MPFI 1.5.4.

Multiple Precision Floating-Point Interval Library
August 2018

AriC, INRIA Grenoble - Rhone-Alpes,
Spaces, INRIA Lorraine,
Arenaire, INRIA Rhone-Alpes,
Lab. ANO, USTL (Univ. of Lille)

MPFI Copying Conditions

This library is *free*; this means that everyone is free to use it and free to redistribute it on a free basis. The library is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of this library that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the library, that you receive source code or else can get it if you want it, that you can change this library or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the MPFI library, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the MPFI library. If it is modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the license for the MPFI library are found in the Lesser General Public License that accompany the source code. As MPFI is built upon MPFR and share its license, see the file COPYING.LESSER in the main MPFR directory.

1 Introduction to MPFI

MPFI is intended to be a portable library written in C for arbitrary precision interval arithmetic with intervals represented using MPFR reliable floating-point numbers. It is based on the GNU MP library and on the MPFR library. The purpose of an arbitrary precision interval arithmetic is on the one hand to get *guaranteed* results, thanks to interval computation, and on the other hand to obtain accurate results, thanks to multiple precision arithmetic. The MPFI library is built upon MPFR in order to benefit from the correct rounding provided, for each operation or function, by MPFR. Further advantages of using MPFR are its portability and compliance with the IEEE 754 standard for floating-point arithmetic.

This version of MPFI is released under the GNU Lesser General Public License. It is permitted to link MPFI to non-free programs, as long as when distributing them the MPFI source code and a means to re-link with a modified MPFI is provided.

As interval arithmetic currently undergoes standardization, through the work of the IEEE-1788 working group, future versions of MPFI may evolve in order to reflect the standardized definitions and behaviours.

2 Installing MPFI

To build MPFI, you first have to install MPFR (version 4.0.1 or above) on your computer. You need a C compiler, preferably GCC, but any reasonable compiler should work. And you need a standard Unix '`make`' program, plus some other standard Unix utility programs.

2.1 How to Install

Here are the steps needed to install the MPFI library on Unix systems. In the MPFI source directory, type the following commands.

1. '`./configure`'

   This will prepare the build and setup the options according to your system. You can give options to specify the install directories (instead of the default `/usr/local`), and so on.

   You can specify the path to GMP and MPFR libraries with configure options: '`--with-gmp=DIR`' assumes that GMP is installed in the '`DIR`' directory. Alternatively, you can use the '`--with-gmp-lib=DIR`' and '`--with-gmp-include=DIR`' to specify respectively the GMP lib and GMP include directories. Options '`--with-mpfr=DIR`', '`--with-mpfr-include=DIR`', and '`--with-mpfr-lib=DIR`' have the same usage for the MPFR library.

   See the `INSTALL` file and the output of '`./configure --help`' for a description of standard options.

2. '`make`'

   This will compile MPFI, and create a library archive file `libmpfi.a`. On most platforms, a dynamic library will be produced too.

3. '`make check`'

   This will make sure MPFI was built correctly. If any test fails, information about this failure can be found in the `tests/test-suite.log` file. If you want the contents of this file to be automatically output in case of failure, you can set the '`VERBOSE`' environment variable to 1 before running '`make check`', for instance by typing:

   '`VERBOSE=1 make check`'

   If you get error messages from the test program, please report this to '`mpfi-users@lists.gforge.inria.fr`'. (See See Chapter 3 [Reporting Bugs], page 5, for information on what to include in useful bug reports.)

4. '`make install`'

   This will copy the file `mpfi.h` to the directory `/usr/local/include`, the library files (`libmpfi.a` and possibly others) to the directory `/usr/local/lib`, the file `mpfi.info` to the directory `/usr/local/share/info`, and some other documentation files to the directory `/usr/local/share/doc/mpfi` (or if you passed the '`--prefix`' option to `configure`, using the prefix directory given as argument to '`--prefix`' instead of `/usr/local`).

2.2 Other 'make' targets

There are some other useful make targets:

- '`info`'

  Create an info version of the manual, in `mpfi.info`.

  This file is already provided in the MPFI archives.

- '`pdf`'

  Create a PDF version of the manual, in `mpfi.pdf`.

- ‘`dvi`’
  Create a DVI version of the manual, in `mpfi.dvi`.
- ‘`ps`’
  Create a Postscript version of the manual, in `mpfi.ps`.
- ‘`html`’ Create a HTML version of the manual, in `mpfi.html`.
- ‘`clean`’
  Delete all object files and archive files, but not the configuration files.
- ‘`uninstall`’ Delete all files copied by ‘`make install`’.

2.3 Known Build Problems

The installation procedure and MPFI itself have been tested only on some Linux distributions. Since it has not been intensively tested, you may discover that MPFI suffers from all bugs of the underlying libraries, plus many many more.

Please report any problem to ‘`mpfi-users@lists.gforge.inria.fr`’. See Chapter 3 [Reporting Bugs], page 5.

2.4 Getting the Latest Version of MPFI

The latest version of MPFI is available from `https://gforge.inria.fr/frs/?group_id=157`.

3 Reporting Bugs

If you think you have found a bug in the MPFI library, please investigate it and report it. We have made this library available to you, and we expect you will report the bugs that you find.

There are a few things you should think about when you put your bug report together.

You have to send us a test case that makes it possible for us to reproduce the bug. Include instructions on how to run the test case.

You also have to explain what is wrong; if you get a crash, or if the results printed are incorrect and in that case, in what way.

Please include compiler version information in your bug report. This can be extracted using '`cc -V`' on some machines or, if you're using gcc, '`gcc -v`'. Also, include the output from '`uname -a`', along with the version of GMP and of MPFR you use.

If your bug report is good, we will do our best to help you to get a corrected version of the library; if the bug report is poor, we won't do anything about it (except kidding you for sending poor bug reports).

Send your bug report to: '`mpfi-users@lists.gforge.inria.fr`'.

If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please send a note to the same address.

4 MPFI Basics

All declarations needed to use MPFI are collected in the include file `mpfi.h`. The declarations useful for inputs and outputs are to be found in `mpfi_io.h`. It is designed to work with both C and C++ compilers. You should include these files in any program using the MPFI library:

```
#include "mpfi.h"
#include "mpfi_io.h"
```

4.1 Nomenclature and Types

As MPFI is built upon MPFR, it is advisable to read MPFR's manual first.

An *interval* is a closed connected set of real numbers, it is represented in MPFI by its endpoints which are MPFR floating-point numbers. The C data type for these objects is `mpfi_t`.

MPFI functions operate on valid intervals (defined below), their behavior with non-valid intervals as input is undefined.

- A *valid interval* can have finite or infinite endpoints, but its left endpoint is not larger than its right endpoint and cannot be +infinity or -0 (respectively, the right endpoint cannot be -infinity or +0).[1] As a consequence, the unique representation of the zero singleton is $[+0, -0]$.

MPFI functions may return intervals that are not valid as input value. Their semantic defined as follows:

- One (or both) NaN endpoint(s) indicates that an invalid operation has been performed and that the resulting interval has no mathematical meaning.
- An *empty interval* has its left endpoint larger than its right endpoint.

Both the meaning of "invalid operation", the representation of the empty set and its handling may change in future versions of MPFI, according to the standardization of interval arithmetic in IEEE-1788.

Some functions on intervals return a floating-point value: among such functions are `mpfi_get_left` that returns the left endpoint of an interval and `mpfi_diam_abs` that gets the width of the input interval.

A *Floating point number* or *Float* for short, is an arbitrary precision significand (aslso called mantissa) with a limited precision exponent. The C data type for such objects is `mpfr_t`.

The *Precision* is the number of bits used to represent the significand of a floating-point number; the corresponding C data type is `mpfr_prec_t` (renamed `mpfr_prec_t` since MPFR version 3.0.0, both types are compatible).

MPFI assumes that both endpoints of an interval use the same precision. However when this does not hold, the largest precision is considered.

4.2 Function Classes

There is only one class of functions in the MPFI library:

[1] The restriction on the infinite values follows the definition of interval, and the sign of the zero bounds allows a simple implementation of the four arithmetic operations as explained in the paper of T. Hickey, Q. Ju, and M. H. Van Emden, *Interval arithmetic: From principles to implementation* (See [References], page 22).

1. Functions for interval arithmetic based on floating-point numbers, with names beginning with `mpfi_`. The associated type is `mpfi_t`. There are around 170 functions in this class.

4.3 MPFI Variable Conventions

As a general rule, all MPFI functions expect output arguments before input arguments. This notation is based on an analogy with the assignment operator.

MPFI allows you to use the same variable for both input and output in the same expression. For example, the function for the exponential, `mpfi_exp`, can be used like this: `mpfi_exp (x, x)`. This computes the set of exponentials of every real belonging to x and puts the result back in x.

Before you can give a value to an MPFI variable, you need to initialize it by calling one of the special initialization functions. When you're done with a variable, you need to clear it out, using one of the appropriate functions.

A variable should be initialized only once, or at least be cleared out between different initializations. After a variable has been initialized, it can be assigned any number of times.

For efficiency reasons, avoid to initialize and clear out a variable in loops. Instead, initialize it before entering the loop, and clear it out after exiting the loop.

You don't need to be concerned about allocating additional space for MPFI variables, since any variable uses a memory space of fixed size. Hence unless you change its precision, or clear and reinitialize it, an interval variable will have the same allocated space during all its lifetime.

5 Interval Functions

The interval functions expect arguments of type `mpfi_t`.

The MPFI interval functions have an interface that is close to the corresponding MPFR functions. The function prefix for interval operations is `mpfi_`.

MPFI intervals are represented by their endpoints; this representation should be invisible to the user, unfortunately it is not... It is assumed that both endpoints have the same precision; however when this does not hold, the largest precision is considered. The user has to specify the precision of each variable. A computation that assigns a variable will take place with the precision of the assigned variable. For more information on precision (precision of a variable, precision of a calculation), see the MPFR documentation.

5.1 Return Values

Four integer values (of C type `int`) can be returned by a typical `mpfi` function. These values indicate whether none, one or two endpoints of the computed interval are exact: since they are rounded values, they can differ from the exact result. Here are their names:

- `MPFI_FLAGS_BOTH_ENDPOINTS_EXACT`
- `MPFI_FLAGS_LEFT_ENDPOINT_INEXACT`: the left endpoint is inexact whereas the right endpoint is exact;
- `MPFI_FLAGS_RIGHT_ENDPOINT_INEXACT`: the right endpoint is inexact whereas the left endpoint is exact;
- `MPFI_FLAGS_BOTH_ENDPOINTS_INEXACT`

To test the exactness of one endpoint, the following functions are available (their names are self-explanatory):

- `MPFI_BOTH_ARE_EXACT`
- `MPFI_LEFT_IS_INEXACT`
- `MPFI_RIGHT_IS_INEXACT`
- `MPFI_BOTH_ARE_INEXACT`

5.2 Precision Handling

The default computing precision is handled by MPFR, getting or setting its value is performed using the following MPFR functions (cf. MPFR documentation):

`void mpfr_set_default_prec (`*mpfr_prec_t prec*`)` [Macro]
Sets the default precision to be **exactly** *prec* bits. The precision of a variable means the number of bits used to store the significands of its endpoints. All subsequent calls to `mpfi_init` will use this precision, but previously initialized variables are unaffected. This default precision is set to 53 bits initially. The precision *prec* can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`.

Note: when MPFR is built with the '`--enable-thread-safe`' configure option, the default precision is local to each thread. See MPFR documentation for more information.

`mpfr_prec_t mpfr_get_default_prec ()` [Macro]
Returns the default MPFR/MPFI precision in bits.

The following two functions are useful for changing the precision during a calculation. A typical use would be for adjusting the precision gradually in iterative algorithms like Newton-Raphson, making the computation precision closely match the actual accurate part of the numbers.

`void mpfi_set_prec` (*mpfi_t x, mpfr_prec_t prec*) [Function]
Resets the precision of *x* to be **exactly** *prec* bits. The previous value stored in *x* is lost. It is equivalent to a call to `mpfi_clear(x)` followed by a call to `mpfi_init2(x, prec)`, but more efficient as no allocation is done in case the current allocated space for the significands of the endpoints of *x* is enough. The precision *prec* can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`. In case you want to keep the previous value stored in *x*, use `mpfi_round_prec` instead.

`mpfr_prec_t mpfi_get_prec` (*mpfi_t x*) [Function]
Return the largest precision actually used for assignments of *x*, i.e.\ the number of bits used to store the significands of its endpoints. Should the two endpoints have different precisions, the largest one is returned.

`int mpfi_round_prec` (*mpfi_t x, mpfr_prec_t prec*) [Function]
Rounds *x* with precision *prec*, which may be different from that of *x*. If *prec* is greater or equal to the precision of *x*, then new space is allocated for the endpoints' significands, and they are filled with zeroes. Otherwise, the significands are rounded outwards to precision *prec*. In both cases, the precision of *x* is changed to *prec*. It returns a value indicating whether the possibly rounded endpoints are exact or not, cf. Section 5.1 [Return Values], page 8.

5.3 Initialization and Assignment Functions

5.3.1 Initialization Functions

An `mpfi_t` object must be initialized before storing the first value in it. The functions `mpfi_init` and `mpfi_init2` are used for that purpose.

`void mpfi_init` (*mpfi_t x*) [Function]
Initializes *x*, and sets its value to NaN, to prevent from using an unassigned variable inadvertently. Normally, a variable should be initialized once only or at least be cleared, using `mpfi_clear`, between consecutive initializations. The precision of *x* is the default precision, which can be changed by a call to `mpfr_set_default_prec`.

Warning! In a given program, some other libraries might change the default precision and not restore it. Thus it is safer to use `mpfi_init2`.

`void mpfi_init2` (*mpfi_t x, mpfr_prec_t prec*) [Function]
Initializes *x*, sets its precision (or more precisely the precision of its endpoints) to be **exactly** *prec* bits, and sets its endpoints to NaN. Normally, a variable should be initialized once only or at least be cleared, using `mpfi_clear`, between consecutive initializations. To change the precision of a variable which has already been initialized, use `mpfi_set_prec` instead, or `mpfi_round_prec` if you want to keep its value.

`void mpfi_clear` (*mpfi_t x*) [Function]
Frees the space occupied by the significands of the endpoints of *x*. Make sure to call this function for all `mpfi_t` variables when you are done with them.

Here is an example on how to initialize interval variables:

```
{
  mpfi_t x, y;
  mpfi_init (x); /* use default precision */
  mpfi_init2 (y, 256); /* precision exactly 256 bits */
  ...
  /* Unless the program is about to exit, do ... */
  mpfi_clear (x);
  mpfi_clear (y);
}
```

`void mpfi_inits (`*mpfi_t x, ...*`)` [Function]
Initialize all the `mpfi_t` variables of the given `va_list`, set their precision to the default precision and their value to NaN. See `mpfi_init` for more details. The `va_list` is assumed to be composed only of type `mpfi_t` (or equivalently `mpfi_ptr`). It begins from *x*, and ends when it encounters a null pointer (whose type must also be `mpfi_ptr`).

Warning! In a given program, some other libraries might change the default precision and not restore it. Thus it is safer to use `mpfi_inits2`.

`void mpfi_inits2 (`*mpfr_prec_t prec, mpfi_t x, ...*`)` [Function]
Initialize all the `mpfi_t` variables of the given variable argument `va_list`, set their precision to be **exactly** *prec* bits and their value to NaN. See `mpfi_init2` for more details. The `va_list` is assumed to be composed only of type `mpfi_t` (or equivalently `mpfi_ptr`). It begins from *x*, and ends when it encounters a null pointer (whose type must also be `mpfi_ptr`).

`void mpfi_clears (`*mpfi_t x, ...*`)` [Function]
Free the space occupied by all the `mpfi_t` variables of the given `va_list`. See `mpfi_clear` for more details. The `va_list` is assumed to be composed only of type `mpfi_t` (or equivalently `mpfi_ptr`). It begins from *x*, and ends when it encounters a null pointer (whose type must also be `mpfi_ptr`).

Here is an example of how to use multiple initialization functions (since `NULL` is not necessarily defined in this context, we use `(mpfi_ptr) 0` instead, but `(mpfi_ptr) NULL` is also correct).

```
{
  mpfi_t a, b, c;

  mpfi_inits (a, b, c, (mpfi_ptr) 0); /* Use default precision */
  ...
  mpfi_clears (a, b, c, (mpfi_ptr) 0);

  /* Another possibility is to specify the precision */
  mpfi_inits2 (200, a, b, c, (mpfi_ptr) 0); /* Use exactly 200 bits */
  ...
  mpfi_clears (a, b, c, (mpfi_ptr) 0);
}
```

5.3.2 Assignment Functions

These functions assign new values to already initialized intervals (see Section 5.3.1 [Initializing Intervals], page 9).

`int mpfi_set` (*mpfi_t* `rop`, *mpfi_t* `op`) [Function]
`int mpfi_set_ui` (*mpfi_t* `rop`, *unsigned long int* `op`) [Function]
`int mpfi_set_si` (*mpfi_t* `rop`, *long int* `op`) [Function]
`int mpfi_set_d` (*mpfi_t* `rop`, *double* `op`) [Function]
`int mpfi_set_flt` (*mpfi_t* `rop`, *float* `op`) [Function]
`int mpfi_set_ld` (*mpfi_t* `rop`, *long double* `op`) [Function]
`int mpfi_set_z` (*mpfi_t* `rop`, *mpz_t* `op`) [Function]
`int mpfi_set_q` (*mpfi_t* `rop`, *mpq_t* `op`) [Function]
`int mpfi_set_fr` (*mpfi_t* `rop`, *mpfr_t* `op`) [Function]
Sets the value of *rop* from *op*, rounded outward to the precision of *rop*: *op* then belongs to *rop*. The returned value indicates whether none, one or both endpoints are exact. Please note that even a `long int` may have to be rounded, if the destination precision is less than the machine word width.

No support (yet) for huge (signed or unsigned) integer, nor for float128, nor for decimal64.

`int mpfi_set_str` (*mpfi_t* `rop`, *char* `*s`, *int* `base`) [Function]
Sets *rop* to the value of the string *s*, in base *base* (between 2 and 36), outward rounded to the precision of *rop*: *op* then belongs to *rop*. The exponent is read in decimal. The string is of the form '`number`' or '`[ number1 , number 2 ]`'. Each endpoint has the form '`M@N`' or, if the base is 10 or less, alternatively '`MeN`' or '`MEN`'. '`M`' is the significand and '`N`' is the exponent. The significand is always in the specified base. The exponent is in decimal. The argument *base* may be in the ranges 2 to 36.

This function returns 1 if the input is incorrect, and 0 otherwise.

`void mpfi_swap` (*mpfi_t* `x`, *mpfi_t* `y`) [Function]
Swaps the values *x* and *y* efficiently. Warning: the precisions are exchanged too; in case the precisions are different, `mpfi_swap` is thus not equivalent to three `mpfi_set` calls using a third auxiliary variable.

5.3.3 Combined Initialization and Assignment Functions

`int mpfi_init_set` (*mpfi_t* `rop`, *mpfi_t* `op`) [Function]
`int mpfi_init_set_ui` (*mpfi_t* `rop`, *unsigned long int* `op`) [Function]
`int mpfi_init_set_si` (*mpfi_t* `rop`, *long int* `op`) [Function]
`int mpfi_init_set_d` (*mpfi_t* `rop`, *double* `op`) [Function]
`int mpfi_init_set_z` (*mpfi_t* `rop`, *mpz_t* `op`) [Function]
`int mpfi_init_set_q` (*mpfi_t* `rop`, *mpq_t* `op`) [Function]
`int mpfi_init_set_fr` (*mpfi_t* `rop`, *mpfr_t* `op`) [Function]
Initializes *rop* and sets its value from *op*, outward rounded so that *op* belongs to *rop*. The precision of *rop* will be taken from the active default precision, as set by `mpfr_set_default_prec`. The returned value indicates whether none, one or both endpoints are exact.

`int mpfi_init_set_str` (*mpfi_t* `rop`, *char* `*s`, *int* `base`) [Function]
Initializes *rop* and sets its value to the value of the string *s*, in base *base* (between 2 and 36), outward rounded to the precision of *rop*: *op* then belongs to *rop*. The exponent is read in decimal. See `mpfi_set_str`.

5.4 Interval Functions with Floating-point Results

Some functions on intervals return floating-point results, such as the center or the width, also called diameter, of an interval.

`int mpfi_diam_abs` (*mpfr_t rop, mpfi_t op*) [Function]
Sets the value of *rop* to the upward rounded diameter of *op*, or in other words to the upward rounded difference between the right endpoint of *op* and its left endpoint. Returns 0 if the diameter is exact and a positive value if the rounded value is greater than the exact diameter.

`int mpfi_diam_rel` (*mpfr_t rop, mpfi_t op*) [Function]
Sets the value of *rop* to the upward rounded relative diameter of *op*, or in other words to the upward rounded difference between the right endpoint of *op* and its left endpoint, divided by the absolute value of the center of *op* if it is not zero. Returns 0 if the result is exact and a positive value if the returned value is an overestimation, in this case the returned value may not be the correct rounding of the exact value.

`int mpfi_diam` (*mpfr_t rop, mpfi_t op*) [Function]
Sets the value of *rop* to the relative diameter of *op* if *op* does not contain zero and to its absolute diameter otherwise. Returns 0 if the result is exact and a positive value if the returned value is an overestimation, it may not be the correct rounding of the exact value in the latter case.

`int mpfi_mag` (*mpfr_t rop, mpfi_t op*) [Function]
Sets the value of *rop* to the magnitude of *op*, i.e.\ to the largest absolute value of the elements of *op*. Returns 0 if the result is exact and a positive value if the returned value is an overestimation.

`int mpfi_mig` (*mpfr_t rop, mpfi_t op*) [Function]
Sets the value of *rop* to the mignitude of *op*, i.e.\ to the smallest absolute value of the elements of *op*. Returns 0 if the result is exact and a negative value if the returned value is an underestimation.

`int mpfi_mid` (*mpfr_t rop, mpfi_t op*) [Function]
Sets *rop* to the middle of *op*. Returns 0 if the result is exact, a positive value if *rop* > the middle of *op* and a negative value if *rop* < the middle of *op*.

`void mpfi_alea` (*mpfr_t rop, mpfi_t op*) [Function]
Sets *rop* to a floating-point number picked up at random in *op*, according to a uniform distribution.

This function is deprecated and may disappear in future versions of MPFI; `mpfi_urandom` should be used instead.

`void mpfi_urandom` (*mpfr_t rop, mpfi_t op, gmp_randstate_t state*) [Function]
Sets *rop* to a floating-point number picked up at random in *op*, according to a uniform distribution.

The argument *state* should be initialized with one of the GMP random state initialization functions (see Section "Random State Initialization" in *GNU MP* manual).

`void mpfi_nrandom` (*mpfr_t rop, mpfi_t op, gmp_randstate_t state*) [Function]
Sets *rop* to a floating-point number picked up at random in *op*, according to a normal distribution.

The argument *state* should be initialized as for `mpfi_urandom`.

Caveat: the normal distribution on the set of floating-point numbers is different from the normal distribution on the set of real numbers. No guarantee is given on the quality of the distribution.

`void mpfi_erandom` (*mpfr_t rop, mpfi_t op, gmp_randstate_t state*) [Function]
Sets *rop* to a floating-point number picked up at random in *op*, according to an exponential distribution.

The argument *state* should be initialized as for `mpfi_urandom`.

Caveat: the exponential distribution on the set of floating-point numbers is different from the exponential distribution on the set of real numbers. No guarantee is given on the quality of the distribution.

5.5 Conversion Functions

`double mpfi_get_d` (*mpfi_t op*) [Function]
Converts *op* to a double, which is the center of *op* rounded to the nearest double.

`void mpfi_get_fr` (*mpfr_t rop, mpfi_t op*) [Function]
Converts *op* to a floating-point number, which is the center of *op* rounded to nearest.

5.6 Basic Arithmetic Functions

`int mpfi_add` (*mpfi_t rop, mpfi_t op1, mpfi_t op2*) [Function]
`int mpfi_add_d` (*mpfi_t rop, mpfi_t op1, double op2*) [Function]
`int mpfi_add_ui` (*mpfi_t rop, mpfi_t op1, unsigned long int op2*) [Function]
`int mpfi_add_si` (*mpfi_t rop, mpfi_t op1, long int op2*) [Function]
`int mpfi_add_z` (*mpfi_t rop, mpfi_t op1, mpz_t op2*) [Function]
`int mpfi_add_q` (*mpfi_t rop, mpfi_t op1, mpq_t op2*) [Function]
`int mpfi_add_fr` (*mpfi_t rop, mpfi_t op1, mpfr_t op2*) [Function]
Sets *rop* to *op1* + *op2*. Returns a value indicating whether none, one or both endpoints are exact.

`int mpfi_sub` (*mpfi_t rop, mpfi_t op1, mpfi_t op2*) [Function]
`int mpfi_sub_d` (*mpfi_t rop, mpfi_t op1, double op2*) [Function]
`int mpfi_d_sub` (*mpfi_t rop, double op1, mpfi_t op2*) [Function]
`int mpfi_sub_ui` (*mpfi_t rop, mpfi_t op1, unsigned long int op2*) [Function]
`int mpfi_ui_sub` (*mpfi_t rop, unsigned long int op1, mpfi_t op2*) [Function]
`int mpfi_sub_si` (*mpfi_t rop, mpfi_t op1, long int op2*) [Function]
`int mpfi_si_sub` (*mpfi_t rop, long int op1, mpfi_t op2*) [Function]
`int mpfi_sub_z` (*mpfi_t rop, mpfi_t op1, mpz_t op2*) [Function]
`int mpfi_z_sub` (*mpfi_t rop, mpz_t op1, mpfi_t op2*) [Function]
`int mpfi_sub_q` (*mpfi_t rop, mpfi_t op1, mpq_t op2*) [Function]
`int mpfi_q_sub` (*mpfi_t rop, mpq_t op1, mpfi_t op2*) [Function]
`int mpfi_sub_fr` (*mpfi_t rop, mpfi_t op1, mpfr_t op2*) [Function]
`int mpfi_fr_sub` (*mpfi_t rop, mpfr_t op1, mpfi_t op2*) [Function]
Sets *rop* to *op1* − *op2*. Returns a value indicating whether none, one or both endpoints are exact.

`int mpfi_mul` (*mpfi_t rop, mpfi_t op1, mpfi_t op2*) [Function]
`int mpfi_mul_d` (*mpfi_t rop, mpfi_t op1, double op2*) [Function]
`int mpfi_mul_ui` (*mpfi_t rop, mpfi_t op1, unsigned long int op2*) [Function]

`int mpfi_mul_si` (*mpfi_t* `rop`, *mpfi_t* `op1`, *long int* `op2`) [Function]
`int mpfi_mul_z` (*mpfi_t* `rop`, *mpfi_t* `op1`, *mpz_t* `op2`) [Function]
`int mpfi_mul_q` (*mpfi_t* `rop`, *mpfi_t* `op1`, *mpq_t* `op2`) [Function]
`int mpfi_mul_fr` (*mpfi_t* `rop`, *mpfi_t* `op1`, *mpfr_t* `op2`) [Function]
Sets *rop* to *op1* * *op2*. Multiplication by an interval containing only zero results in 0. Returns a value indicating whether none, one or both endpoints are exact.

Division is defined even if the divisor contains zero: when the divisor contains zero in its interior, the result is the whole real interval $[-\infty, \infty]$. When the divisor has one of its endpoints equal to 0, the rules defined by the IEEE 754 norm for the division by signed zeroes apply: for instance, $[1, 2]/[0^+, 1]$ results in $[1, \infty]$. In this example, both endpoints are exact.

The extended interval division, returning two semi-infinite intervals when the divisor contains 0, should be available soon.

`int mpfi_div` (*mpfi_t* `rop`, *mpfi_t* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_div_d` (*mpfi_t* `rop`, *mpfi_t* `op1`, *double* `op2`) [Function]
`int mpfi_d_div` (*mpfi_t* `rop`, *double* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_div_ui` (*mpfi_t* `rop`, *mpfi_t* `op1`, *unsigned long int* `op2`) [Function]
`int mpfi_ui_div` (*mpfi_t* `rop`, *unsigned long int* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_div_si` (*mpfi_t* `rop`, *mpfi_t* `op1`, *long int* `op2`) [Function]
`int mpfi_si_div` (*mpfi_t* `rop`, *long int* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_div_z` (*mpfi_t* `rop`, *mpfi_t* `op1`, *mpz_t* `op2`) [Function]
`int mpfi_z_div` (*mpfi_t* `rop`, *mpz_t* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_div_q` (*mpfi_t* `rop`, *mpfi_t* `op1`, *mpq_t* `op2`) [Function]
`int mpfi_q_div` (*mpfi_t* `rop`, *mpq_t* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_div_fr` (*mpfi_t* `rop`, *mpfi_t* `op1`, *mpfr_t* `op2`) [Function]
`int mpfi_fr_div` (*mpfi_t* `rop`, *mpfr_t* `op1`, *mpfi_t* `op2`) [Function]
Sets *rop* to *op1*/*op2*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_neg` (*mpfi_t* `rop`, *mpfi_t* `op`) [Function]
Sets *rop* to $-op$. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_sqr` (*mpfi_t* `rop`, *mpfi_t* `op`) [Function]
Sets *rop* to op^2. Returns an indication of whether none, one or both endpoints are exact. Indeed, in interval arithmetic, the square of an interval is a nonnegative interval whereas the product of an interval by itself can contain negative values.

`int mpfi_inv` (*mpfi_t* `rop`, *mpfi_t* `op`) [Function]
Sets *rop* to $1/op$. Inversion is defined even if the interval contains zero: when the denominator contains zero, the result is the whole real interval $[-\infty, \infty]$. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_sqrt` (*mpfi_t* `rop`, *mpfi_t* `op`) [Function]
Sets *rop* to $\sqrt{op}$. Sets *rop* to NaN if *op* is negative. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_cbrt` (*mpfi_t* `rop`, *mpfi_t* `op`) [Function]
Sets *rop* to the cubic root of *op*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_abs` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
Sets *rop* to $|op|$, the absolute value of *op*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_mul_2exp` (*mpfi_t* **rop**, *mpfi_t* **op1**, *unsigned long int* **op2**) [Function]
`int mpfi_mul_2ui` (*mpfi_t* **rop**, *mpfi_t* **op1**, *unsigned long int* **op2**) [Function]
`int mpfi_mul_2si` (*mpfi_t* **rop**, *mpfi_t* **op1**, *long int* **op2**) [Function]
Sets *rop* to $op1 \times 2^{op2}$. Returns an indication of whether none, one or both endpoints are exact. Just increases the exponents of the endpoints by *op2* when *rop* and *op1* are identical.

`int mpfi_div_2exp` (*mpfi_t* **rop**, *mpfi_t* **op1**, *unsigned long int* **op2**) [Function]
`int mpfi_div_2ui` (*mpfi_t* **rop**, *mpfi_t* **op1**, *unsigned long int* **op2**) [Function]
`int mpfi_div_2si` (*mpfi_t* **rop**, *mpfi_t* **op1**, *long int* **op2**) [Function]
Sets *rop* to $op1/2^{op2}$. Returns an indication of whether none, one or both endpoints are exact. Just decreases the exponents of the endpoints by *op2* when *rop* and *op1* are identical.

5.7 Special Functions

These functions are based on their MPFR counterparts. For more information, see the MPFR documentation or related bibliography.

`int mpfi_log` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
Sets *rop* to the natural logarithm of *op*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact. If *op* contains negative numbers, then *rop* has at least one NaN endpoint.

`int mpfi_exp` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
Sets *rop* to the exponential of *op*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_exp2` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
Sets *rop* to 2 to the power *op*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_cos` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
`int mpfi_sin` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
`int mpfi_tan` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
Sets *rop* to the cosine, sine or tangent of *op*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_sec` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
`int mpfi_csc` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
`int mpfi_cot` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
Sets *rop* to the secant, cosecant or cotangent of *op*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_acos` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
`int mpfi_asin` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
`int mpfi_atan` (*mpfi_t* **rop**, *mpfi_t* **op**) [Function]
Sets *rop* to the arc-cosine, arc-sine or arc-tangent of *op*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_atan2` (*mpfi_t rop, mpfi_t op1, mpfi_t op2*) [Function]
Sets *rop* to the arc-tangent2 of *op1* and *op2*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_cosh` (*mpfi_t cop, mpfi_t op*) [Function]
`int mpfi_sinh` (*mpfi_t sop, mpfi_t op*) [Function]
`int mpfi_tanh` (*mpfi_t top, mpfi_t op*) [Function]
Sets *cop* to the hyperbolic cosine of *op*, *sop* to the hyperbolic sine of *op*, *top* to the hyperbolic tangent of *op*, with the precision of the result. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_sech` (*mpfi_t rop, mpfi_t op*) [Function]
`int mpfi_csch` (*mpfi_t rop, mpfi_t op*) [Function]
`int mpfi_coth` (*mpfi_t rop, mpfi_t op*) [Function]
Sets *rop* to the hyperbolic secant, cosecant or cotangent of *op*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_acosh` (*mpfi_t rop, mpfi_t op*) [Function]
`int mpfi_asinh` (*mpfi_t rop, mpfi_t op*) [Function]
`int mpfi_atanh` (*mpfi_t rop, mpfi_t op*) [Function]
Sets *rop* to the inverse hyperbolic cosine, sine or tangent of *op*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_log1p` (*mpfi_t rop, mpfi_t op*) [Function]
Sets *rop* to the natural logarithm of one plus *op*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact. If *op* contains negative numbers, then *rop* has at least one NaN endpoint.

`int mpfi_expm1` (*mpfi_t rop, mpfi_t op*) [Function]
Sets *rop* to the exponential of *op*, minus one, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_log2` (*mpfi_t rop, mpfi_t op*) [Function]
`int mpfi_log10` (*mpfi_t rop, mpfi_t op*) [Function]
Sets *rop* to $\log_t op$, with $t = 2$ or 10 the base for the logarithm, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact. If *op* contains negative numbers, then *rop* has at least one NaN endpoint.

`int mpfi_hypot` (*mpfi_t rop, mpfi_t op1, mpfi_t op2*) [Function]
Sets *rop* to the euclidean distance between points in *op1* and points in *op2*, with the precision of *rop*. Returns an indication of whether none, one or both endpoints are exact.

`int mpfi_const_log2` (*mpfi_t rop*) [Function]
`int mpfi_const_pi` (*mpfi_t rop*) [Function]
`int mpfi_const_euler` (*mpfi_t rop*) [Function]
`int mpfi_const_catalan` (*mpfi_t rop*) [Function]
Sets *rop* respectively to the logarithm of 2, to the value of π, to the Euler's constant, and to the Catalan's constant, with the precision of *rop*.

Returns an indication of whether none, one or both endpoints are exact.

5.8 Comparison Functions

The comparison of two intervals is not clearly defined when they overlap. MPFI proposes default comparison functions, but they can easily be customized according to the user's needs. The default comparison functions return a positive value if the first interval has all its elements strictly greater than all elements of the second one, a negative value if the first interval has all its elements strictly lower than all elements of the second one and 0 otherwise, i.e.\ if they overlap or if one is contained in the other.

`int mpfi_cmp` (*mpfi_t op1*, *mpfi_t op2*) [Function]
`int mpfi_cmp_d` (*mpfi_t op1*, *double op2*) [Function]
`int mpfi_cmp_ui` (*mpfi_t op1*, *unsigned long int op2*) [Function]
`int mpfi_cmp_si` (*mpfi_t op1*, *long int op2*) [Function]
`int mpfi_cmp_z` (*mpfi_t op1*, *mpz_t op2*) [Function]
`int mpfi_cmp_q` (*mpfi_t op1*, *mpq_t op2*) [Function]
`int mpfi_cmp_fr` (*mpfi_t op1*, *mpfr_t op2*) [Function]
Compares *op1* and *op2*. Return a positive value if $op1 > op2$, zero if *op1* overlaps, contains or is contained in *op2*, and a negative value if $op1 < op2$. In case one of the operands is invalid (which is represented by at least one NaN endpoint), it returns 1, even if both are invalid.

`int mpfi_is_pos` (*mpfi_t op*) [Function]
Returns a positive value if *op* contains only positive numbers, the left endpoint can be zero.

`int mpfi_is_strictly_pos` (*mpfi_t op*) [Function]
Returns a positive value if *op* contains only positive numbers.

`int mpfi_is_nonneg` (*mpfi_t op*) [Function]
Returns a positive value if *op* contains only nonnegative numbers.

`int mpfi_is_neg` (*mpfi_t op*) [Function]
Returns a positive value if *op* contains only negative numbers, the right endpoint can be zero.

`int mpfi_is_strictly_neg` (*mpfi_t op*) [Function]
Returns a positive value if *op* contains only negative numbers.

`int mpfi_is_nonpos` (*mpfi_t op*) [Function]
Returns a positive value if *op* contains only nonpositive numbers.

`int mpfi_is_zero` (*mpfi_t op*) [Function]
Returns a positive value if *op* contains only 0.

`int mpfi_has_zero` (*mpfi_t op*) [Function]
Returns a positive value if *op* contains 0 (and possibly other numbers).

`int mpfi_nan_p` (*mpfi_t op*) [Function]
Returns non-zero if *op* is invalid, i.e.\ at least one of its endpoints is a Not-a-Number (NaN), zero otherwise.

`int mpfi_inf_p` (*mpfi_t op*) [Function]
Returns non-zero if at least one of the endpoints of *op* is plus or minus infinity, zero otherwise.

`int mpfi_bounded_p` (*mpfi_t op*) [Function]
Returns non-zero if *op* is a bounded interval, i.e.\ neither invalid nor (semi-)infinite.

5.9 Input and Output Functions

Functions that perform input from a stdio stream, and functions that output to a stdio stream. Passing a NULL pointer for a *stream* argument to any of these functions will make them read from `stdin` and write to `stdout`, respectively.

When using any of these functions, it is a good idea to include `stdio.h` before `mpfr.h`, since that will allow `mpfr.h` to define prototypes for these functions.

The input and output functions are based on the representation by endpoints. The input function has to be improved. For the time being, it is mandatory to insert spaces between the interval brackets and the endpoints and also around the comma separating the endpoints.

`size_t mpfi_out_str` (*FILE *stream*, *int* `base`, *size_t* `n_digits`, *mpfi_t* *op*) [Function]

Outputs *op* on stdio stream *stream*, as a string of digits in base *base*. The output is an opening square bracket "[", followed by the lower endpoint, a separating comma, the upper endpoint and a closing square bracket "]".

For each endpoint, the output is performed by `mpfr_out_str`. The following piece of information is taken from MPFR documentation. The base may vary from 2 to 36. For each endpoint, it prints at most *n_digits* significant digits, or if *n_digits* is 0, the maximum number of digits accurately representable by *op*. In addition to the significant digits, a decimal point at the right of the first digit and a trailing exponent, in the form '`eNNN`', are printed. If *base* is greater than 10, '`@`' will be used instead of '`e`' as exponent delimiter.

Returns the number of bytes written, or if an error occurred, return 0.

As `mpfi_out_str` outputs an enclosure of the input interval, and as `mpfi_inp_str` provides an enclosure of the interval it reads, these functions are not reciprocal. More precisely, when they are called one after the other, the resulting interval contains the initial one, and this inclusion may be strict.

`size_t mpfi_inp_str` (*mpfi_t* *rop*, *FILE *stream*, *int* `base`) [Function]

Inputs a string in base *base* from stdio stream *stream*, and puts the read float in *rop*. The string is of the form '`number`' or '`[ number1 , number 2 ]`'. Each endpoint has the form '`M@N`' or, if the base is 10 or less, alternatively '`MeN`' or '`MEN`'. '`M`' is the significand and '`N`' is the exponent. The significand is always in the specified base. The exponent is in decimal.

The argument *base* may be in the ranges 2 to 36.

Unlike the corresponding `mpz` function, the base will not be determined from the leading characters of the string if *base* is 0. This is so that numbers like '`0.23`' are not interpreted as octal.

Returns the number of bytes read, or if an error occurred, return 0.

`void mpfi_print_binary` (*mpfi_t* *op*) [Function]

Outputs *op* on stdout in raw binary format for each endpoint (the exponent is in decimal, yet). The last bits from the least significant limb which do not belong to the significand are printed between square brackets; they should always be zero.

5.10 Functions Operating on Endpoints

`int mpfi_get_left` (*mpfr_t* `rop`, *mpfi_t* `op`) [Function]
Sets *rop* to the left endpoint of *op*, rounded toward minus infinity. It returns a negative value if *rop* differs from the left endpoint of *op* (due to rounding) and 0 otherwise.

`int mpfi_get_right` (*mpfr_t* `rop`, *mpfi_t* `op`) [Function]
Sets *rop* to the right endpoint of *op*, rounded toward plus infinity. It returns a positive value if *rop* differs from the right endpoint of *op* (due to rounding) and 0 otherwise.

The following function should never be used... but it helps to return correct intervals when there is a bug.

`int mpfi_revert_if_needed` (*mpfi_t* `rop`) [Function]
Swaps the endpoints of *rop* if they are not properly ordered, i.e.\ if the lower endpoint is greater than the right one. It returns a non-zero value if the endpoints have been swapped, zero otherwise.

`int mpfi_put` (*mpfi_t* `rop`, *mpfi_t* `op`) [Function]
`int mpfi_put_d` (*mpfi_t* `rop`, *double* `op`) [Function]
`int mpfi_put_ui` (*mpfi_t* `rop`, *unsigned long int* `op`) [Function]
`int mpfi_put_si` (*mpfi_t* `rop`, *long int* `op`) [Function]
`int mpfi_put_z` (*mpfi_t* `rop`, *mpz_t* `op`) [Function]
`int mpfi_put_q` (*mpfi_t* `rop`, *mpq_t* `op`) [Function]
`int mpfi_put_fr` (*mpfi_t* `rop`, *mpfr_t* `op`) [Function]
Extends the interval *rop* so that it contains *op*. In other words, *rop* is set to the convex hull of *rop* and *op*. It returns a value indicating whether none, one or both endpoints are inexact (due to possible roundings).

`int mpfi_interv_d` (*mpfi_t* `rop`, *double* `op1`, *double* `op2`) [Function]
`int mpfi_interv_ui` (*mpfi_t* `rop`, *unsigned long int* `op1`, *unsigned long int* `op2`) [Function]
`int mpfi_interv_si` (*mpfi_t* `rop`, *long int* `op1`, *long int* `op2`) [Function]
`int mpfi_interv_z` (*mpfi_t* `rop`, *mpz_t* `op1`, *mpz_t* `op2`) [Function]
`int mpfi_interv_q` (*mpfi_t* `rop`, *mpq_t* `op1`, *mpq_t* `op2`) [Function]
`int mpfi_interv_fr` (*mpfi_t* `rop`, *mpfr_t* `op1`, *mpfr_t* `op2`) [Function]
Sets *rop* to the interval having as endpoints *op1* and *op2*. The values of *op1* and *op2* are given in any order, the left endpoints of *rop* is always the minimum of *op1* and *rop2*. It returns a value indicating whether none, one or both endpoints are inexact (due to possible roundings).

5.11 Set Functions on Intervals

`int mpfi_is_strictly_inside` (*mpfi_t* `op1`, *mpfi_t* `op2`) [Function]
Returns a positive value if the second interval *op2* is contained in the interior of *op1*, 0 otherwise.

`int mpfi_is_inside` (*mpfi_t* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_is_inside_d` (*double* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_is_inside_ui` (*unsigned long* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_is_inside_si` (*long int* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_is_inside_z` (*mpz_t* `op1`, *mpfi_t* `op2`) [Function]
`int mpfi_is_inside_q` (*mpq_t* `op1`, *mpfi_t* `op2`) [Function]

`int mpfi_is_inside_fr` (*mpfr_t op1*, *mpfi_t op2*) [Function]
Returns a positive value if *op1* is contained in *op2*, 0 otherwise. Return 0 if at least one argument is NaN or an invalid interval.

`int mpfi_is_empty` (*mpfi_t op*) [Function]
Returns a positive value if *op* is empty (its endpoints are in reverse order) and 0 otherwise. Nothing is done in arithmetic or special functions to handle empty intervals: this is the responsibility of the user to avoid computing with empty intervals.

`int mpfi_intersect` (*mpfi_t rop*, *mpfi_t op1*, *mpfi_t op2*) [Function]
Sets *rop* to the intersection (possibly empty) of the intervals *op1* and *op2*. It returns a value indicating whether none, one or both endpoints are inexact (due to possible roundings). Warning: this function can return an empty interval (i.e.\ with endpoints in reverse order).

`int mpfi_union` (*mpfi_t rop*, *mpfi_t op1*, *mpfi_t op2*) [Function]
Sets *rop* to the convex hull of the union of the intervals *op1* and *op2*. It returns a value indicating whether none, one or both endpoints are inexact (due to possible roundings).

5.12 Miscellaneous Interval Functions

`int mpfi_increase` (*mpfi_t rop*, *mpfr_t op*) [Function]
Subtracts *op* to the lower endpoint of *rop* and adds it to the upper endpoint of *rop*, sets the resulting interval to *rop*. It returns a value indicating whether none, one or both endpoints are inexact.

`int mpfi_blow` (*mpfi_t rop*, *mpfi_t op1*, *double op2*) [Function]
Sets *rop* to the interval whose center is the center of *op1* and whose radius is the radius of *op1* multiplied by $(1 + |op2|)$. It returns a value indicating whether none, one or both endpoints are inexact.

`int mpfi_bisect` (*mpfi_t rop1*, *mpfi_t rop2*, *mpfi_t op*) [Function]
Splits *op* into two halves and sets them to *rop1* and *rop2*. Due to outward rounding, the two halves *rop1* and *rop2* may overlap. It returns a value >0 if the splitting point is greater than the exact centre, <0 if it is smaller and 0 if it is the exact centre.

`const char * mpfi_get_version` () [Function]
Returns the MPFI version number as a NULL terminated string.

5.13 Error Handling

`void MPFI_ERROR` (*char * msg*) [Macro]
If there is no previous error, sets the error number to 1 and prints the message *msg* to the standard error stream. If the error number is already set, do nothing.

`int mpfi_is_error` () [Function]
Returns 1 if the error number is set (to 1).

`void mpfi_set_error` (*int op*) [Function]
Sets the error number to *op*.

`void mpfi_reset_error` () [Function]
Resets the error number to 0.

Contributors

MPFI has been written by Fabrice Rouillier, Nathalie Revol, Sylvain Chevillard, Hong Diep Nguyen, Christoph Lauter and Philippe Théveny. Its development has greatly benefited from the patient and supportive help of the MPFR team.

References

This is a largely lacunary list of introductory references.

- MPFR team (SPACES project, INRIA Lorraine and LORIA), "MPFR. The Multiple Precision Floating-Point Reliable Library", available at `http://www.mpfr.org`.
- The main Web site for interval computations is `http://cs.utep.edu/interval-comp/main.html`.
- The Web site of the IEEE-1788 working group for the standardization of interval arithmetic is `http://grouper.ieee.org/groups/1788/`.
- G. Alefeld and J. Herzberger, "Introduction to interval analysis", Academic Press, 1983.
- R. Baker Kearfott, "Rigorous global search: continuous problems", Kluwer, 1996.
- T. Hickey and Q. Ju and M. H. Van Emden, "Interval arithmetic: From principles to implementation", Journal of the ACM, vol. 48, no 4, pp 1038–1068, September 2001.
- E. Hansen, "Global optimization using interval analysis", Marcel Dekker, 1992.
- A. Neumaier, "Interval methods for systems of equations", Cambridge University Press, 1990.
- H. Ratschek and J. Rokne, "New computer methods for global optimization", Ellis Horwood Ltd, 1988.
- N. Revol and F. Rouillier, "Motivations for an arbitrary precision interval arithmetic and the MPFI library", Reliable Computing, vol. 11, no 4, pp 275–290, 2005.

Concept Index

A

Arithmetic functions ... 13

C

Comparison functions ... 17
Conditions for copying MPFI ... 1
Conversion functions ... 13
Copying conditions ... 1

E

Error handling ... 20

F

Floating-point number ... 6
Functions operating on endpoints ... 19

I

I/O functions ... 18
Initialization and assignment functions ... 11
Input functions ... 18
Installation ... 3
Interval ... 6
Interval arithmetic functions ... 13
Interval assignment functions ... 10
Interval comparisons functions ... 17
Interval functions ... 8
Interval functions with floating-point results ... 11
Interval initialization functions ... 9
Interval input and output functions ... 18

M

Miscellaneous interval functions ... 20
`mpfi.h` ... 6

O

Output functions ... 18

P

Precision ... 6, 8

R

Reporting bugs ... 5
Return values ... 8

S

Set functions on intervals ... 19
Special functions ... 15

U

User-defined precision ... 8

Function and Type Index

mpfi_abs ... 15
mpfi_acos ... 15
mpfi_acosh ... 16
mpfi_add ... 13
mpfi_add_d ... 13
mpfi_add_fr ... 13
mpfi_add_q ... 13
mpfi_add_si ... 13
mpfi_add_ui ... 13
mpfi_add_z ... 13
mpfi_alea ... 12
mpfi_asin ... 15
mpfi_asinh ... 16
mpfi_atan ... 15
mpfi_atan2 ... 16
mpfi_atanh ... 16
mpfi_bisect ... 20
mpfi_blow ... 20
mpfi_bounded_p ... 17
mpfi_cbrt ... 14
mpfi_clear ... 9
mpfi_clears ... 10
mpfi_cmp ... 17
mpfi_cmp_d ... 17
mpfi_cmp_fr ... 17
mpfi_cmp_q ... 17
mpfi_cmp_si ... 17
mpfi_cmp_ui ... 17
mpfi_cmp_z ... 17
mpfi_const_catalan ... 16
mpfi_const_euler ... 16
mpfi_const_log2 ... 16
mpfi_const_pi ... 16
mpfi_cos ... 15
mpfi_cosh ... 16
mpfi_cot ... 15
mpfi_coth ... 16
mpfi_csc ... 15
mpfi_csch ... 16
mpfi_d_div ... 14
mpfi_d_sub ... 13
mpfi_diam ... 12
mpfi_diam_abs ... 12
mpfi_diam_rel ... 12
mpfi_div ... 14
mpfi_div_2exp ... 15
mpfi_div_2si ... 15
mpfi_div_2ui ... 15
mpfi_div_d ... 14
mpfi_div_fr ... 14
mpfi_div_q ... 14
mpfi_div_si ... 14
mpfi_div_ui ... 14
mpfi_div_z ... 14
mpfi_erandom ... 13
mpfi_exp ... 15
mpfi_exp2 ... 15
mpfi_expm1 ... 16
mpfi_fr_div ... 14
mpfi_fr_sub ... 13
mpfi_get_d ... 13
mpfi_get_fr ... 13
mpfi_get_left ... 19
mpfi_get_prec ... 9
mpfi_get_right ... 19
mpfi_get_version ... 20
mpfi_has_zero ... 17
mpfi_hypot ... 16
mpfi_increase ... 20
mpfi_inf_p ... 17
mpfi_init ... 9
mpfi_init_set ... 11
mpfi_init_set_d ... 11
mpfi_init_set_fr ... 11
mpfi_init_set_q ... 11
mpfi_init_set_si ... 11
mpfi_init_set_str ... 11
mpfi_init_set_ui ... 11
mpfi_init_set_z ... 11
mpfi_init2 ... 9
mpfi_inits ... 10
mpfi_inits2 ... 10
mpfi_inp_str ... 18
mpfi_intersect ... 20
mpfi_interv_d ... 19
mpfi_interv_fr ... 19
mpfi_interv_q ... 19
mpfi_interv_si ... 19
mpfi_interv_ui ... 19
mpfi_interv_z ... 19
mpfi_inv ... 14
mpfi_is_empty ... 20
mpfi_is_error ... 20
mpfi_is_inside ... 19
mpfi_is_inside_d ... 19
mpfi_is_inside_fr ... 19
mpfi_is_inside_q ... 19
mpfi_is_inside_si ... 19
mpfi_is_inside_ui ... 19
mpfi_is_inside_z ... 19
mpfi_is_neg ... 17
mpfi_is_nonneg ... 17
mpfi_is_nonpos ... 17
mpfi_is_pos ... 17
mpfi_is_strictly_inside ... 19
mpfi_is_strictly_neg ... 17
mpfi_is_strictly_pos ... 17
mpfi_is_zero ... 17
mpfi_log ... 15
mpfi_log10 ... 16
mpfi_log1p ... 16
mpfi_log2 ... 16
mpfi_mag ... 12
mpfi_mid ... 12
mpfi_mig ... 12
mpfi_mul ... 13
mpfi_mul_2exp ... 15
mpfi_mul_2si ... 15
mpfi_mul_2ui ... 15
mpfi_mul_d ... 13
mpfi_mul_fr ... 14
mpfi_mul_q ... 14

mpfi_mul_si 13
mpfi_mul_ui 13
mpfi_mul_z 14
mpfi_nan_p 17
mpfi_neg 14
mpfi_nrandom 12
mpfi_out_str 18
mpfi_print_binary 18
mpfi_put 19
mpfi_put_d 19
mpfi_put_fr 19
mpfi_put_q 19
mpfi_put_si 19
mpfi_put_ui 19
mpfi_put_z 19
mpfi_q_div 14
mpfi_q_sub 13
mpfi_reset_error 20
mpfi_revert_if_needed 19
mpfi_round_prec 9
mpfi_sec 15
mpfi_sech 16
mpfi_set 11
mpfi_set_d 11
mpfi_set_error 20
mpfi_set_flt 11
mpfi_set_fr 11
mpfi_set_ld 11
mpfi_set_prec 9
mpfi_set_q 11
mpfi_set_si 11
mpfi_set_str 11
mpfi_set_ui 11
mpfi_set_z 11
mpfi_si_div 14
mpfi_si_sub 13
mpfi_sin 15
mpfi_sinh 16
mpfi_sqr 14
mpfi_sqrt 14
mpfi_sub 13
mpfi_sub_d 13
mpfi_sub_fr 13
mpfi_sub_q 13
mpfi_sub_si 13
mpfi_sub_ui 13
mpfi_sub_z 13
mpfi_swap 11
mpfi_t 6, 7
mpfi_tan 15
mpfi_tanh 16
mpfi_ui_div 14
mpfi_ui_sub 13
mpfi_union 20
mpfi_urandom 12
mpfi_z_div 14
mpfi_z_sub 13
MPFI_BOTH_ARE_EXACT 8
MPFI_BOTH_ARE_INEXACT 8
MPFI_ERROR 20
MPFI_FLAGS_BOTH_ENDPOINTS_EXACT 8
MPFI_FLAGS_BOTH_ENDPOINTS_INEXACT 8
MPFI_FLAGS_LEFT_ENDPOINT_INEXACT 8
MPFI_FLAGS_RIGHT_ENDPOINT_INEXACT 8
MPFI_LEFT_IS_INEXACT 8
MPFI_RIGHT_IS_INEXACT 8
mpfr_get_default_prec 8
mpfr_prec_t 6
mpfr_set_default_prec 8
mpfr_t 6

Table of Contents

MPFI Copying Conditions 1

1 Introduction to MPFI 2

2 Installing MPFI 3
- 2.1 How to Install 3
- 2.2 Other 'make' targets 3
- 2.3 Known Build Problems 4
- 2.4 Getting the Latest Version of MPFI 4

3 Reporting Bugs 5

4 MPFI Basics 6
- 4.1 Nomenclature and Types 6
- 4.2 Function Classes 6
- 4.3 MPFI Variable Conventions 7

5 Interval Functions 8
- 5.1 Return Values 8
- 5.2 Precision Handling 8
- 5.3 Initialization and Assignment Functions 9
  - 5.3.1 Initialization Functions 9
  - 5.3.2 Assignment Functions 10
  - 5.3.3 Combined Initialization and Assignment Functions 11
- 5.4 Interval Functions with Floating-point Results 11
- 5.5 Conversion Functions 13
- 5.6 Basic Arithmetic Functions 13
- 5.7 Special Functions 15
- 5.8 Comparison Functions 17
- 5.9 Input and Output Functions 18
- 5.10 Functions Operating on Endpoints 19
- 5.11 Set Functions on Intervals 19
- 5.12 Miscellaneous Interval Functions 20
- 5.13 Error Handling 20

Contributors 21

References 22

Concept Index 23

Function and Type Index 24