/pub/gnu/cln.tar.z, 1996.

[8] Thomas Papanikolaou, Ingrid Biehl, and Johannes Buchmann. LiDIA: a library for computational number theory. SFB 124 report, Universität des Saarlandes, 1995.

[9] Simon Plouffe. ISC: Inverse Symbolic Calculator. Tables of records of computation, http://www.cecm.sfu.ca/projects/ISC/records2.html.

[10] Eugene Salamin. Computation of $\pi$ using arithmetic-geometric mean. *Mathematics of Computation* **30** (1976), 565–570.

[11] Arnold Schönhage, and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing* **7** (1971), 281–292.