MPC

The Multiple Precision Complex Library
Edition 0.5.1
November 2008

Andreas Enge, Philippe Théveny, Paul Zimmermann

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 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 4.2.2 or higher) and MPFR (version 2.3.1 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-0.5.1.tar.gz'
2. 'cd mpc-0.5.1'
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>'.
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

No build problems are known. Please report any 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 `mp_prec_t`. See the MPFR documentation for more details on the allowed precision range.

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.

Most MPC functions have a return value of type `int`, which is used to indicate the position of the rounded real or 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. However, some functions do not completely fulfill this: in some cases the sign is not guaranteed, and in some cases a non-zero value is returned although the result is exact; in these cases the function documentation explains the exact meaning of the return value. However, the return value never wrongly indicates an exact computation.

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

`void mpc_set_default_prec` (*mp_prec_t* **`prec`**) [Function]

Set the default precision to be **exactly** *prec* bits. All subsequent calls to `mpc_init` will use this precision, but previously initialized variables are unaffected. This default precision is set to 53 bits initially. It is valid for the real and the imaginary parts alike.

`mp_prec_t mpc_get_default_prec` () [Function]

Returns the default MPC precision in bits.

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

`void mpc_init` (*mpc_t* **`z`**) [Function]

Initialize *z*, 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. The precision of *x* is the default precision, which can be changed by a call to `mpc_set_default_prec`.

`void mpc_init2` (*mpc_t* **`z`**, *mp_prec_t* **`prec`**) [Function]

Initialize *z*, set its precision to be *prec* bits, and set its real and imaginary parts to NaN.

`void mpc_init3` (*mpc_t* **`z`**, *mp_prec_t* **`prec_r`**, *mp_prec_t* **`prec_i`**) [Function]

Initialize *z*, set the precision of its real part to *prec_r* bits, the precision of its imaginary part to *prec_i* bits, and set its 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, z;
  mpc_init  (x); /* use default precision */
  mpc_init2 (y, 256); /* precision exactly 256 bits */
  mpc_init3 (z, 100, 50); /* 100/50 bits for the real/imaginary part */
  ...
  /* Unless the program is about to exit, do ... */
  mpc_clear (x);
  mpc_clear (y);
  mpc_clear (z);
}
```

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`, *mp_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.

`mp_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` (*mp_prec_t** `pr`, *mp_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).

`int mpc_set` (*mpc_t* `rop`, *mpc_t* `op`, *mpc_rnd_t* `rnd`) [Function]
`int mpc_set_ui` (*mpc_t* `rop`, *unsigned long int* `op`, *mpc_rnd_t* `rnd`) [Macro]
`int mpc_set_si` (*mpc_t* `rop`, *long int* `op`, *mpc_rnd_t* `rnd`) [Macro]
`int mpc_set_d` (*mpc_t* `rop`, *double* `op`, *mpc_rnd_t* `rnd`) [Macro]
`int mpc_set_fr` (*mpc_t* `rop`, *mpfr_t* `op`, *mpc_rnd_t* `rnd`) [Macro]

Set the value of *rop* from *op*, rounded to the precision of *rop* with the given rounding mode *rnd*. Except for `mpc_set`, 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_d_d` (*mpc_t* `rop`, *double* `op1`, *double* `op2`, *mpc_rnd_t* `rnd`) [Function]
`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_fr_fr (*mpc_t* **rop**, *mpfr_t* **op1**, *mpfr_t* **op2**, *mpc_rnd_t* **rnd**) [Function]
int mpc_set_ui_fr (*mpc_t* **rop**, *unsigned long int* **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*.

5.3 Combined Initialization and Assignment Functions

int mpc_init_set (*mpc_t* **rop**, *mpc_t* **op**, *mpc_rnd_t* **rnd**) [Macro]
int mpc_init_set_ui (*mpc_t* **rop**, *unsigned long int* **op**, *mpc_rnd_t* **rnd**) [Macro]

Initialize *rop* and set its value from *op*, rounded with the rounding mode *rnd*. The precision of *rop* will be taken from the active default precision, as set by `mpc_set_default_prec`.

int mpc_init_set_ui_ui (*mpc_t* **rop**, *unsigned long int* **op1**, *unsigned long int* **op2**, *mpc_rnd_t* **rnd**) [Macro]
int mpc_init_set_ui_fr (*mpc_t* **rop**, *unsigned long int* **op1**, *mpfr_t* **op2**, *mpc_rnd_t* **rnd**) [Macro]
int mpc_init_set_si_si (*mpc_t* **rop**, *long int* **op1**, *long int* **op2**, *mpc_rnd_t* **rnd**) [Macro]

Initialize *rop*, set its real part from *op1*, and its imaginary part from *op2*, rounded with the rounding mode *rnd*. The precision of *rop* will be taken from the active default precision, as set by `mpc_set_default_prec`.

5.4 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`.

5.5 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.

int mpc_arg (*mpfr_t* **rop**, *mpc_t* **op**, *mpc_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*.

5.6 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**, *mp_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. Note the destination is of type `mpfr_t`, not `mpc_t`.

`int mpc_norm` (*mpfr_t* **rop**, *mpc_t* **op**, *mp_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.

5.7 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*. Here, when the return value is 0, it means the result is exact, but the contrary may be false.

`void 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*.

`void 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 $]-\pi,\pi]$.

5.8 Trigonometric Functions

`void 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*.

`void 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*.

`void 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*.

`void 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*.

`void 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*.

`void 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*.

5.9 Input and Output Functions

Functions that perform input from a standard input/output stream, and functions that output to a standard input/output 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, you need to include '`stdio.h`' before '`mpc.h`'.

`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*. First the real part is printed, then `+I*` followed by the imaginary part. The base may vary from 2 to 36. Print at most *n_digits* significant digits for each part, or if *n_digits* is 0, enough digits so that *op* can be read back exactly rounding to nearest.

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.

Return the number of characters written.

`size_t mpc_inp_str` (*mpc_t* **rop**, *FILE* ***stream**, *int* **base**, *mpc_rnd_t* **rnd**) [Function]

Input a string in base *base* from stdio stream *stream*, rounded according to *rnd*, and put the read complex in *rop*. Each of the real and imaginary part should be of the form '`M@N`' or, if the base is 10 or less, alternatively '`MeN`' or '`MEN`'. '`M`' is the mantissa and '`N`' is the exponent. The mantissa is always in the specified base. The exponent is always read in decimal. This function first reads the real part, then `+` or `-` followed by `I*` or `i*` and the absolute value of the imaginary part.

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

Note: since the real part is read with `mpfr_inp_str`, it should be followed by a whitespace, thus `17+I*42` is invalid, but `17 +I*42` is valid.

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

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.

`void mpc_random (mpc_t rop)` [Function]

Generate a random complex, with real and imaginary parts uniformly distributed in the interval -1 < X < 1.

`void mpc_random2 (mpc_t rop, mp_size_t max_size, mp_exp_t max_exp)` [Function]

Generate a random complex, with real and imaginary part of at most *max_size* limbs, with long strings of zeros and ones in the binary representation. The exponent of the real (resp. imaginary) part is in the interval $-exp$ to *exp*. This function is useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger corner-case bugs. Negative parts are generated when *max_size* is negative.

`const char * mpc_get_version (void)` [Function]

Return the MPC version, as a null-terminated string.

`MPC_VERSION_MAJOR` [Macro]
`MPC_VERSION_MINOR` [Macro]
`MPC_VERSION_PATCHLEVEL` [Macro]
`MPC_VERSION_STRING` [Macro]

`MPC_VERSION_MAJOR`, `MPC_VERSION_MINOR` and `MPC_VERSION_PATCHLEVEL` are respectively the major, minor and patch level of MPFR 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).

5.11 Internals

These types and functions were 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 `mpc_t` type consists of two fields of type `mpfr_t`, one for the real part, one for the imaginary part. These fields can be accessed through `MPC_RE(z)` and `MPC_IM(z)`.

Normally the real and imaginary part have the same precision, but the function `mpc_init3` enables one to have different precisions, and the user may also use `mpfr_set_prec` to change their precision. The macro `MPC_MAX_PREC(z)` gives the maximum of the precisions of the real and imaginary parts.

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 4.2.2, `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 2.3.1, `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.

Concept Index

A

Arithmetic functions 11

C

Comparison functions 10
Complex arithmetic functions 11
Complex assignment functions 9
Complex comparisons functions 10
Complex functions 8
Complex input and output functions 13
Complex number 6
Conditions for copying MPC 1
Copying conditions 1

I

I/O functions 13
Initialization and assignment functions 10
Input functions 13
Installation 3

L

Logarithm 12

M

Miscellaneous complex functions 13
'mpc.h' 6

O

Output functions 13

P

Power functions 12
Precision 6
Projection and Decomposing Functions 10

R

Reporting bugs 5
Rounding Mode 6

T

Trigonometric functions 12

U

User-defined precision 8

Function and Type Index

mp_prec_t ... 6
mpc_abs ... 12
mpc_add ... 11
mpc_add_fr ... 11
mpc_add_ui ... 11
mpc_arg ... 10
mpc_clear ... 8
mpc_cmp ... 10
mpc_cmp_si ... 10
mpc_cmp_si_si ... 10
mpc_conj ... 12
mpc_cos ... 12
mpc_cosh ... 13
mpc_div ... 11
mpc_div_2exp ... 12
mpc_div_fr ... 11
mpc_div_ui ... 11
mpc_exp ... 12
mpc_fr_div ... 11
mpc_fr_sub ... 11
mpc_get_default_prec ... 8
mpc_get_prec ... 9
mpc_get_prec2 ... 9
mpc_get_version ... 14
mpc_imag ... 10
mpc_imagref ... 10
mpc_init ... 8
mpc_init_set ... 10
mpc_init_set_si_si ... 10
mpc_init_set_ui ... 10
mpc_init_set_ui_fr ... 10
mpc_init_set_ui_ui ... 10
mpc_init2 ... 8
mpc_init3 ... 8
mpc_inp_str ... 13
mpc_log ... 12
mpc_mul ... 11
mpc_mul_2exp ... 12
mpc_mul_fr ... 11
mpc_mul_i ... 11
mpc_mul_si ... 11
mpc_mul_ui ... 11
mpc_neg ... 12
mpc_norm ... 12
mpc_out_str ... 13
mpc_proj ... 11
mpc_random ... 14
mpc_random2 ... 14
mpc_real ... 10
mpc_realref ... 10
mpc_rnd_t ... 6
mpc_set ... 9
mpc_set_d ... 9
mpc_set_d_d ... 9
mpc_set_default_prec ... 8
mpc_set_fr ... 9
mpc_set_fr_fr ... 9
mpc_set_prec ... 9
mpc_set_si ... 9
mpc_set_si_si ... 9
mpc_set_ui ... 9
mpc_set_ui_fr ... 10
mpc_set_ui_ui ... 9
mpc_sin ... 12
mpc_sinh ... 13
mpc_sqr ... 11
mpc_sqrt ... 12
mpc_sub ... 11
mpc_sub_fr ... 11
mpc_sub_ui ... 11
mpc_t ... 6
mpc_tan ... 12
mpc_tanh ... 13
mpc_ui_div ... 11
mpc_ui_sub ... 11
mpc_ui_ui_sub ... 11
mpc_urandom ... 13
MPC_VERSION_MAJOR ... 14
MPC_VERSION_MINOR ... 14
MPC_VERSION_PATCHLEVEL ... 14
MPC_VERSION_STRING ... 14

Table of Contents

MPC Copying Conditions **1**

1 Introduction to MPC **2**
- 1.1 How to use this Manual 2

2 Installing MPC **3**
- 2.1 Other 'make' Targets 3
- 2.2 Known Build Problems 4

3 Reporting Bugs **5**

4 MPC Basics **6**
- 4.1 Nomenclature and Types 6
- 4.2 Function Classes 6
- 4.3 MPC Variable Conventions 6
- 4.4 Rounding Modes 6

5 Complex Functions **8**
- 5.1 Initialization Functions 8
- 5.2 Assignment Functions 9
- 5.3 Combined Initialization and Assignment Functions 10
- 5.4 Comparison Functions 10
- 5.5 Projection and Decomposing Functions 10
- 5.6 Basic Arithmetic Functions 11
- 5.7 Power Functions and Logarithm 12
- 5.8 Trigonometric Functions 12
- 5.9 Input and Output Functions 13
- 5.10 Miscellaneous Functions 13
- 5.11 Internals 14

Contributors **15**

References **16**

Concept Index **17**

Function and Type Index **18**