Multiple-Precision Arithmetic: from MP to MPFR

Paul Zimmermann

Number Theory Down Under, Newcastle, September 23, 2016

ANTS-7, Berlin, July 2006

From MP ...

The MP package

A Fortran Multiple-Precision Arithmetic Package, *ACM Transactions on Mathematical Software*, Richard P. Brent, 1978.

November 1973: first working version (731101)

Version 770217: matches the TOMS publication.

1978: Augment interface added

1979: storage allocation improved, rounding options implemented, dependance on Fortran REAL eliminated, added packed numbers

Main MP Features

base b, t digits, with $b \geq 2$, $t \geq 2$, $8b^2 - 1$ representable as a single-precision integer

wordlength	b
48 bits	$2^{22} = 4194304$ or 10^6
36 bits	$2^{16} = 65536$ or 10^4
32 bits	$2^{14} = 16384$ or 10^4
24 bits	2^{10} or 1000
18 bits	2^7 or 100
16 bits	2^6 or 10
12 bits	2^4 or 10

Assumption $8b^2 - 1$ representable as a single-precision integer: storage waste of about 50%

Exponent in $[-m, m]$, with $4m$ representable as a single-precision integer

Internal representation

$$\underbrace{\boxed{\text{sign } (0, +1, \text{-}1)}}_{\text{word } 1}\underbrace{\boxed{\text{exponent (base } b\text{)}}}_{\text{word } 2}\underbrace{\boxed{\text{fraction}}}_{\text{word } 3}\cdots\underbrace{\boxed{\text{fraction}}}_{\text{word } t+2}$$

The precision t in words is global for a given MP session, thus does not need to be represented (idem for base b).

Example: representation of 17 with $b = 2^{10}$ and $t = 2$:

$$\underbrace{\boxed{+1}}_{\text{word } 1}\ \underbrace{\boxed{0}}_{\text{word } 2}\ \underbrace{\boxed{17}}_{\text{word } 3}\ \underbrace{\boxed{0}}_{\text{word } 4}$$

The subroutines are machine independent and the precision is arbitrary, subject to storage limitations

We have attempted to make it efficient at a high level by implementing good algorithms

Compressed/packed numbers to avoid the storage waste of about 50% (at the expense of increased timings by about 1.5)

Rounding in MP

RNDRL $= 0$ means truncated (chopped) arithmetic

RNDRL $= 1$ means rounded (to nearest) arithmetic

RNDRL $= 2$ means round down (towards $-\infty$)

RNDRL $= 3$ means round up (towards $+\infty$)

Underflow and Overflow in MP

underflow: set to zero

overflow: fatal error

no Infinity, no Not-a-Number

Implemented functions

MPROOT for $x^{-1/n}$

MPEXP1 for $\exp(x) - 1$, MPEXP for $\exp(x)$

MPLNS for $\log(1 + x)$, MPLN for $\log(x)$

MPSIN, MPTAN, MPATAN, MPASIN,

MPPI for π, MPEUL for γ,

MPGAM for $\Gamma(x)$,

MPBERN for the Bernoulli numbers

MPEI, MPERF, MPERFC

MPBESJ

The Augment Interface

(with J. A. Hooper and J. M. Yohe)

```
MULTIPLE X, Y, Z
...
X = Y + Z*EXP(X+1)/Y
```

Applications of MP

Knuth's constants to 1000 decimal and 1100 octal places, Richard P. Brent, Technical Report 47, Computer Centre, Australian National University, 1975.

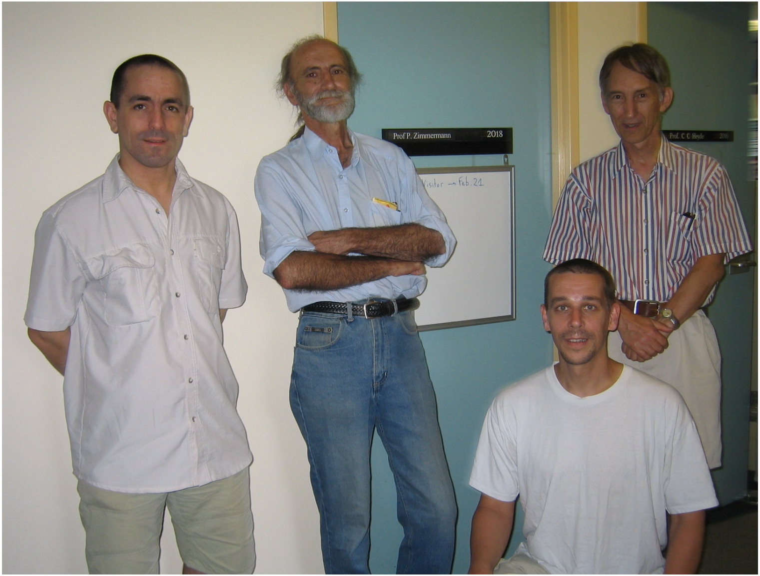
The constants were computed twice, once with base 10000 and 260 floating-point digits, and once with base 11701 and 250 digits ($10000^{260} = 10^{1040}, 11701^{250} \approx 10^{1017}$).

Future plans (from June 1981)

It is also impracticable to formally prove correctness of any nontrivial MP routines using present theorem-proving techniques.

In the future we hope to implement rounding options for more MP routines, and write a multiple-precision interval arithmetic package which uses MP and takes advantage of the directed rounding options.

A never-ending project is to implement multiple-precision versions of ever more special functions, and to improve the efficiency of those multiple-precision routines already implemented.



Visit to ANU, February 2007

Visit to ANU, February 2007

… to MPFR

Notations

MPFR uses GMP's `mpn` layer for its internal representation

limb: a GMP machine word (usually 32 or 64 bits)

For simplicity we will assume the number of bits in a limb is 64 in this talk.

Number Representation in MPFR

- precision $p \geq 1$ (in bits)
- sign (-1 or $+1$)
- exponent (between $E_{\min}$ and $E_{\max}$), also used to represent special numbers (NaN, $\pm\infty$, ± 0)
- significand (array of $\lceil p/64 \rceil$ limbs), only defined for *regular* numbers (neither NaN, nor $\pm\infty$, nor ± 0)

Most significant limbs/bits will be represented left in this talk.
Regular numbers are *normalized*: the most significant bit of the most significant limb should be set.
Example : $x = 17$ with a precision of 10 bits is stored with a 6-bit limb as

$$\underbrace{10}_{\textit{precision}}\quad \underbrace{+1}_{\textit{sign}}\quad \underbrace{5}_{\textit{exponent}}\quad \underbrace{\boxed{100010}}_{\text{word 1}}\underbrace{\boxed{000000}}_{\text{word 0}}$$

Round and sticky bit

$$v = \underbrace{xxx\ldots yyy}_{m \text{ of } p \text{ bits}} \quad \underbrace{r}_{\text{round bit}} \quad \underbrace{sss\ldots}_{\text{sticky bit}}$$

The *round bit* r is the value of bit at position $p+1$.

The *sticky bit* s is zero iff `sss...` is identically zero.

The round and sticky bits are enough to get *correct rounding* for all rounding modes:

r	s	zero	nearest	away
0	0	m	m	m
0	1	m	m	$m+1$
1	0	m	$m + (m \bmod 2)$	$m+1$
1	1	m	$m+1$	$m+1$

The mpfr_add function

The mpfr_add(a, b, c) function works as follows ($a \leftarrow b + c$):

- first check for singular values ($NaN, \pm Inf, \pm 0$)
- if b and c have different signs, call mpfr_sub1
- if a, b, c have the same precision, call mpfr_add1sp
- otherwise call the generic code mpfr_add1 described in: Vincent Lefèvre, *The Generic Multiple-Precision Floating-Point Addition With Exact Rounding (as in the MPFR Library)*, 6th Conference on Real Numbers and Computers 2004 - RNC 6, Nov 2004, Dagstuhl, Germany, pp.135-145, 2004.

The mpfr_add1sp function

- if $p < 64$, call mpfr_add1sp1
- if $64 < p < 128$, call mpfr_add1sp2
- else execute the generic addition code for same precision

Note: $p = 64$ and $p = 128$ will use the generic code, thus should be avoided unless really needed.

The mpfr_add1sp1 function

Case 1, $e_b = e_c$:

$$b = \boxed{110100}$$

$$c = \boxed{111000}$$

```
ap[0] = MPFR_LIMB_HIGHBIT | ((bp[0] + cp[0]) >> 1);
e_a = e_b + 1;
rb = ap[0] & (MPFR_LIMB_ONE << (sh - 1));
ap[0] ^= rb;
sb = 0;
```

Since b and c are normalized, the most significant bits of $bp[0]$ and $cp[0]$ are set.

Thus adding $bp[0]$ and $cp[0]$ will always produce a carry, and the exponent of a will be $e_b + 1$.

$$b = \boxed{110100}$$

$$c = \boxed{111000}$$

```
ap[0] = MPFR_LIMB_HIGHBIT | ((bp[0] + cp[0]) >> 1);
e_a = e_b + 1;
rb = ap[0] & (MPFR_LIMB_ONE << (sh - 1));
ap[0] ^= rb;
sb = 0;
```

The sum might have up to $p + 1$ bits, but since $p < 64$ ($p < 6$ here), it fits on 64 bits.

`sh` is the number $64 - p$ of trailing bits, here $6 - p = 2$.

The round bit is the $(p + 1)$-th bit of the addition, the sticky bit is always zero.

An overflow might happen, but no underflow.

The mpfr_sub function

The mpfr_sub(a, b, c) function works as follows ($a \leftarrow b - c$):

- first check for singular values ($NaN, \pm Inf, \pm 0$)
- if b and c have different signs, call mpfr_add1
- if b and c have the same precision, call mpfr_sub1sp
- otherwise call the generic code mpfr_sub1

The mpfr_sub1sp function

- if $p < 64$, call mpfr_sub1sp1
- if $64 < p < 128$, call mpfr_sub1sp2
- else execute the generic subtraction code for same precision

Note: $p = 64$ and $p = 128$ will use the generic code, thus should be avoided unless really needed.

The mpfr_sub1sp1 function

- if the exponents differ, swap b and c so that $e_b > e_c$
- case 1: $e_b = e_c$
- case 2: $e_b > e_c$

Case 1, $e_b = e_c$:

$$b = \boxed{110100}$$

$$c = \boxed{111000}$$

subtract $bp[0] - cp[0]$ and put the result in $ap[0]$, which is $bp[0] - cp[0] \bmod 2^{64}$

if $ap[0] = 0$, then the result is zero

if $ap[0] > bp[0]$, then a borrow occurred thus $|c| > |b|$: change $ap[0]$ to $-ap[0]$ and change the sign of a

otherwise no borrow occurred thus $|c| < |b|$

count the number of leading zeros in $ap[0]$, shift $ap[0]$ accordingly and decrease the exponent

in that case both the round bit and the sticky bit are zero

An underflow might happen, no overflow since $|a| \leq \max(|b|, |c|)$

The mpfr_mul(a,b,c) function

$$a \leftarrow \circ(b \cdot c)$$

- if $p_a < 64$ and $p_b, p_c \leq 64$, call mpfr_mul_1
- if $64 < p_a < 128$ and $64 < p_b, p_c \leq 128$, call mpfr_mul_2
- else use the generic code

The mpfr_mul_1 function

$$a \leftarrow \circ(b \cdot c)$$

a: at most one limb (minus 1 bit); b, c: at most one limb

$$h \cdot 2^{64} + \ell \leftarrow bp[0] \cdot cp[0]$$

Since $2^{63} \leq bp[0], cp[0] < 2^{64}$, we have $2^{62} \leq h$

If $h < 2^{63}$, shift h, ℓ by 1 bit to the left, and decrease the exponent

The round bit is formed by the $(p+1)$-th bit of h

The sticky bit is formed by the remaining bits of h, and those of ℓ

Both underflow and overflow can happen

Warning: MPFR considers underflow *after* rounding (with an infinite exponent range)

The mpfr_div(a,b,c) function

$$a \leftarrow \circ(b/c)$$

- if $p_a < 64$ and $p_b, p_c \leq 64$, call mpfr_div_1
- if $64 < p_a < 128$ and $64 < p_b, p_c \leq 128$, call mpfr_div_2
- else use the generic code

The mpfr_div_1 function

$$a \leftarrow \circ(b/c)$$

Assume $p_a < 64$ and $p_b, p_c \leq 64$

1. $bp[0] \geq cp[0]$: one extra quotient bit
2. $bp[0] < cp[0]$: no extra quotient bit

Deal separately with the special case where the target precision is less than 32, and the divisor $cp[0]$ has at most 32 bits. Then a single 64/32-bit division suffices. (Code used when dividing two binary32 numbers.)

General case: perform a 128/64 integer division, calling GMP's `udiv_qrnnd_preinv` routine. This yields a quotient of 64 bits, and a remainder, from which the round and sticky bit are deduced.

$$bp[0] \cdot 2^{64} = q \cdot cp[0] + r$$

With `-enable-gmp-internals`, `udiv_qrnnd_preinv` uses GMP's `mpn_invert_limb` routine, which given $2^{63} \leq d < 2^{64}$, returns $\lfloor (2^{128} - 1)/d - 2^{64} \rfloor$.

$$i = \lfloor (2^{128} - 1)/cp[0] - 2^{64} \rfloor$$

$$q \approx bp[0] + (i \cdot bp[0]/2^{64})$$

Without `-enable-gmp-internals`, we let $d = d_1 2^{32} + d_0$, and perform two divisions by d_1 using its pseudo-inverse $i = \lfloor (2^{64} - 1)/d_1 \rfloor$. This is slightly slower.

The mpfr_sqrt(r,u) function

$$r \leftarrow \circ(\sqrt{u})$$

- if $p_r < 64$ and $p_u < 64$, call mpfr_sqrt1
- if $64 < p_r < 128$ and $64 < p_u \leq 128$, call mpfr_sqrt2
- else use the generic code

The mpfr_sqrt1 function

Input: $2^{63} \le u < 2^{64}$ representing a p-bit number with $p < 64$ (thus its least significant bit is 0)

- if the exponent of u is odd, shift u by one bit to the right

Now $2^{62} \le u < 2^{64}$

- call `mpn_sqrtrem2`, a routine returning r and s such that

$$u \cdot 2^{64} = r^2 + s \qquad \text{with } 0 \le s \le 2r$$

We have $2^{63} \le r < 2^{64}$ and $0 \le s < 2^{65}$, thus s is represented by one 64-bit word and one bit.

- deduce the round bit from r, and the sticky bit from s and the last bits of r (if $p < 63$).

The mpfr_sqrtrem2 function

Input: $u := u_3 \cdot 2^{192} + u_2 \cdot 2^{128} + u_1 \cdot 2^{64} + u_0$ with $0 \le u_j < 2^{64}$

Output: r and s such that $u = r^2 + s$ with $u < (r+1)^2$.

GMP provides a `mpn_sqrtrem` function but it is slow.

mpfr_sqrtrem2 works as follows:

• using a bipartite table reading the leading $12 = 4 + 4 + 4$ bits of u, obtain a 17-bit approximation of $u^{-1/2}$ with about 9 correct bits

$$u = \underbrace{xxxx}_{a}\,\underbrace{yyyy}_{b}\,\underbrace{zzzz}_{c}\cdots$$

$$x_0 = T_1[a,b] + T_2[b,c]$$

• using Newton's iteration for the inverse square root, obtain a 32-bit approximation of $u^{-1/2}$ with about 19 correct bits

$$x_1 \approx x_0 + \frac{x}{2}(1 - u x_0^2)$$

• using Newton's iteration for the inverse square root, obtain a 41-bit approximation x of $u^{-1/2}$ with about 38 correct bits, ensuring $x_2 \leq u^{-1/2}$

$$x_2 \approx x_1 + \frac{x}{2}(1 - u x_1^2)$$

• use Karp-Markstein trick to deduce a 64-bit approximation y' of $u^{1/2}$

$$y \approx a x_2, \qquad y' \approx y + \frac{x_2}{2}(a - y^2)$$

MPFR 3.1.4 against MPF (from GMP 6.1.1)

`bavette.loria.fr`, Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz, running at 3.3Ghz, with GMP 6.1.1 and GCC 6.1.1.

MPFR 3.1.4

bits	24	53	113
mpfr_add	**37**	**44**	48
mpfr_sub	**44**	**50**	56
mpfr_mul	**42**	44	59
mpfr_div	115	116	**131**
mpfr_sqrt	**152**	**153**	**244**

MPF from GMP 6.1.1

limbs	24	53	113
mpfr_add	49	49	**45**
mpfr_sub	52	52	**48**
mpfr_mul	43	**43**	**46**
mpfr_div	**81**	**81**	146
mpfr_sqrt	236	234	339

Timings are in cycles.

MPFR 3.1.4 against MPFR 4.0-dev

MPFR 4.0-dev is configured with `-enable-gmp-internals`.

MPFR 3.1.4

bits	24	53	113
mpfr_add	37	44	48
mpfr_sub	44	50	56
mpfr_mul	42	44	59
mpfr_div	115	116	131
mpfr_sqrt	152	153	244

MPFR 4.0-dev

bits	24	53	113
mpfr_add	**25**	**26**	**29**
mpfr_sub	**29**	**31**	**32**
mpfr_mul	**22**	**21**	**33**
mpfr_div	**48**	**57**	**87**
mpfr_sqrt	**48**	**72**	**128**

Timings are in cycles.

MPFR 4.0-dev against MPF (from GMP 6.1.1)

MPFR is configured with `-enable-gmp-internals`.

MPFR 4.0-dev

bits	24	53	113
mpfr_add	**25**	**26**	**29**
mpfr_sub	**29**	**31**	**32**
mpfr_mul	**22**	**21**	**33**
mpfr_div	**48**	**57**	**87**
mpfr_sqrt	**48**	**72**	**128**

MPF from GMP 6.1.1

limbs	24	53	113
mpfr_add	49	49	45
mpfr_sub	52	52	48
mpfr_mul	43	43	46
mpfr_div	81	81	146
mpfr_sqrt	236	234	339

Timings are in cycles.

Visit to France, Ligne Maginot, April 2007