Introduction to the GNU MPFR Library

Vincent LEFÈVRE

AriC, INRIA Grenoble – Rhône-Alpes / LIP, ENS-Lyon

GdT AriC, 2014-04-22

Outline

Introduction: Arbitrary Precision

Arbitrary precision: the ability to perform calculations on numbers whose "size" can be arbitrarily large, with environmental limits (available memory, size of basic data types, etc.).

Introduction: Arbitrary Precision

Arbitrary precision: the ability to perform calculations on numbers whose "size" can be arbitrarily large, with environmental limits (available memory, size of basic data types, etc.).

Different kinds of (arbitrary-precision) arithmetics: integer (e.g., GMP/mpz), rational (e.g., GMP/mpq), fixed-point, **floating-point**, etc.

Introduction: Arbitrary Precision

Arbitrary precision: the ability to perform calculations on numbers whose "size" can be arbitrarily large, with environmental limits (available memory, size of basic data types, etc.).

Different kinds of (arbitrary-precision) arithmetics: integer (e.g., GMP/mpz), rational (e.g., GMP/mpq), fixed-point, **floating-point**, etc.

Different kinds of floating-point-based arbitrary-precision arithmetics:

- **digit-based** (floating-point arithmetic):
  - high radix (GMP/mpf: 2^{32} or 2^{64}) or small radix (e.g., 2 [MPFR] or 10) with digits packed in a basic (fixed-size) data type;
  - basic data type: integer (high or small radix) or floating-point (high radix);

Introduction: Arbitrary Precision

Arbitrary precision: the ability to perform calculations on numbers whose "size" can be arbitrarily large, with environmental limits (available memory, size of basic data types, etc.).

Different kinds of (arbitrary-precision) arithmetics: integer (e.g., GMP/mpz), rational (e.g., GMP/mpq), fixed-point, **floating-point**, etc.

Different kinds of floating-point-based arbitrary-precision arithmetics:

- **digit-based** (floating-point arithmetic):
  - high radix (GMP/mpf: 2^{32} or 2^{64}) or small radix (e.g., 2 [MPFR] or 10) with digits packed in a basic (fixed-size) data type;
  - basic data type: integer (high or small radix) or floating-point (high radix);
- floating-point *expansions*. Value x represented by a tuple $(x_1, x_2, x_3, \ldots, x_n)$ of fixed-precision floating-point numbers, with constraints like: $x_i = 0$ or $|x_i| < \mathrm{ulp}(x_{i-1})$. Value represented: $x = x_1 + x_2 + x_3 + \cdots + x_n$ (exactly). **Note:** In practice, n small (e.g. $n = 2$ for the `long double` C type of the PowerPC ABI), thus no arbitrary precision for these implementations.

What About Accuracy and Reproducibility?

About the same problems as in fixed precision, though no standards similar to IEEE 754.

Accuracy and/or reproducibility (in a language, library or application) can be documented for:

- Operations. But most often, no documented error bounds. In a fixed radix, reproducibility across platforms can be implied by correct rounding.

  Example of Maple: radix 10; the *precision* can be chosen by setting the `Digits` variable. No directed roundings, *accuracy* is *not* documented. Problem for interval arithmetic: intpak assumed a maximum error of 0.6 ulp, but counter-examples had been found. Now intpakX assumes a maximum error of 1 ulp, but is that the case? $\rightarrow$ Results are *not* guaranteed.

  Example of GMP/mpf: the radix depends on the platform (2^{32} or 2^{64}). $\rightarrow$ The results (though accurate) depend on the platform.

- Expressions (anything above operations). Usually, no dynamic error tracking. Alternatives: interval arithmetic, exact real arithmetic (see last slide).

GNU MPFR in a Few Words

- GNU MPFR is an efficient multiple-precision floating-point library with well-defined semantics (copying the good ideas from the IEEE-754 standard), in particular correct rounding.
- 80 mathematical functions, in addition to utility functions (assignments, conversions...).
- Special data (*Not a Number*, infinities, signed zeros).
- Originally developed at LORIA, INRIA Nancy – Grand Est.
  Since the end of 2006, a joint project between the INRIA project-teams Arénaire (LIP, ENS-Lyon), now AriC, and CACAO (LORIA), now Caramel.
- Written in C (ISO + optional extensions); based on GMP (mpn/mpz).
- Licence: LGPL (version 3 or later, for GNU MPFR 3).

MPFR History

1998–2000	ARC INRIA *Fiable*.
November 1998	Foundation text (Guillaume Hanrot, Jean-Michel Muller, Joris van der Hoeven, Paul Zimmermann).
Early 1999	First lines of code (G. Hanrot, P. Zimmermann).
9 June 1999	First commit into CVS (later, SVN).
2000–2002	ARC AOC (*Arithmétique des Ordinateurs Certifiée*).
February 2000	First public version.
June 2000	Copyright assigned to the Free Software Foundation.
December 2000	Vincent Lefèvre joins the MPFR team.
2003–2005	Patrick Pélissier (in particular, optimization in small precision).
2004	GNU Fortran uses MPFR (evaluation of constant expressions).
September 2005	**v2.2.0** (shared library, TLS support). [60094 lines]
October 2005	The MPFR team won the *Many Digits* Friendly Competition.

MPFR History [2]

August 2007	**v2.3.0** (shared library enabled by default). [72177 lines]
2007–2009	Philippe Théveny.
October 2007	CEA-EDF-INRIA School *Certified Numerical Computation*.
March 2008	GCC 4.3.0 release: GCC now uses MPFR in its middle-end.
January 2009	**v2.4.0** (now a GNU package). [83771 lines]
March 2009	MPFR switches to LGPL v3+ (trunk, for MPFR 3.x).
June 2009	*Certified Numerical Computation* 2 Summer School.
June 2010	**v3.0.0** (API clean-up). [90744 lines]
October 2011	**v3.1.0** (TLS enabled by default if supported). [95132 lines]
??? 2014	**v3.2.0** (mini-gmp support, export/import). [> 101600 lines]

Other contributions: Sylvie Boldo, David Daney, Mathieu Dutour, Laurent Fousse, Emmanuel Jeandel, Fabrice Rouillier, Kevin Ryde, and others.

More: `http://www.mpfr.org/history.html`

Why MPFR?

In general, exact computations on real numbers are not possible: they would be far too slow or even undecidable.

A floating-point arbitrary-precision system allows a large range and high precision.

Criteria:

- performance (time and memory);
- accuracy, correctness[1], and consistency;
- portability;
- reproducibility of the results (on different platforms, with different software).

Some compromise between the performance and the other criteria.

MPFR focuses on the last 3 criteria, while still being very efficient.

A new criterion: emulation of other arithmetics, say IEEE 754, i.e. not just for multiple precision.

[1] A $\frac{1}{2}$ ulp error bound is *not* enough, see `14.0/7.0` on old Cray machines!

Example: $\sin(10^{22})$

Environment	Computed value of $\sin 10^{22}$
Exact result	$-\mathbf{0.852200849767188801772}7\ldots$
MPFR (53 bits)	$-0.8522008497671887 9$
Glibc 2.3.6 / x86	0.4626130407646017 5
Glibc 2.3.6 / x86_64	$-0.8522008497671887 9$
Mac OS X 10.4.11 / PowerPC	$-0.8522008497790920 5$
Maple 10 (Digits = 17)	$-0.8522008497671888 0$
Mathematica 5.0 (x86?)	0.462613
MuPAD 3.2.0	-0.9873536182
HP 700	0.0
HP 375, 425t (4.3 BSD)	$-0.65365288\ldots$
Solaris/SPARC	$-0.852200849\ldots$
IBM 3090/600S-VF AIX 370	0.0
PC: Borland TurboC 2.0	4.67734e−240
Sharp EL5806	-0.090748172

Note: $10^{22} = 5^{22} \times 2^{22}$, and 5^{22} fits on 53 bits.

MPFR Program to Compute $\sin(10^{22})$

```
#include <stdio.h>   /* for mpfr_printf, before #include <mpfr.h> */
#include <assert.h>
#include <gmp.h>     /* optional, automatically done by mpfr.h */
#include <mpfr.h>

int main (void)
{
  mpfr_t x;  int inex;
  mpfr_init2 (x, 53);  /* x: 53-bit precision */
  inex = mpfr_set_ui (x, 10, MPFR_RNDN);     assert (inex == 0);
  inex = mpfr_pow_ui (x, x, 22, MPFR_RNDN);  assert (inex == 0);
  mpfr_sin (x, x, MPFR_RNDN);
  mpfr_printf ("sin(10^22) = %.17Rg\n", x);
  mpfr_clear (x);
  return 0;
}
```

Compile with: `gcc -Wall -O2 sin10p22.c -o sin10p22 -lmpfr -lgmp`

Evaluating a Sine (from Glibc)

MPFR can be useful if one cannot rely on the standard C library...

at least for testing / debugging.

Evaluating a Sine (from Glibc): 1st Test

For all tests: GCC 4.4.3 and glibc 2.10.2 under Linux/x86_64 (Debian/sid).

```
double test1 (void)
{
  double x = D;
  double x2, y;

  x2 = x;
  y = sin (x2);
  return y;
}
```

compiled with: `-O2 -DD=2.522464e-1`

Evaluating a Sine (from Glibc): 1st Test

For all tests: GCC 4.4.3 and glibc 2.10.2 under Linux/x86_64 (Debian/sid).

```
double test1 (void)
{
  double x = D;
  double x2, y;

  x2 = x;
  y = sin (x2);
  return y;
}
```

compiled with: `-O2 -DD=2.522464e-1`

Result: 0.24957989804940911016 (correct)

Evaluating a Sine (from Glibc): 2nd Test

For all tests: GCC 4.4.3 and glibc 2.10.2 under Linux/x86_64 (Debian/sid).

```
double test2 (void)
{
  volatile double x = D;
  double x2, y;

  x2 = x;
  y = sin (x2);
  return y;
}
```

compiled with: `-O2 -DD=2.522464e-1` (like `test1`)

`test1`: 0.24957989804940911016 (correct)

Evaluating a Sine (from Glibc): 2nd Test

For all tests: GCC 4.4.3 and glibc 2.10.2 under Linux/x86_64 (Debian/sid).

```
double test2 (void)
{
  volatile double x = D;
  double x2, y;

  x2 = x;
  y = sin (x2);
  return y;
}
```

compiled with: `-O2 -DD=2.522464e-1` (like `test1`)

`test1`: 0.24957989804940911016 (correct)
Result: 0.24957989804940913792

Evaluating a Sine (from Glibc): 3rd Test

```
double test3 (void)
{
  volatile double x = D, z;
  double x2, y;

  x2 = x;
  y = sin (x2);
  z = cos (x2);
  return y;
}
```

compiled with: `-O2 -DD=1e22` (note the new value of `D`)

Results:

`test1`: −0.852200849767188794 99

`test2`: −0.852200849767188794 99

Evaluating a Sine (from Glibc): 3rd Test

```
double test3 (void)
{
  volatile double x = D, z;
  double x2, y;

  x2 = x;
  y = sin (x2);
  z = cos (x2);
  return y;
}
```

compiled with: `-O2 -DD=1e22` (note the new value of `D`)

Results:

`test1`: −0.852200849767188794 99

`test2`: −0.852200849767188794 99

`test3`: 0.462613040764601746 17

Evaluating a Sine (from Glibc): The Explanations

- `test1`: The variable `x` has a constant value (and known at compile time), so does `x2`, and GCC can evaluate the expression `sin(x2)`. As of version 4.3.0, GCC uses MPFR, which provides correct rounding.

Evaluating a Sine (from Glibc): The Explanations

- `test1`: The variable `x` has a constant value (and known at compile time), so does `x2`, and GCC can evaluate the expression `sin(x2)`. As of version 4.3.0, GCC uses MPFR, which provides correct rounding.
- `test2`: Due to the `volatile` qualifier, GCC does not perform the above optimization (assuming possible side effects). The `sin()` function is called. At run time, this function is provided by the glibc math library, based (in 64-bit mode) on IBM's MathLib, which provides correct rounding.

Evaluating a Sine (from Glibc): The Explanations

- `test1`: The variable `x` has a constant value (and known at compile time), so does `x2`, and GCC can evaluate the expression `sin(x2)`. As of version 4.3.0, GCC uses MPFR, which provides correct rounding.
- `test2`: Due to the `volatile` qualifier, GCC does not perform the above optimization (assuming possible side effects). The `sin()` function is called. At run time, this function is provided by the glibc math library, based (in 64-bit mode) on IBM's MathLib, which provides correct rounding.
  But there is a bug for $0.25 < |x| < 0.855469$, due to incorrect error analysis (found by Paul Zimmermann, glibc bug 10709).

Evaluating a Sine (from Glibc): The Explanations

- `test1`: The variable `x` has a constant value (and known at compile time), so does `x2`, and GCC can evaluate the expression `sin(x2)`. As of version 4.3.0, GCC uses MPFR, which provides correct rounding.
- `test2`: Due to the `volatile` qualifier, GCC does not perform the above optimization (assuming possible side effects). The `sin()` function is called. At run time, this function is provided by the glibc math library, based (in 64-bit mode) on IBM's MathLib, which provides correct rounding.
  But there is a bug for $0.25 < |x| < 0.855469$, due to incorrect error analysis (found by Paul Zimmermann, glibc bug 10709).
- `test3`: The optimization is still not possible, but GCC notices that both `sin()` and `cos()` are called on the same value `x2` (not volatile), and calls the `sincos()` function, assuming the glibc math library will be used (indeed, `sincos()` is a GNU extension). This function, not provided by MathLib, is implemented by the `fsincos` x87 instruction.

Representation and Computation Model

Extension of the IEEE-754 standard to the arbitrary precision:

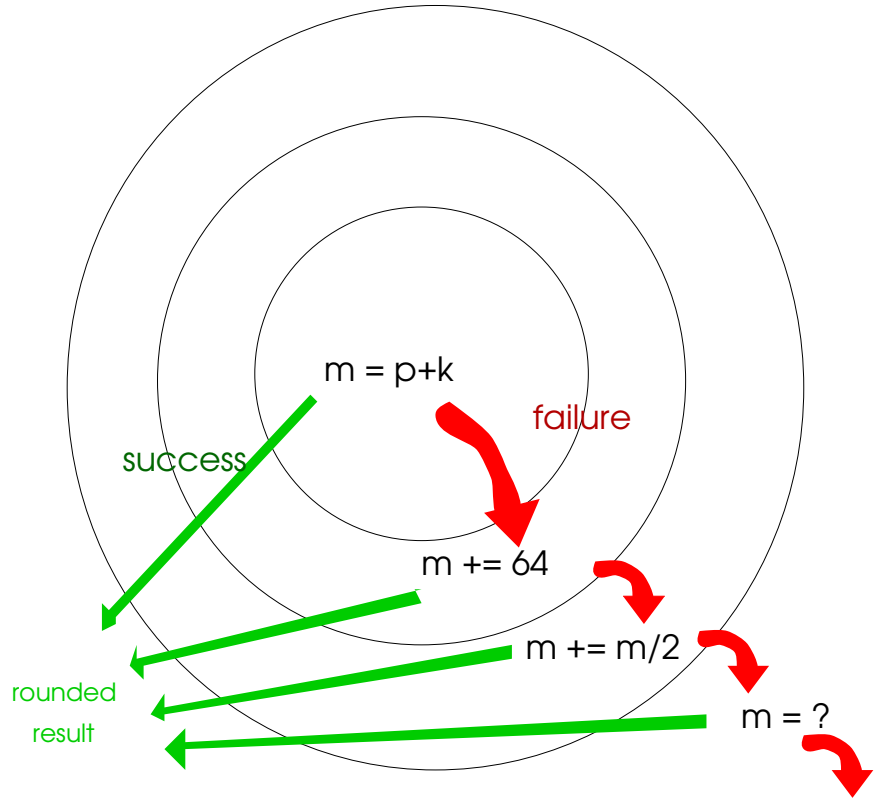
- Radix 2, precision $p \geq 2$ associated with each MPFR number.
- Format of normal numbers: $\pm 0.\underbrace{1 b_2 b_3 \ldots b_p}_{p \text{ bits}} \cdot 2^e$ with $E_{\min} \leq e \leq E_{\max}$

  ($E_{\min}$ and $E_{\max}$ are chosen by the user, $1 - 2^{30}$ and $2^{30} - 1$ by default).
- No *subnormals*, but can be emulated with `mpfr_subnormalize`.
- Special MPFR data: ± 0, $\pm\infty$, NaN (only one kind, similar to sNaN).
- Correct rounding in the 4 rounding modes of IEEE 754-1985:
  Nearest-even, Downward, Upward, toward Zero.

  Also supports: Away from zero (new in MPFR 3.0.0).
- Correct rounding in *any* precision for *any* function. More than the accuracy, needed for reproducibility of the results and for testing arithmetics.
  Note: Before MPFR 3.1.0, the results of the random functions depended on the platform (32-bit / 64-bit). This is no longer the case.

Caveats

- Correct rounding, variable precision and special numbers $\rightarrow$ noticeable overhead in very small precisions.
- Correct rounding $\rightarrow$ much slower on (mostly rare) "bad" cases (due to the *Table Maker's Dilemma*), but slightly slower in average. Ziv's strategy in MPFR:
  - first evaluate the result with slightly more precision (m) than the target (p);
  - if rounding is not possible, then $m \leftarrow m +$ (32 or 64), and recompute;
  - for the following failures: $m \leftarrow m + \lfloor m/2 \rfloor$.



- Huge exponent range and meaningful results $\rightarrow$ functions sin, cos and tan on huge arguments are very slow and take a lot of memory.

Exceptions (Global/Per-Thread Sticky Flags)

Invalid The MPFR (floating-point) result is not defined (NaN).

Ex.: $0/0$, $\log(-17)$, but also `mpfr_set` on a NaN.

DivideByZero a.k.a. Infinitary (LIA-2). An exact infinite result is defined for a function on finite operands.

Ex.: $1/\pm 0$, $\log(\pm 0)$.

Overflow The exponent of the rounded result with unbounded exponent range would be larger than $E_{\max}$.

Ex.: $2^{E_{\max}}$, and even `mpfr_set(y,x,MPFR_RNDU)` with $x = \text{nextbelow}(+\infty)$ and $\text{prec}(y) < \text{prec}(x)$.

Underflow The exponent of the rounded result with unbounded exponent range would be smaller than $E_{\min}$.

Ex.: If $E_{\min} = -17$, underflow occurs with `0.1e-17 / 2` and `0.11e-17 - 0.1e-17` (no subnormals).

Inexact The returned result is different from the exact result.

Erange Range error when the result is not a MPFR datum.

Ex.: `mpfr_get_ui` on negative value, `mpfr_cmp` on (NaN, x).

The Ternary Value

Most functions that return a MPFR number as a result (pointer passed as the first argument) also return a value of type `int`, called the *ternary value*:

- $= 0$ The value stored in the destination is exact (no rounding) or NaN.
- > 0 The value stored in the destination is greater than the exact result.
- < 0 The value stored in the destination is less than the exact result.

When not already set, the *inexact* flag is set if and only if the ternary value is nonzero.

Some Differences Between MPFR and IEEE 754

- No subnormals in MPFR, but can be emulated with `mpfr_subnormalize`.
- MPFR has only one kind of NaN (behavior is similar to signaling NaNs).
- No DivideByZero exception before MPFR 3.1.
- The Invalid exception is a bit different (see NaNs).
- Memory representation is different, but the mapping of a bit string (specified by IEEE 754) into memory is implementation-defined anyway.
- Some operations are not implemented.
- And other minor differences...

Memory Handling

- Type `mpfr_t`: `typedef __mpfr_struct mpfr_t[1];`
  - when a `mpfr_t` variable is declared, the structure is automatically allocated (the variable must still be initialized with `mpfr_init2` for the significand);
  - in a function, the pointer itself is passed, so that in `mpfr_add(a,b,c,rnd)`, the object `*a` is modified;
  - associated pointer: `typedef __mpfr_struct *mpfr_ptr;`
- MPFR numbers with more precision can be created internally.
  Warning! Possible crash in extreme cases (like in most software).
- Some MPFR functions may create caches, e.g. when computing constants such as π. Caches can be freed with `mpfr_free_cache`.
- MPFR internal data (exception flags, exponent range, caches...) are either global or per-thread (if MPFR has been built with TLS support).

Logging

When MPFR has been built with `-enable-logging` (on supported platforms), environment variables can be defined for logging:

MPFR_LOG_FILE	Name of the log file (default: `mpfr.log`).
MPFR_LOG_PREC	Number of digits of the output (default: 6).
MPFR_LOG_LEVEL	Max recursive level (default: 7).
MPFR_LOG_INPUT	Log the function input.
MPFR_LOG_OUTPUT	Log the function output.
MPFR_LOG_TIME	Log the time spent inside the function.
MPFR_LOG_INTERNAL	Log some particular variables if any.
MPFR_LOG_MSG	Log the messages if any.
MPFR_LOG_ZIV	Log what the Ziv loops do.
MPFR_LOG_STAT	Log how many times a Ziv loop failed.
MPFR_LOG_ALL	Log everything.

Output Functions

	Simple output	**Formatted output**
To file	`mpfr_out_str`	`mpfr_fprintf`, `mpfr_printf`
To string	`mpfr_get_str`	`mpfr_sprintf`
MPFR version	old	2.4.0
Locale-sensitive	yes (2.2.0)	yes
Base	2 to 36 (2.x) 2 to 62 (3.x)	2, 10, 16
Read-back exactly	yes (prec = 0)	yes[2] (empty precision field)
Efficiency	-	before MPFR 3.1.0, very slow in base 10

[2] Except for the conversion specifier g (or G) — documentation of MPFR 2.4.1 is incorrect.

Simple Output (mpfr_out_str, mpfr_get_str)

```
size_t mpfr_out_str (FILE *stream, int base, size_t n,
                     mpfr_t op, mp_rnd_t rnd)
```

Base b: from 2 to 62 (from 2 to 36 before MPFR 3.0.0).

Precision n: number of digits or 0. If $n = 0$:

- The number of digits m is chosen large enough so that re-reading the printed value with the same precision, assuming both output and input use rounding to nearest, will recover the original value of op.
- More precisely, if p is the precision of op, then $m = 1 + \lceil p. \log(2)/\log(b) \rceil$, and $m = 1 + \lceil (p-1). \log(2)/\log(b) \rceil$ when b is a power of 2 (it has been checked that these formulas are computed exactly for practical values of p). See: David W. Matula, *In-and-Out Conversions*, CACM, 1968.

Output to string: `mpfr_get_str` (on which `mpfr_out_str` is based).

Formatted Output Functions (printf-like)

Conversion specification:

```
% [flags] [width] [.[precision]] [type] [rounding] conv
```

Examples (32-bit $x \approx 10000/81 \approx 123.45679012$):

```
  mpfr_printf ("%Rf %.6RDe %.6RUe\n", x, x, x);
> 123.45679012 1.234567e+02 1.234568e+02
  mpfr_printf ("%11.1R*A\n", MPFR_RNDD, x);
>    0X7.BP+4
  mpfr_printf ("%.*Rb\n", 6, x);
> 1.111011p+6
  mpfr_printf ("%.9Rg %#.9Rg\n", x, x);
> 123.45679 123.456790
  mpfr_printf ("%#.*R*g %#.9g\n", 8, MPFR_RNDU, x, 10000./81.);
> 123.45680 123.456790
```

Test of MPFR (make check)

In the GCC development mailing-list, on 2007-12-29:

http://gcc.gnu.org/ml/gcc/2007-12/msg00707.html

```
> On 29 December 2007 20:07, Dennis Clarke wrote:
>
>>
>> Do you have a testsuite ?  Some battary of tests that can be thrown at the
>> code to determine correct responses to various calculations, error
>> conditions, underflows and rounding errors etc etc ?
>
> There's a "make check" target in the tarball.  I don't know how thorough
> it is.

That is what scares me.

Dennis
```

Test of MPFR (`make check`) [2]

Exhaustive testing is not possible.
→ Particular and generic tests (random or not).

- Complete branch coverage (or almost), but not sufficient.
- Function-specific or algorithm-specific values and other difficulties (e.g., based on bugs that have been found).
  1. Bug found in some function.
  2. Corresponding particular test added.
  3. Analysis:
     - ⋆ Reason of the bug?
     - ⋆ Can a similar bug be found somewhere else in the MPFR code (current or future)?
  4. Corresponding generic test(s) added.
- Tests with various gcc options, with valgrind.

In addition to `make check`, potential bugs detected by mpfrlint.

What Is Tested

- Special data in input or output: NaN, infinities, ± 0.
- Inputs that yield exceptions, exact cases, or midpoint cases in rounding-to-nearest.
- Discontinuity points.
- Bit patterns: for some functions (arithmetic operations, integer power), random inputs with long sequence of 0's and/or 1's.
- Thresholds: *hard-to-round cases*, underflow/overflow thresholds (currently for a few functions only).
- Extreme cases: tiny or huge input values.
- Reuse of variables (`reuse.c`), e.g. in `mpfr_exp(x,x,rnd)`.
- The influence of previous data: exception flags, sign of the output variable.
- Weird exponent range, e.g. $[17, 59]$.

The Generic Tests (`tgeneric.c`)

Basic Principle

A function is first evaluated on some input x in some target precision $p + k$, and if one can deduce the result in precision p (i.e., the TMD does not occur), then one evaluates f on the same input x in the target precision p, and compare the results.

- The precision p and the inputs are chosen randomly (in some ranges). Special values (tiny and huge inputs) can be tested too.
- Functions with 2 inputs (possibly integer) are supported.
- The exceptions are supported (with a consistency test of flags and values).
- The ternary value is checked.
- The evaluations can be performed in different flag contexts (to check the sensitivity to the flags).
- An evaluation can be redone in an extremely reduced exponent range.
- In the second evaluation, the precision of the inputs can be increased.
- The exponent range is checked at the end (bug if not restored).

Testing Bad Cases for Correct Rounding (TMD)

- Small-precision worst cases found by exhaustive search (in practice, in double precision), by using function `data_check` of `tests.c`. These worst cases are currently *not* in the repository. Each hard-to-round case is tested
  - in rounding-to-nearest, in target precision $p - 1$,
  - in all the directed rounding modes in target precision p,

  where p is the minimal precision of the corresponding *breakpoint*.
- Random hard-to-round cases (when the inverse function is implemented), using the fact that the input can have more precision than the output (function `bad_cases` of `tests.c`):
  1. A precision p_y and a MPFR number y of precision p_y are chosen randomly.
  2. One computes $x = f^{-1}(y)$ in a precision $p_x = p_y + k$.
     $\rightarrow$ In general, x is a bad case for f in precision p_y for directed rounding modes (and rounding-to-nearest for some smaller precision).
  3. One tests x in all the rounding modes (see above).

  TODO: use Newton's iteration for the other functions?

Application 1: Test of Sum Sticky-Rounded

Algorithm OddRoundedAdd

function $z = \mathrm{OddRoundedAdd}(x, y)$
$d = \mathrm{RD}(x + y)$;
$u = \mathrm{RU}(x + y)$;
$e' = \mathrm{RN}(d + u)$;
$e = e' \times 0.5$; { exact }
$z = (u - e) + d$; { exact }

This algorithm returns the sum $z = x + y$ rounded-to-odd (sticky-rounded):

- z if it is a machine number; otherwise the value among $\mathrm{RD}(z)$ and $\mathrm{RU}(z)$ whose least significant bit is a 1;
- equivalently, the significand truncated on $p - 1$ bits + sticky bit.

The corresponding MPFR instructions:

```
mpfr_add (d, x, y, MPFR_RNDD);
mpfr_add (u, x, y, MPFR_RNDU);
mpfr_add (e, d, u, MPFR_RNDN);
mpfr_div_2ui (e, e, 1, MPFR_RNDN);
mpfr_sub (z, u, e, MPFR_RNDN);
mpfr_add (z, z, d, MPFR_RNDN);
```

Application 1: Test of Sum Sticky-Rounded [2]

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <mpfr.h>

#define LIST x, y, d, u, e, z

int main (int argc, char **argv)
{
  mpfr_t LIST;
  mp_prec_t prec;
  int pprec;        /* will be prec - 1 for mpfr_printf */

  prec = atoi (argv[1]);
  pprec = prec - 1;

  mpfr_inits2 (prec, LIST, (mpfr_ptr) 0);
```

Application 1: Test of Sum Sticky-Rounded [3]

```
if (mpfr_set_str (x, argv[2], 0, MPFR_RNDN))
  {
    fprintf (stderr, "rndo-add: bad x value\n");
    exit (1);
  }
mpfr_printf ("x = %.*Rb\n", pprec, x);

if (mpfr_set_str (y, argv[3], 0, MPFR_RNDN))
  {
    fprintf (stderr, "rndo-add: bad y value\n");
    exit (1);
  }
mpfr_printf ("y = %.*Rb\n", pprec, y);
```

Application 1: Test of Sum Sticky-Rounded [4]

```
  mpfr_add (d, x, y, MPFR_RNDD);
  mpfr_printf ("d = %.*Rb\n", pprec, d);

  mpfr_add (u, x, y, MPFR_RNDU);
  mpfr_printf ("u = %.*Rb\n", pprec, u);

  mpfr_add (e, d, u, MPFR_RNDN);
  mpfr_div_2ui (e, e, 1, MPFR_RNDN);
  mpfr_printf ("e = %.*Rb\n", pprec, e);

  mpfr_sub (z, u, e, MPFR_RNDN);
  mpfr_add (z, z, d, MPFR_RNDN);
  mpfr_printf ("z = %.*Rb\n", pprec, z);

  mpfr_clears (LIST, (mpfr_ptr) 0);
  return 0;
}
```

Application 2: Test of the Double Rounding Effect

Arguments: $d_{\max}$, target precision n, extended precision p (by default, $p = n$).

Return all the pairs of positive machine numbers (x, y) such that $1/2 \leq y < 1$, $0 \leq E_x - E_y \leq d_{\max}$, $x - y$ is exactly representable in precision n and the results of $\lfloor \circ_n(\circ_p(x/y)) \rfloor$ in the rounding modes toward 0 and to nearest are different.

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

#define PRECN x, y, z  /* in precision n, t in precision p */

static unsigned long
eval (mpfr_t x, mpfr_t y, mpfr_t z, mpfr_t t, mpfr_rnd_t rnd)
{
  mpfr_div (t, x, y, rnd);  /* the division x/y in precision p */
  mpfr_set (z, t, rnd);     /* the rounding to the precision n */
  mpfr_rint_floor (z, z, rnd);  /* rnd shouldn't matter */
  return mpfr_get_ui (z, rnd);  /* rnd shouldn't matter */
}
```

Application 2: Test of the Double Rounding Effect [2]

```
int main (int argc, char *argv[])
{
  int dmax, n, p;
  mpfr_t PRECN, t;

  if (argc != 3 && argc != 4)
    { fprintf (stderr, "Usage: divworst <dmax> <n> [ <p> ]\n");
      exit (EXIT_FAILURE); }

  dmax = atoi (argv[1]);
  n = atoi (argv[2]);
  p = argc == 3 ? n : atoi (argv[3]);
  if (p < n)
    { fprintf (stderr, "p must be greater or equal to n\n");
      exit (EXIT_FAILURE); }

  mpfr_inits2 (n, PRECN, (mpfr_ptr) 0);
  mpfr_init2 (t, p);
```

Application 2: Test of the Double Rounding Effect [3]

```
  for (mpfr_set_ui_2exp (x, 1, -1, MPFR_RNDN);
       mpfr_get_exp (x) <= dmax; mpfr_nextabove (x))
    for (mpfr_set_ui_2exp (y, 1, -1, MPFR_RNDN);
         mpfr_get_exp (y) == 0; mpfr_nextabove (y))
      {
        unsigned long rz, rn;

        if (mpfr_sub (z, x, y, MPFR_RNDZ) != 0)
          continue;  /* x - y not representable in precision n */
        rz = eval (x, y, z, t, MPFR_RNDZ);
        rn = eval (x, y, z, t, MPFR_RNDN);
        if (rz != rn)
          mpfr_printf ("x = %.*Rb ; y = %.*Rb ; Z: %lu ; N: %lu\n",
                       n - 1, x, n - 1, y, rz, rn);
      }

  mpfr_clears (PRECN, t, (mpfr_ptr) 0);
  return 0;
}
```

Application 3: Continuity Test

Compute $f(1/2)$ in some given (global) precision for $f(x) = (g(x) + 1) - g(x)$ and $g(x) = \tan(\pi x)$.

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

int main (int argc, char *argv[])
{
  mp_prec_t prec;
  mpfr_t f, g;

  if (argc != 2)
    {
      fprintf (stderr, "Usage: continuity2 <prec>\n");
      exit (EXIT_FAILURE);
    }
```

Application 3: Continuity Test [2]

```
  prec = atoi (argv[1]);
  mpfr_inits2 (prec, f, g, (mpfr_ptr) 0);

  mpfr_const_pi (g, MPFR_RNDD);
  mpfr_div_2ui (g, g, 1, MPFR_RNDD);
  mpfr_tan (g, g, MPFR_RNDN);

  mpfr_add_ui (f, g, 1, MPFR_RNDN);
  mpfr_sub (f, f, g, MPFR_RNDN);
  mpfr_printf ("g(1/2) = %-17Rg  f(1/2) = %Rg\n", g, f);

  mpfr_clears (f, g, (mpfr_ptr) 0);
  return 0;
}
```

Application 3: Continuity Test [3]

```
Precision  2    g(1/2) = 16        f(1/2) = 0
Precision  3    g(1/2) = 14        f(1/2) = 2
Precision  4    g(1/2) = 14        f(1/2) = 1
Precision  5    g(1/2) = 120       f(1/2) = 0
Precision  6    g(1/2) = 120       f(1/2) = 0
Precision  7    g(1/2) = 121       f(1/2) = 1
Precision  8    g(1/2) = 2064      f(1/2) = 0
Precision  9    g(1/2) = 2064      f(1/2) = 0
Precision 10    g(1/2) = 2068      f(1/2) = 0
Precision 11    g(1/2) = 2066      f(1/2) = 2
Precision 12    g(1/2) = 2067      f(1/2) = 1
Precision 13    g(1/2) = 4172      f(1/2) = 1
Precision 14    g(1/2) = 8502      f(1/2) = 1
Precision 15    g(1/2) = 17674     f(1/2) = 1
Precision 16    g(1/2) = 38368     f(1/2) = 1
Precision 17    g(1/2) = 92555     f(1/2) = 1
Precision 18    g(1/2) = 314966    f(1/2) = 2
```

Application 3: Continuity Test [4]

```
Precision 19    g(1/2) = 314967          f(1/2) = 1
Precision 20    g(1/2) = 788898          f(1/2) = 1
Precision 21    g(1/2) = 3.18556e+06     f(1/2) = 0
Precision 22    g(1/2) = 3.18556e+06     f(1/2) = 1
Precision 23    g(1/2) = 1.32454e+07     f(1/2) = 2
Precision 24    g(1/2) = 1.32454e+07     f(1/2) = 1
Precision 25    g(1/2) = 6.29198e+07     f(1/2) = 2
Precision 26    g(1/2) = 6.29198e+07     f(1/2) = 1
Precision 27    g(1/2) = 1.00797e+09     f(1/2) = 0
Precision 28    g(1/2) = 1.00797e+09     f(1/2) = 0
Precision 29    g(1/2) = 1.00797e+09     f(1/2) = 2
Precision 30    g(1/2) = 1.00797e+09     f(1/2) = 1
Precision 31    g(1/2) = 1.64552e+10     f(1/2) = 0
Precision 32    g(1/2) = 1.64552e+10     f(1/2) = 0
Precision 33    g(1/2) = 1.64552e+10     f(1/2) = 0
Precision 34    g(1/2) = 1.64552e+10     f(1/2) = 1
Precision 35    g(1/2) = 3.90115e+11     f(1/2) = 0
```

Application 4: Proof of the Minimality of TwoSum

Based on the article *On the computation of correctly-rounded sums*, by Peter Kornerup, Vincent Lefèvre, Nicolas Louvet and Jean-Michel Muller, IEEE Transactions on Computers, 2012.

Full version on `http://hal.inria.fr/inria-00475279` [RR-7262 (2010)].

Algorithm TwoSum*

$$
\begin{aligned}
s &= RN(a+b)\\
b' &= RN(s-a)\\
a' &= RN(s-b')\\
\delta_b &= RN(b-b')\\
\delta_a &= RN(a-a')\\
t &= RN(\delta_a+\delta_b)
\end{aligned}
$$

* due to Knuth and Møller.

- Floating-point system in radix 2.
- Correct rounding in rounding to nearest.
- Two finite floating-point numbers a and b.

$\rightarrow$ Assuming no overflows, this algorithm computes two floating-point numbers s and t such that:

$$s = RN(a+b) \quad \text{and} \quad s+t = a+b.$$

Is this algorithm minimal (number of operations $+$ and $-$, and depth of the computation DAG) in any precision $p \geq 2$?

Application 4: Proof of the Minimality of TwoSum [2]

→ **Search for minimal algorithms.**

For the number of operations ($+$ and $-$):

- enumerate all the possible algorithms (DAGs labelled by the operators) with at most n operations;
- equivalent DAGs (through obvious transformations in the order of operations and sign/add/sub changes) can be ordered, thus only one DAG can be kept;
- test each algorithm on 3 well-chosen pairs of inputs in precision p by comparing the result with the correct one;
- reject the algorithm if a result does not match.

For the depth of the DAG:

- build a single DAG containing all the possible nodes at depth at most d;
- test the result of each node on 3 well-chosen pairs of inputs in precision p by comparing it with the correct one;
- take the maximum depth for which there is no match on the 3 pairs.

Application 4: Proof of the Minimality of TwoSum [3]

The number of possible precisions is infinite (or arbitrarily large).

The idea: choose the pairs of inputs in some form so that one can prove that a counter-example in one precision yields a counter-example in all (large enough) precisions.

Let us take $\varepsilon = \mathrm{ulp}(1) = 2^{1-p}$ and choose numbers of the form $u + v\varepsilon$, where u and v are small integers. Example of TwoSum on one of the pairs:

algorithm	precision 12	precision 17	expr.
a	1000.00000001	1000.0000000000001	$8 + 8\,\varepsilon$
b	1.00000000011	1.0000000000000011	$1 + 3\,\varepsilon$
$s = RN(a+b)$	1001.00000001	1001.0000000000001	$9 + 8\,\varepsilon$
$b' = RN(s-a)$	1.00000000000	1.0000000000000000	$1 + 0\,\varepsilon$
$a' = RN(s-b')$	1000.00000001	1000.0000000000001	$8 + 8\,\varepsilon$
$\delta_b = RN(b-b')$	0.00000000011	0.0000000000000011	$0 + 3\,\varepsilon$
$\delta_a = RN(a-a')$	0	0	$0 + 0\,\varepsilon$
$t = RN(\delta_a+\delta_b)$	0.00000000011	0.0000000000000011	$0 + 3\,\varepsilon$

Application 4: Proof of the Minimality of TwoSum [4]

Chosen pairs after testing various ones, where $\uparrow x$ denotes nextUp(x), i.e. the least floating-point number that compares greater than x:

$$\begin{array}{lll} a_1 = \uparrow 8 & \qquad & b_1 = \uparrow^3 1 \\ a_2 = \uparrow^5 1 & & b_2 = \uparrow 8 \\ a_3 = 3 & & b_3 = \uparrow 3 \end{array}$$

In precision $p \geq 4$, this gives:

$$\begin{array}{lll} a_1 = 8 + 8\,\varepsilon & \qquad & b_1 = 1 + 3\,\varepsilon \\ a_2 = 1 + 5\,\varepsilon & & b_2 = 8 + 8\,\varepsilon \\ a_3 = 3 & & b_3 = 3 + 2\,\varepsilon \end{array}$$

Precisions 2 to 12 (or 11) are tested. Results in precisions $p \geq 13$ can be deduced from the results in precision 12 (or 11).

Application 4: Proof of the Minimality of TwoSum [5]

Let us consider a computation DAG of maximum depth n. Here, $n = 6$. Assume that $p \geq n + 6$ (here, $p \geq 12$). We define: $\varepsilon_p = \mathrm{ulp}(1) = 2^{1-p}$. Main properties that have been proved:

- the value of any node of the DAG has the form $u + v\varepsilon_p$, where u and v are "small" integers ($|v\varepsilon_p| < 1/2$) that do not depend on the precision p;
- since the integers u and v are small enough (see the full proof), two values $u_1 + v_1\varepsilon_p$ and $u_2 + v_2\varepsilon_p$ are equal if and only if $u_1 = u_2$ and $v_1 = v_2$.

Note: we know that the depth of a minimal algorithm is bounded by 5, so that we could take $n = 5$ (but spurious algorithms might be obtained if the test is done in precision 11).

Application 4: Proof of the Minimality of TwoSum [6]

- The minimal add/sub algorithm giving the correct result is TwoSum (that is, with 6 operations); all the other equivalent algorithms reduce to TwoSum by using trivial transformations.

  Test on 33,467,556 DAGs.

- For add/sub algorithms, the depth minimality for precision $p \geq 4$ is proved by computing the 89,903,977 values of depth less or equal to 4 for each pair, in precisions 4 to 12.

  The proof of the minimality for precisions 2 and 3 needs a 4th pair to test:

  - Precision 2: $a_4 = 1$ and $b_4 = 6$.
  - Precision 3: $a_4 = 10$ and $b_4 = 1$.

  $\rightarrow$ In precision $p \geq 2$, the depth is at least 5 (depth of TwoSum).

Timings

Source: `http://www.mpfr.org/mpfr-3.1.0/timings.html`

Maple	Mathematica	Sage	GMP MPF	MPFR	PARI	NTL	CLN
commercial	commercial	GPL	LGPL	LGPL	GPL	GPL	GPL
12.00	6.0.1	4.7	5.0.2	3.1.0	2.5.0	5.5.2	1.3.2
interactive	interactive	interactive	library	library	library	library	library

100 digits	Maple	Mathematica	Sage	MPF	MPFR	Pari	NTL	CLN
mult	0.0020	0.0006	0.00056	**0.00011**	0.00013	0.00012	0.000367	0.00018
sqr			0.00051	**0.00009**	0.00010	0.00011		0.00015
div	0.0029	0.0017	0.00078	**0.00031**	**0.00031**	0.00034	0.00070	0.00049
sqrt	0.032	0.0018	0.00114	0.00056	**0.00049**	**0.00049**	0.00442	0.00067
exp	0.070	0.019	0.0098	na	**0.0073**	0.0106	0.069	0.0197
log	0.100	0.028	0.0172	na	**0.0108**	0.0117	0.386	0.0278
sin	0.131	0.017	0.0107	na	**0.0074**	0.0095	0.074	0.0253
cos	0.119	0.018	0.0075	na	**0.0054**	0.0085	0.082	0.0212
acos	0.450	0.053	0.062	na	0.045	**0.028**	na	0.033
atan	0.280	0.048	0.053	na	0.039	**0.026**	na	0.028

Timings [2]

1000 digits	Maple	Mathematica	Sage	MPF	MPFR	Pari	NTL	CLN
mult	0.0200	0.007	0.0040	0.0036	**0.0030**	0.0035	0.0137	0.0037
sqr			0.0029	0.0024	**0.0018**	0.0024		0.0026
div	0.0200	0.015	0.0070	**0.0041**	0.0048	0.0060	0.0201	0.0080
sqrt	0.160	0.011	0.0061	0.0050	**0.0047**	**0.0047**	0.187	0.0063
exp	0.90	0.63	0.196	na	**0.183**	0.364	5.96	0.332
log	0.300	0.67	0.192	na	**0.162**	0.203	48.1	0.400
sin	1.89	0.41	0.200	na	**0.167**	0.310	6.78	0.291
cos	1.91	0.40	0.187	na	**0.157**	0.300	6.98	0.266
acos	2.50	0.82	0.77	na	**0.36**	0.73	na	0.49
atan	2.10	0.80	0.69	na	**0.34**	0.72	na	0.45

10000 digits	Maple	Mathematica	Sage	MPF	MPFR	Pari	NTL	CLN
mult	0.80	0.28	0.113	0.107	**0.095**	0.108	0.508	0.106
sqr			0.086	0.076	**0.064**	0.076		0.076
div	0.80	0.56	0.267	0.198	**0.183**	0.264	1.662	0.503
sqrt	3.70	0.36	0.183	0.178	**0.176**	**0.176**	20.48	0.295
exp	50.0	17.6	9.5	na	**8.7**	12.6	1560	13.6
log	20.0	15.9	7.7	na	**7.1**	8.2	16080	16.7
sin	93.0	44.4	17.3	na	**15.4**	21.7	1650	17.6
cos	92.0	44.4	17.0	na	**15.5**	21.6	7710	16.5
acos	87.0	91.2	28.9	na	**15.2**	31.1	na	28.6
atan	82.0	87.2	26.5	na	**13.9**	31.0	na	27.0

The Future – Work in Progress

Already done in the MPFR trunk (not exhaustive):

- Mini-gmp support. [PZ]
- Export/import functions (portable and compact format).
- Operations on groups of flags (IEEE 754-2008). [VL]
- Conversions with GCC's `__float128` type. [PZ]
- Partial emulation of rounding to nearest-away. [PZ / PP]
- Assertions: static assertions / `MPFR_ASSUME` (hints). [PP]
- Better Automake 1.13 support. [VL]

Work in progress:

- Shared caches for multithreading. [PP]
- Advanced `mpfr_sum` tests (with sequences of cancellations), new specification (sign of exact zero), and new summation algorithm. [VL]

New `mpfr_sum` Implementation

`mpfr_sum`: sum of n floating-point numbers ($n \geq 0$), each of them having its own precision p_i ($0 \leq i < n$), with correct rounding on p bits.

Current implementation: naive algorithm with the conventional Ziv test.
The problem: inputs with huge differences in magnitude, huge cancellations.
$\rightarrow$ Not enough memory, or takes too much time.

Proposed algorithm (April 2014):

- Cases $n \leq 1$ handled separately.
- Addition by blocks (whose size is determined at each iteration), using the two's-complement representation.
- Use of two windows for the accumulation of partial sums, in order to limit carry propagation in some particular cases.
- Worst-case complexity in $O(np + \sum p_i)$?
- One can theoretically do better for the worst case, but this may not be interesting in practice.

New `mpfr_sum` Implementation [2]

1. In a first pass, look at the exponent field of each input (fast); also track the signs of Inf's and zeros.
   $\rightarrow$ Detect the singular cases and determine the maximum exponent $E_{\max}$ of the n' regular inputs.
2. Compute the truncated sum with `mpn`, in a window around the exponents $E_{\max} + \log_2(n')$ to $E_{\max} - p - \log_2(n')$. In the same loop over the inputs, determine the $E_{\max}$ for the next iteration (in case it is needed).
3. If applicable (see below), add both windows.
4. Determine the number of cancelled bits (by looking at the partial sum).
5. If the truncated sum is 0, reiterate at (2).
6. If the error is too large, shift the truncated sum to the left of the window, and reiterate at (2) with a second window (with p = shift count).
7. If only the sign of the error term is unknown, reiterate at (2) to compute it, using a second window where the output precision p is 0.
8. Copy the rounded result to the destination.

New `mpfr_sum` Implementation [3]

The choice of the sign of the result when it is an *exact zero*...

- Not specified by IEEE 754-2008 (not even correct rounding).
- IEEE P1788 draft: -0 in roundTowardNegative, otherwise $+0$.
  $\rightarrow$ Inconsistent with everything.

Current behavior (but unspecified):

- if $n = 0$, then the result is $+0$ (this is consistent with `mpfr_set_si/ui`);
- if all the inputs have the same sign (i.e. all $+0$ or all -0), then the result has the same sign as the inputs (for $n = 1$, this is consistent with `mpfr_set`);
- otherwise, either because all inputs are zeros with at least a $+0$ and a -0, or because some inputs are non-zero (but they cancel), then the result is $+0$.

Third rule: consistent with non-arithmetic functions, e.g. $\log(1) = +0$, but inconsistent with IEEE 754's rules for addition (-0 with `MPFR_RNDD`).

Proposed change for the third rule: -0 with `MPFR_RNDD`, otherwise $+0$, to be consistent with `mpfr_add` for $n = 2$, and get the same result as a sequence of 2-ary exact additions.

Support

- MPFR manual in info, HTML and PDF formats (if installed).
- MPFR web site: `http://www.mpfr.org/` (manual, FAQ, patches...).
- MPFR project page: `https://gforge.inria.fr/projects/mpfr/` (with Subversion repository).
- Mailing-list `mpfr@inria.fr` with
  - official archives: `https://sympa.inria.fr/sympa/arc/mpfr`;
  - Gmane mirror: `http://dir.gmane.org/gmane.comp.lib.mpfr.general`.

  25 messages per month in average.

How To Contribute to GNU MPFR

- Improve the documentation.
- Find, report and fix bugs.
- Improve the code coverage and/or contribute new test cases.
- Measure and improve the efficiency of the code.
- Contribute a new mathematical function.
  - Assign (you or your employer) the copyright of your code to the FSF.
  - Mathematical definition, specification (including the special data).
  - Choose one or several algorithms (with error analysis).
  - Implementation: conform to ISO C89, C99, and GNU Coding Standards.
  - Write a test program in `tests` (see slides on the tests).
  - Write the documentation (`mpfr.texi`), including the special cases.
  - Test the efficiency of your implementation (optional).
  - Send your contribution as a patch (obtained with `svn diff`).

More information: `http://www.mpfr.org/contrib.html`

Other Projects Based on MPFR

GNU MPFR does not track the errors, though this is partly done internally to implement correct rounding. Other software can be used for this purpose:

- Norbert Müller's C++ package **iRRAM** implements an exact real arithmetic (with limitations).
- Alternatively, interval arithmetic can be used: **MPFI**. An exact value x is represented by a pair $(\underline{x}, \overline{x})$ such that $x \in [\underline{x}, \overline{x}]$ (inf-sup representation).

For complex numbers: **GNU MPC**. Similar to MPFR, as if the real and imaginary parts were computed separately, i.e.:

- for each complex number, **2 precisions**;
- for each operation, **2 rounding modes** (packed: `MPC_RNDxy` macros, where `x` and `y` respectively refer to the rounding of the real and imaginary parts) and **2 ternary values** (also packed, extraction with `MPC_INEX_RE(`t`)` and `MPC_INEX_IM(`t`)` macros).