The MPFR library: well-defined semantics for multiple-precision floating-point numbers

Paul Zimmermann

INRIA Nancy - Grand Est

SAGE Days 6, November 11, 2007

Workshop on Open Source Computer Algebra

Lyon, France, May 21-23, 2002, 60 participants

- state of the art of the existing free CASs
- discussion about a collaborative effort for a free CAS/platform

What makes the strength of your system? Its weaknesses?

What development model was used? Encountered difficulties?

Why did your system succeed or fail? What about future?

Are you ready to distribute your system under an open source license?

Day 1: legal issues (copyright, patent, trade mark), TeXmacs, GINAC, GIAC, Linbox, TRIP

Day 2: OCaml, SYNAPS, Scilab, GAP, MuPAD, Magma, Axiom, Maxima, FOC, PARI, Singular

Day 3: g++, ACE, mu-EC, main discussion

Day 1: legal issues (copyright, patent, trade mark), TeXmacs, GINAC, GIAC, Linbox, TRIP

Day 2: OCaml, SYNAPS, Scilab, GAP, MuPAD, Magma, Axiom, Maxima, FOC, PARI, Singular

Day 3: g++, ACE, mu-EC, main discussion

Gabriel Dos Reis: *for a CAS to be successful, give the possibility to extend the system to the user by a sort of glue*

Day 1: legal issues (copyright, patent, trade mark), TeXmacs, GINAC, GIAC, Linbox, TRIP

Day 2: OCaml, SYNAPS, Scilab, GAP, MuPAD, Magma, Axiom, Maxima, FOC, PARI, Singular

Day 3: g++, ACE, mu-EC, main discussion

Gabriel Dos Reis: *for a CAS to be successful, give the possibility to extend the system to the user by a sort of glue*

Jacques Laskar: *I would have liked to find a system able to access types at a very low level, a light kernel with a language allowing to use it completely and to design specialized routines*

Day 1: legal issues (copyright, patent, trade mark), TeXmacs, GINAC, GIAC, Linbox, TRIP

Day 2: OCaml, SYNAPS, Scilab, GAP, MuPAD, Magma, Axiom, Maxima, FOC, PARI, Singular

Day 3: g++, ACE, mu-EC, main discussion

Gabriel Dos Reis: *for a CAS to be successful, give the possibility to extend the system to the user by a sort of glue*

Jacques Laskar: *I would have liked to find a system able to access types at a very low level, a light kernel with a language allowing to use it completely and to design specialized routines*

Bill Allombert: *Lots of developers of specialized system, make a library with a small interface to incorporate all existing stuff?*

Tim Daly: *interaction between free software and scientific publication: both can learn from the other. We need to develop methods of reviewing software and standards for publication*

Tim Daly: *interaction between free software and scientific publication: both can learn from the other. We need to develop methods of reviewing software and standards for publication*

Tim Daly: *we need standardized test suites. [...] Test suites need to be validated by mathematicians*

Tim Daly: *interaction between free software and scientific publication: both can learn from the other. We need to develop methods of reviewing software and standards for publication*

Tim Daly: *we need standardized test suites. [. . .] Test suites need to be validated by mathematicians*

Frédéric Lehobey: *The main reason I did not follow an academic carreer is that I missed a free CAS [. . .] It's only a question of time that a free CAS will exist, with or without this community.*

My List of Software Publications

- 1991: PhD in CS (advisor Flajolet), Automatic Average-Case Analysis of Algorithms, the ΛΥΩ system
- 1992: GFUN package for Maple (with B. Salvy), D-finite functions and recurrences
- 1993: Combstruct package for Maple (with E. Murray), counting and drawing combinatorial structures
- 1994-1995: contributions to the MuPAD CAS
- 1997-now: GMP-ECM, integer factorization (Elliptic Curve Method)
- 1998-now: the MPFR library (this talk)
- contributions to GMP (Toom-Cook 3-way, FFT multiplication, subquadratic division and square root, modular exponentiation, k-th integer root)

Plan of the talk

- Motivation
- The IEEE 754 standard
- The MPFR library
- Can we do better than other CAS?

Motivation

Useful Computations Need Useful Numbers
David R. Stoutemyer
ACM Communications in Computer Algebra
September 2007.

> *Most of us have taken the exact rational and approximate numbers in our computer algebra systems for granted for a long time, not thinking to ask if they could be significantly better.*

Increase the precision is enough!

```
    |\^/|     Maple 10 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2005
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                 -126
                                0.2604007480 10
```

Increase the precision is enough!

```
    |\^/|     Maple 10 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2005
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                    -126
                                   0.2604007480 10

> Digits:=20: evalf(Int(exp(-x^2)*ln(x),x=17..42));
```

Increase the precision is enough!

```
    |\^/|     Maple 10 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2005
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                              -126
                              0.2604007480 10

> Digits:=20: evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                -126
                     0.34288028340847034512 10
```

Increase the precision is enough!

```
    |\^/|     Maple 10 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2005
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                  -126
                                   0.2604007480 10

> Digits:=20: evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                  -126
                            0.34288028340847034512 10

> Digits:=50: evalf(Int(exp(-x^2)*ln(x),x=17..42));
```

Increase the precision is enough!

```
    |\^/|     Maple 10 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2005
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                    -126
                                   0.2604007480 10

> Digits:=20: evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                    -126
                         0.34288028340847034512 10

> Digits:=50: evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                                      -128
0.49076783443012876473973482836733778547443399549250 10
```

Increase the precision is enough!

```
    |\^/|     Maple 10 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2005
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                  -126
                                 0.2604007480 10

> Digits:=20: evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                             -126
                   0.34288028340847034512 10

> Digits:=50: evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                            -128
0.49076783443012876473973482836733778547443399549250 10

> Digits:=100: evalf(Int(exp(-x^2)*ln(x),x=17..42));
```

Increase the precision is enough!

```
    |\^/|     Maple 10 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2005
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                         -126
                            0.2604007480 10

> Digits:=20: evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                              -126
                     0.34288028340847034512 10

> Digits:=50: evalf(Int(exp(-x^2)*ln(x),x=17..42));
                                                            -128
0.49076783443012876473973482836733778547443399549250 10

> Digits:=100: evalf(Int(exp(-x^2)*ln(x),x=17..42));
0.4907678344301287647397348283673377854744339954925038421\
                                                          -128
    143549866501350633128535963552537205078506221 2 10
```

It makes me nervous to fly on airplanes, since I know they are designed using floating-point arithmetic.

Alston Householder

(famous architect of floating-point algorithms and error analysis)

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

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

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

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

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

0.4491999480

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

-0.58645356896925826300

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

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

> evalf(sin(2^100),30);
```

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

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

> evalf(sin(2^100),30);
           0.199885621653625738215132811525
```

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

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

> evalf(sin(2^100),30);
       0.199885621653625738215132811525

> evalf(sin(2^100),40);
```

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

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

> evalf(sin(2^100),30);
          0.199885621653625738215132811525

> evalf(sin(2^100),40);
   -0.8721836054182673097807197782134705593243
```

What does the documentation say?

```
> ?evalf
- The evalf command numerically evaluates expressions
  (or subexpressions) involving constants (for
  example, Pi, exp(1), and gamma) and mathematical
  functions (for example, exp, ln, sin, arctan, cosh,
  GAMMA, and erf).

Output
- The evalf command returns a floating-point or
  complex floating-point number or expression.
```

What does the documentation say?

```
> ?evalf
- The evalf command numerically evaluates expressions
  (or subexpressions) involving constants (for
  example, Pi, exp(1), and gamma) and mathematical
  functions (for example, exp, ln, sin, arctan, cosh,
  GAMMA, and erf).

Output
- The evalf command returns a floating-point or
  complex floating-point number or expression.

  For detailed information including:
- Complete description of all parameters
- Controlling numeric precision of computations
- Special evaluation for user-defined constants and
  functions
  see the ?evalf/details (evalf,details) help page.
```

Look at the details

```
> ?evalf/details
- The evalf command numerically evaluates expressions
  (or subexpressions) involving constants (for example,
  Pi, exp(1), and gamma) and mathematical functions
  (for example, exp, ln, sin, arctan, cosh, GAMMA, and
  erf.

- You can control the precision of all numeric
  computations using the environment variable Digits.
  By default, Digits is assigned the value 10, so the
  evalf command uses 10-digit floating-point
  arithmetic.

See Also: numeric_overview
```

Look at the numeric overview

```
> ?numeric_overview
  The Maple numeric computation environment is designed
  to achieve the following goals.

  1.    Consistency with IEEE standards.
```

Look at the numeric overview

```
> ?numeric_overview
  The Maple numeric computation environment is designed
  to achieve the following goals.

  1.   Consistency with IEEE standards.

> Digits:=3:
> Rounding := 0:
> 1.0 - 9e-5;
                         1.0
```

2. Consistency across different types of numeric computations (hardware, software, and exact).

2. Consistency across different types of numeric computations (hardware, software, and exact).

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

2. Consistency across different types of numeric computations (hardware, software, and exact).

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

> evalhf(sin(2^100));
                -0.872183605418267338
```

What are the semantics of Maple's `evalf()`, Mathematica's `N[]`, Magma's `RR()`, SAGE's `n()`?

```
-------------------------------------------------------------------
| SAGE Version 2.8.12, Release Date: 2007-11-06                    |
| Type notebook() for the GUI, and license() for information.      |
-------------------------------------------------------------------
```

```
----------------------------------------------------------------------
| SAGE Version 2.8.12, Release Date: 2007-11-06                        |
| Type notebook() for the GUI, and license() for information.          |
----------------------------------------------------------------------
sage: n?
Definition:     n(x, prec=None, digits=None)
   Return a numerical approximation of x with
   at least prec bits of precision.
```

```
----------------------------------------------------------------------
| SAGE Version 2.8.12, Release Date: 2007-11-06                      |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------
sage: n?
Definition:     n(x, prec=None, digits=None)
   Return a numerical approximation of x with
   at least prec bits of precision.

sage: f=exp(pi*sqrt(163))-262537412640768744
sage: n(f, digits=15)
-1024.00000000000
```

```
----------------------------------------------------------------------
| SAGE Version 2.8.12, Release Date: 2007-11-06                      |
| Type notebook() for the GUI, and license() for information.        |
----------------------------------------------------------------------
sage: n?
Definition:     n(x, prec=None, digits=None)
   Return a numerical approximation of x with
   at least prec bits of precision.

sage: f=exp(pi*sqrt(163))-262537412640768744
sage: n(f, digits=15)
-1024.00000000000
sage: n(f, digits=30)
-0.000000000000767386154620908200740814208984
```

```
sage: f=sin(x)^2+cos(x)^2-1
```

```
sage: f=sin(x)^2+cos(x)^2-1
sage:  f.nintegrate(x,0,1)
(-1.11896007892848996e-18, 6.44784360322443e-18,
  8379, 5)
```

```
sage: f=sin(x)^2+cos(x)^2-1
sage:  f.nintegrate(x,0,1)
(-1.11896007892848 99e-18, 6.44784360322443 4e-18,
  8379, 5)
```

```
sage: f.nintegrate?
OUTPUT:
  -- float: approximation to the integral
  -- float: estimated absolute error of the approximation
  -- the number of integrand evaluations
  -- an error code:
       0 -- no problems were encountered
       1 -- too many subintervals were done
       2 -- excessive roundoff error
       3 -- extremely bad integrand behavior
       4 -- failed to converge
       5 -- integral is probably divergent or slowly convergent
       6 -- the input is invalid
```

```
sage: f=exp(pi*sqrt(163))-262537412640768744
sage: f.nintegrate(x,0,1)
(-480.000000000000011, 5.3290705182007538e-12, 21, 0)
```

```
sage: f=exp(pi*sqrt(163))-262537412640768744
sage: f.nintegrate(x,0,1)
(-480.000000000000011, 5.3290705182007538e-12, 21, 0)
```

Is this a bug?

What you should have learned so far. . .

no specification $\Longrightarrow$ no bug

What you should have learned so far...

no specification $\Longrightarrow$ no bug

...but useless

What you should have learned so far. . .

no specification $\Longrightarrow$ no bug

. . . but useless

well-defined semantics

What you should have learned so far...

no specification $\Longrightarrow$ no bug

...but useless

well-defined semantics

$\Longrightarrow$ maybe useful

The IEEE standard

The IEEE standard

- Approved in 1985
- four different binary formats (single, single extended, double, double extended)
- four rounding modes
- requires correct rounding for $+$, $-$, $\times$, $\div$, $\sqrt{\cdot}$
- exceptions (underflow, overflow, inexact, invalid)
- well supported by modern processors

Sterbenz Lemma

Lemma *If x and y are two floating-point numbers such that $y/2 < x < 2y$, then:*

$$\circ(x - y)$$

is exact.

```
sage: R=RealField(42)
sage: x=R(catalan)
sage: y=R(euler_gamma)
sage: x, y
(0.915965594177, 0.577215664902)
sage: z=x-y
sage: z
0.338749929276
sage: x.exact_rational() - y.exact_rational()
1489837944587/4398046511104
sage: z.exact_rational()
1489837944587/4398046511104
```

FastTwoSum

Theorem *If* $|a| \geq |b|$, *and:*
$s \leftarrow \circ(a+b)$
$t \leftarrow \circ(s-a)$
$u \leftarrow \circ(b-t)$
then we have

$$a+b=s+u.$$

```
sage: R=RealField(53,sci_not=1)
sage: a=R(pi)
sage: b=R(exp(1))
sage: s=a+b
sage: t=s-a
sage: u=b-t
sage: s, u
(5.85987448204884e0, 4.44089209850063e-16)
sage: a.exact_rational()+b.exact_rational()\
     -s.exact_rational()-u.exact_rational()
0
```

The following code returns the internal base:

```
sage: A = 1.0
sage: B = 1.0
sage: while ((A + 1.0) - A) - 1.0 == 0.0:
....:       A = 2.0 * A
sage: while ((A + B) - A) - B <> 0.0:
....:       B = B + 1.0
sage: B
2.00000000000000
```

```
> A := 1.0:
> B := 1.0:
> while (evalf(A + 1.0) - A) - 1.0 = 0.0 do
     A := 2.0 * A
  od:
> while ((A + B) - A) - B <> 0.0 do
     B := B + 1.0
  od:
> B;
                              10.0
```

The MPFR library

www.mpfr.org

A lot of code involving a little floating-point will be written by many people who have ***never attended*** *my (nor anyone else's) numerical analysis classes. We had to enhance the likelihood that* ***their programs would get correct results.*** *At the same time we had to ensure that people who really are expert in floating-point could write* ***portable software*** *and prove that it worked, since so many of us would have to rely upon it. There were a lot of almost* ***conflicting requirements*** *on the way to a balanced design.*

William Kahan, An Interview with the Old Man of Floating-Point, February 1998.

History of MPFR

- Nov 1998: founding text (G. Hanrot, J.-M. Muller, J. van der Hoeven, PZ)
- early 1999: first lines of code (Hanrot, PZ)
- Dec 2000: V. Lefèvre joins the “MPFR-team”
- 2001: postdoc David Daney
- 2003-2005: Patrick Pélissier
- Oct 2004: version 2.1.0, **gfortran** uses MPFR
- Oct 2005: MPFR wins the *Many Digits* competition
- 2007: GCC 4.3 uses MPFR, version 2.3.0, article in ACM TOMS
- 2007-2009: Philippe Théveny

Computing Model

Extension of IEEE 754 to arbitrary precision:

- formats: arbitrary precision p

$$x = \pm 0.\underbrace{b_1 b_2 \ldots b_p}_{p \text{ bits}} \cdot 2^e$$

  with $E_{\min} \le e \le E_{\max}$;
- 5 special numbers ± 0, $\pm\infty$, NaN;
- rounding modes: four IEEE 754 modes.

Limits of MPFR

- radix is fixed (2);
- precision $p \geq 2$;
- $E_{\min}$ and $E_{\max}$ are global (default $E_{\min} = -2^{30} + 1$, $E_{\max} = 2^{30} - 1$);
- no denormal/subnormal numbers (but `mpfr_subnormalize`);
- operations are **atomic** (like IEEE 754).

Differences with IEEE 754

- each variable has its own precision:

```
mpfr_init2 (a, 17);
mpfr_init2 (b, 42);
```

Differences with IEEE 754

- each variable has its own precision:

```
mpfr_init2 (a, 17);
mpfr_init2 (b, 42);
```

- mixed operations are allowed:

```
mpfr_sqrt (a, b, GMP_RNDN);
```

Differences with IEEE 754

- each variable has its own precision:

```
mpfr_init2 (a, 17);
mpfr_init2 (b, 42);
```

- mixed operations are allowed:

```
mpfr_sqrt (a, b, GMP_RNDN);
```

- in-place operations are allowed:

```
mpfr_sqrt (a, a, GMP_RNDN);
```

Correct rounding

For any operation, MPFR guarantees **correct rounding**:

- basic arithmetic operations ($+, -, \times, \div$);
- algebraic functions $\sqrt{\cdot}, \sqrt{x^2+y^2}, x^n, \ldots$;
- elementary and special functions: $\exp, \log, \sin, \ldots$, erf, Bessel, …
- **conversions** (types `long`, `char *`, `double`, `long double`, `mpz_t`, `mpq_t`).

Main consequence: **one and only one** correct result!

Correct rounding

For any operation, MPFR guarantees **correct rounding**:

- basic arithmetic operations $(+, -, \times, \div)$;
- algebraic functions $\sqrt{\cdot}, \sqrt{x^2+y^2}, x^n, \ldots$;
- elementary and special functions: $\exp, \log, \sin, \ldots$, erf, Bessel, ...
- **conversions** (types `long`, `char *`, `double`, `long double`, `mpz_t`, `mpq_t`).

Main consequence: **one and only one** correct result!

Corollary 1: portability of code (across platforms **and** versions)

Correct rounding

For any operation, MPFR guarantees **correct rounding**:

- basic arithmetic operations $(+, -, \times, \div)$;
- algebraic functions $\sqrt{\cdot}, \sqrt{x^2 + y^2}, x^n, \ldots$;
- elementary and special functions: $\exp, \log, \sin, \ldots$, erf, Bessel, ...
- **conversions** (types `long`, `char *`, `double`, `long double`, `mpz_t`, `mpq_t`).

Main consequence: **one and only one** correct result!
Corollary 1: portability of code (across platforms **and** versions)
Corollary 2: directed rounding allows interval arithmetic (MPFI).

MPFR inside

3 levels of functions:

- low level (addition, subtraction, multiplication, division, square root);
- basic elementary functions $(\exp, \log, \sin)$;
- other elementary and special functions.

Low level functions

Native implementation on top of GMP's `mpn`:

$$0.\underbrace{1101011100}_{10} + 0.\underbrace{1111011110110}_{13} \cdot 2^{-3} \rightarrow \underbrace{xxxxxx}_{6} \cdot 2^{y}$$

.1101011100
    .1111011110110
.1111010111110110

Basic elementary functions

Example: exp in precision n.

- argument reduction:

$$x \to r = x/2^k$$

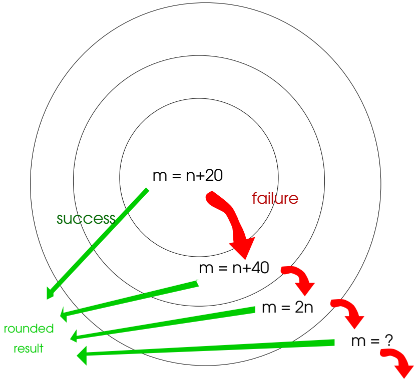
- Taylor series:

$$\exp r \approx 1 + r + \frac{r^2}{2!} + \frac{r^3}{3!} + \cdots + \frac{r^\ell}{\ell!}$$

- reconstruction:

$$x = r2^k \Longrightarrow \exp x = (\exp r)^{2^k}$$

Ziv's strategy



- if one failure, $p \leftarrow p + 32$ or $p + 64$;
- if two or more failures, $p \leftarrow p + \lfloor p/2 \rfloor$.

Other mathematical functions

Reduce to basic mathematical functions, plus Ziv's strategy;

$$\cosh x = \frac{e^x + e^{-x}}{2}$$

$$\begin{array}{lcll} u & \leftarrow & \circ(e^x) & \\ v & \leftarrow & \circ(u^{-1}) & \\ w & \leftarrow & \circ(u+v) & \\ s & \leftarrow & \frac{1}{2}w & [exact] \end{array}$$

Tests

- coverage tests
- non-regression tests (fixed bugs);
- random tests (each night):

$$\begin{array}{rcl} y & \leftarrow & \circ_p(f(x)) \\ t & \leftarrow & \circ_{p+10}(f(x)) \\ z & \leftarrow & \circ_p(t) \end{array}$$

  If no double-rounding problem, we should have $y = z$.
- data bases (warning . . .).

Efficiency (small precision)

Precision 53 bits on Pentium 4 and Athlon (cycles):

version	machine	add	sub	mul	div	sqrt
2.0.1	Pentium 4	298	398	331	1024	1211
2.1.0	Pentium 4	211	213	268	549	1084
2.0.1	Athlon	222	323	270	886	975
2.1.0	Athlon	132	151	183	477	919

Comparison MPFR, CLN, PARI, NTL

Athlon 1.8Ghz, milliseconds, $x = \sqrt{3} - 1$, $y = \sqrt{5}$:

	digits	MPFR 2.2.0	CLN 1.1.11	PARI 2.2.12-beta	NTL 5.4
$x \cdot y$	10^2	**0.00048**	0.00071	0.00056	0.00079
	10^4	**0.48**	0.81	0.58	0.57
x/y	10^2	**0.0010**	0.0013	0.0011	0.0020
	10^4	**1.2**	2.4	**1.2**	**1.2**
$\sqrt{x}$	10^2	**0.0014**	0.0016	0.0015	0.0037
	10^4	**0.81**	1.58	0.82	1.23

Comparison MPFR, CLN, PARI, NTL

	digits	MPFR 2.2.0	CLN 1.1.11	PARI 2.2.12-beta	NTL 5.4
$\exp x$	10^2	**0.017**	0.060	0.032	0.140
	10^4	**54**	70	68	1740
$\log x$	10^2	**0.031**	0.076	0.037	0.772
	10^4	**34**	79	40	17940
$\sin x$	10^2	**0.022**	0.056	0.032	0.155
	10^4	**78**	129	134	1860
$\operatorname{atan} x$	10^2	0.28	**0.067**	0.076	NA
	10^4	610	**149**	151	NA

Limits of MPFR

- roundoff error only for **atomic** operations: interval arithmetic (MPFI), RealRAM implementation (iRRAM, RealLib)

Limits of MPFR

- roundoff error only for **atomic** operations: interval arithmetic (MPFI), RealRAM implementation (iRRAM, RealLib)
- **no automatic precision setting** wrt accuracy

Limits of MPFR

- roundoff error only for **atomic** operations: interval arithmetic (MPFI), RealRAM implementation (iRRAM, RealLib)
- **no automatic precision setting** wrt accuracy
- radix is fixed to 2 (cf `decNumber` for radix 10)

Limits of MPFR

- roundoff error only for **atomic** operations: interval arithmetic (MPFI), RealRAM implementation (iRRAM, RealLib)
- **no automatic precision setting** wrt accuracy
- radix is fixed to 2 (cf `decNumber` for radix 10)
- no high-level numerical algorithms: polynomial equations, linear algebra (ALGLIB.NET), quadrature (CRQ)

Limits of MPFR

- roundoff error only for **atomic** operations: interval arithmetic (MPFI), RealRAM implementation (iRRAM, RealLib)
- **no automatic precision setting** wrt accuracy
- radix is fixed to 2 (cf `decNumber` for radix 10)
- no high-level numerical algorithms: polynomial equations, linear algebra (ALGLIB.NET), quadrature (CRQ)
- only "paper and pencil" proofs

Your first MPFR program

```
#include <stdio.h>
#include "mpfr.h"

int main ()
{
   unsigned long i;
   mpfr_t s, t;
   mpfr_init2 (s, 100); mpfr_init2 (t, 100);
   mpfr_set_ui (t, 1, GMP_RNDN);
   mpfr_set (s, t, GMP_RNDN);
   for (i = 1; i <= 29; i++)
      {
         mpfr_div_ui (t, t, i, GMP_RNDN);
         mpfr_add (s, s, t, GMP_RNDN);
      }
   mpfr_out_str (stdout, 10, 0, s, GMP_RNDN);
   printf ("\n");
   mpfr_clear (s); mpfr_clear (t);
}
```

```
#include "mpfr.h"
```

Includes the MPFR header file.

```
#include "mpfr.h"
```

Includes the MPFR header file.

```
   mpfr_t s, t;
```

Declares two variables s and t.

```
#include "mpfr.h"
```
Includes the MPFR header file.
```
   mpfr_t s, t;
```
Declares two variables *s* and *t*.
```
   mpfr_init2 (s, 100); mpfr_init2 (t, 100);
```
Initializes *s* and *t*, with a precision of 100 bits.

```
#include "mpfr.h"
```

Includes the MPFR header file.

```
   mpfr_t s, t;
```

Declares two variables *s* and *t*.

```
   mpfr_init2 (s, 100); mpfr_init2 (t, 100);
```

Initializes *s* and *t*, with a precision of 100 bits.

```
   mpfr_set_ui (t, 1, GMP_RNDN);
   mpfr_set (s, t, GMP_RNDN);
```

Sets *t* to 1, rounded to nearest, and copies *t*, rounded to nearest, into *s*.

```
mpfr_div_ui (t, t, i, GMP_RNDN);
```

Divides t by i, rounded to nearest.

```
mpfr_div_ui (t, t, i, GMP_RNDN);
```

Divides t by i, rounded to nearest.

```
mpfr_add (s, s, t, GMP_RNDN);
```

Adds t to s, with rounding to nearest.

```
mpfr_div_ui (t, t, i, GMP_RNDN);
```

Divides t by i, rounded to nearest.

```
mpfr_add (s, s, t, GMP_RNDN);
```

Adds t to s, with rounding to nearest.

```
mpfr_out_str (stdout, 10, 0, s, GMP_RNDN);
```

Prints s in decimal, with rounding to nearest (number of digits is deduced from precision of s).

```
mpfr_div_ui (t, t, i, GMP_RNDN);
```

Divides *t* by *i*, rounded to nearest.

```
mpfr_add (s, s, t, GMP_RNDN);
```

Adds *t* to *s*, with rounding to nearest.

```
mpfr_out_str (stdout, 10, 0, s, GMP_RNDN);
```

Prints *s* in decimal, with rounding to nearest (number of digits is deduced from precision of *s*).

```
mpfr_clear (s); mpfr_clear (t);
```

Frees the memory of *s* and *t*.

```
mpfr_div_ui (t, t, i, GMP_RNDN);
```

Divides t by i, rounded to nearest.

```
mpfr_add (s, s, t, GMP_RNDN);
```

Adds t to s, with rounding to nearest.

```
mpfr_out_str (stdout, 10, 0, s, GMP_RNDN);
```

Prints s in decimal, with rounding to nearest (number of digits is deduced from precision of s).

```
mpfr_clear (s); mpfr_clear (t);
```

Frees the memory of s and t.

If one replaces `GMP_RNDN` by `GMP_RNDZ`, one gets a lower bound for $e = \sum_{n \geq 0} \frac{1}{n!}$.

Compiling and running

```
$ gcc sample.c -lmpfr -lgmp
```

Compiling and running

```
$ gcc sample.c -lmpfr -lgmp
```

or:

```
$ echo $MPFR
/usr/local/mpfr-2.3.0
$ gcc -I$MPFR/include sample.c $MPFR/lib/libmpfr.a
```

Compiling and running

```
$ gcc sample.c -lmpfr -lgmp
```
or:
```
$ echo $MPFR
/usr/local/mpfr-2.3.0
$ gcc -I$MPFR/include sample.c $MPFR/lib/libmpfr.a $
```
and:
```
$ ./a.out
2.7182818284590452353602874713481
```

Applications

The CRQ library (L. Fousse): numerical quadrature with rigorous bounds

Applications

The CRQ library (L. Fousse): numerical quadrature with rigorous bounds

FPLLL (D. Stehlé): fast lattice reduction using floating-point numbers

$$\mu_{i,j} = \left\lfloor \frac{b_i b_j^*}{||b_j^*||^2} \right\rceil$$

Can we do better than other CAS?

A Challenge

Prove $\exp 1 < 3$ with your favorite CAS.

A Challenge

Prove $\exp 1 < 3$ with your favorite CAS.

```
> evalf(3-exp(1));
                    0.281718172
```

A Challenge

Prove $\exp 1 < 3$ with your favorite CAS.

```
> evalf(3-exp(1));
                      0.281718172
```

```
sage: n(3-exp(1),digits=10)
0.2817181716
```

A Challenge

Prove $\exp 1 < 3$ with your favorite CAS.

```
> evalf(3-exp(1));
                    0.281718172
```

```
sage: n(3-exp(1),digits=10)
0.2817181716
```

Is this a proof?

```
sage: R = RealIntervalField(3)
sage: 3-exp(R(1))
[-0.00 .. 0.50]
sage: R = RealIntervalField(4)
sage: 3-exp(R(1))
[0.250 .. 0.500]
```

```
sage: R = RealIntervalField(3)
sage: 3-exp(R(1))
[-0.00 .. 0.50]
sage: R = RealIntervalField(4)
sage: 3-exp(R(1))
[0.250 .. 0.500]

sage: R = RealIntervalField(105)
sage: exp(R(pi)*sqrt(R(163)))-262537412640768744
[-0.00000000000142108547152020037174224853515625 ..
 -0.000000000000582645043323282152414321899414062]
```

Still from David R. Soutemyer:

> *The astounding increase in computer speed and memory size since floating-point arithmetic was first implemented makes it affordable to use interval arithmetic and self-validating algorithms for almost all approximate scientific computation. It should be the default approximate arithmetic — especially in computer algebra [. . .]*

Let's have a dream. . .

Theorem. $e^{\pi\sqrt{163}} < 262537412640768744.$

Let's have a dream...

Theorem. $e^{\pi\sqrt{163}} < 262537412640768744$.

Proof.

```
------------------------------------------------------------
| SAGE Version 17.5.13, Release Date: 2011-11-11          |
| Type notebook() for the GUI.                             |
------------------------------------------------------------

sage: f=exp(pi*sqrt(163))-262537412640768744
sage n(f, accuracy_goal = 1 digit)
[-8e-13, -7e-13]
```