Reliable computing with GNU MPFR

Paul Zimmermann

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche NANCY - GRAND-EST

14 September 2010
International Conference on Mathematical Software
Kobe, Japan

Overview of 3rd International Workshop on Symbolic-Numeric Computation (SNC 2009), Kyoto, August 2009:

> *Algorithms that combine ideas from symbolic and numeric computation have been of increasing interest over the past decade. The growing demand for speed, accuracy and reliability in mathematical computing has accelerated the process of blurring the distinction between two areas of research that were previously quite separate. [...]*

Proceedings of ICMS 2010:

> *However, floating-point arithmetic carries a reputation of being unreliable and not trustworthy*

Overview of 3rd International Workshop on Symbolic-Numeric Computation (SNC 2009), Kyoto, August 2009:

> *Algorithms that combine ideas from symbolic and numeric computation have been of increasing interest over the past decade. The growing demand for speed, accuracy and reliability in mathematical computing has accelerated the process of blurring the distinction between two areas of research that were previously quite separate. [...]*

Proceedings of ICMS 2010:

> *However, floating-point arithmetic carries a reputation of being unreliable and not trustworthy (Rump, p. 105)*

Sign of $\sin(2^{100})$?

Maple 13:

```
> evalf(sin(2^100));
                 0.4491999480

> evalf(sin(2^100), 20);
          -0.58645356896925826300
```

Sign of $c = \exp(\pi\sqrt{163}) - 262537412640768744$?

Sage 4.5.3:

```
sage: c=exp(pi*sqrt(163))-262537412640768744
sage: numerical_approx(c, digits=15)
448.000000000000
sage: numerical_approx(c, digits=30)
-5.96855898038484156131744384766e-13
```

Mathematica 6.0:

```
In[1]:= c = Exp[Pi*Sqrt[163]]-262537412640768744;

In[2]:= N[c]

Out[2]= -480.

In[3]:= N[c, 20]

                                     -13
Out[3]= -7.4992740280181431112 10
```

FFTW

On `http://www.fftw.org/accuracy/comments.html`:

> *Our benchmark shows that certain FFT routines are more accurate than others. In other cases, a routine is accurate on one machine but not on another. [...]*

FFTW uses *trigonometric recurrences* to compute the *twiddle factors* $e^{2ik\pi/2^n}$, which are evaluated in double-extended precision (64 bits) or double precision (53 bits).

Why this chaos?

Developers/users want:

- first speed (easy to measure)
- then reliability (how to define it?)
- then reproducibility among different hardwares, compilers, operating systems

Why this chaos?

Developers/users want:

- first speed (easy to measure)
- then reliability (how to define it?)
- then reproducibility among different hardwares, compilers, operating systems

People usually prefer a to a

Why this chaos?

Developers/users want:

- first speed (easy to measure)
- then reliability (how to define it?)
- then reproducibility among different hardwares, compilers, operating systems

People usually prefer a to a

Still prefer to ?

Lessons from last millenium...

25 years ago, hardware vendors sat together and agreed on the IEEE 754 standard.

Lessons from last millenium...

25 years ago, hardware vendors sat together and agreed on the IEEE 754 standard.

Before it was a chaos. The same program gave different results on different hardware, maybe even from the same vendor!

Lessons from last millenium...

25 years ago, hardware vendors sat together and agreed on the IEEE 754 standard.

Before it was a chaos. The same program gave different results on different hardware, maybe even from the same vendor!

IEEE 754 standardizes the *double-precision* format and the corresponding operations.

Lessons from last millenium...

25 years ago, hardware vendors sat together and agreed on the IEEE 754 standard.

Before it was a chaos. The same program gave different results on different hardware, maybe even from the same vendor!

IEEE 754 standardizes the *double-precision* format and the corresponding operations.

25 years later, a given program (usually) gives only one possible result, independent of the hardware, compiler, operating system.

Lessons from last millenium...

25 years ago, hardware vendors sat together and agreed on the IEEE 754 standard.

Before it was a chaos. The same program gave different results on different hardware, maybe even from the same vendor!

IEEE 754 standardizes the *double-precision* format and the corresponding operations.

25 years later, a given program (usually) gives only one possible result, independent of the hardware, compiler, operating system.

Computer algebra systems: we are still in the chaos!

How to exit the chaos?

Ideally: follow IEEE 754-2008 extended and extendable precisions (Section 3.7)

How to exit the chaos?

Ideally: follow IEEE 754-2008 extended and extendable precisions (Section 3.7)

At least: provide a rigorous error bound for each computation (either in the documentation, or returned with the result)

```
sage: numerical_integral?
...
OUTPUT:
numerical_integral returns a tuple whose
first component is the answer and whose
second component is an error estimate.
...
sage: numerical_integral(sin(x^2),0,1)
(0.31026830172338105, 3.4446701238464278e-15)
```

GNU MPFR

Project started in 1999 with a few people (G. Hanrot, F. Rouillier, P. Z.) after discussions with J. van der Hoeven and J.-M. Muller.

Supported by INRIA and LORIA (Nancy, Lyon). Current main developers are V. Lefèvre and P. Z.

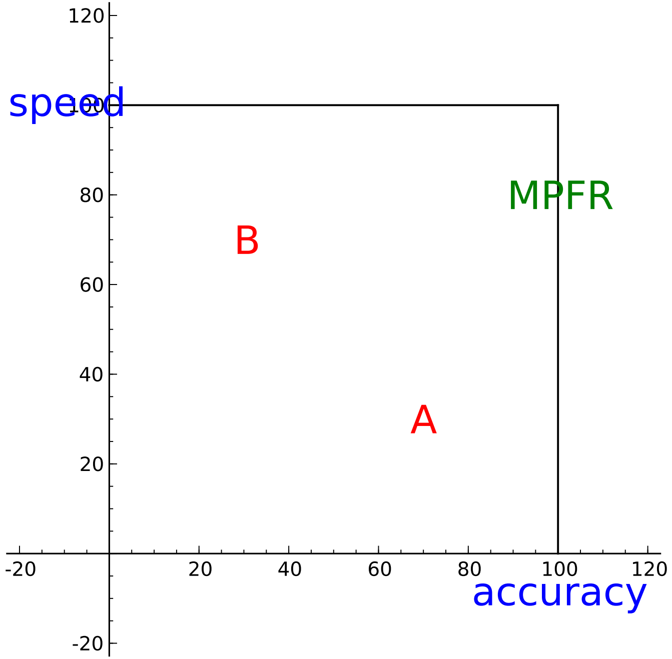
Arbitrary-precision extension of IEEE 754-1985 (implements Section 3.7 of IEEE 754-2008).

Guarantees *correct rounding* for all functions (including Erf, Zeta, Ai, ..., and base-conversion functions)

Why did we develop GNU MPFR?

Need of an arbitrary precision tool where:

- reliability is first goal
- *correct rounding* as in IEEE 754: only one correct result for each atomic computation
- reproducibility comes as extra bonus
- speed still very important, but comes after reliability



120
100
80
60
40
20
-20
speed
-20
20
40
60
80
100
120
accuracy
MPFR
B
A

How did we develop GNU MPFR?

Based on GMP's MPN class (array of unsigned long).

GMP provides efficiency (assembly), portability and reproducibility (integer types only).

MPFR does not use hardware floating-point registers (except for conversions).

Atomic vs complex operations

IEEE 754 only requests correct rounding for atomic operations.

Problem A. 2D-plot of $|\cos((x+iy)^4)| = 1$ for $-3 \leq x, y \leq 3$

Problem B. 20 decimal places of $\rho(10)$ where

$$x\rho'(x) + \rho(x-1) = 0 \qquad \text{with } \rho(x) = 1 \text{ for } 0 \leq x \leq 1$$

Problem C. 20 decimal places of the singular values of the Hilbert matrix of order 50, defined by $M_{i,j} = 1/(i+j)$

IEEE-754 alone (as MPFR) is unable to solve those problems!

Hints: interval arithmetic, RealRAM, ...

Applications of GNU MPFR in LNCS 6327

Floating-point numbers (`RealField`) of Sage (Eröcal, Stein, pages 12–27). Remember also Witty's tutorial on Sage.

Applications of GNU MPFR in LNCS 6327

Floating-point numbers (`RealField`) of Sage (Eröcal, Stein, pages 12–27). Remember also Witty's tutorial on Sage.

Evaluating dag expressions (Mörig, Friday 14:40-15:20, pages 109-120):

> *Interestingly, creating a tree is faster with MPFR [...]*

Applications of GNU MPFR in LNCS 6327

Floating-point numbers (`RealField`) of Sage (Eröcal, Stein, pages 12–27). Remember also Witty's tutorial on Sage.

Evaluating dag expressions (Mörig, Friday 14:40-15:20, pages 109-120):

> *Interestingly, creating a tree is faster with MPFR [...]*

Exact numeric computation (Yu, Yap, Du, Pion, Brönnimann, Friday 15:20-16:00, pages 121-141):

> *The timing in Figure 8 show that Core 2 is about 25 times faster, thanks purely to MPFR.*

Mathemagix (Lecerf, session Reliable Computation B, Friday 14:00, pages 329-332):

> ***numerix*** *[...] is essentially a wrapper of GMP and MPFR [...]*
> ***linalg*** *is a C++ templated version of the* LAPACK *library, that allows the user for instance to benefit of the* LAPACK *algorithms on the arbitrarily large floating-point numbers of MPFR*

Other selected applications of GNU MPFR

- constant folding in GCC and Gfortran
- companion libraries MPFI and MPC
- MPFR in Sage (also in Magma, and Mathemagix/numerix)
- real solutions of polynomial systems within Maple
- reference correctly-rounded implementation
- study of waves produced by a disc (Chiba and Ushijima)
- finding new worst approximable pairs (Briggs)
- simulate the Longxin/Longsoon processor (Joloboff)
- videos of the Mandelbrot/Julia sets (de Rauglaudre)
- ...

Constant folding in GCC (and Gfortran)

```
#include <stdio.h>
#include <math.h>
int main() { printf ("y=%.16e\n", sin(0.2522464));
```

Constant folding in GCC (and Gfortran)

```
#include <stdio.h>
#include <math.h>
int main() { printf ("y=%.16e\n", sin(0.2522464));
```

```
$ gcc f.c; ./a.out
y=2.4957989804940911e-01
```

Constant folding in GCC (and Gfortran)

```
#include <stdio.h>
#include <math.h>
int main() { printf ("y=%.16e\n", sin(0.2522464));
```

```
$ gcc f.c; ./a.out
y=2.4957989804940911e-01
```

```
$ gcc -fno-builtin f.c
/tmp/ccZTbrpq.o: In function ‘main’:
f.c:(.text+0xd): undefined reference to ‘sin’
collect2: ld returned 1 exit status
```

Constant folding in GCC (and Gfortran)

```
#include <stdio.h>
#include <math.h>
int main() { printf ("y=%.16e\n", sin(0.2522464));

$ gcc f.c; ./a.out
y=2.4957989804940911e-01

$ gcc -fno-builtin f.c
/tmp/ccZTbrpq.o: In function ‘main’:
f.c:(.text+0xd): undefined reference to ‘sin’
collect2: ld returned 1 exit status

$ gcc -fno-builtin f.c -lm; ./a.out
y=2.4957989804940914e-01
```

MPFI

Multiple-precision floating-point interval library on top of GNU MPFR.

Originally designed by N. Revol and F. Rouillier.

Each interval is represented by two MPFR numbers (inf-sup representation):

$$\ell \leq x \leq h$$

Implementing MPFI on top of MPFR is trivial for monotonic functions.

Non-trivial for non-monotonic functions: does $\cos([103992, 103993])$ contain 1?

MPFI is included within Sage (`RealIntervalField`).

MPC

Complex multiple-precision floating-point library on top of MPFR.

Developed by A. Enge, Ph. Théveny and P. Zimmermann.

Implements all functions from the C99 standard.

Required by GCC up from version 4.5 (constant folding).

Cartesian representation: $z = x + iy$.

Both x and y are correctly rounded.

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)
```

Sign of $\sin(2^{100})$?

```
#include <stdio.h>
#include <mpfr.h>
int main() {
  mpfr_t x;
  mpfr_init2 (x, 10);
  mpfr_set_ui (x, 1, MPFR_RNDN);
  mpfr_mul_2exp (x, x, 100, MPFR_RNDN);
  mpfr_sin (x, x, MPFR_RNDZ);
  mpfr_printf ("sin(2^100)=%RZf\n", x);
  mpfr_clear (x); }
```

```
$ ./test_sin
sin(2^100)=-0.872070
```

Or simply using Sage

With Sage 4.5.3 and MPFI:

```
sage: R=RealIntervalField(2)
sage: R(sin(2^100)).str(style='brackets')
'[-1.0 .. -0.75]'
```

Future plans

- improve efficiency of existing functions (e.g., `mpfr_root`)
- improve test coverage to 100% (97.1% for 3.0.0)
- better deal with different input and output precisions
- implement random distribution functions (suggested by Charles Karney)
- implement more mathematical functions (LIA-2, GSL, gnumeric, ...)
- (very hard) obtain a formal proof of say `mpfr_sub`

Can I contribute my preferred function to GNU MPFR?

Yes you can! See `http://www.mpfr.org/contrib.html`:

- define precisely the function you want to implement
- write down a detailed algorithm
- perform its error analysis (mathematical and rounding)
- implement it within MPFR
- write a test program and a documentation
- compare the efficiency of your code with other tools

Mandelbrot/Julia set (credit de Rauglaudre)