# 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

# 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 ($n \approx 53$ bits)

- Extensively studied - elsewhere!

# Precision ranges

Hardware precision ($n \approx 53$ bits)

- Extensively studied - elsewhere!

**Medium precision ($n \approx 100$-$10\,000$ 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 ($n \approx 53$ bits)

- Extensively studied - elsewhere!

**Medium precision ($n \approx 100$-$10\,000$ 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 ($n \gg 10\,000$ 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

$\exp(x)$ | $\sin(x), \cos(x)$ | $\log(1+x)$ | $\text{atan}(x)$

$$\downarrow$$

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

$$\downarrow$$

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

# Recipe for elementary functions

$\exp(x)$    $\sin(x), \cos(x)$    $\log(1+x)$    $\operatorname{atan}(x)$

$\downarrow$

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

$\downarrow$

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

$\downarrow$

**Argument-halving $r \approx 8$ times**

$$\exp(x) = [\exp(x/2)]^2$$

$$\log(1+x) = 2\log(\sqrt{1+x})$$

$\downarrow$

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

$\downarrow$

**Taylor series**

# Better recipe at medium precision

$\exp(x)$ $\sin(x), \cos(x)$ $\log(1+x)$ $\text{atan}(x)$

$\downarrow$

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

$\downarrow$

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

$\downarrow$

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

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

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

$\downarrow$

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

$\downarrow$

Taylor series

# Argument reduction formulas

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

# Argument reduction formulas

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

$$\exp(x) = \exp(t)\exp(y), \quad y = x - i/q$$

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

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

# Argument reduction formulas

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

$$\exp(x) = \exp(t)\exp(y), \quad y = x - i/q$$

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

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

$$\log(1 + x) = \log(1 + t) + \log(1 + y), \quad y = (qx - i)/(i + q)$$

$$\operatorname{atan}(x) = \operatorname{atan}(t) + \operatorname{atan}(y), \quad y = (qx - i)/(ix + q)$$

# Optimizing lookup tables

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

| Function | Precision | $m$ | $r$ | Entries | Size (KiB) |
|---|---|---|---|---|---|
| exp | $\leq 512$ | 1 | 8 | 178 | 11.125 |
| exp | $\leq 4608$ | 2 | 5 | 23+32 | 30.9375 |
| sin | $\leq 512$ | 1 | 8 | 203 | 12.6875 |
| sin | $\leq 4608$ | 2 | 5 | 26+32 | 32.625 |
| cos | $\leq 512$ | 1 | 8 | 203 | 12.6875 |
| cos | $\leq 4608$ | 2 | 5 | 26+32 | 32.625 |
| log | $\leq 512$ | 2 | 7 | 128+128 | 16 |
| log | $\leq 4608$ | 2 | 5 | 32+32 | 36 |
| atan | $\leq 512$ | 1 | 8 | 256 | 16 |
| atan | $\leq 4608$ | 2 | 5 | 32+32 | 36 |
| Total | | | | | 236.6875 |

# Taylor series

**Logarithmic series**:

$\operatorname{atan}(x) = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \ldots$

$\log(1+x) = 2\operatorname{atanh}(x/(x+2))$

With $x < 2^{-10}$, need 230 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

# 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:

$\sum_{i=0}^{n} \square x^i$ in $O(n)$ cheap steps $+$ $O(n^{1/2})$ expensive steps

# Evaluating Taylor series using rectangular splitting

Paterson and Stockmeyer, 1973:

$\sum_{i=0}^{n} \square x^i$ in $O(n)$ cheap steps + $O(n^{1/2})$ expensive steps

$$
\begin{array}{lcccccccccl}
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^4 & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^8 & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^{12} &
\end{array}
$$

# Evaluating Taylor series using rectangular splitting

Paterson and Stockmeyer, 1973:

$\sum_{i=0}^{n} \square x^i$ in $O(n)$ cheap steps $+$ $O(n^{1/2})$ expensive steps

$$
\begin{array}{lllllllllll}
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^4 & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^8 & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^{12} &
\end{array}
$$

- Smith, 1989: elementary and hypergeometric functions

# Evaluating Taylor series using rectangular splitting

Paterson and Stockmeyer, 1973:

$\sum_{i=0}^{n} \square x^i$ in $O(n)$ cheap steps + $O(n^{1/2})$ expensive steps

$$
\begin{array}{lllllllllll}
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^4 & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^8 & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^{12} &
\end{array}
$$

- Smith, 1989: elementary and hypergeometric functions
- Brent & Zimmermann, 2010: improvements to Smith

# Evaluating Taylor series using rectangular splitting

Paterson and Stockmeyer, 1973:

$\sum_{i=0}^{n} \square x^i$ in $O(n)$ cheap steps + $O(n^{1/2})$ expensive steps

$$
\begin{array}{ccccccccccc}
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^4 & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^8 & + \\
( & \square & + & \square x & + & \square x^2 & + & \square x^3 & ) & x^{12} &
\end{array}
$$

- 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:

$\sum_{i=0}^{n} \square x^i$ in $O(n)$ cheap steps $+$ $O(n^{1/2})$ expensive steps

$$
\begin{array}{l}
(\ \square + \square x + \square x^2 + \square x^3\ ) \qquad\quad + \\
(\ \square + \square x + \square x^2 + \square x^3\ )\ x^4 + \\
(\ \square + \square x + \square x^2 + \square x^3\ )\ x^8 + \\
(\ \square + \square x + \square x^2 + \square x^3\ )\ x^{12}
\end{array}
$$

- 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:

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

# Exponential series

Rectangular splitting:

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

Improved algorithm with fewer divisions:

$$1+x+\frac{1}{24}\left[12x^2+x^3\left\{4+1\left[x+\frac{1}{30}\left[6x^2+x^3\left\{1+\frac{1}{56}\left[8x+x^2\right]\right\}\right]\right]\right\}\right]$$

# Coefficients for exp series (on a 64-bit machine)

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

```
2432902008176640000 2432902008176640000 1216451004088320000
405483668029440000 101370917007360000 20274183401472000 3379030566912000
482718652416000 60339831552000 6704425728000 670442572800 60949324800
5079110400 390700800 27907200 1860480 116280 6840 380 20 1
```

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

```
1699580639877120000 77253665448960000 3358855019520000 139952292480000
5598091699200 215311219200 7974489600 284803200 9820800 327360 10560 330 1
```

Numerators 288-294, denominator $294!/287! = 17667663522953472000$

```
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!

# Taylor series evaluation using `mpn` arithmetic

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

**Example:**

```
// sum = sum + term * coeff
sum[n] += mpn_addmul_1(sum, term, n, 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

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

# Taylor series summation

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

```
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

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

```
sum[n] += mpn_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]
```

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

# Taylor series summation

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

```
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]
```

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

```
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]
```

# 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

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

```
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]
```

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

# 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]
```

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

```
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)

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

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

# Including divisions (exponential series)

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

```
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)

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

```
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])
```

```
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 (exponential series)

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

```
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])

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])
```

# Including divisions (logarithmic series)

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

# 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)

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

```
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])
```

# 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)

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

```
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]

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://fredrikj.net/arb`

Open source (GPL version 2 or later)

# Timings (microseconds / function evaluation)

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

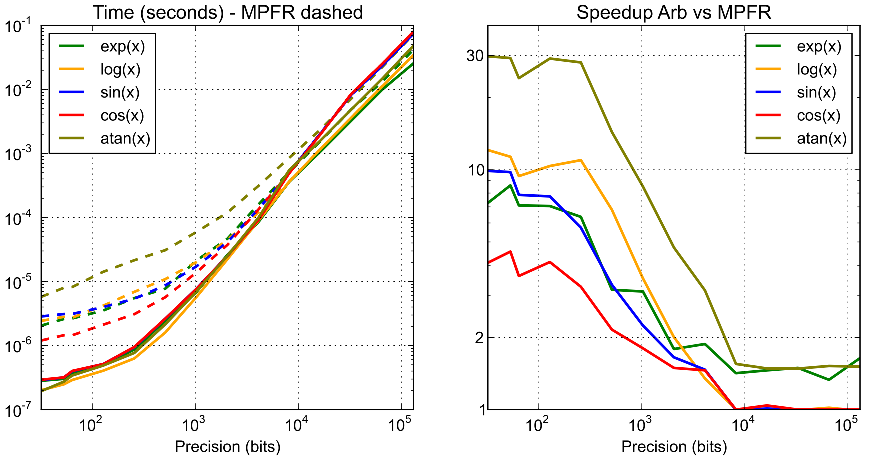
Measurements done on an Intel i7-2600S CPU.

# Speedup vs MPFR

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

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:

| | exp | sin | cos | log | atan |
|---|---|---|---|---|---|
| EGLIBC | **0.045** | **0.056** | **0.058** | **0.061** | **0.072** |
| MPFR | 2.45 | 3.19 | 1.48 | 2.83 | 7.77 |
| Arb | 0.27 | 0.39 | 0.38 | 0.26 | 0.30 |

Double (53 bits) precision, microseconds/eval, Intel i7-2600S:

| | exp | sin | cos | log | atan |
|---|---|---|---|---|---|
| EGLIBC | **0.045** | **0.056** | **0.058** | **0.061** | **0.072** |
| MPFR | 2.45 | 3.19 | 1.48 | 2.83 | 7.77 |
| Arb | 0.27 | 0.39 | 0.38 | 0.26 | 0.30 |

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

| | exp | sin | cos | log | atan |
|---|---|---|---|---|---|
| MPFR | 5.76 | 7.29 | 3.42 | 8.01 | 21.30 |
| libquadmath | 4.51 | 4.71 | 4.57 | 5.39 | 4.32 |
| QD | 0.73 | **0.69** | **0.69** | 0.82 | 1.08 |
| Arb | **0.65** | 0.81 | 0.79 | **0.61** | **0.68** |

Double (53 bits) precision, microseconds/eval, Intel i7-2600S:

| | exp | sin | cos | log | atan |
|---|---|---|---|---|---|
| EGLIBC | **0.045** | **0.056** | **0.058** | **0.061** | **0.072** |
| MPFR | 2.45 | 3.19 | 1.48 | 2.83 | 7.77 |
| Arb | 0.27 | 0.39 | 0.38 | 0.26 | 0.30 |

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

| | exp | sin | cos | log | atan |
|---|---|---|---|---|---|
| MPFR | 5.76 | 7.29 | 3.42 | 8.01 | 21.30 |
| libquadmath | 4.51 | 4.71 | 4.57 | 5.39 | 4.32 |
| QD | 0.73 | **0.69** | **0.69** | 0.82 | 1.08 |
| Arb | **0.65** | 0.81 | 0.79 | **0.61** | **0.68** |

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

| | exp | sin | cos | log | atan |
|---|---|---|---|---|---|
| MPFR | 7.87 | 9.23 | 5.06 | 12.60 | 33.00 |
| QD | 6.09 | 5.77 | 5.76 | 20.10 | 24.90 |
| Arb | **1.29** | **1.49** | **1.49** | **1.26** | **1.23** |

# Summary

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

# 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.)

# 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

# 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!