Introduction to the GNU MPFR Library

Vincent LEFÈVRE

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

CNC'2, LORIA, 2009-06-25

Outline

- Presentation, History
- MPFR Basics
- Output Functions
- Test of MPFR (`make check`)
- Applications
- Conclusion

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, MPFR has become a joint project between the Arénaire and CACAO project-teams.
- Written in C (ISO + optional extensions); based on GMP (mpn/mpz).
- Licence: LGPL (currently 2.1 or later).

[ecnc2009.tex 30104 2009-06-24 08:17:14Z vinc17/vin]

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).
June-July 1999	Sylvie Boldo (AGM, log).
2000–2002	ARC AOC (*Arithmétique des Ordinateurs Certifiée*).
February 2000	First public version.
March 2000	APP (*Agence pour la Protection des Programmes*) deposit.
June 2000	Copyright assigned to the Free Software Foundation.
December 2000	Vincent Lefèvre joins the MPFR team.
2001–2002	David Daney (1-year postdoc).

MPFR History [2]

2003–2005	Patrick Pélissier.
2004	GNU Fortran uses MPFR.
September 2005	MPFR 2.2.0 is released (shared library, TLS support).
October 2005	The MPFR team won the *Many Digits* Friendly Competition.
August 2007	MPFR 2.3.0 is released (shared library enabled by default).
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	GNU MPFR 2.4.0 is released (now a GNU package).
March 2009	MPFR switches to LGPL v3+ (trunk, for MPFR 3.x).

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

Representation and Computation Model

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

- Base 2, precision $p \geq 2$ associated with each MPFR number.
- Format of normal numbers: $\pm 0.\underbrace{1b_2b_3\ldots b_p}_{p \text{ bits}} \cdot 2^e$ with $E_{\min} \leq e \leq E_{\max}$

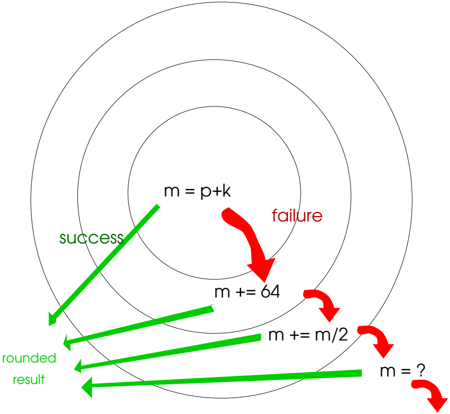
  ($E_{\min}$ and $E_{\max}$ are chosen by the user, $1 - 2^{30}$ and $2^{30} - 1$ by default).
- No subnormals, but can be emulated with `mpfr_subnormalize`.
- Special MPFR data: ± 0, $\pm\infty$, NaN (only one kind, similar to sNaN).
- Correct rounding in the 4 rounding modes of IEEE 754-1985: Nearest-even,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

  In the future MPFR 3: Away from zero.
- Correct rounding in *any* precision for *any* function. More than the accuracy, needed for reproducibility of the results and for testing arithmetics.

Caveats

- Correct rounding, variable precision and special numbers → noticeable overhead in very small precisions.

- Correct rounding → much slower on (mostly rare) “bad” cases, 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 $m \leftarrow m + (32 \text{ or } 64)$, and recompute;
    - for the following failures: $m \leftarrow m + \lfloor m/2 \rfloor$.



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

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

Environment	Computed value of $\sin 10^{22}$
Exact result	$-$ **0.8522008497671888017727...**
MPFR (53 bits)	-0.85220084976718879
Glibc 2.3.6 / x86	0.46261304076460175
Glibc 2.3.6 / x86_64	-0.85220084976718879
Mac OS X 10.4.11 / PowerPC	-0.85220084977909205
Maple 10 (Digits = 17)	-0.85220084976718880
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: 5^{22} fits on 53 bits.

[ecnc2009.tex 30104 2009-06-24 08:17:14Z vinc17/vin]

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

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

int main (void)
{
  mpfr_t x;  int inex;
  mpfr_init2 (x, 53);
  inex = mpfr_set_ui (x, 10, GMP_RNDN);     assert (inex == 0);
  inex = mpfr_pow_ui (x, x, 22, GMP_RNDN);  assert (inex == 0);
  mpfr_sin (x, x, GMP_RNDN);
  mpfr_printf ("sin(10^22) = %.17Rg\n", x);  /* new in MPFR 2.4.0 */
  mpfr_clear (x);
  return 0;
}
```

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

Exceptions (Global/Per-Thread Sticky Flags)

Invalid	Result is not defined (NaN). Examples: $0/0$, $\log(-17)$, but also `mpfr_set` on a NaN.
Overflow	The exponent of the rounded result with unbounded exponent range would be larger than $E_{\max}$. Examples: $2^{E_{\max}}$, and even `mpfr_set(y,x,GMP_RNDU)` with $x = \mathrm{nextbelow}(+\infty)$ and $\mathrm{prec}(y) < \mathrm{prec}(x)$.
Underflow	The exponent of the rounded result with unbounded exponent range would be smaller than $E_{\min}$. Examples: 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. Examples: `mpfr_get_ui` on negative value, `mpfr_cmp` on (NaN, x).

TODO: Add a DivideByZero (IEEE 754) or Infinitary (LIA-2) exception:
an exact infinite result is defined for a function on finite operands.

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.

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).

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. Invalid is a bit different (see NaNs).
- Mathematical functions on special values follow the ISO C99 standard rather than IEEE 754-2008 (more recent than the MPFR specifications).
- 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…

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, 10, 16
Read-back exactly	yes (prec = 0)	yes[1] (empty precision field)

---

[1] 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 36 (will be from 2 to 62 in MPFR 3).

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 $m = \lceil p.\log(2)/\log(b)\rceil$, and $m = \lceil (p-1).\log(2)/\log(b)\rceil$ when b is a power of 2 (it has been check that these formulas are computed exactly for practical values of p).

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

Formatted Output Functions (printf-like)

New feature in MPFR 2.4.0!

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", GMP_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, GMP_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.

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: *bad 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 (not in the generic tests yet), 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).
- 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 bad 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 bad 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 Rounded to Odd

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:

- $\mathrm{RO}(z) = z$ if z is a machine number;
- otherwise $\mathrm{RO}(z)$ is the value among $\mathrm{RD}(z)$ and $\mathrm{RU}(z)$ whose least significant bit is a one.

The corresponding MPFR instructions:

```
mpfr_add (d, x, y, GMP_RNDD);
mpfr_add (u, x, y, GMP_RNDU);
mpfr_add (e, d, u, GMP_RNDN);
mpfr_div_2ui (e, e, 1, GMP_RNDN);
mpfr_sub (z, u, e, GMP_RNDN);
mpfr_add (z, z, d, GMP_RNDN);
```

Application 1: Test of Sum Rounded to Odd [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 Rounded to Odd [3]

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

Application 1: Test of Sum Rounded to Odd [4]

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

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

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

  mpfr_sub (z, u, e, GMP_RNDN);
  mpfr_add (z, z, d, GMP_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 couples 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, GMP_RNDN);
       mpfr_get_exp (x) <= dmax; mpfr_nextabove (x))
    for (mpfr_set_ui_2exp (y, 1, -1, GMP_RNDN);
         mpfr_get_exp (y) == 0; mpfr_nextabove (y))
      {
        unsigned long rz, rn;

        if (mpfr_sub (z, x, y, GMP_RNDZ) != 0)
          continue;  /* x - y not representable in precision n */
        rz = eval (x, y, z, t, GMP_RNDZ);
        rn = eval (x, y, z, t, GMP_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, GMP_RNDD);
  mpfr_div_2ui (g, g, 1, GMP_RNDD);
  mpfr_tan (g, g, GMP_RNDN);

  mpfr_add_ui (f, g, 1, GMP_RNDN);
  mpfr_sub (f, f, g, GMP_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
```

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@loria.fr` with
  - official archives: `http://websympa.loria.fr/wwsympa/arc/mpfr`;
  - Gmane mirror: `http://dir.gmane.org/gmane.comp.lib.mpfr.general`.

  43 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`

[ecnc2009.tex 30104 2009-06-24 08:17:14Z vinc17/vin]

The Future (MPFR 3)

- Licence upgrade to LGPL version 3 or later.
- **Not compatible** with previous versions (API / ABI).
  But only minor changes.
- Rounding modes `GMP_RNDx` renamed as `MPFR_RNDx` (the old ones are still available for compatibility, but are deprecated).
- New rounding mode `MPFR_RNDA` (away from zero).
- Faithful rounding (`MPFR_RNDF`)? Probably not in MPFR 3.0.
- New functions `mpfr_buildopt_tls_p` and `mpfr_buildopt_decimal_p` giving information about options used at MPFR build time.
- Other small changes.

MPFR 3.0 planned for the end of the year.