Efficient implementation of elementary functions in the medium-precision range

Fredrik Johansson
(LFANT, INRIA Bordeaux)

22nd IEEE Symposium on Computer Arithmetic (ARITH 22), Lyon, France, June 2015
Elementary functions

**Functions**: exp, log, sin, cos, atan

My goal is to do interval arithmetic with arbitrary precision.

**Input**: floating-point number \( x = a \cdot 2^b \), precision \( p \geq 2 \)

**Output**: \( m, r \) with \( f(x) \in [m-r, m+r] \) and \( r \approx 2^{-p} |f(x)| \)

\[ 2^{27} \]
Elementary functions

**Functions**: exp, log, sin, cos, atan

My goal is to do interval arithmetic with arbitrary precision.
Elementary functions

**Functions**: exp, log, sin, cos, atan

My goal is to do *interval arithmetic* with *arbitrary precision*.

**Input**: floating-point number $x = a \cdot 2^b$, precision $p \geq 2$

**Output**: $m, r$ with $f(x) \in [m - r, m + r]$ and $r \approx 2^{-p}|f(x)|$
Precision ranges

Hardware precision \((n \approx 53 \text{ bits})\)
- Extensively studied - elsewhere!

Medium precision \((100 - 10000 \text{ bits})\)
- Multiplication costs \(M(n) = O(n^2) \text{ or } O(n^{1.6})\)
- Argument reduction + rectangular splitting: \(O(n^{1/3}M(n))\)
- In the lower range, software overhead is significant

Very high precision \((n \gg 10000 \text{ bits})\)
- Multiplication costs \(M(n) = O(n \log n \log \log n)\)
- Asymptotically fast algorithms: binary splitting, arithmetic-geometric mean (AGM) iteration: \(O(M(n) \log(n))\)
Precision ranges

Hardware precision \((n \approx 53 \text{ bits})\)
- Extensively studied - elsewhere!

Medium precision \((n \approx 100-10000 \text{ bits})\)
- Multiplication costs \(M(n) = O(n^2)\) or \(O(n^{1.6})\)
- Argument reduction + rectangular splitting: \(O(n^{1/3}M(n))\)
- In the lower range, software overhead is significant
Precision ranges

Hardware precision \((n \approx 53 \text{ bits})\)
- Extensively studied - elsewhere!

Medium precision \((n \approx 100\text{-}10\,000 \text{ bits})\)
- Multiplication costs \(M(n) = O(n^2)\) or \(O(n^{1.6})\)
- Argument reduction + rectangular splitting: \(O(n^{1/3}M(n))\)
- In the lower range, software overhead is significant

Very high precision \((n \gg 10\,000 \text{ bits})\)
- Multiplication costs \(M(n) = O(n \log n \log \log n)\)
- Asymptotically fast algorithms: binary splitting, arithmetic-geometric mean (AGM) iteration: \(O(M(n) \log(n))\)
Improvements in this work

1. The **low-level** `mpn` layer of GMP is used exclusively
   - Most libraries (e.g. MPFR) use more convenient types, e.g. `mpz` and `mpfr`, to implement elementary functions
Improvements in this work

1. The **low-level** *mpn* layer of GMP is used exclusively
   ▶ Most libraries (e.g. MPFR) use more convenient types, e.g. *mpz* and *mpfr*, to implement elementary functions

2. Argument reduction based on **lookup tables**
   ▶ Old idea, not well explored in high precision
Improvements in this work

1. The **low-level** `mpn` layer of GMP is used exclusively
   - Most libraries (e.g. MPFR) use more convenient types, e.g. `mpz` and `mpfr`, to implement elementary functions

2. Argument reduction based on **lookup tables**
   - Old idea, not well explored in high precision

3. Faster **evaluation of Taylor series**
   - Optimized version of Smith’s rectangular splitting algorithm
   - Takes advantage of `mpn` level functions
Recipe for elementary functions

\[
\begin{align*}
\exp(x) & \quad \sin(x), \cos(x) & \quad \log(1 + x) & \quad \text{atan}(x) \\
\downarrow & & & \\
\text{Domain reduction using } \pi \text{ and } \log(2) \\
\downarrow & & & \\
x & \in [0, \log(2)) & x & \in [0, \pi/4) & x & \in [0, 1) & x & \in [0, 1)
\end{align*}
\]
Recipe for elementary functions

\[
\begin{align*}
\text{exp}(x) & \quad \sin(x), \cos(x) & \log(1 + x) & \atan(x) \\
\text{Domain reduction using } \pi \text{ and } \log(2) \\
x & \in [0, \log(2)) & x & \in [0, \pi/4) & x & \in [0, 1) & x & \in [0, 1) \\
\text{Argument-halving } r \approx 8 \text{ times} \\
\text{exp}(x) & = [\text{exp}(x/2)]^2 \\
\log(1 + x) & = 2 \log(\sqrt{1 + x}) \\
x & \in [0, 2^{-r}) \\
\text{Taylor series}
\end{align*}
\]
Better recipe at medium precision

exp(x)  sin(x), cos(x)  log(1 + x)  atan(x)

↓

Domain reduction using $\pi$ and $\log(2)$

$\downarrow$

$x \in [0, \log(2))$  $x \in [0, \pi/4)$  $x \in [0, 1)$  $x \in [0, 1)$

↓

Lookup table with $2^r \approx 2^8$ entries

$\downarrow$

$\exp(t + x) = \exp(t) \exp(x)$

$\log(1 + t + x) = \log(1 + t) + \log(1 + x/(1 + t))$

↓

$x \in [0, 2^{-r})$

↓

Taylor series
Argument reduction formulas

What we want to compute: \( f(x) \), \( x \in [0, 1) \)

Table size: \( q = 2^r \)

Precomputed value: \( f(t) \), \( t = i/q, \ i = \lfloor 2^r x \rfloor \)

Remaining value to compute: \( f(y) \), \( y \in [0, 2^{-r}) \)
Argument reduction formulas

What we want to compute: $f(x)$, $x \in [0, 1)$
Table size: $q = 2^r$
Precomputed value: $f(t)$, $t = i/q$, $i = \lfloor 2^r x \rfloor$
Remaining value to compute: $f(y)$, $y \in [0, 2^{-r})$

\[
\begin{align*}
\exp(x) &= \exp(t) \exp(y), \quad y = x - i/q \\
\sin(x) &= \sin(t) \cos(y) + \cos(t) \sin(y), \quad y = x - i/q \\
\cos(x) &= \cos(t) \cos(y) - \sin(t) \sin(y), \quad y = x - i/q
\end{align*}
\]
Argument reduction formulas

What we want to compute: $f(x), \ x \in [0, 1)$
Table size: $q = 2^r$
Precomputed value: $f(t), \ t = i/q, \ i = \lfloor 2^r x \rfloor$
Remaining value to compute: $f(y), \ y \in [0, 2^{-r})$

$$\exp(x) = \exp(t) \exp(y), \ y = x - i/q$$
$$\sin(x) = \sin(t) \cos(y) + \cos(t) \sin(y), \ y = x - i/q$$
$$\cos(x) = \cos(t) \cos(y) - \sin(t) \sin(y), \ y = x - i/q$$

$$\log(1 + x) = \log(1 + t) + \log(1 + y), \ y = (qx - i)/(i + q)$$
$$\tan(x) = \tan(t) + \tan(y), \ y = (qx - i)/(ix + q)$$
**Optimizing lookup tables**

$m = 2$ tables with $2^5 + 2^5$ entries gives same reduction as
$m = 1$ table with $2^{10}$ entries

<table>
<thead>
<tr>
<th>Function</th>
<th>Precision</th>
<th>$m$</th>
<th>$r$</th>
<th>Entries</th>
<th>Size (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>$\leq 512$</td>
<td>1</td>
<td>8</td>
<td>178</td>
<td>11.125</td>
</tr>
<tr>
<td>exp</td>
<td>$\leq 4608$</td>
<td>2</td>
<td>5</td>
<td>$23+32$</td>
<td>30.9375</td>
</tr>
<tr>
<td>sin</td>
<td>$\leq 512$</td>
<td>1</td>
<td>8</td>
<td>203</td>
<td>12.6875</td>
</tr>
<tr>
<td>sin</td>
<td>$\leq 4608$</td>
<td>2</td>
<td>5</td>
<td>$26+32$</td>
<td>32.625</td>
</tr>
<tr>
<td>cos</td>
<td>$\leq 512$</td>
<td>1</td>
<td>8</td>
<td>203</td>
<td>12.6875</td>
</tr>
<tr>
<td>cos</td>
<td>$\leq 4608$</td>
<td>2</td>
<td>5</td>
<td>$26+32$</td>
<td>32.625</td>
</tr>
<tr>
<td>log</td>
<td>$\leq 512$</td>
<td>2</td>
<td>7</td>
<td>$128+128$</td>
<td>16</td>
</tr>
<tr>
<td>log</td>
<td>$\leq 4608$</td>
<td>2</td>
<td>5</td>
<td>$32+32$</td>
<td>36</td>
</tr>
<tr>
<td>atan</td>
<td>$\leq 512$</td>
<td>1</td>
<td>8</td>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td>atan</td>
<td>$\leq 4608$</td>
<td>2</td>
<td>5</td>
<td>$32+32$</td>
<td>36</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>236.6875</strong></td>
</tr>
</tbody>
</table>
Taylor series

Logarithmic series:
\[ \text{atan}(x) = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \ldots \]
\[ \log(1 + x) = 2 \text{atanh}(x/(x + 2)) \]
With \( x < 2^{-10} \), need 230 terms for 4600-bit precision
Taylor series

**Logarithmic series:**

\[ \tan^{-1}(x) = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \ldots \]

\[ \log(1 + x) = 2 \tan^{-1}(x/(x + 2)) \]

With \( x < 2^{-10} \), need 230 terms for 4600-bit precision

**Exponential series:**

\[ \exp(x) = 1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \frac{1}{4!}x^4 + \ldots \]

\[ \sin(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \ldots, \quad \cos(x) = 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \ldots \]

With \( x < 2^{-10} \), need 280 terms for 4600-bit precision
Taylor series

**Logarithmic series:**

\[ \text{atan}(x) = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \ldots \]

\[ \log(1 + x) = 2 \text{atanh}(x/(x + 2)) \]

With \( x < 2^{-10} \), need 230 terms for 4600-bit precision

**Exponential series:**

\[ \exp(x) = 1 + x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + \frac{1}{4!}x^4 + \ldots \]

\[ \sin(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \ldots, \quad \cos(x) = 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \ldots \]

With \( x < 2^{-10} \), need 280 terms for 4600-bit precision

Above 300 bits: \( \cos(x) = \sqrt{1 - \sin^2(x)} \)

Above 800 bits: \[ \exp(x) = \sinh(x) + \sqrt{1 + \sinh^2(x)} \]
Evaluating Taylor series using rectangular splitting

Paterson and Stockmeyer, 1973:
\[ \sum_{i=0}^{n} x^i \text{ in } O(n) \text{ cheap steps} + O(n^{1/2}) \text{ expensive steps} \]
Evaluating Taylor series using rectangular splitting

Paterson and Stockmeyer, 1973:
\[ \sum_{i=0}^{n} \square x^i \text{ in } O(n) \text{ cheap steps} + O(n^{1/2}) \text{ expensive steps} \]

\[
( \square + \square x + \square x^2 + \square x^3 ) + \\
( \square + \square x + \square x^2 + \square x^3 ) x^4 + \\
( \square + \square x + \square x^2 + \square x^3 ) x^8 + \\
( \square + \square x + \square x^2 + \square x^3 ) x^{12} \]
Evaluating Taylor series using rectangular splitting

Paterson and Stockmeyer, 1973:

\[ \sum_{i=0}^{n} \Box x^i \text{ in } O(n) \text{ cheap steps} + O(n^{1/2}) \text{ expensive steps} \]

\[ (\Box + \Box x + \Box x^2 + \Box x^3) + \]
\[ (\Box + \Box x + \Box x^2 + \Box x^3) x^4 + \]
\[ (\Box + \Box x + \Box x^2 + \Box x^3) x^8 + \]
\[ (\Box + \Box x + \Box x^2 + \Box x^3) x^{12} + \]

▷ Smith, 1989: elementary and hypergeometric functions
Evaluating Taylor series using rectangular splitting

Paterson and Stockmeyer, 1973:

\[
\sum_{i=0}^{n} \Box x^i \text{ in } O(n) \text{ cheap steps } + O(n^{1/2}) \text{ expensive steps}
\]

\[
( \Box + \Box x + \Box x^2 + \Box x^3 ) + \\
( \Box + \Box x + \Box x^2 + \Box x^3 ) x^4 + \\
( \Box + \Box x + \Box x^2 + \Box x^3 ) x^8 + \\
( \Box + \Box x + \Box x^2 + \Box x^3 ) x^{12}
\]

- Smith, 1989: elementary and hypergeometric functions
- Brent & Zimmermann, 2010: improvements to Smith
Evaluating Taylor series using rectangular splitting

Paterson and Stockmeyer, 1973:
\[ \sum_{i=0}^{n} \Box x^i \text{ in } O(n) \text{ cheap steps} + O(n^{1/2}) \text{ expensive steps} \]

- Smith, 1989: elementary and hypergeometric functions
- Brent & Zimmermann, 2010: improvements to Smith
- FJ, 2014: generalization to D-finite functions
Evaluating Taylor series using rectangular splitting

Paterson and Stockmeyer, 1973:

\[ \sum_{i=0}^{n} \Box x^i \text{ in } O(n) \text{ cheap steps} + O(n^{1/2}) \text{ expensive steps} \]

\[
( \Box + \Box x + \Box x^2 + \Box x^3 ) + \\
( \Box + \Box x + \Box x^2 + \Box x^3 ) x^4 + \\
( \Box + \Box x + \Box x^2 + \Box x^3 ) x^8 + \\
( \Box + \Box x + \Box x^2 + \Box x^3 ) x^{12}
\]

- Smith, 1989: elementary and hypergeometric functions
- Brent & Zimmermann, 2010: improvements to Smith
- FJ, 2014: generalization to D-finite functions
- New: optimized algorithm for elementary functions
Logarithmic series

Rectangular splitting:

\[ x + \frac{1}{2}x^2 + x^3\left\{\frac{1}{3} + \frac{1}{4}x + \frac{1}{5}x^2 + x^3\left\{\frac{1}{6} + \frac{1}{7}x + \frac{1}{8}x^2\right\}\right\} \]
Logarithmic series

Rectangular splitting:

\[
x + \frac{1}{2}x^2 + x^3 \left\{ \frac{1}{3} + \frac{1}{4}x + \frac{1}{5}x^2 + x^3 \left\{ \frac{1}{6} + \frac{1}{7}x + \frac{1}{8}x^2 \right\} \right\}
\]

Improved algorithm with fewer divisions:

\[
x + \frac{1}{60} \left[ 30x^2 + x^3 \left\{ 20 + 15x + 12x^2 + x^3 \left\{ 10 + \frac{1}{56} \left[ 60 \left[ 8x + 7x^2 \right] \right] \right\} \right\} \right]
\]
Exponential series

Rectangular splitting:

\[ 1 + x + \frac{1}{2} \left( x^2 + \frac{1}{3} x^3 \right) \left( 1 + \frac{1}{4} \left[ x + \frac{1}{5} \left( x^2 + \frac{1}{6} x^3 \left( 1 + \frac{1}{7} \left[ x + \frac{1}{8} x^2 \right] \right) \right] \right) \]
Exponential series

Rectangular splitting:

\[ 1 + x + \frac{1}{2} \left( x^2 + \frac{1}{3} x^3 \right) \left\{ 1 + \frac{1}{4} \left[ x + \frac{1}{5} \left( x^2 + \frac{1}{6} x^3 \right) \left\{ 1 + \frac{1}{7} \left[ x + \frac{1}{8} x^2 \right] \right\} \right\} \right\} \]

Improved algorithm with fewer divisions:

\[ 1 + x + \frac{1}{24} \left[ 12 x^2 + x^3 \right] \left\{ 4 + 1 \left[ x + \frac{1}{30} \left[ 6 x^2 + x^3 \left\{ 1 + \frac{1}{56} \left[ 8 x + x^2 \right] \right\} \right\} \right\} \]
Coefficients for exp series (on a 64-bit machine)

Numerators 0-20, denominator 20!/0! = 2432902008176640000

2432902008176640000 2432902008176640000 1216451004088320000
405483668029440000 101370917007360000 202741834014720000 3379030566912000
482718652416000 60339831552000 6704425728000 6704425728000 60949324800
5079110400 390700800 27907200 1860480 116280 6840 380 20 1

Numerators 21-33, denominator 33!/20! = 3569119343741952000

169958063987712000 7725366544896000 335885501952000 13995229248000
559809169920 21531121920 797448960 28480320 982080 32736 1056 33 1

Numerators 288-294, denominator 294!/287! = 17667663529534720

613460538991440 2122700826960 7319658024 25153464 86142 294 1
Taylor series evaluation using mpn arithmetic

We use $n$-word fixed-point numbers ($\text{ulp} = 2^{-64n}$)
Negative numbers implicitly or using two's complement!

Example:

```c
// sum = sum + term * coeff
sum[n] += mpn_addmul_1(sum, term, n, coeff)
```
Taylor series evaluation using \texttt{mpn} arithmetic

We use \( n \)-word fixed-point numbers (ulp = \( 2^{-64n} \))
Negative numbers implicitly or using two’s complement!

Example:

\[
\text{sum} = \text{sum} + \text{term} \times \text{coeff}
\]
\[
\text{sum}[n] += \text{mpn_addmul_1}(\text{sum}, \text{term}, n, \text{coeff})
\]

- term is \( n \) words: real number in \([0, 1)\)
- sum is \( n + 1 \) words: real number in \([0, 2^{64})\)
- coeff is 1 word: integer in \([0, 2^{64})\)
Taylor series summation

\[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + x^4 [c_4 + c_5 x + c_6 x^2 + c_7 x^3] \]
Taylor series summation

\[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + x^4 \left[ c_4 + c_5 x + c_6 x^2 + c_7 x^3 \right] \]

```c
sum[n] += mpn_addmul_1(sum, xpowers[3], n, c[7])
sum[n] += mpn_addmul_1(sum, xpowers[2], n, c[6])
sum[n] += mpn_addmul_1(sum, xpowers[1], n, c[5])
sum[n] += c[4]
```
Taylor series summation

\[ c_0 + c_1x + c_2x^2 + c_3x^3 + x^4 [c_4 + c_5x + c_6x^2 + c_7x^3] \]

\[
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[3], n, c[7])
\]

\[
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[2], n, c[6])
\]

\[
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[1], n, c[5])
\]

\[
\text{sum}[n] += c[4]
\]

\[
\text{mpn_mul}(\text{tmp}, \text{sum}, n+1, \text{xpowers}[4], n)
\]

\[
\text{mpn_copyi}(\text{sum}, \text{tmp}+n, n+1)
\]
Taylor series summation

\[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + x^4 [c_4 + c_5 x + c_6 x^2 + c_7 x^3] \]

\[
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[3], n, c[7]) \\
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[2], n, c[6]) \\
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[1], n, c[5]) \\
\text{sum}[n] += c[4] \\
\]

\[
\text{mpn_mul}(\text{tmp}, \text{sum}, n+1, \text{xpowers}[4], n) \\
\text{mpn_copyi}(\text{sum}, \text{tmp}+n, n+1) \\
\]

\[
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[3], n, c[3]) \\
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[2], n, c[2]) \\
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[1], n, c[1]) \\
\text{sum}[n] += c[0] \\
\]
Alternating signs

\[ c_0 - c_1 x + c_2 x^2 - c_3 x^3 + x^4 \left[ c_4 - c_5 x + c_6 x^2 - c_7 x^3 \right] \]
Alternating signs

\[ c_0 - c_1 x + c_2 x^2 - c_3 x^3 + x^4 \left[ c_4 - c_5 x + c_6 x^2 - c_7 x^3 \right] \]

```
sum[n] -= mpn_submul_1(sum, xpowers[3], n, c[7])
sum[n] += mpn_addmul_1(sum, xpowers[2], n, c[6])
sum[n] -= mpn_submul_1(sum, xpowers[1], n, c[5])
sum[n] += c[4]
```
Alternating signs

\[ c_0 - c_1 x + c_2 x^2 - c_3 x^3 + x^4 [c_4 - c_5 x + c_6 x^2 - c_7 x^3] \]

\[
\begin{align*}
\text{sum}[n] &= \text{mpn_submul}_1(\text{sum}, \text{xpowers}[3], n, c[7]) \\
\text{sum}[n] &= + \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[2], n, c[6]) \\
\text{sum}[n] &= - \text{mpn_submul}_1(\text{sum}, \text{xpowers}[1], n, c[5]) \\
\text{sum}[n] &= + c[4]
\end{align*}
\]

\[
\begin{align*}
\text{mpn_mul}(\text{tmp}, \text{sum}, n+1, \text{xpowers}[4], n) \\
\text{mpn_copyi}(\text{sum}, \text{tmp}+n, n+1)
\end{align*}
\]
Alternating signs

\[ c_0 - c_1 x + c_2 x^2 - c_3 x^3 + x^4 \left[ c_4 - c_5 x + c_6 x^2 - c_7 x^3 \right] \]

```c
sum[n] -= mpn_submul_1(sum, xpowers[3], n, c[7])
sum[n] += mpn_addmul_1(sum, xpowers[2], n, c[6])
sum[n] -= mpn_submul_1(sum, xpowers[1], n, c[5])
sum[n] += c[4]

mpn_mul(tmp, sum, n+1, xpowers[4], n)
mpn_copyi(sum, tmp+n, n+1)
```

```c
sum[n] -= mpn_submul_1(sum, xpowers[3], n, c[3])
sum[n] += mpn_addmul_1(sum, xpowers[2], n, c[2])
sum[n] -= mpn_submul_1(sum, xpowers[1], n, c[1])
sum[n] += c[0]
```
Including divisions (exponential series)

\[
\frac{1}{q_0} \left[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \frac{1}{q_4} \left[ c_4 x^4 + c_5 x^5 \right] \right]
\]
Including divisions (exponential series)

\[
\frac{1}{q_0} \left[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \frac{1}{q_4} \left[ c_4 x^4 + c_5 x^5 \right]\right]
\]

\[
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[5], n, c[5])
\]

\[
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[4], n, c[4])
\]
Including divisions (exponential series)

\[
\frac{1}{q_0} \left[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \frac{1}{q_4} \left[ c_4 x^4 + c_5 x^5 \right] \right]
\]

```plaintext
sum[n] += mpn_addmul_1(sum, xpowers[5], n, c[5])
sum[n] += mpn_addmul_1(sum, xpowers[4], n, c[4])
mpn_divrem_1(sum, 0, sum, n+1, q[4])
```
Including divisions (exponential series)

\[
\frac{1}{q_0} \left[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \frac{1}{q_4} \left[ c_4 x^4 + c_5 x^5 \right] \right]
\]

\begin{align*}
\text{sum}[n] & \text{ += mpn_addmul}_1(\text{sum}, \text{xpowers}[5], n, c[5]) \\
\text{sum}[n] & \text{ += mpn_addmul}_1(\text{sum}, \text{xpowers}[4], n, c[4]) \\
\text{mpn_divrem}_1(\text{sum}, 0, \text{sum}, n+1, q[4]) \\
\text{sum}[n] & \text{ += mpn_addmul}_1(\text{sum}, \text{xpowers}[3], n, c[3]) \\
\text{sum}[n] & \text{ += mpn_addmul}_1(\text{sum}, \text{xpowers}[2], n, c[2]) \\
\text{sum}[n] & \text{ += mpn_addmul}_1(\text{sum}, \text{xpowers}[1], n, c[1]) \\
\text{sum}[n] & \text{ += c[0]}
\end{align*}
Including divisions (exponential series)

\[ \frac{1}{q_0} \left[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \frac{1}{q_4} \left[ c_4 x^4 + c_5 x^5 \right] \right] \]

\[
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[5], n, c[5]) \\
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[4], n, c[4]) \\
\text{mpn_divrem}_1(\text{sum}, 0, \text{sum}, n+1, q[4])
\]

\[
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[3], n, c[3]) \\
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[2], n, c[2]) \\
\text{sum}[n] += \text{mpn_addmul}_1(\text{sum}, \text{xpowers}[1], n, c[1]) \\
\text{sum}[n] += c[0] \\
\text{mpn_divrem}_1(\text{sum}, 0, \text{sum}, n+1, q[0])
\]
Including divisions (logarithmic series)

\[
\frac{1}{q_0} \left[c_0 + c_1x + c_2x^2 + c_3x^3\right] + \frac{1}{q_4} \left[c_4x^4 + c_5x^5\right]
\]
Including divisions (logarithmic series)

\[ \frac{1}{q_0} \left[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \frac{q_0}{q_4} \left[ c_4 x^4 + c_5 x^5 \right] \right] \]
Including divisions (logarithmic series)

\[
\frac{1}{q_0} \left[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \frac{q_0}{q_4} \left[ c_4 x^4 + c_5 x^5 \right] \right]
\]

sum[n] += mpn_addmul_1(sum, xpowers[5], n, c[5])
sum[n] += mpn_addmul_1(sum, xpowers[4], n, c[4])
Including divisions (logarithmic series)

\[
\frac{1}{q_0} \left[ \begin{array}{c} c_0 + c_1x + c_2x^2 + c_3x^3 + \frac{q_0}{q_4} \left[ c_4x^4 + c_5x^5 \right] \end{array} \right]
\]

```c
sum[n] += mpn_addmul_1(sum, xpowers[5], n, c[5])
sum[n] += mpn_addmul_1(sum, xpowers[4], n, c[4])
sum[n+1] = mpn_mul_1(sum, sum, n+1, q[0])
mpn_divrem_1(sum, 0, sum, n+2, q[4])
```
Including divisions (logarithmic series)

\[
\frac{1}{q_0} \left[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \frac{q_0}{q_4} \left[ c_4 x^4 + c_5 x^5 \right] \right]
\]

```
sum[n] += mpn_addmul_1(sum, xpowers[5], n, c[5])
sum[n] += mpn_addmul_1(sum, xpowers[4], n, c[4])

sum[n+1] = mpn_mul_1(sum, sum, n+1, q[0])
mpn_divrem_1(sum, 0, sum, n+2, q[4])
```

```
sum[n] += mpn_addmul_1(sum, xpowers[3], n, c[3])
sum[n] += mpn_addmul_1(sum, xpowers[2], n, c[2])
sum[n] += mpn_addmul_1(sum, xpowers[1], n, c[1])
sum[n] += c[0]
```
Including divisions (logarithmic series)

\[
\frac{1}{q_0} \left[ c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \frac{q_0}{q_4} [ c_4 x^4 + c_5 x^5 ] \right]
\]

```c
sum[n] += mpn_addmul_1(sum, xpowers[5], n, c[5])
sum[n] += mpn_addmul_1(sum, xpowers[4], n, c[4])

sum[n+1] = mpn_mul_1(sum, sum, n+1, q[0])
mpn_divrem_1(sum, 0, sum, n+2, q[4])

sum[n] += mpn_addmul_1(sum, xpowers[3], n, c[3])
sum[n] += mpn_addmul_1(sum, xpowers[2], n, c[2])
sum[n] += mpn_addmul_1(sum, xpowers[1], n, c[1])
sum[n] += c[0]

mpn_divrem_1(sum, 0, sum, n+1, q[0])
```
Error bounds and correctness

The algorithm evaluates each truncated Taylor series with $\leq 2$ fixed-point ulp error
Error bounds and correctness

The algorithm evaluates each truncated Taylor series with \( \leq 2 \) fixed-point ulp error

How does that work?

- mpn_addmul_1 type operations are exact
- Clearing denominators gives up to 64 guard bits
- Evaluation is done backwards: multiplications and divisions remove error
Error bounds and correctness

The algorithm evaluates each truncated Taylor series with \( \leq 2 \) fixed-point ulp error

How does that work?

- `mpn_addmul_1` type operations are exact
- Clearing denominators gives up to 64 guard bits
- Evaluation is done backwards: multiplications and divisions remove error

Must also show: no overflow, correct handling of signs
Error bounds and correctness

The algorithm evaluates each truncated Taylor series with \( \leq 2 \) fixed-point ulp error

How does that work?

- mpn_addmul_1 type operations are exact
- Clearing denominators gives up to 64 guard bits
- Evaluation is done backwards: multiplications and divisions remove error

Must also show: no overflow, correct handling of signs

Proof depends on the precise sequence of numerators and denominators used.
Error bounds and correctness

The algorithm evaluates each truncated Taylor series with \( \leq 2 \) fixed-point ulp error

How does that work?

- `mpn_addmul_1` type operations are exact
- Clearing denominators gives up to 64 guard bits
- Evaluation is done backwards: multiplications and divisions remove error

Must also show: no overflow, correct handling of signs

Proof depends on the precise sequence of numerators and denominators used.

Proof by exhaustive side computation!
Benchmarks

The code is part of the Arb library
http://fredrikkj.net/arb

Open source (GPL version 2 or later)
## Timings (microseconds / function evaluation)

<table>
<thead>
<tr>
<th>Bits</th>
<th>exp</th>
<th>sin</th>
<th>cos</th>
<th>log</th>
<th>atan</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.26</td>
<td>0.35</td>
<td>0.35</td>
<td>0.21</td>
<td>0.20</td>
</tr>
<tr>
<td>53</td>
<td>0.27</td>
<td>0.39</td>
<td>0.38</td>
<td>0.26</td>
<td>0.30</td>
</tr>
<tr>
<td>64</td>
<td>0.33</td>
<td>0.47</td>
<td>0.47</td>
<td>0.30</td>
<td>0.34</td>
</tr>
<tr>
<td>128</td>
<td>0.48</td>
<td>0.59</td>
<td>0.59</td>
<td>0.42</td>
<td>0.47</td>
</tr>
<tr>
<td>256</td>
<td>0.83</td>
<td>1.05</td>
<td>1.08</td>
<td>0.66</td>
<td>0.73</td>
</tr>
<tr>
<td>512</td>
<td>2.06</td>
<td>2.88</td>
<td>2.76</td>
<td>1.69</td>
<td>2.20</td>
</tr>
<tr>
<td>1024</td>
<td>6.79</td>
<td>7.92</td>
<td>7.84</td>
<td>5.84</td>
<td>6.97</td>
</tr>
<tr>
<td>2048</td>
<td>22.70</td>
<td>25.50</td>
<td>25.60</td>
<td>22.80</td>
<td>25.90</td>
</tr>
<tr>
<td>4096</td>
<td>82.90</td>
<td>97.00</td>
<td>98.00</td>
<td>99.00</td>
<td>104.00</td>
</tr>
</tbody>
</table>

Measurements done on an Intel i7-2600S CPU.
### Speedup vs MPFR

<table>
<thead>
<tr>
<th>Bits</th>
<th>exp</th>
<th>sin</th>
<th>cos</th>
<th>log</th>
<th>atan</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>7.9</td>
<td>8.2</td>
<td>3.6</td>
<td>11.8</td>
<td>29.7</td>
</tr>
<tr>
<td>53</td>
<td>9.1</td>
<td>8.2</td>
<td>3.9</td>
<td>10.9</td>
<td>25.9</td>
</tr>
<tr>
<td>64</td>
<td>7.6</td>
<td>6.9</td>
<td>3.2</td>
<td>9.3</td>
<td>23.7</td>
</tr>
<tr>
<td>128</td>
<td>6.9</td>
<td>6.9</td>
<td>3.6</td>
<td>10.4</td>
<td>30.6</td>
</tr>
<tr>
<td>256</td>
<td>5.6</td>
<td>5.4</td>
<td>2.9</td>
<td>10.7</td>
<td>31.3</td>
</tr>
<tr>
<td>512</td>
<td>3.7</td>
<td>3.2</td>
<td>2.1</td>
<td>6.9</td>
<td>14.5</td>
</tr>
<tr>
<td>1024</td>
<td>2.7</td>
<td>2.2</td>
<td>1.8</td>
<td>3.6</td>
<td>8.8</td>
</tr>
<tr>
<td>2048</td>
<td>1.9</td>
<td>1.6</td>
<td>1.4</td>
<td>2.0</td>
<td>4.9</td>
</tr>
<tr>
<td>4096</td>
<td>1.7</td>
<td>1.5</td>
<td>1.3</td>
<td>1.3</td>
<td>3.1</td>
</tr>
</tbody>
</table>

Measurements done on an Intel i7-2600S CPU.
Comparison to MPFR

Measurements done on an Intel i7-2600S CPU.
Double (53 bits) precision, microseconds/eval, Intel i7-2600S:

<table>
<thead>
<tr>
<th></th>
<th>exp</th>
<th>sin</th>
<th>cos</th>
<th>log</th>
<th>atan</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGLIBC</td>
<td>0.045</td>
<td>0.056</td>
<td>0.058</td>
<td>0.061</td>
<td>0.072</td>
</tr>
<tr>
<td>MPFR</td>
<td>2.45</td>
<td>3.19</td>
<td>1.48</td>
<td>2.83</td>
<td>7.77</td>
</tr>
<tr>
<td>Arb</td>
<td>0.27</td>
<td>0.39</td>
<td>0.38</td>
<td>0.26</td>
<td>0.30</td>
</tr>
</tbody>
</table>
Double (53 bits) precision, microseconds/eval, Intel i7-2600S:

<table>
<thead>
<tr>
<th></th>
<th>exp</th>
<th>sin</th>
<th>cos</th>
<th>log</th>
<th>atan</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGLIBC</td>
<td>0.045</td>
<td>0.056</td>
<td>0.058</td>
<td>0.061</td>
<td>0.072</td>
</tr>
<tr>
<td>MPFR</td>
<td>2.45</td>
<td>3.19</td>
<td>1.48</td>
<td>2.83</td>
<td>7.77</td>
</tr>
<tr>
<td>Arb</td>
<td>0.27</td>
<td>0.39</td>
<td>0.38</td>
<td>0.26</td>
<td>0.30</td>
</tr>
</tbody>
</table>

Quad (113 bits) precision, microseconds/eval, Intel T4400:

<table>
<thead>
<tr>
<th></th>
<th>exp</th>
<th>sin</th>
<th>cos</th>
<th>log</th>
<th>atan</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPFR</td>
<td>5.76</td>
<td>7.29</td>
<td>3.42</td>
<td>8.01</td>
<td>21.30</td>
</tr>
<tr>
<td>libquadmath</td>
<td>4.51</td>
<td>4.71</td>
<td>4.57</td>
<td>5.39</td>
<td>4.32</td>
</tr>
<tr>
<td>QD</td>
<td>0.73</td>
<td><strong>0.69</strong></td>
<td><strong>0.69</strong></td>
<td>0.82</td>
<td>1.08</td>
</tr>
<tr>
<td>Arb</td>
<td><strong>0.65</strong></td>
<td>0.81</td>
<td>0.79</td>
<td><strong>0.61</strong></td>
<td><strong>0.68</strong></td>
</tr>
</tbody>
</table>
Double (53 bits) precision, microseconds/eval, Intel i7-2600S:

<table>
<thead>
<tr>
<th></th>
<th>exp</th>
<th>sin</th>
<th>cos</th>
<th>log</th>
<th>atan</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGLIBC</td>
<td>0.045</td>
<td>0.056</td>
<td>0.058</td>
<td>0.061</td>
<td>0.072</td>
</tr>
<tr>
<td>MPFR</td>
<td>2.45</td>
<td>3.19</td>
<td>1.48</td>
<td>2.83</td>
<td>7.77</td>
</tr>
<tr>
<td>Arb</td>
<td>0.27</td>
<td>0.39</td>
<td>0.38</td>
<td>0.26</td>
<td>0.30</td>
</tr>
</tbody>
</table>

Quad (113 bits) precision, microseconds/eval, Intel T4400:

<table>
<thead>
<tr>
<th></th>
<th>exp</th>
<th>sin</th>
<th>cos</th>
<th>log</th>
<th>atan</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPFR</td>
<td>5.76</td>
<td>7.29</td>
<td>3.42</td>
<td>8.01</td>
<td>21.30</td>
</tr>
<tr>
<td>libquadmath</td>
<td>4.51</td>
<td>4.71</td>
<td>4.57</td>
<td>5.39</td>
<td>4.32</td>
</tr>
<tr>
<td>QD</td>
<td>0.73</td>
<td><strong>0.69</strong></td>
<td><strong>0.69</strong></td>
<td>0.82</td>
<td>1.08</td>
</tr>
<tr>
<td>Arb</td>
<td><strong>0.65</strong></td>
<td>0.81</td>
<td>0.79</td>
<td><strong>0.61</strong></td>
<td><strong>0.68</strong></td>
</tr>
</tbody>
</table>

Quad-double (212 bits) precision, microseconds/eval, Intel T4400:

<table>
<thead>
<tr>
<th></th>
<th>exp</th>
<th>sin</th>
<th>cos</th>
<th>log</th>
<th>atan</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPFR</td>
<td>7.87</td>
<td>9.23</td>
<td>5.06</td>
<td>12.60</td>
<td>33.00</td>
</tr>
<tr>
<td>QD</td>
<td>6.09</td>
<td>5.77</td>
<td>5.76</td>
<td>20.10</td>
<td>24.90</td>
</tr>
<tr>
<td>Arb</td>
<td><strong>1.29</strong></td>
<td><strong>1.49</strong></td>
<td><strong>1.49</strong></td>
<td><strong>1.26</strong></td>
<td><strong>1.23</strong></td>
</tr>
</tbody>
</table>
Summary

- Elementary functions with error bounds
- Variable precision up to 4600 bits
Summary

- Elementary functions with error bounds
- Variable precision up to 4600 bits
- \texttt{mpn} arithmetic + 256 KB of lookup tables + efficient algorithm to evaluate Taylor series (rectangular splitting, optimized denominator sequence)
- Similar algorithm for all functions (no Newton iteration, etc.)
Summary

- Elementary functions with error bounds
- Variable precision up to 4600 bits
- `mpn` arithmetic + 256 KB of lookup tables + efficient algorithm to evaluate Taylor series (rectangular splitting, optimized denominator sequence)
- Similar algorithm for all functions (no Newton iteration, etc.)
- Improvement over MPFR: up to 3-4x for cos, 8-10x for sin/exp/log, 30x for atan
- Gap to double precision LIBM (EGLIBC): 4-7x
Future work

- Optimizations
  - Gradually change precision in Taylor series summation
  - Use short multiplications (no GMP support)
  - Use precomputed inverses (no GMP support)
  - Assembly for low precision (1-3 words?)
  - Further tuning of lookup tables

What if floating-point expansions or carry-save bignums are used instead of GMP/64-bit full words? Vectorization? What about formally verified implementation? Port to other libraries (e.g. MPFR)?
Future work

- Optimizations
  - Gradually change precision in Taylor series summation
  - Use short multiplications (no GMP support)
  - Use precomputed inverses (no GMP support)
  - Assembly for low precision (1-3 words?)
  - Further tuning of lookup tables

- What if floating-point expansions or carry-save bignums are used instead of GMP/64-bit full words? Vectorization?
Future work

- Optimizations
  - Gradually change precision in Taylor series summation
  - Use short multiplications (no GMP support)
  - Use precomputed inverses (no GMP support)
  - Assembly for low precision (1-3 words?)
  - Further tuning of lookup tables

- What if floating-point expansions or carry-save bignums are used instead of GMP/64-bit full words? Vectorization?

- Formally verified implementation?
Future work

- Optimizations
  - Gradually change precision in Taylor series summation
  - Use short multiplications (no GMP support)
  - Use precomputed inverses (no GMP support)
  - Assembly for low precision (1-3 words?)
  - Further tuning of lookup tables

- What if floating-point expansions or carry-save bignums are used instead of GMP/64-bit full words? Vectorization?
- Formally verified implementation?
- Port to other libraries (e.g. MPFR)
Thank you!