MPC

The Multiple Precision Complex Library
Edition 0.9
February 2011

INRIA

MPC Copying Conditions

1 Introduction to MPC

MPC is a portable library written in C for arbitrary precision arithmetic on complex numbers providing correct rounding. Ultimately, it should implement a multiprecision equivalent of the C99 standard. It builds upon the GNU MP and the GNU MPFR libraries.

1.1 How to use this Manual

Everyone should read Chapter 4 [MPC Basics], page 6. If you need to install the library yourself, you need to read Chapter 2 [Installing MPC], page 3, too.

The remainder of the manual can be used for later reference, although it is probably a good idea to skim through it.

2 Installing MPC

To build MPC, you first have to install GNU MP (version 4.3.2 or higher) and GNU MPFR (version 2.4.2 or higher) 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.

Here are the steps needed to install the library on Unix systems:

1. '`tar xzf mpc-0.9.tar.gz`'
2. '`cd mpc-0.9`'
3. '`./configure`'

   if GMP and MPFR are installed into standard directories, that is, directories that are searched by default by the compiler and the linking tools.

   '`./configure --with-gmp=<gmp_install_dir>`'

   is used to indicate a different location where GMP is installed. Alternatively, you can specify directly GMP include and GMP lib directories with '`./configure --with-gmp-lib=<gmp_lib_dir> --with-gmp-include=<gmp_include_dir>`'.

   '`./configure --with-mpfr=<mpfr_install_dir>`'

   is used to indicate a different location where MPFR is installed. Alternatively, you can specify directly MPFR include and MPFR lib directories with '`./configure --with-mpf-lib=<mpfr_lib_dir> --with-mpfr-include=<mpfr_include_dir>`'.

   Another useful parameter is '`--prefix`', which can be used to specify an alternative installation location instead of '`/usr/local`'; see '`make install`' below.

   If for debugging purposes you wish to log calls to MPC functions from within your code, add the parameter '`--enable-logging`'. In your code, replace the inclusion of '`mpc.h`' by '`mpc-log.h`' and link the executable dynamically. Then all calls to functions with only complex arguments are printed to '`stderr`' in the following form: First, the function name is given, followed by its type such as '`c_cc`', meaning that the function has one complex result (one '`c`' in front of the '`_`'), computed from two complex arguments (two '`c`' after the '`_`'). Then, the precisions of the real and the imaginary part of the first result is given, followed by the second one and so on. Finally, for each argument, the precisions of its real and imaginary part are specified and the argument itself is printed in hexadecimal via the function `mpc_out_str` (see [String and Stream Input and Output], page 10).

   Use '`./configure --help`' for an exhaustive list of parameters.
4. '`make`'

   This compiles MPC in the working directory.
5. '`make check`'

   This will make sure MPC was built correctly.

   If you get error messages, please report them to '`mpc-discuss@lists.gforge.inria.fr`' (See Chapter 3 [Reporting Bugs], page 5, for information on what to include in useful bug reports).
6. '`make install`'

   This will copy the file '`mpc.h`' to the directory '`/usr/local/include`', the file '`libmpc.a`' to the directory '`/usr/local/lib`', and the file '`mpc.info`' to the directory '`/usr/local/share/info`' (or if you passed the '`--prefix`' option to '`configure`', using the prefix directory given as argument to '`--prefix`' instead of '`/usr/local`'). Note: you need write permissions on these directories.

2.1 Other 'make' Targets

There are some other useful make targets:

- '`info`'
  Create an info version of the manual, in '`mpc.info`'.
- '`pdf`'
  Create a PDF version of the manual, in '`doc/mpc.pdf`'.
- '`dvi`'
  Create a DVI version of the manual, in '`doc/mpc.dvi`'.
- '`ps`'
  Create a Postscript version of the manual, in '`doc/mpc.ps`'.
- '`html`'
  Create an HTML version of the manual, in several pages in the directory '`doc/mpc.html`'; if you want only one output HTML file, then type '`makeinfo --html --no-split mpc.texi`' instead.
- '`clean`'
  Delete all object files and archive files, but not the configuration files.
- '`distclean`'
  Delete all files not included in the distribution.
- '`uninstall`'
  Delete all files copied by '`make install`'.

2.2 Known Build Problems

On AIX, if GMP was built with the 64-bit ABI, before building and testing MPC, it might be necessary to set the '`OBJECT_MODE`' environment variable to 64 by, e.g.,

'`export OBJECT_MODE=64`'

This has been tested with the C compiler IBM XL C/C++ Enterprise Edition V8.0 for AIX, version: 08.00.0000.0021, GMP 4.2.4 and MPFR 2.4.1.

Please report any other problems you encounter to '`mpc-discuss@lists.gforge.inria.fr`'. See Chapter 3 [Reporting Bugs], page 5.

3 Reporting Bugs

If you think you have found a bug in the MPC library, please investigate and report it. We have made this library available to you, and it is not to ask too much from you, to ask you to 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 '`gcc -v`', or '`cc -V`' on some machines. Also, include the output from '`uname -a`'.

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 will not do anything about it (aside of chiding you to send better bug reports).

Send your bug report to: '`mpc-discuss@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 MPC Basics

All declarations needed to use MPC are collected in the include file '`mpc.h`'. It is designed to work with both C and C++ compilers. You should include that file in any program using the MPC library by adding the line

```
#include "mpc.h"
```

4.1 Nomenclature and Types

Complex number or *Complex* for short, is a pair of two arbitrary precision floating-point numbers (for the real and imaginary parts). The C data type for such objects is `mpc_t`.

The *Precision* is the number of bits used to represent the mantissa of the real and imaginary parts; the corresponding C data type is `mpfr_prec_t`. For more details on the allowed precision range, see Section "Nomenclature and Types" in *MPFR*.

The *rounding mode* specifies the way to round the result of a complex operation, in case the exact result can not be represented exactly in the destination mantissa; the corresponding C data type is `mpc_rnd_t`. A complex rounding mode is a pair of two rounding modes: one for the real part, one for the imaginary part.

4.2 Function Classes

There is only one class of functions in the MPC library, namely functions for complex arithmetic. The function names begin with `mpc_`. The associated type is `mpc_t`.

4.3 MPC Variable Conventions

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

MPC allows you to use the same variable for both input and output in the same expression. For example, the main function for floating-point multiplication, `mpc_mul`, can be used like this: `mpc_mul (x, x, x, rnd_mode)`. This computes the square of x with rounding mode `rnd_mode` and puts the result back in x.

Before you can assign to an MPC variable, you need to initialize it by calling one of the special initialization functions. When you are done with a variable, you need to clear it out, using one of the functions for that purpose.

A variable should only be initialized once, or at least cleared out between each initialization. After a variable has been initialized, it may be assigned to 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 the loop has exited.

You do not need to be concerned about allocating additional space for MPC variables, since each of its real and imaginary part has a mantissa of fixed size. Hence unless you change its precision, or clear and reinitialize it, a complex variable will have the same allocated space during all its life.

4.4 Rounding Modes

A complex rounding mode is of the form `MPC_RNDxy` where `x` and `y` are one of `N` (to nearest), `Z` (towards zero), `U` (towards plus infinity), `D` (towards minus infinity). The first letter refers to the rounding mode for the real part, and the second one for the imaginary part. For example

`MPC_RNDZU` indicates to round the real part towards zero, and the imaginary part towards plus infinity.

The '`round to nearest`' mode works as in the IEEE P754 standard: in case the number to be rounded lies exactly in the middle of two representable numbers, it is rounded to the one with the least significant bit set to zero. For example, the number 5, which is represented by (101) in binary, is rounded to (100)=4 with a precision of two bits, and not to (110)=6.

4.5 Return Value

Most MPC functions have a return value of type `int`, which is used to indicate the position of the rounded real and imaginary parts with respect to the exact (infinite precision) values. If this integer is `i`, the macros `MPC_INEX_RE(i)` and `MPC_INEX_IM(i)` give 0 if the corresponding rounded value is exact, a negative value if the rounded value is less than the exact one, and a positive value if it is greater than the exact one. Similarly, functions computing a result of type `mpfr_t` return an integer that is 0, positive or negative depending on whether the rounded value is the same, larger or smaller then the exact result.

Some functions, such as `mpc_sin_cos`, compute two complex results; the macros `MPC_INEX1(i)` and `MPC_INEX2(i)`, applied to the return value `i` of such a function, yield the exactness value corresponding to the first or the second computed value, respectively.

4.6 Branch Cuts And Special Values

Some complex functions have branch cuts, across which the function is discontinous. In MPC, the branch cuts chosen are the same as those specified for the corresponding functions in the ISO C99 standard.

Likewise, when evaluated at a point whose real or imaginary part is either infinite or a NaN or a signed zero, a function returns the same value as those specified for the corresponding function in the ISO C99 standard.

5 Complex Functions

The complex functions expect arguments of type `mpc_t`.

The MPC floating-point functions have an interface that is similar to the GNU MP integer functions. The function prefix for operations on complex numbers is `mpc_`.

The precision of a computation is defined as follows: Compute the requested operation exactly (with "infinite precision"), and round the result to the destination variable precision with the given rounding mode.

The MPC complex functions are intended to be a smooth extension of the IEEE P754 arithmetic. The results obtained on one computer should not differ from the results obtained on a computer with a different word size.

5.1 Initialization Functions

An `mpc_t` object must be initialized before storing the first value in it. The functions `mpc_init2` and `mpc_init3` are used for that purpose.

`void mpc_init2 (`*mpc_t* `z`, *mpfr_prec_t* `prec``)` [Function]
Initialize *z* to precision *prec* bits and set its real and imaginary parts to NaN. Normally, a variable should be initialized once only or at least be cleared, using `mpc_clear`, between initializations.

`void mpc_init3 (`*mpc_t* `z`, *mpfr_prec_t* `prec_r`, *mpfr_prec_t* `prec_i``)` [Function]
Initialize *z* with the precision of its real part being *prec_r* bits and the precision of its imaginary part being *prec_i* bits, and set the real and imaginary parts to NaN.

`void mpc_clear (`*mpc_t* `z``)` [Function]
Free the space occupied by *z*. Make sure to call this function for all `mpc_t` variables when you are done with them.

Here is an example on how to initialize complex variables:

```
{
  mpc_t x, y;
  mpc_init2 (x, 256); /* precision exactly 256 bits */
  mpc_init3 (y, 100, 50); /* 100/50 bits for the real/imaginary part */
  ...
  mpc_clear (x);
  mpc_clear (y);
}
```

The following function is 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 mpc_set_prec (`*mpc_t* `x`, *mpfr_prec_t* `prec``)` [Function]
Reset the precision of *x* to be **exactly** *prec* bits, and set its real/imaginary parts to NaN. The previous value stored in *x* is lost. It is equivalent to a call to `mpc_clear(x)` followed by a call to `mpc_init2(x, prec)`, but more efficient as no allocation is done in case the current allocated space for the mantissa of *x* is sufficient.

`mpfr_prec_t mpc_get_prec` (*mpc_t* `x`) [Function]
If the real and imaginary part of *x* have the same precision, it is returned, otherwise, 0 is returned.

`void mpc_get_prec2` (*mpfr_prec_t** `pr`, *mpfr_prec_t** `pi`, *mpc_t* `x`) [Function]
Returns the precision of the real part of *x* via *pr* and of its imaginary part via *pi*.

5.2 Assignment Functions

These functions assign new values to already initialized complex numbers (see Section 5.1 [Initializing Complex Numbers], page 8). When using any functions with `intmax_t` or `uintmax_t` parameters, you must include `<stdint.h>` or `<inttypes.h>` *before* '`mpc.h`', to allow '`mpc.h`' to define prototypes for these functions. Similarly, functions with parameters of type `complex` or `long complex` are defined only if `<complex.h>` is included *before* '`mpc.h`'. If you need assignment functions that are not in the current API, you can define them using the `MPC_SET_X_Y` macro (see Section 5.11 [Advanced Functions], page 16).

`int mpc_set` (*mpc_t* `rop`, *mpc_t* `op`, *mpc_rnd_t* `rnd`) [Function]
Set the value of *rop* from *op*, rounded to the precision of *rop* with the given rounding mode *rnd*.

`int mpc_set_ui` (*mpc_t* `rop`, *unsigned long int* `op`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_si` (*mpc_t* `rop`, *long int* `op`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_uj` (*mpc_t* `rop`, *uintmax_t* `op`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_sj` (*mpc_t* `rop`, *intmax_t* `op`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_d` (*mpc_t* `rop`, *double* `op`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_ld` (*mpc_t* `rop`, *long double* `op`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_dc` (*mpc_t* `rop`, *double _Complex* `op`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_ldc` (*mpc_t* `rop`, *long double _Complex* `op`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_z` (*mpc_t* `rop`, *mpz_t* `op` *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_q` (*mpc_t* `rop`, *mpq_t* `op` *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_f` (*mpc_t* `rop`, *mpf_t* `op` *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_fr` (*mpc_t* `rop`, *mpfr_t* `op`, *mpc_rnd_t* `rnd`) [Function]
Set the value of *rop* from *op*, rounded to the precision of *rop* with the given rounding mode *rnd*. The argument *op* is interpreted as real, so the imaginary part of *rop* is set to zero with a positive sign. Please note that even a `long int` may have to be rounded, if the destination precision is less than the machine word width. For `mpc_set_d`, be careful that the input number *op* may not be exactly representable as a double-precision number (this happens for 0.1 for instance), in which case it is first rounded by the C compiler to a double-precision number, and then only to a complex number.

`int mpc_set_ui_ui` (*mpc_t* `rop`, *unsigned long int* `op1`, *unsigned long int* `op2`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_si_si` (*mpc_t* `rop`, *long int* `op1`, *long int* `op2`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_uj_uj` (*mpc_t* `rop`, *uintmax_t* `op1`, *uintmax_t* `op2`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_sj_sj` (*mpc_t* `rop`, *intmax_t* `op1`, *intmax_t* `op2`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_d_d` (*mpc_t* `rop`, *double* `op1`, *double* `op2`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_ld_ld` (*mpc_t* `rop`, *long double* `op1`, *long double* `op2`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_z_z` (*mpc_t* `rop`, *mpz_t* `op1`, *mpz_t* `op2`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_q_q` (*mpc_t* `rop`, *mpq_t* `op1`, *mpq_t* `op2`, *mpc_rnd_t* `rnd`) [Function]

`int mpc_set_f_f` (*mpc_t* **rop**, *mpf_t* **op1**, *mpf_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_set_fr_fr` (*mpc_t* **rop**, *mpfr_t* **op1**, *mpfr_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
Set the real part of *rop* from *op1*, and its imaginary part from *op2*, according to the rounding mode *rnd*.

Beware that the behaviour of `mpc_set_fr_fr` is undefined if *op1* or *op2* is a pointer to the real or imaginary part of *rop*. To exchange the real and the imaginary part of a complex number, either use `mpfr_swap (mpc_realref (rop), mpc_imagref (rop))`, which also exchanges the precisions of the two parts; or use a temporary variable.

For functions assigning complex variables from strings or input streams, see [String and Stream Input and Output], page 10.

`void mpc_set_nan` (*mpc_t* **rop**) [Function]
Set *rop* to Nan+i*NaN.

`void mpc_swap` (*mpc_t* **op1**, *mpc_t* **op2**) [Function]
Swap the values of *op1* and *op2* efficiently. Warning: The precisions are exchanged, too; in case these are different, `mpc_swap` is thus not equivalent to three `mpc_set` calls using a third auxiliary variable.

5.3 Conversion Functions

The following functions are available only if `<complex.h>` is included *before* '`mpc.h`'.

`double _Complex` *mpc_get_dc* (*mpc_t* **op**, *mpc_rnd_t* **rnd**) [Function]
`long double _Complex mpc_get_ldc` (*mpc_t* **op**, *mpc_rnd_t* **rnd**) [Function]
Convert *op* to a C complex number, using the rounding mode *rnd*.

For functions converting complex variables to strings or stream output, see [String and Stream Input and Output], page 10.

5.4 String and Stream Input and Output

`int mpc_strtoc` (*mpc_t* **rop**, *const char* ***nptr**, *char* ****endptr**, *int* **base**, *mpc_rnd_t* **rnd**) [Function]
Read a complex number from a string *nptr* in base *base*, rounded to the precision of *rop* with the given rounding mode *rnd*. The *base* must be either 0 or a number from 2 to 36 (otherwise the behaviour is undefined). If *nptr* starts with valid data, the result is stored in *rop*, the usual inexact value is returned (see [Return Value], page 7) and, if *endptr* is not the null pointer, **endptr* points to the character just after the valid data. Otherwise, *rop* is set to `NaN + i * NaN`, -1 is returned and, if *endptr* is not the null pointer, the value of *nptr* is stored in the location referenced by *endptr*.

The expected form of a complex number string is either a real number (an optional leading whitespace, an optional sign followed by a floating-point number), or a pair of real numbers in parentheses separated by whitespace. If a real number is read, the missing imaginary part is set to +0. The form of a floating-point number depends on the base and is described in the documentation of `mpfr_strtofr` in the MPFR manual. For instance, `"3.1415926"`, `"(1.25e+7 +.17)"`, `"(@nan@ 2)"` and `"(-0 -7)"` are valid strings for *base* = 10. If *base* = 0, then a prefix may be used to indicate the base in which the floating-point number is written. Use prefix '0b' for binary numbers, prefix '0x' for hexadecimal numbers, and no prefix for decimal numbers. The real and imaginary part may then be written in different bases. For

instance, `"(1.024e+3 +2.05e+3)"` and `"(0b1p+10 +0x802)"` are valid strings for `base`=0 and represent the same value.

`int mpc_set_str` (*mpc_t* **`rop`**, *const char* ***`s`**, *int* **`base`**, *mpc_rnd_t rnd*) [Function]
Set *rop* to the value of the string *s* in base *base*, rounded to the precision of *rop* with the given rounding mode *rnd*. See the documentation of `mpc_strtoc` for a detailed description of the valid string formats. Contrarily to `mpc_strtoc`, `mpc_set_str` requires the *whole* string to represent a valid complex number (potentially followed by additional white space). This function returns the usual inexact value (see [Return Value], page 7) if the entire string up to the final null character is a valid number in base *base*; otherwise it returns −1, and *rop* is set to NaN+i*NaN.

`char * mpc_get_str` (*int* **`b`**, *size_t* **`n`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
Convert *op* to a string containing its real and imaginary parts, separated by a space and enclosed in a pair of parentheses. The numbers are written in base *b* (which may vary from 2 to 36) and rounded according to *rnd*. The number of significant digits, at least 2, is given by *n*. It is also possible to let *n* be zero, in which case the number of digits 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*. Note that `mpc_get_str` uses the decimal point of the current locale if available, and '`.`' otherwise.

The string is generated using the current memory allocation function (`malloc` by default, unless it has been modified using the custom memory allocation interface of `gmp`); once it is not needed any more, it should be freed by calling `mpc_free_str`.

`void mpc_free_str` (*char* ***`str`**) [Function]
Free the string *str*, which needs to have been allocated by a call to `mpc_get_str`.

The following two functions read numbers from input streams and write them to output streams. When using any of these functions, you need to include '`stdio.h`' *before* '`mpc.h`'.

`int mpc_inp_str` (*mpc_t* **`rop`**, *FILE* ***`stream`**, *size_t* ***`read`**, *int* **`base`**, *mpc_rnd_t* **`rnd`**) [Function]
Input a string in base *base* in the same format as for `mpc_strtoc` from stdio stream *stream*, rounded according to *rnd*, and put the read complex number into *rop*. If *stream* is the null pointer, *rop* is read from `stdin`. Return the usual inexact value; if an error occurs, set *rop* to `NaN + i * NaN` and return -1. If *read* is not the null pointer, it is set to the number of read characters.

Unlike `mpc_strtoc`, the function `mpc_inp_str` does not possess perfect knowledge of the string to transform and has to read it character by character, so it behaves slightly differently: It tries to read a string describing a complex number and processes this string through a call to `mpc_set_str`. Precisely, after skipping optional whitespace, a minimal string is read according to the regular expression `mpfr | '(' \s* mpfr \s+ mpfr \s* ')'`, where `\s` denotes a whitespace, and `mpfr` is either a string containing neither whitespaces nor parentheses, or `nan(n-char-sequence)` or `@nan@(n-char-sequence)` (regardless of capitalisation) with `n-char-sequence` a string of ascii letters, digits or `'_'`.

For instance, upon input of `"nan(13 1)"`, the function `mpc_inp_str` starts to recognise a value of NaN followed by an n-char-sequence indicated by the opening parenthesis; as soon as the space is reached, it becocmes clear that the expression in parentheses is not an n-char-sequence, and the error flag -1 is returned after 6 characters have been consumed from the stream (the whitespace itself remaining in the stream). The function `mpc_strtoc`, on the other hand, may track back when reaching the whitespace; it treats the string as the

two successive complex numbers `NaN + i * 0` and `13 + i`. It is thus recommended to have a whitespace follow each floating point number to avoid this problem.

`size_t mpc_out_str` (*FILE *stream*, *int* `base`, *size_t* `n_digits`, *mpc_t* `op`, *mpc_rnd_t* `rnd`) [Function]

Output *op* on stdio stream *stream* in base *base*, rounded according to *rnd*, in the same format as for `mpc_strtoc` If *stream* is the null pointer, *rop* is written to `stdout`.

Return the number of characters written.

5.5 Comparison Functions

`int mpc_cmp` (*mpc_t* `op1`, *mpc_t* `op2`) [Function]

`int mpc_cmp_si_si` (*mpc_t* `op1`, *long int* `op2r`, *long int* `op2i`) [Function]

`int mpc_cmp_si` (*mpc_t* `op1`, *long int* `op2`) [Macro]

Compare *op1* and *op2*, where in the case of `mpc_cmp_si_si`, *op2* is taken to be *op2r* + i *op2i*. The return value *c* can be decomposed into `x = MPC_INEX_RE(c)` and `y = MPC_INEX_IM(c)`, such that *x* is positive if the real part of *op1* is greater than that of *op2*, zero if both real parts are equal, and negative if the real part of *op1* is less than that of *op2*, and likewise for *y*. Both *op1* and *op2* are considered to their full own precision, which may differ. It is not allowed that one of the operands has a NaN (Not-a-Number) part.

The storage of the return value is such that equality can be simply checked with `mpc_cmp (op1, op2) == 0`.

5.6 Projection and Decomposing Functions

`int mpc_real` (*mpfr_t* `rop`, *mpc_t* `op`, *mpfr_rnd_t* `rnd`) [Function]

Set *rop* to the value of the real part of *op* rounded in the direction *rnd*.

`int mpc_imag` (*mpfr_t* `rop`, *mpc_t* `op`, *mpfr_rnd_t* `rnd`) [Function]

Set *rop* to the value of the imaginary part of *op* rounded in the direction *rnd*.

`mpfr_t mpc_realref` (*mpc_t* `op`) [Macro]

`mpfr_t mpc_imagref` (*mpc_t* `op`) [Macro]

Return a reference to the real part and imaginary part of *op*, respectively. The `mpfr` functions can be used on the result of these macros (note that the `mpfr_t` type is itself a pointer).

`int mpc_arg` (*mpfr_t* `rop`, *mpc_t* `op`, *mpfr_rnd_t* `rnd`) [Function]

Set *rop* to the argument of *op*, with a branch cut along the negative real axis.

`int mpc_proj` (*mpc_t* `rop`, *mpc_t* `op`, *mpc_rnd_t* `rnd`) [Function]

Compute a projection of *op* onto the Riemann sphere. Set *rop* to *op* rounded in the direction *rnd*, except when at least one part of *op* is infinite (even if the other part is a NaN) in which case the real part of *rop* is set to plus infinity and its imaginary part to a signed zero with the same sign as the imaginary part of *op*.

5.7 Basic Arithmetic Functions

All the following functions are designed in such a way that, when working with real numbers instead of complex numbers, their complexity should essentially be the same as with the MPFR library, with only a marginal overhead due to the MPC layer.

`int mpc_add` (*mpc_t* **rop**, *mpc_t* **op1**, *mpc_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_add_ui` (*mpc_t* **rop**, *mpc_t* **op1**, *unsigned long int* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_add_fr` (*mpc_t* **rop**, *mpc_t* **op1**, *mpfr_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to *op1* + *op2* rounded according to *rnd*.

`int mpc_sub` (*mpc_t* **rop**, *mpc_t* **op1**, *mpc_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_sub_fr` (*mpc_t* **rop**, *mpc_t* **op1**, *mpfr_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_fr_sub` (*mpc_t* **rop**, *mpfr_t* **op1**, *mpc_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_sub_ui` (*mpc_t* **rop**, *mpc_t* **op1**, *unsigned long int* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_ui_sub` (*mpc_t* **rop**, *unsigned long int* **op1**, *mpc_t* **op2**, *mpc_rnd_t* **rnd**) [Macro]
`int mpc_ui_ui_sub` (*mpc_t* **rop**, *unsigned long int* **re1**, *unsigned long int* **im1**, *mpc_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to *op1* − *op2* rounded according to *rnd*. For `mpc_ui_ui_sub`, *op1* is *re1* + *im1*.

`int mpc_mul` (*mpc_t* **rop**, *mpc_t* **op1**, *mpc_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_mul_ui` (*mpc_t* **rop**, *mpc_t* **op1**, *unsigned long int* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_mul_si` (*mpc_t* **rop**, *mpc_t* **op1**, *long int* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_mul_fr` (*mpc_t* **rop**, *mpc_t* **op1**, *mpfr_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to *op1* times *op2* rounded according to *rnd*.

`int mpc_mul_i` (*mpc_t* **rop**, *mpc_t* **op**, *int* **sgn**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to *op* times the imaginary unit i if *sgn* is non-negative, set *rop* to *op* times -i otherwise, in both cases rounded according to *rnd*.

`int mpc_sqr` (*mpc_t* **rop**, *mpc_t* **op**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to the square of *op* rounded according to *rnd*.

`int mpc_div` (*mpc_t* **rop**, *mpc_t* **op1**, *mpc_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_div_ui` (*mpc_t* **rop**, *mpc_t* **op1**, *unsigned long int* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_ui_div` (*mpc_t* **rop**, *unsigned long int* **op1**, *mpc_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_div_fr` (*mpc_t* **rop**, *mpc_t* **op1**, *mpfr_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_fr_div` (*mpc_t* **rop**, *mpfr_t* **op1**, *mpc_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to *op1*/*op2* rounded according to *rnd*. For `mpc_div` and `mpc_ui_div`, the return value may fail to recognize some exact results. The sign of returned value is significant only for `mpc_div_ui`.

`int mpc_neg` (*mpc_t* **rop**, *mpc_t* **op**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to −*op* rounded according to *rnd*. Just changes the sign if *rop* and *op* are the same variable.

`int mpc_conj` (*mpc_t* **rop**, *mpc_t* **op**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to the conjugate of *op* rounded according to *rnd*. Just changes the sign of the imaginary part if *rop* and *op* are the same variable.

`int mpc_abs` (*mpfr_t* **rop**, *mpc_t* **op**, *mpfr_rnd_t* **rnd**) [Function]
Set the floating-point number *rop* to the absolute value of *op*, rounded in the direction *rnd*. The returned value is zero iff the result is exact.

`int mpc_norm` (*mpfr_t* **rop**, *mpc_t* **op**, *mpfr_rnd_t* **rnd**) [Function]
Set the floating-point number *rop* to the norm of *op* (i.e., the square of its absolute value), rounded in the direction *rnd*. The returned value is zero iff the result is exact. Note that the destination is of type `mpfr_t`, not `mpc_t`.

`int mpc_mul_2exp` (*mpc_t* **rop**, *mpc_t* **op1**, *unsigned long int* **op2**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to *op1* times 2 raised to *op2* rounded according to *rnd*. Just increases the exponents of the real and imaginary parts by *op2* when *rop* and *op1* are identical.

`int mpc_div_2exp` (*mpc_t* **rop**, *mpc_t* **op1**, *unsigned long int* **op2**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to *op1* divided by 2 raised to *op2* rounded according to *rnd*. Just decreases the exponents of the real and imaginary parts by *op2* when *rop* and *op1* are identical.

`int mpc_fma` (*mpc_t* **rop**, *mpc_t* **op1**, *mpc_t* **op2**, *mpc_t* **op3**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to *op1* times *op2* plus *op3*, rounded according to *rnd*, with only one final rounding.

5.8 Power Functions and Logarithm

`int mpc_sqrt` (*mpc_t* **rop**, *mpc_t* **op**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to the square root of *op* rounded according to *rnd*.

`int mpc_pow` (*mpc_t* **rop**, *mpc_t* **op1**, *mpc_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_pow_d` (*mpc_t* **rop**, *mpc_t* **op1**, *double* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_pow_ld` (*mpc_t* **rop**, *mpc_t* **op1**, *long double* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_pow_si` (*mpc_t* **rop**, *mpc_t* **op1**, *long* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_pow_ui` (*mpc_t* **rop**, *mpc_t* **op1**, *unsigned long* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_pow_z` (*mpc_t* **rop**, *mpc_t* **op1**, *mpz_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
`int mpc_pow_fr` (*mpc_t* **rop**, *mpc_t* **op1**, *mpfr_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to *op1* raised to the power *op2*, rounded according to *rnd*. For `mpc_pow_d`, `mpc_pow_ld`, `mpc_pow_si`, `mpc_pow_ui`, `mpc_pow_z` and `mpc_pow_fr`, the imaginary part of *op2* is considered as +0.

`int mpc_exp` (*mpc_t* **rop**, *mpc_t* **op**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to the exponential of *op*, rounded according to *rnd* with the precision of *rop*.

`int mpc_log` (*mpc_t* **rop**, *mpc_t* **op**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to the logarithm of *op*, rounded according to *rnd* with the precision of *rop*. The principal branch is chosen, with the branch cut on the negative real axis, so that the imaginary part of the result lies in $]-\pi, \pi]$.

5.9 Trigonometric Functions

`int mpc_sin` (*mpc_t* **rop**, *mpc_t* **op**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to the sine of *op*, rounded according to *rnd* with the precision of *rop*.

`int mpc_cos` (*mpc_t* **rop**, *mpc_t* **op**, *mpc_rnd_t* **rnd**) [Function]
Set *rop* to the cosine of *op*, rounded according to *rnd* with the precision of *rop*.

`int mpc_sin_cos` (*mpc_t* **`rop_sin`**, *mpc_t* **`rop_cos`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd_sin`**, *mpc_rnd_t* **`rnd_cos`**) [Function]
Set *rop_sin* to the sine of *op*, rounded according to *rnd_sin* with the precision of *rop_sin*, and *rop_cos* to the cosine of *op*, rounded according to *rnd_cos* with the precision of *rop_cos*.

`int mpc_tan` (*mpc_t* **`rop`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
Set *rop* to the tangent of *op*, rounded according to *rnd* with the precision of *rop*.

`int mpc_sinh` (*mpc_t* **`rop`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
Set *rop* to the hyperbolic sine of *op*, rounded according to *rnd* with the precision of *rop*.

`int mpc_cosh` (*mpc_t* **`rop`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
Set *rop* to the hyperbolic cosine of *op*, rounded according to *rnd* with the precision of *rop*.

`int mpc_tanh` (*mpc_t* **`rop`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
Set *rop* to the hyperbolic tangent of *op*, rounded according to *rnd* with the precision of *rop*.

`int mpc_asin` (*mpc_t* **`rop`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
`int mpc_acos` (*mpc_t* **`rop`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
`int mpc_atan` (*mpc_t* **`rop`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
Set *rop* to the inverse sine, inverse cosine, inverse tangent of *op*, rounded according to *rnd* with the precision of *rop*.

`int mpc_asinh` (*mpc_t* **`rop`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
`int mpc_acosh` (*mpc_t* **`rop`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
`int mpc_atanh` (*mpc_t* **`rop`**, *mpc_t* **`op`**, *mpc_rnd_t* **`rnd`**) [Function]
Set *rop* to the inverse hyperbolic sine, inverse hyperbolic cosine, inverse hyperbolic tangent of *op*, rounded according to *rnd* with the precision of *rop*. The branch cut of *mpc_acosh* is $(-\infty, 1)$.

5.10 Miscellaneous Functions

`int mpc_urandom` (*mpc_t* **`rop`**, *gmp_randstate_t* **`state`**) [Function]
Generate a uniformly distributed random complex in the unit square $[0, 1] \times [0, 1]$. Return 0, unless an exponent in the real or imaginary part is not in the current exponent range, in which case that part is set to NaN and a zero value is returned. The second argument is a `gmp_randstate_t` structure which should be created using the GMP `rand_init` function, see the GMP manual.

`const char * mpc_get_version` (*void*) [Function]
Return the MPC version, as a null-terminated string.

`MPC_VERSION` [Macro]
`MPC_VERSION_MAJOR` [Macro]
`MPC_VERSION_MINOR` [Macro]
`MPC_VERSION_PATCHLEVEL` [Macro]
`MPC_VERSION_STRING` [Macro]
`MPC_VERSION` is the version of MPC as a preprocessing constant. `MPC_VERSION_MAJOR`, `MPC_VERSION_MINOR` and `MPC_VERSION_PATCHLEVEL` are respectively the major, minor and patch level of MPC version, as preprocessing constants. `MPC_VERSION_STRING` is the version as a string constant, which can be compared to the result of `mpc_get_version` to check at run time the header file and library used match:

```
if (strcmp (mpc_get_version (), MPC_VERSION_STRING))
  fprintf (stderr, "Warning: header and library do not match\n");
```

Note: Obtaining different strings is not necessarily an error, as in general, a program compiled with some old MPC version can be dynamically linked with a newer MPC library version (if allowed by the library versioning system).

`long MPC_VERSION_NUM (`*`major`*`, `*`minor`*`, `*`patchlevel`*`)` [Macro]
Create an integer in the same format as used by `MPC_VERSION` from the given *major*, *minor* and *patchlevel*. Here is an example of how to check the MPC version at compile time:

```
#if (!defined(MPC_VERSION) || (MPC_VERSION<MPC_VERSION_NUM(2,1,0)))
# error "Wrong MPC version."
#endif
```

5.11 Advanced Functions

`MPC_SET_X_Y (`*`real_suffix`*`, `*`imag_suffix`*`, `*`rop`*`, `*`real`*`, `*`imag`*`, `*`rnd`*`)` [Macro]
The macro MPC_SET_X_Y is designed to serve as the body of an assignment function and cannot be used by itself. The *real_suffix* and *imag_suffix* parameters are the types of the real and imaginary part, that is, the `x` in the `mpfr_set_x` function one would use to set the part; for the mpfr type, use `fr`. *real* (respectively *imag*) is the value you want to assign to the real (resp. imaginary) part, its type must conform to *real_suffix* (resp. *imag_suffix*). *rnd* is the `mpc_rnd_t` rounding mode. The return value is the usual inexact value (see [Return Value], page 7).

For instance, you can define mpc_set_ui_fr as follows:

```
int mpc_set_ui_fr (mpc_t rop, long int re, double im, mpc_rnd_t rnd)
    MPC_SET_X_Y (ui, fr, rop, re, im, rnd);
```

5.12 Internals

These macros and functions are mainly designed for the implementation of MPC, but may be useful for users too. However, no upward compatibility is guaranteed. You need to include `mpc-impl.h` to use them.

The macro `MPC_MAX_PREC(z)` gives the maximum of the precisions of the real and imaginary parts of a complex number.

Contributors

The main developers of the MPC library are Andreas Enge, Philippe Théveny and Paul Zimmermann. Patrick Pélissier has helped cleaning up the code. Marc Helbling contributed the `mpc_ui_sub` and `mpc_ui_ui_sub` functions.

References

- Torbjörn Granlund et al. `gmp` – GNU multiprecision library. Version 4.2.4, http://gmplib.org/.
- Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann et al. `mpfr` – A library for multiple-precision floating-point computations with exact rounding. Version 2.4.1, http://www.mpfr.org.
- IEEE standard for binary floating-point arithmetic, Technical Report ANSI-IEEE Standard 754-1985, New York, 1985. Approved March 21, 1985: IEEE Standards Board; approved July 26, 1985: American National Standards Institute, 18 pages.
- Donald E. Knuth, "The Art of Computer Programming", vol 2, "Seminumerical Algorithms", 2nd edition, Addison-Wesley, 1981.
- ISO/IEC 9899:1999, Programming languages C.

Concept Index

Function and Type Index

Table of Contents