GNU MPFR: back to the future

Paul Zimmermann

inria
(inventors for the digital world)

23 September 2011

MaGiX@LiX 2011 conference

What is GNU MPFR?

An arbitrary precision floating-point library

What is GNU MPFR?

An arbitrary precision floating-point library

written in C

What is GNU MPFR?

An arbitrary precision floating-point library

written in C

providing correct rounding

What is GNU MPFR?

An arbitrary precision floating-point library

written in C

providing correct rounding

which aims to be efficient too

What is GNU MPFR?

An arbitrary precision floating-point library

written in C

providing correct rounding

which aims to be efficient too

used by several software tools: Mathemagix, TRIP, Macaulay2, fpLLL,

What is GNU MPFR?

An arbitrary precision floating-point library

written in C

providing correct rounding

which aims to be efficient too

used by several software tools: Mathemagix, TRIP, Macaulay2, fpLLL, MPC, MPFI, CGAL, Gappa,

What is GNU MPFR?

An arbitrary precision floating-point library

written in C

providing correct rounding

which aims to be efficient too

used by several software tools: Mathemagix, TRIP, Macaulay2, fpLLL, MPC, MPFI, CGAL, Gappa, Sage, Magma, Maple, GCC, ...

Using MPFR in Mathemagix

```
1] type_mode? := true;
2] a:Double == 3.14159265359
3.14159265359: Double
3] exp a
23.1406926328: Double
```

Using MPFR in Mathemagix

```
1] type_mode? := true;
2] a:Double == 3.14159265359
3.14159265359: Double
3] exp a
23.1406926328: Double

4] use "numerix"
5] bit_precision := 53;
6] b:Floating == 3.14159265359
3.141592653590001: Floating
7] exp b
23.14069263278056: Floating
```

Using MPFR in Mathemagix

```
1] type_mode? := true;
2] a:Double == 3.14159265359
3.14159265359: Double
3] exp a
23.1406926328: Double

4] use "numerix"
5] bit_precision := 53;
6] b:Floating == 3.14159265359
3.14159265359000001: Floating
7] exp b
23.14069263278405 6: Floating

8] bit_precision := 97;
9] c:Floating := exp (exp (exp 3.0))
2.050986436051648895044869200806e229520860:
Alias (Floating)
```

MPFR in Sage

```
sage: D = RealField(42, rnd='RNDD');
      U = RealField(42, rnd='RNDU')

sage: D(pi), U(pi)
(3.14159265358, 3.14159265360)

sage: D(pi).exact_rational()
3454217652357/1099511627776

sage: x = RealIntervalField(42)(pi);
      x.lower(), x.upper()
(3.14159265358, 3.14159265360)
```

Plan of the talk

- history of GNU MPFR
- some design choices
- some recent developments
- GNU MPFR in 2022

History of GNU MPFR

1998: discussion with Joris van der Hoeven, Jean-Michel Muller, and Guillaume Hanrot in a *café* in Paris, and by mail with Torbjörn Granlund

History of GNU MPFR

1998: discussion with Joris van der Hoeven, Jean-Michel Muller, and Guillaume Hanrot in a *café* in Paris, and by mail with Torbjörn Granlund

November 1998: *Proposal for a Portable and Reliable Multiple Precision Floating-Point Library* (5 pages)

History of GNU MPFR

1998: discussion with Joris van der Hoeven, Jean-Michel Muller, and Guillaume Hanrot in a *café* in Paris, and by mail with Torbjörn Granlund

November 1998: *Proposal for a Portable and Reliable Multiple Precision Floating-Point Library* (5 pages)

4 February 2000: first public version is released (MPFR 0.4)

History of GNU MPFR

1998: discussion with Joris van der Hoeven, Jean-Michel Muller, and Guillaume Hanrot in a *café* in Paris, and by mail with Torbjörn Granlund

November 1998: *Proposal for a Portable and Reliable Multiple Precision Floating-Point Library* (5 pages)

4 February 2000: first public version is released (MPFR 0.4)

June 2000: copyright assigned to the FSF

History of GNU MPFR

1998: discussion with Joris van der Hoeven, Jean-Michel Muller, and Guillaume Hanrot in a *café* in Paris, and by mail with Torbjörn Granlund

November 1998: *Proposal for a Portable and Reliable Multiple Precision Floating-Point Library* (5 pages)

4 February 2000: first public version is released (MPFR 0.4)

June 2000: copyright assigned to the FSF

24 August 2000: version 1.0 is released

History of GNU MPFR

1998: discussion with Joris van der Hoeven, Jean-Michel Muller, and Guillaume Hanrot in a *café* in Paris, and by mail with Torbjörn Granlund

November 1998: *Proposal for a Portable and Reliable Multiple Precision Floating-Point Library* (5 pages)

4 February 2000: first public version is released (MPFR 0.4)

June 2000: copyright assigned to the FSF

24 August 2000: version 1.0 is released

15 April 2002: version 2.0.1 is released

History of GNU MPFR

1998: discussion with Joris van der Hoeven, Jean-Michel Muller, and Guillaume Hanrot in a *café* in Paris, and by mail with Torbjörn Granlund

November 1998: *Proposal for a Portable and Reliable Multiple Precision Floating-Point Library* (5 pages)

4 February 2000: first public version is released (MPFR 0.4)

June 2000: copyright assigned to the FSF

24 August 2000: version 1.0 is released

15 April 2002: version 2.0.1 is released

October 2005: winner of the Many Digits Friendly Competition

History of GNU MPFR

1998: discussion with Joris van der Hoeven, Jean-Michel Muller, and Guillaume Hanrot in a *café* in Paris, and by mail with Torbjörn Granlund

November 1998: *Proposal for a Portable and Reliable Multiple Precision Floating-Point Library* (5 pages)

4 February 2000: first public version is released (MPFR 0.4)

June 2000: copyright assigned to the FSF

24 August 2000: version 1.0 is released

15 April 2002: version 2.0.1 is released

October 2005: winner of the Many Digits Friendly Competition

October 2007: CEA-EDF-INRIA School on Certified Numerical Computation

2007: Mathemagix and Sage use MPFR

History of GNU MPFR

2007: Mathemagix and Sage use MPFR

March 2008: GCC 4.3.0 uses MPFR in its middle-end

History of GNU MPFR

2007: Mathemagix and Sage use MPFR

March 2008: GCC 4.3.0 uses MPFR in its middle-end

26 January 2009: version 2.4.0 is released (*andouillette sauce moutarde*) and becomes a GNU package

History of GNU MPFR

2007: Mathemagix and Sage use MPFR

March 2008: GCC 4.3.0 uses MPFR in its middle-end

26 January 2009: version 2.4.0 is released (*andouillette sauce moutarde*) and becomes a GNU package

25-26 June 2009: CNC'2 Summer School on MPFR and MPC

History of GNU MPFR

2007: Mathemagix and Sage use MPFR

March 2008: GCC 4.3.0 uses MPFR in its middle-end

26 January 2009: version 2.4.0 is released (*andouillette sauce moutarde*) and becomes a GNU package

25-26 June 2009: CNC'2 Summer School on MPFR and MPC

10 June 2010: version 3.0.0 is released (*boudin aux pommes*)

History of GNU MPFR

2007: Mathemagix and Sage use MPFR

March 2008: GCC 4.3.0 uses MPFR in its middle-end

26 January 2009: version 2.4.0 is released (*andouillette sauce moutarde*) and becomes a GNU package

25-26 June 2009: CNC'2 Summer School on MPFR and MPC

10 June 2010: version 3.0.0 is released (*boudin aux pommes*)

13-14 January 2011: MPFR-MPC developers meeting

History of GNU MPFR

2007: Mathemagix and Sage use MPFR

March 2008: GCC 4.3.0 uses MPFR in its middle-end

26 January 2009: version 2.4.0 is released (*andouillette sauce moutarde*) and becomes a GNU package

25-26 June 2009: CNC'2 Summer School on MPFR and MPC

10 June 2010: version 3.0.0 is released (*boudin aux pommes*)

13-14 January 2011: MPFR-MPC developers meeting

28 August 2011: Presentation at the GNU Hackers Meeting in Paris

2007: Mathemagix and Sage use MPFR

March 2008: GCC 4.3.0 uses MPFR in its middle-end

26 January 2009: version 2.4.0 is released (*andouillette sauce moutarde*) and becomes a GNU package

25-26 June 2009: CNC'2 Summer School on MPFR and MPC

10 June 2010: version 3.0.0 is released (*boudin aux pommes*)

13-14 January 2011: MPFR-MPC developers meeting

28 August 2011: Presentation at the GNU Hackers Meeting in Paris

September 2011: version 3.1.0 is released (*canard à l'orange*)

History of GNU MPFR

2007: Mathemagix and Sage use MPFR

March 2008: GCC 4.3.0 uses MPFR in its middle-end

26 January 2009: version 2.4.0 is released (*andouillette sauce moutarde*) and becomes a GNU package

25-26 June 2009: CNC'2 Summer School on MPFR and MPC

10 June 2010: version 3.0.0 is released (*boudin aux pommes*)

13-14 January 2011: MPFR-MPC developers meeting

28 August 2011: Presentation at the GNU Hackers Meeting in Paris

September 2011: version 3.1.0 is released (*canard à l'orange*)

Early 2012: 2nd MPFR-MPC developers meeting?

The `mpfr_t` type

Each MPFR variable has:

- a precision $p \geq 2$ in bits (`long`)
- a sign $s \in \{-1, 1\}$ (`int`)
- an exponent e (`long`)
- a pointer to the significand m (`mp_limb_t*`)

The corresponding value is

$$s \cdot m \cdot 2^e$$

where m is an integer multiple of 2^{-p} with $1/2 \leq m < 1$

On a 64-bit computer, a 53-bit variable takes 40 bytes (32 bytes for `mpfr_t`, 8 bytes for the significand)

Some design choices

- use of the `mpn` layer from GMP
- local vs global fields
- base 2 or 2^w?
- padding or not?

Use of the `mpn` layer from GMP

- ⊖ dependency on GMP
- ⊕ portability and efficiency of GMP
- ⊕ no assembly code in MPFR, only C code
- ⊖ some basic routines are missing or inefficient in GMP (short product and division, floating-point exponentiation, middle product, *k*th root)

One limb = one GMP base word (usually corresponds to a computer word)

Local vs global fields

- Each MPFR variable has its own precision p: enables to mix variables with different precisions (Newton's iteration). We decided to allow any precision in *bits*, not only multiples of the number w of bits per limb ($w = 32$ or $w = 64$ usually).
- The memory allocated for the significand is exactly $\lceil p/w \rceil$ limbs. No field for allocated space, but requires to reallocate if the precision changes.
- The exceptions are global (contrary to what was planned originally).

Base 2 or 2^w?

Consider a 17-bit significand $b_{16} \ldots b_1 b_0$ on a 10-bit computer. There are several ways to store it:

Base 2, right-aligned (most significant bits left):

$$\boxed{000 b_{16} \ldots b_{10}}\boxed{b_9 \ldots b_0} \cdot 2^{e+3}$$

Base 2 or 2^w?

Consider a 17-bit significand $b_{16} \ldots b_1 b_0$ on a 10-bit computer. There are several ways to store it:

Base 2, right-aligned (most significant bits left):

$$\boxed{000 b_{16} \ldots b_{10}}\boxed{b_9 \ldots b_0} \cdot 2^{e+3}$$

Base 2, left-aligned:

$$\boxed{b_{16} \ldots b_7}\boxed{b_6 \ldots b_0 000} \cdot 2^{e}$$

Base 2 or 2^w?

Consider a 17-bit significand $b_{16} \ldots b_1 b_0$ on a 10-bit computer. There are several ways to store it:

Base 2, right-aligned (most significant bits left):

$$\boxed{000 b_{16} \ldots b_{10}}\boxed{b_9 \ldots b_0} \cdot 2^{e+3}$$

Base 2, left-aligned:

$$\boxed{b_{16} \ldots b_7}\boxed{b_6 \ldots b_0 000} \cdot 2^{e}$$

Base 2^{10}:

$$\boxed{00000 b_{16} \ldots b_{12}}\boxed{b_{11} \ldots b_2}\boxed{b_1 b_0 0 \ldots 0} \cdot 2^{10e'}$$

$$\text{or } \boxed{0 b_{16} \ldots b_8}\boxed{b_7 \ldots b_0 00} \cdot 2^{10e''}$$

Base 2, left aligned: addition $a + b$

$$a = \boxed{1011010111}\boxed{0110111000} \cdot 2^4$$
$$b = \boxed{1101101010}\boxed{1010101000} \cdot 2^0$$

Base 2, left aligned: addition $a + b$

$$\begin{array}{l}
a = \begin{array}{|c|c|}\hline 1011010111 & 0110111000 \\ \hline\end{array} \cdot 2^4 \\
b = \begin{array}{|c|c|}\hline 1101101010 & 1010101000 \\ \hline\end{array} \cdot 2^0
\end{array}$$

We have to shift the smaller operand, which might need another limb:

$$\begin{array}{r|c|c|c|}
\cline{2-3} a = & 1011010111 & 0110111000 & \multicolumn{1}{c}{} \\
\hline b = & 0000110110 & 1010101010 & 1000000000 \\
\cline{2-4}
\end{array}$$

In some cases `mpn_add` might return a carry, which will require another shift

Base 2^w: addition $a + b$

a= | 0000001011 | 0101110110 | 1110000000 | $\cdot 2^0$

b= | 1101101010 | 1010101000 | $\cdot 2^0$

No need to shift:

0000001011	0101110110	1110000000
	1101101010	1010101000

No post-shift needed (except in rare cases, but only limb shift).

Base 2, left aligned: multiplication $a \times b$

a= | 1011010111 | 0110111000 | $\cdot 2^4$

b= | 1101101010 | 1010101000 | $\cdot 2^0$

We perform a 2×2 product, and round:

1001101101	0101100000	1001101100	0011000000

Post-shift needed when product is 01...

Base 2^w: multiplication $a \times b$

$$a = \boxed{0000001011}\boxed{0101110110}\boxed{1110000000} \cdot 2^0$$

$$b = \boxed{0110110101}\boxed{0101010100} \cdot 2^0$$

We need to perform a 3×2 product, and round:

$$\boxed{0000000100}\boxed{1101101010}\boxed{1100000100}\boxed{1101100001}\boxed{1000000000}$$

- base 2: smaller memory usage, number of limbs only depends on precision, multiplication cheaper
- base 2^w: no bit shifts

Base 2, right- vs left-aligned: the latter is better for GMP division, and when we truncate an input

A clever idea?

Instead of flushing to zero least significant padding bits:

$$a = \boxed{1011010111}\boxed{0110111000} \cdot 2^4$$

why not use them to store extra bits?

$$a = \boxed{1011010111}\boxed{0110111101} \cdot 2^4$$

A clever idea?

Instead of flushing to zero least significant padding bits:

$$a = \boxed{1011010111}\boxed{0110111000} \cdot 2^4$$

why not use them to store extra bits?

$$a = \boxed{1011010111}\boxed{0110111101} \cdot 2^4$$

Not a so good idea:

- could not emulate IEEE-754 arithmetic ($p = 53$)
- would be non-portable between $w = 16$, $w = 32$, $w = 64$, ...

Constant folding in GCC

```
$ cat bug10709.c
#include <stdio.h>
#include <math.h>
main()
{
  printf ("sin(0.2522464)=%.17f\n", sin(0.2522464));
}
```

Constant folding in GCC

```
$ cat bug10709.c
#include <stdio.h>
#include <math.h>
main()
{
  printf ("sin(0.2522464)=%.17f\n", sin(0.2522464));
}
```

```
$ gcc bug10709.c; ./a.out
sin(0.2522464)=0.24957989804940911
```

Constant folding in GCC

```
$ cat bug10709.c
#include <stdio.h>
#include <math.h>
main()
{
  printf ("sin(0.2522464)=%.17f\n", sin(0.2522464));
}
```

```
$ gcc bug10709.c; ./a.out
sin(0.2522464)=0.24957989804940911
```

```
$ gcc -fno-builtin bug10709.c
/tmp/ccL6YmL8.o: In function `main':
bug10709.c: undefined reference to `sin'
collect2: ld returned 1 exit status
```

Constant folding in GCC

```
$ cat bug10709.c
#include <stdio.h>
#include <math.h>
main()
{
  printf ("sin(0.2522464)=%.17f\n", sin(0.2522464));
}
```

```
$ gcc bug10709.c; ./a.out
sin(0.2522464)=0.24957989804940911
```

```
$ gcc -fno-builtin bug10709.c
/tmp/ccL6YmL8.o: In function ‘main’:
bug10709.c: undefined reference to ‘sin’
collect2: ld returned 1 exit status
```

```
$ gcc -fno-builtin bug10709.c -lm; ./a.out
sin(0.2522464)=0.24957989804940914
```

Some recent developments (*canard à l'orange* release)

automatic TLS (thread local storage) support

new division by zero exception and flag

improved division and squaring using Mulders' algorithm

Some recent developments

We recently improved (with David Harvey) the *short division* in GNU MPFR.

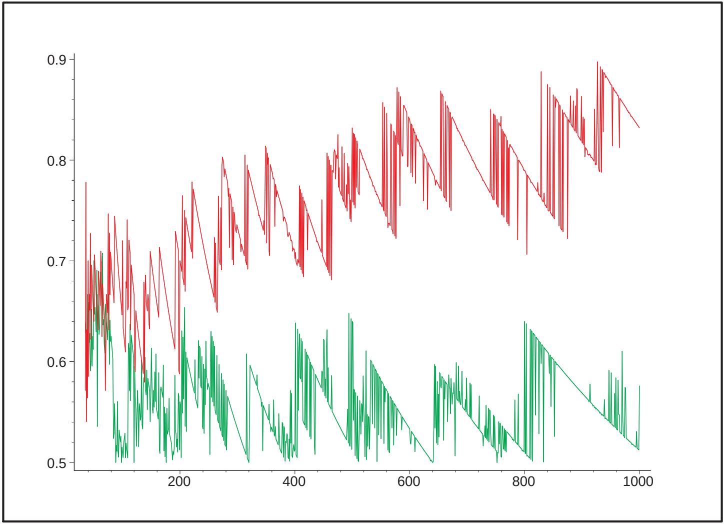
Example: division of two 1000-digits floating-point numbers on a 2.66GHz Intel Xeon X7460.

GMP MPF 5.0.1: 0.0040ms

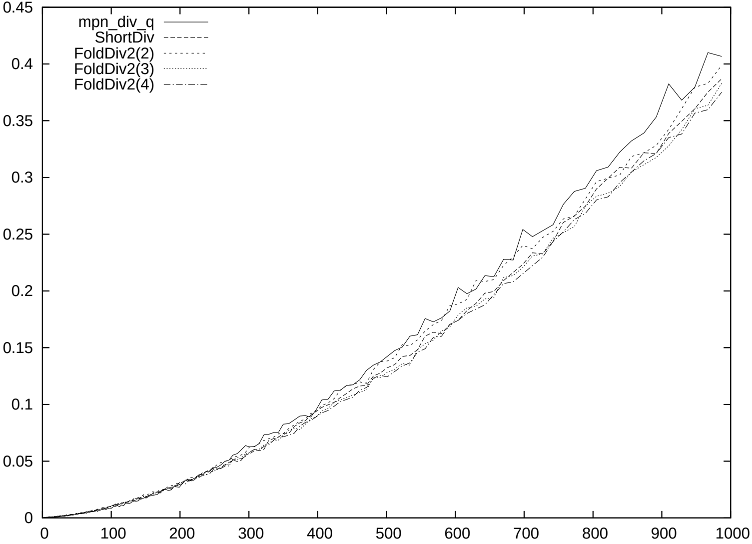
MPFR 3.0.0: 0.0058ms

MPFR 3.1.0-dev: 0.0040ms (without mulmid patch)

Short product (green) and division (red)



Short division timings



Compiler bugs found by MPFR

www.loria.fr/~zimmerma/software/compilerbugs.html

- a bug in 32-bit sparc gcc 2.95.2, when a *double* is passed as last argument of a C function, which produced Bus errors. Reported in revision 1949 of MPFR.
- a bug in GCC on m68040-unknown-netbsd1.4.1, where DBL_MIN gives $(1 - 2^{-52}) \cdot 2^{-1022}$ (rev. 2218)
- bug in LONG_MIN / 1 under FreeBSD (this is a bug of the C library of FreeBSD 5.20 on Alpha with GCC 3.3.3), reported in revision 2982 of MPFR
- bug of the Solaris memset function, revealed when testing MPFR 2.4.1 on some Solaris machines with GCC 4.4.0
- bug with the Sun C compiler with the -xO3 optimization level on sparc/Solaris, reported on August 3, 2011 [affects Sun C 5.9 SunOS_sparc Patch 124867-16 2010/08/11]
- a bug with GCC 4.3.2 (and 4.4.1) found while testing MPFR 3.1.0-rc1 on gcc54.fsffrance.org (UltraSparc IIe under Debian) with –enable-thread-safe

Efficient and machine-independent file input/output (in progress)

GNU MPFR in 2022

Efficient and machine-independent file input/output (in progress)

Companion programs: isolation and refinement of real and complex roots of a polynomial, arbitrary-precision quadrature, ...

GNU MPFR in 2022

Efficient and machine-independent file input/output (in progress)

Companion programs: isolation and refinement of real and complex roots of a polynomial, arbitrary-precision quadrature, ...

Faster internal computations with faithful rounding mode

GNU MPFR in 2022

Efficient and machine-independent file input/output (in progress)

Companion programs: isolation and refinement of real and complex roots of a polynomial, arbitrary-precision quadrature, ...

Faster internal computations with faithful rounding mode

Ball arithmetic (van der Hoeven 2011): an engineer will implement a midrad arithmetic $[m - r, m + r]$ where m has arbitrary precision, r has small precision. Cf the P1788 IEEE group about a new standard for interval arithmetic (`http://grouper.ieee.org/groups/1788/`).

GNU MPFR in 2022

Efficient and machine-independent file input/output (in progress)

Companion programs: isolation and refinement of real and complex roots of a polynomial, arbitrary-precision quadrature, ...

Faster internal computations with faithful rounding mode

Ball arithmetic (van der Hoeven 2011): an engineer will implement a midrad arithmetic $[m - r, m + r]$ where m has arbitrary precision, r has small precision. Cf the P1788 IEEE group about a new standard for interval arithmetic (`http://grouper.ieee.org/groups/1788/`).

Better deal with intermediate underflow or overflow, e.g. $\sqrt{x^2 + y^2}$

Improve robustness and efficiency of the library

Improve robustness and efficiency of the library

Generic algorithms for D-finite functions (cf work of Mezzarobba and Chevillard)

Improve robustness and efficiency of the library

Generic algorithms for D-finite functions (cf work of Mezzarobba and Chevillard)

Improve code coverage to 100% (currently 95.3% for src)

Improve robustness and efficiency of the library

Generic algorithms for D-finite functions (cf work of Mezzarobba and Chevillard)

Improve code coverage to 100% (currently 95.3% for src)

Formally prove (some of) the algorithms *implemented* in MPFR