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,
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, ...

1] type_mode? := true;
2] a:Double == 3.14159265359
3.14159265359: Double
3] exp a
23.1406926328: Double
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
3.1415926535900001: Floating
7] exp b
23.140692632784056: Floating
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
3.1415926535900001: Floating
7] exp b
23.140692632784056: Floating

8] bit_precision := 97;
9] c:Floating := exp (exp (exp 3.0))
2.050986436051648895044869200806e229520860: Alias (Floating)
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
1998: discussion with Joris van der Hoeven, Jean-Michel Muller, and Guillaume Hanrot in a café in Paris, and by mail with Torbjörn Granlund
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)
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)
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
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
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
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
2007: Mathemagix and Sage use MPFR

March 2008: GCC 4.3.0 uses MPFR in its middle-end
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
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
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)
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 \texttt{mpfr\_t} type

Each MPFR variable has:

- a precision $p \geq 2$ in bits (\texttt{long})
- a sign $s \in \{-1, 1\}$ (\texttt{int})
- an exponent $e$ (\texttt{long})
- a pointer to the significand $m$ (\texttt{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 \texttt{mpfr\_t}, 8 bytes for the significand)
Some design choices

- use of the \texttt{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)
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).
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):

$$000b_{16} \ldots b_{10} \ | \ b_9 \ldots b_0 \cdot 2^{e+3}$$
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):**

\[
\begin{array}{c|c}
000 & b_{16}\ldots b_{10} | b_9\ldots b_0 \\
\end{array}
\cdot 2^{e+3}
\]

**Base 2, left-aligned:**

\[
\begin{array}{c}
b_{16}\ldots b_7 & b_6\ldots b_0 000 \\
\end{array}
\cdot 2^e
\]
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):**

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

**Base 2, left-aligned:**

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

**Base $2^{10}$:**

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

or $$0b_{16} \ldots b_8 b_7 \ldots b_0 00 \cdot 2^{10e''}$$
Base 2, left aligned: addition $a + b$

\[
\begin{align*}
a &= 1011010111 \, 0110111000 \cdot 2^4 \\
b &= 1101101010 \, 1010101000 \cdot 2^0
\end{align*}
\]
We have to shift the smaller operand, which might need another limb:

\[
\begin{align*}
a &= \begin{array}{c}1011010111 \end{array} \begin{array}{c}0110111000 \end{array} \cdot 2^4 \\
b &= \begin{array}{c}1101101010 \end{array} \begin{array}{c}1010101000 \end{array} \cdot 2^0
\end{align*}
\]

In some cases `mpn_add` might return a carry, which will require another shift.
Base $2^w$: addition $a + b$

\[
a = \begin{array}{c}
0000001011 \\
0101110110 \\
1110000000
\end{array} \cdot 2^0
\]

\[
b = \begin{array}{c}
1101101010 \\
1010101000 \\
1010101000
\end{array} \cdot 2^0
\]

No need to shift:

\[
\begin{array}{c}
0000001011 \\
0101110110 \\
1110000000 \\
1101101010 \\
1010101000
\end{array}
\]

No post-shift needed (except in rare cases, but only limb shift).
Base 2, left aligned: multiplication $a \times b$

\[
a = \begin{array}{c|c}
1011010111 & 0110111000 \\
\hline
1101101010 & 1010101000
\end{array} \cdot 2^4
\]

\[
b = \begin{array}{c|c}
1011010111 & 0110111000 \\
\hline
1101101010 & 1010101000
\end{array} \cdot 2^0
\]

We perform a $2 \times 2$ product, and round:

\[
\begin{array}{c|c|c|c}
1001101101 & 0101100000 & 1001101100 & 0011000000
\end{array}
\]

Post-shift needed when product is 01...
Base \( 2^w \): multiplication \( a \times b \)

\[ a = \begin{array}{c} 0000001011 \\ 0101110110 \\ 1110000000 \end{array} \times 2^0 \\
\[ b = \begin{array}{c} 0110110101 \\ 0101010100 \end{array} \times 2^0 \]

We need to perform a \( 3 \times 2 \) product, and round:

\[ \begin{array}{c} 0000000100 \\ 1101101010 \\ 1100000100 \\ 1101100001 \\ 1000000000 \end{array} \]
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
Instead of flushing to zero least significant padding bits:

\[ a = \begin{array}{c} 1011010111 \\ 0110111000 \end{array} \cdot 2^4 \]

why not use them to store extra bits?

\[ a = \begin{array}{c} 1011010111 \\ 0110111101 \end{array} \cdot 2^4 \]
A clever idea?

Instead of flushing to zero least significant padding bits:

\[ a = 1011010111 \, 0110111000 \cdot 2^4 \]

why not use them to store extra bits?

\[ a = 1011010111 \, 0110111101 \cdot 2^4 \]

Not a so good idea:

- could not emulate IEEE-754 arithmetic \((p = 53)\)
- would be non-portable between \(w = 16, w = 32, w = 64, \ldots\)
$ 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)=0.24957989804940914

Paul Zimmermann GNU MPFR: back to the future
$ 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
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 division timings

Paul Zimmermann  GNU MPFR: back to the future
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 Ile under Debian) with --enable-thread-safe
Efficient and machine-independent file input/output (in progress)
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. $\sqrt{x^2+y^2}$
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