

HIGH PRECISION ARITHMETIC LIBRARY PROGRAMMER'S MANUAL

Copyright (C) 2005 - 2010 Ivano Primi [jivprimi \(a\) libero \(dot\) it](mailto:jivprimi (a) libero (dot) it)

Last Update: 2010-07-21

Contents

1	Summary	3
2	License	3
3	General Technical Comments	4
4	General overview	4
5	Dealing with runtime errors	7
6	Compiling and linking	9
7	Real arithmetic	12
7.1	Real constants	13
7.2	Extended Precision Floating Point Arithmetic	14
7.3	Extended Precision Math Library	26
7.4	Applications of Extended Precision Arithmetic	29
8	Complex Arithmetic	30
8.1	Complex constants	31
8.2	Extended Precision Complex Arithmetic	31
8.3	Extended Precision Complex Math Library	45
9	The C++ interface	49
10	Compiling and linking with the C++ wrapper	51
11	The xreal class	52
12	The xcomplex class	60
13	Acknowledgments	71
14	GNU Free Documentation License	71

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the [section "GNU Free Documentation License"](#).

1 Summary

The High Precision Arithmetic (HPA) library implements a high precision floating point arithmetic together with a comprehensive set of support functions. The general areas covered by these functions include:

- Extended Precision Arithmetic,
- Extended Precision Math Library,
- Applications of High Precision Computation.

The math library support includes evaluation of trigonometric, inverse trigonometric, hyperbolic, logarithm, and exponential functions at the same precision as the floating point math itself. The HPA library also supports high precision complex arithmetic and includes an Extended Precision Complex Math Library.

2 License

The HPA library has been derived from a branch of the source code of the CCMath library, which is a work by Daniel A. Atkinson. Since Daniel A. Atkinson released the code of the CCMath Library under GNU Lesser General Public License, it has been possible for Ivano Primi to modify, complete and redistribute this source code under the same terms.

The HPA (abbreviation of High Precision Arithmetic) Library is then copyrighted by Ivano Primi <ivprimi (at) libero (dot) it> and Daniel A. Atkinson. As for the CCMath Library, its source code is released under the terms of the GNU Lesser General Public License, as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

The HPA library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, see <http://www.gnu.org/licenses/>. Write to the email address

<ivprimi (at) libero (dot) it> .

if you want to ask for additional information, report bugs or submit patches.

3 General Technical Comments

The functions forming the HPA library are all implemented in a portable fashion in the C language. The IEEE 754 standard for floating point hardware and software is assumed in the PC/Unix version of this library. The normal configuration of the library employs a floating point mantissa of 112 bits, which corresponds approximately to a 32 decimal digit precision. Higher precision is available as an option. An extended floating point number is represented as a combination of the following elements:

- sign bit(s): 0 -> positive, 1 -> negative ;
- exponent(e): 15-bit biased integer (bias=16383) ;
- mantissa(m): 7 words of 16 bit length with the leading 1 explicitly represented .

The range of representable numbers is then given by

$$2^{16384} > x > 2^{-16383}$$

or

$$1.19 \cdot 10^{4932} > x > 1.68 \cdot 10^{-4932} .$$

Special values of the exponent are:

- all ones -> infinity (floating point overflow)
- all zeros -> number = zero.

Underflow in operations is handled by a flush to zero. Thus, a number with the exponent zero and nonzero mantissa is invalid (not-a-number). From the point of view of the HPA library, a complex number is simply a structure formed by two extended floating point numbers, representing respectively the real and the imaginary part of the complex number.

4 General overview

The HPA library is composed by *two modules*. The first one is formed by the functions for real arithmetic, i.e. by the functions operating on real arguments. The second one is formed by all functions which manipulate complex arguments. The lists of the functions which compose the HPA library are in the header

files `xpre.h` and `cxpre.h`. `xpre.h` is the header file for real arithmetic, it contains the definition of an extended precision real number (`struct xpr`), and the declarations of the functions provided by the library to manipulate real arguments. The numeric type `struct xpr` can be used to declare and define real variables, just as in

```
struct xpr q;
```

The size of a variable of type `struct xpr` is given by $(2 \times \text{XDIM} + 2)$ bytes, where `XDIM` is a constant defined in the file `xpre.h` (namely, in the file `hpaconf.h` which is included by `xpre.h`).

`cxpre.h` is the header file for complex arithmetic, it contains the definition of an extended precision complex number (`struct cxpr`), and the declarations of the functions supplied by the library to manipulate complex arguments. The numeric type `struct cxpr` can be used to declare and define complex variables, just as in

```
struct cxpr q;
```

The size of a variable of type `struct cxpr` is given by $(4 \times \text{XDIM} + 4)$ bytes, where `XDIM` is the same constant as above.

Before declaring or defining variables of type `struct xpr` and before using anyone of the functions declared in the header file `xpre.h`, you have to insert the line

```
#include <xpre.h>
```

in your source code file. Similarly, before declaring or defining variables of type `struct cxpr` and before using anyone of the functions declared in the header file `cxpre.h`, you have to add the line

```
#include <cxpre.h>
```

to your source code file.

After including in your source code the header file `xpre.h` or, if you also need functions handling complex arguments, the header file `cxpre.h`, you can start to use the HPA library by defining all variables and calling all functions which are necessary for your computations. In fact, the HPA library DOES NOT REQUIRE that a special initialization routine must be called before any other function of the library. Moreover, variables of type `struct xpr` or `struct cxpr` DO NOT NEED to be initialized before they can be used.

When the HPA library was written, a sort of namespace was created for all the identifiers used by the library. This has been achieved by sticking to the following rules:

1. Identifiers of functions and types are lowercase; the unique exception to this rule is the function `xisNaN()`. Their names start by `x`, if they are defined in `xpre.h`, or by `cx`, if they are defined in `cxpre.h`, with the only exception of a few functions, which have a name ending by `tox` or `tocx`:

```

strttox(), strtocx(), atox(), atocx(),

dbltox(), dctocx(), fltttox(), fctocx(),

inttox(), ictocx(), uinttox(), uctocx());

```

2. The names of the macros defined by the HPA library are all uppercase and start by **X** or **CX**;
3. The names of the global constants defined by the HPA library start by **x** (real constants) or by **cx** (complex constants) and the letter which immediately follows this prefix is always uppercase, just as in **xZero**, **xPi**, **cxOne**, **cxIU** (IU stays for imaginary unit);
4. The unique global variable defined by the HPA library is an error indicator of **int** type, whose name is **xErrNo**.

This is a trivial program showing the use of the HPA library:

```

#include <stdio.h>
#include <xpre.h>

int main (void)
{
    struct xpr s;
    int i, n;

    do
    {
        printf ("Give me a number, n = ? \t");
        scanf ("%d", &n);
        s = xZero;
        for (i = 0; i <= n; i++)
            s = xadd (s, xpr2(xOne, i), 0);
        printf ("The sum 2^0 + 2^1 + ... + 2^n is equal to\n");
        xprxpr (s, 30);
        putchar ('\n');
    } while (n > 0);
    return 0;
}

```

This program takes in input from the user an integer value **n** and prints on the screen the sum of the first **n** powers of 2. In the program we use the functions **xpr2()** and **xprxpr()**. **xpr2(x, n)**, where **n** is an integer, returns **x * 2ⁿ**,

while `xprxpr(x, m)`, where `m` is an integer, prints on the screen the number `x` with `m` decimal digits after the decimal point (`.`) .

The HPA library is **NOT** thread safe. Some of the HPA internal data could get corrupted if multiple HPA functions are active at the same time. This is due to the fact that some functions of the HPA library use static variables to store information. The user should then guarantee that only one thread is performing HPA functions. This can usually be achieved by a call to the operating system to obtain a *semaphore*, *mutex*, or *critical code section*; the operating system will then guarantee that only one HPA thread will be active at a time.

5 Dealing with runtime errors

During the use of the HPA library it could happen to pass to a function an illegal argument, namely, an argument whose value is not compatible with the mathematical definition of the function. For example, this occurs when a negative value is passed to the function `xsqrt()`. This function computes and returns the (real) square root of its argument, but the square root of a number is defined only for non-negative numbers. So, if `x` is less than zero, `xsqrt(x)` can not be computed and a mathematical error occurs, a so called *domain error*. Another type of mathematical error occurs when the second argument of the division function (`xdiv()`) is zero: since it is impossible to divide a number by zero, a *division by zero* error occurs. What exactly happens when a mathematical error is detected during the execution of a function depends on the way the HPA library was compiled when it was installed on the system where you are working. If, during the installation process, the default setting was left unchanged, then, whenever a runtime error occurs within a function of the HPA library, this function will set an external error indicator to a suitable value. This value can be looked up later to know what exactly went wrong. The name of the variable of type `int` used as error indicator is `xErrNo`. Before any function of the HPA library is executed, the value of `xErrNo` is 0. If a mathematical error occurs when the first HPA function is called, then `xErrNo` is set to a suitable positive value, which indicates the exact type of the error. After, `xErrNo` is modified if and only if, during the execution of an HPA function, another mathematical error occurs. `xErrNo` is never reset to 0 by the HPA library; therefore, in order to detect possible errors, one has to set `xErrNo` to zero before calling any function of the HPA library. An example of error handling is given by:

```
#include <stdio.h>
#include <xpre.h>

extern int xErrNo;

int main (void)
```

```

{
    int n;
    struct xpr sr;

    do
    {
        printf ("Give me a number, n = ? \t");
        scanf ("%d", &n);
        xErrNo = 0;
        sr = xsqrt (inttox (n));
        if (xErrNo == 0)
        {
            printf ("The square root of %d is\n", n);
            xprxpr (sr, 30);
            putchar ('\n');
        }
        else
            fprintf (stderr, "*** Error: Out of domain\n");
    } while (n != 0);
    return 0;
}

```

In this program `xErrNo` is reset to zero, at each execution of the `do {...} while();` loop, before the call of the `xsqrt()` function.

However, the HPA library could be compiled to deal differently with runtime errors. For example, in case of error a suitable message could be printed on *stderr* and the library could also cause the termination of the calling program via `exit(1)`. Finally, the library could also be compiled to ignore any mathematical error (sigh !). To know how the routines of the HPA library deal with errors is sufficient to examine the file `hpaconf.h` (which is automatically included by `xpre.h` and `cxpre.h`). This file defines the macro:

- `XERR_DFL` to mean that, in case of error, `xErrNo` is suitably set;
- `XERR_WARN` to mean that, in case of error, a suitable message is printed on *stderr*;
- `XERR_EXIT` to mean that, in case of error, the calling program is terminated through a call to `exit(1)` after printing a message on *stderr*;
- `XERR_IGN` to mean that, in case of error, nothing is done or signaled.

When the macro `XERR_DFL` is defined, the header file `xpre.h` also defines the macros `XENONE`, `XEDIV`, `XEDOM`, `XEBADEXP`, `XFPOFLOW` and `XNERR`:

```

#define XENONE    0    /* No error          */
#define XEDIV     1    /* Division by zero */

```



```

#define XEDOM    2    /* Out of domain    */
#define XEBADEXP 3    /* Bad exponent    */
#define XFPOFLOW 4    /* Floating point overflow */

#define XNERR     4    /* Number of the non-null error codes */

```

These macros can be used, together with `xErrNo`, to recover the exact type of the error occurred during the execution of a routine of the HPA library.

6 Compiling and linking

Together with the HPA library is installed a little and simple program called `hpaconf`. You can use it to quickly compile and build your programs. If `PREFIX` is the root directory chosen to install the HPA library (the default value for `PREFIX` is `/usr/local`), then `hpaconf` should be installed inside `PREFIX/bin`. You can know where `hpaconf` is installed by launching the command

```
which hpaconf
```

in your console or terminal. In the following it is assumed that the directory `PREFIX/bin` is included in your `PATH` environment variable (This is surely true if the command `which` was able to find `hpaconf`).

`hpaconf` recognizes four options:

- `-v` to return the version of HPA installed on your system,
- `-c` to return the flags needed to compile with HPA,
- `-l` to return the flags needed to link against HPA,
- `-n` to print a newline at the end of the output.

The option `-v` cannot be used together with the options `-c` and `-l`, but it may be used together with `-n`:

```
hpaconf -v
```

prints on the standard output (console) the version of HPA installed on your system,

```
hpaconf -v -n
```

or

```
hpaconf -n -v
```

(order does not matter) behaves exactly the same but also prints a newline to force the following output to be written on the next line.

The options `-c` and `-l` cannot be used together with `-v`, but they can be used both at the same time and can also be accompanied by the option `-n`. Of course, order does not matter.

```
hpaconf -c
```

prints on the standard output the flags needed to compile with HPA,

```
hpaconf -l
```

prints on the standard output the flags needed to link against HPA,

```
hpaconf -c -l
```

or

```
hpaconf -l -c
```

prints both the flags to compile with HPA and the flags to link against HPA.

If the option `-n` is added, then the information printed is followed by a newline. An example will show the usefulness of `hpaconf`. To compile the source file `example.c` you should tell the compiler where looking for the header files of HPA and for the library itself; to do this it is sufficient to specify the related paths through the options `-I` and `-L`, at least if you are using GCC/G++ as C/C++ compiler. But in this way you are constrained to remember the path where HPA was installed, and this is quite uncomfortable. With `hpaconf` you can simply use the command

```
cc -c $(hpaconf -c) example.c
```

or

```
cc -c 'hpaconf -c' example.c
```

to compile the file `example.c` and obtain the object file `example.o`. The previous one is the right form of the command for a shell sh compatible, like ash, bash or ksh. If you are using another shell, probably the right form to obtain the expansion of the command `hpaconf -c` will be another one (see the manual of your preferred shell for this). On GNU/Linux, bash is the default shell for all users. If this is not true for your machine, ask your system administrator. Once you have obtained the object file `example.o`, you may do the linkage by using the command (for a shell sh-compatible):

```
cc example.o $(hpaconf -l) -o example
```

or

```
cc example.o 'hpaconf -l' -o example
```

If you want, you may also compile and build at the same time by using

```
cc example.c $(hpaconf -c -l) -o example
```

or

```
cc example.c 'hpaconf -c -l' -o example
```

which will compile `example.c` and build the program `example`. Naturally, compiling and building at the same time is only possible when the source code of your program is entirely contained in one file.

The `hpaconf` program may also tell you which options were used to compile the HPA library for the system where you are working. This information is displayed, together with hints about usage, when `hpaconf` is called with no options. This is the output obtained on my personal machine:

```
ivano@darkstar[~]$ hpaconf
*** Usage:  hpaconf [-v] [-n] or
***          hpaconf [-c] [-l] [-n]

*** Meaning of the options:
    -v      returns the current version of the HPA library,
    -c      returns the flags to compile with HPA library,
    -l      returns the flags to link against HPA library,
    -n      adds a newline at the end of the output.

----- Features of the HPA library (including build options) -----

Size of an extended precision floating point value (in bytes): 16
Number of bits available for the sign:      1
Number of bits available for the exponent: 15
Number of bits available for the mantissa: 112
Decimal digits of accuracy:                  ~33
Dynamic range supported:                      $2^{16384} > x > 2^{-(16383)}$  i.e.
                                            $1.19 \cdot 10^{4932} > x > 1.68 \cdot 10^{-(4932)}$ 

In case of floating point error
the global (extern) variable 'xErrNo' is suitably set
```

The first value shown after the header `Features of the HPA library` is the size of a variable of `struct xpr` type. When I installed the HPA library on my machine, I chose to compile it by setting `XDIM` to 7. Then a variable of type `struct xpr` turns out to have a size of $16 = 2 * 7 + 2$ bytes. Since `XDIM` could have been set to another value on the system where you are working (`XDIM` could also have the values 11, 15, 19, 23, 27 and 31), the first value shown by `hpaconf` could differ on your machine. The next 2 values (bits available

for sign and exponent) are the same for all installations. The decimal digits of accuracy depend on the value of `XDIM`, namely, they increase together with `XDIM` till to a maximum of 149 when `XDIM` is 31. As you can see, the value of `XDIM` determines the accuracy provided by the mathematical functions of the HPA library. Even if a larger value for `XDIM` implies a greater accuracy, together with `XDIM` increase the memory and the time requested by the routines of the HPA library to perform their computations.

The dynamic range supported by the HPA library is always the same, or almost. The HPA library can be compiled to deal differently with the error conditions (see [previous section](#)). In the last line of the output of `hpaconf`, you can find information about the way runtime errors are handled. This information can also be retrieved from the file `hpaconf.h`, as explained in the previous section.

The file `hpaconf.h` also defines the macro `HPA_VERSION`, a string containing the version number of the release of the HPA library currently in use.

7 Real arithmetic

The first module of the HPA library is made of functions for Extended Precision Floating Point Arithmetic, functions of the Extended Precision Math Library, and of applications of the Extended Precision Arithmetic. They are all declared in the file `xpre.h` together with some macros and numerical constants. The header file `xpre.h` also defines the structure `xoutflags`:

```
struct xoutflags
{
    short fmt, notat, sf, mfw, lim;
    signed char padding, ldel, rdel;
};
```

A structure of such kind is used by the output functions `xfout()`, `xout()` and `xsout()` to know how they have to print numbers.

The field `notat` refers to the notation: it can be equal to `XOUT_SCIENTIFIC` (scientific notation) or to `XOUT_FIXED` (floating point notation).

Both `XOUT_SCIENTIFIC` and `XOUT_FIXED` are macros defined inside `xpre.h`.

The field `sf` refers to the sign: if `sf` is not zero, then every non-negative number is printed with a plus sign (+) ahead.

The field `mfw` indicates the minimum field width to use in printing numbers. When `mfw` is zero no minimum field width is used. When `mfw` is negative, then the actual minimum field width is given by `-mfw` and the printed number is left adjusted on the field boundary (default is right justification).

`lim` has a different meaning depending on the notation in use. Together with the scientific notation, `lim` gives the number of decimal digits to the right of the decimal point (`lim+1` = total digits displayed). Otherwise, `lim + 1` is the

number of significant digits displayed. When `lim` is negative, the default value (6) is used.

Finally, `padding` defines the padding character to use together with a non-zero minimum field width. If `padding` is negative, then the default padding char (i.e. the blank character) is used.

The fields `fmt`, `ldel` and `rdel` are ignored by the functions `xfout()`, `xout()` and `xsout()`. They are only used by the functions `cxfout()`, `cxout()` and `cx्सout()` to format and print complex numbers.

`fmt` specifies the format to use in printing complex numbers. The possible values for `fmt` are `XFMT_STD`, `XFMT_RAW` and `XFMT_ALT` (these macros are declared inside `cxpre.h`). If `fmt == XFMT_STD`, then the complex number (`a`, `b`) is printed using the notation `a+bi` or `a-bi` (depending on the sign of `b`). Of course, `a` and `b` are printed according to the rules exposed above. If `fmt == XFMT_RAW`, then (`a`, `b`) is printed in the form

```
a<two blank spaces>b
```

just like in

```
1.0  2.5
```

assuming that `a = 1.0` and `b = 2.5`. Finally, if `fmt == XFMT_ALT`, then (`a`,`b`) is printed as

```
<left delimiter>a, b<right delimiter>
```

where `<left_delimiter>` and `<right_delimiter>` are the characters given by the fields `ldel` and `rdel` respectively. If `ldel < 0` or `rdel < 0`, then (or) is used as default `<left_delimiter>` / `<right_delimiter>`.

Care that none of the functions `xfout()`, `xout()`, `xsout()`, `cxfout()`, `cxout()` or `cx्सout()` adds a newline at the end of the printed number.

7.1 Real constants

The header file `xpre.h` defines several constants. Between the constants defined in `xpre.h` there are those ones corresponding to particular mathematical values:

```
extern const struct xpr xZero, xOne, xTwo, xTen;
extern const struct xpr xPinf, xMinf, xNaN;
extern const struct xpr xPi, xPi2, xPi4, xEe, xSqrt2;
extern const struct xpr xLn2, xLn10, xLog2_e, xLog2_10,
                        xLog10_e;
```

`xZero` (= 0), `xOne` (= 1), `xTwo` (= 2) and `xTen` (= 10) do not need a comment. `xPi`, `xPi2`, `xPi4`, `xEe`, `xSqrt2`, `xLn2`, `xLn10`, `xLog2_e`, `xLog2_10`, `xLog10_e` represent, respectively, the values π (= 3.14159...), $\pi/2$, $\pi/4$, e (= 2.7182818...),

square root of 2 ($= 1.4142135\dots$), natural logarithm of 2 and 10, base-2 logarithm of e and 10, 10-base logarithm of e .

`xPinf`, `xMinf` and `xNaN` are *special values*: `xPinf` represents the value `+oo` (plus infinity), `xMinf` the value `-oo` (minus infinity) and `xNaN` is used to mean an invalid number (NaN stays for Not a Number). `xPinf` and `xMinf` are usually returned by a function to signal a floating point overflow, positive and negative respectively, while `xNaN` is returned by the functions converting ASCII strings to floating point numbers to indicate that the string given to them as argument does not contain any valid number. `xPinf`, `xMinf` and `xNaN` should never be used as arguments for functions, since this has unpredictable results.

7.2 Extended Precision Floating Point Arithmetic

The arithmetic functions support the basic computations and input/output operations for extended precision floating point mathematics. Some of the operations supply capabilities designed to enhance the computational efficiency of this arithmetic (e.g., `xpwr`). What follows is their complete list including the synopsis for each of them.

xadd - Add or subtract two extended precision numbers.

```
struct xpr xadd(struct xpr s, struct xpr t, int f)
```

- `s` = structure containing first number;
- `t` = structure containing second number;
- `f` = control flag: if 0, then `s` and `t` are added, else they are subtracted (`s-t`).

The value returned by `xadd()` is the result of the addition/subtraction. `xadd()` can return `xPinf` or `xMinf` to signal a floating point overflow.

xmul - Multiply two extended precision numbers.

```
struct xpr xmul(struct xpr s, struct xpr t)
```

- `s` = structure containing first number;
- `t` = structure containing second number.

The value returned by `xmul()` is the product `s*t`. It can be `xPinf` or `xMinf` in case of overflow.

xdiv - Divide one extended precision number by a second.

`struct xpr xdiv(struct xpr s,struct xpr t)`

- `s` = structure containing numerator;
- `t` = structure containing denominator.

The value returned by `xdiv()` is the quotient `s/t`.

xneg - Change sign (unary minus).

`struct xpr xneg(struct xpr s)`

- `s` = structure containing input number.

The value returned by `xneg()` is a structure containing its argument with the changed sign.

xabs - Compute absolute value.

`struct xpr xabs(struct xpr s)`

- `s` = structure containing input number.

The value returned by `xabs()` is a structure containing the absolute value of its argument.

x_exp - Extract the binary exponent.

`int x_exp(const struct xpr *p)`

- `p` = pointer to an extended precision number.

The value returned by `x_exp()` is the binary exponent (power of 2) of the number pointed to by `p`.

x_neg - Test the sign of an extended precision number.

`int x_neg(const struct xpr *p)`

- `p` = pointer to an extended precision number.

The value returned by `x_neg()` is a sign flag, with 0 meaning positive input, 1 negative input. The input is, of course, the number pointed to by `p`. *Note that neither `x_exp()` nor `x_neg()` alter the input number.*

xpwr - Raise to integer powers.

`struct xpr xpwr(struct xpr s,int n)`

- `s` = structure containing input number;

- `n` = power desired.

The return value is the `n`th power of the first argument.

xpr2 - Multiplication by a power of 2.

```
struct xpr xpr2(struct xpr s,int m)
```

- `s` = structure containing input number;
- `m` = power of two desired.

The return value is the product of the first argument by the `m`th power of two.
`xpr2()` returns `xZero` in case of underflow, `xPinf` or `xMinf` in case of overflow.

xpow - Power function.

```
struct xpr xpow (struct xpr x, struct xpr y)
```

- `x` = base;
- `y` = exponent.

The return value is the power of the first argument raised to the second one.
 Note that the first argument must be greater than zero.

xprcmp - Compare two extended precision numbers.

```
int xprcmp (const struct xpr *p, const struct xpr *q)
```

- `p` = pointer to first number;
- `q` = pointer to second number.

The value returned by `xprcmp()` is a comparison flag, with 1 meaning `*p` greater than `*q`, 0 meaning `*p` equal to `*q`, and -1 meaning `*p` less than `*q`.
Note that the input numbers are not altered by `xprcmp()`.

xeq - Check if two numbers are or are not equal.

```
int xeq (struct xpr x1, struct xpr x2)
```

- `x1` = first number;
- `x2` = second number.

The return value is 0 if `x1` and `x2` are different, else a non-null value.

xneq - Check if two numbers are or are not equal.


```
int xneq (struct xpr x1, struct xpr x2)
```

- **x1** = first number;
- **x2** = second number.

The return value is 0 if **x1** and **x2** are equal, else a non-null value.

xgt - Check if a number is greater than another one.

```
int xgt (struct xpr x1, struct xpr x2)
```

- **x1** = first number;
- **x2** = second number.

The return value is 0 if **x1** is less or equal to **x2**, else a non-null value.

xge - Check if a number is greater or equal to another one.

```
int xge (struct xpr x1, struct xpr x2)
```

- **x1** = first number;
- **x2** = second number.

The return value is 0 if **x1** is less than **x2**, else a non-null value.

xlt - Check if a number is less than another one.

```
int xlt (struct xpr x1, struct xpr x2)
```

- **x1** = first number;
- **x2** = second number.

The return value is 0 if **x1** is greater or equal to **x2**, else a non-null value.

xle - Check if a number is less or equal to another one.

```
int xle (struct xpr x1, struct xpr x2)
```

- **x1** = first number;
- **x2** = second number.

The return value is 0 if **x1** is greater than **x2**, else a non-null value.

xisNaN - Check if a number is valid or not.

```
int xisNaN (const struct xpr *u)
```

- `u` = pointer to a structure containing a number.

The return value is 0 if `*u` is a valid number, else a non-null value.

Remark: A number is considered invalid (not-a-number) when its exponent is zero but not its mantissa.

xisPinf - Check if a number is equal to `+oo`.

```
int xisPinf (const struct xpr *u)
```

- `u` = pointer to a structure containing a number.

The return value is 1 if `*u` is equal to `xPinf (+oo)`, 0 otherwise.

xisMinf - Check if a number is equal to `-oo`.

```
int xisMinf (const struct xpr *u)
```

- `u` = pointer to a structure containing a number.

The return value is 1 if `*u` is equal to `xMinf (-oo)`, 0 otherwise.

xisordnumb - Check if a given number is an ordinary number.

```
int xisordnumb (const struct xpr *u)
```

- `u` = pointer to a structure containing a number.

The return value is 1 if `*u` is a valid number and is neither `xPinf (+oo)` nor `xMinf (-oo)`, else 0.

xis0 - Compare a number with zero.

```
int xis0 (const struct xpr *u)
```

- `u` = pointer to a structure containing a number.

The return value is 0 if `*u` is not zero, else a non-zero value.

xnot0 - Compare a number with zero.

```
int xnot0 (const struct xpr *u)
```

- `u` = pointer to a structure containing a number.

The return value is 0 if `*u` is zero, else a non-zero value.

xsgn - Obtain the sign of a number.

```
int xsgn (const struct xpr *u)
```

- `u` = pointer to a structure containing a number.

The return value is 0 when `*u` is zero or is an invalid number (not-a-number), 1 if `*u` is positive, -1 if `*u` is negative.

Remark: `xPinf` is considered a positive value, `xMinf` a negative value.

xtodbl - Cast extended precision numbers to double precision ones.

`double xtodbl(struct xpr s)`

- `s` = structure containing extended precision input.

The return value is the double precision float corresponding to `s`.

dbltox - Convert double precision numbers to extended precision ones.

`struct xpr dbltox(double y)`

- `y` = double precision floating point input.

The return value is a structure containing extended equivalent of `y`.

xtoflt - Cast extended precision numbers to single precision ones.

`float xtoflt(struct xpr s)`

- `s` = structure containing extended precision input.

The return value is the single precision float corresponding to `s`.

flttox - Convert single precision numbers to extended precision ones.

`struct xpr flttox(float y)`

- `y` = single precision floating point input.

The return value is a structure containing extended equivalent of `y`.

inttox - Convert signed integers to extended precision numbers.

`struct xpr inttox(long n)`

- `n` = integer input.

The return value is a structure containing extended equivalent of `n`.

uinttox - Convert unsigned integers to extended precision numbers.

`struct xpr uinttox(unsigned long n)`

- **n** = integer input.

The return value is a structure containing extended equivalent of **n**.

strtox - Convert a floating point number, expressed as a decimal ASCII string in a form consistent with C, into the extended precision format.

```
struct xpr strttox (const char* s, char** endptr)
```

- **s** = pointer to a null terminated ASCII string expressing a decimal number;
- **endptr** = NULL, or address of a pointer defined outside **strttox()**.

The value returned by **strttox()** is a structure containing the input number in the extended precision format.

Remarks: The **strttox()** function converts the initial portion of the string pointed to by **s** to its extended precision representation.

The expected form of the (initial portion of the) string is: optional leading white space as recognized by the standard library function **isspace()**, an optional plus (+) or minus sign (-), and then a decimal number. A decimal number consists of a nonempty sequence of decimal digits possibly containing a radix character (decimal point, i.e. '.'), optionally followed by a decimal exponent. A decimal exponent consists of an E or e, followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits, and indicates multiplication by a power of 10.

This function returns the converted value, if any. If the correct value would cause overflow, then xPinf or xMInf is returned, according to the sign of the value. If the correct value would cause underflow, xZero is returned. If no conversion is performed, xNaN is returned.

If **endptr** is not NULL, a pointer to the character after the last character used in the conversion is stored in the location referenced by **endptr**. If no conversion is performed, the value of **s** is stored in the location referenced by **endptr**.

atox - Convert a floating point number, expressed as a decimal ASCII string in a form consistent with C, into the extended precision format.

```
struct xpr atox(const char *s)
```

- **s** = pointer to a null terminated ASCII string expressing a decimal number.

Remark: The call **atox(s)** is equivalent to **strttox(s, NULL)**.

xfmod - This function is the extended precision analog of the **fmod** function from the C standard library.

`struct xpr xfmod(struct xpr s, struct xpr t, struct xpr* q)`

- `s` = structure containing argument of `fmod`;
- `t` = structure containing base number (`t` must be different from zero);
- `q` = pointer to store for output integer `m`.

The return value is the extended number with same sign as `s` and absolute value less than that of `t`, satisfying $s = m*t + x$ if $s*t > 0$, or $s = -m*t + x$ if $s*t < 0$.

xfrexp - This function is the extended precision analog of the `frexp` function from the C standard library.

`struct xpr xfrexp(struct xpr s, int *p)`

- `s` = structure containing argument;
- `p` = pointer to store for output exponent `e`.

The return value is the extended number satisfying $x = s*2^{(-e)}$ with $(-1 < x < +1)$.

xfrac - This function returns the fractional part of the input number.

`struct xpr xfrac (struct xpr s)`

- `s` = structure containing argument.

The return value is the fractional part of the number `s`, with the same sign as `s`.

Remark: The fractional part of the number `s` is 0 if `s` is an integer number, otherwise is given by $(-)0.xyz\dots$, where `xyz...` are the digits of `s` following the radix character `(.)` in the decimal representation. `xfrac(s)` has always the same sign as `s`.

xtrunc - This function returns the integer part of the input number.

`struct xpr xtrunc (struct xpr s)`

- `s` = structure containing argument.

The return value is the integer part of the number `s`, with the same sign as `s`.

Remark:

The integer part of the number `s` is given by $(-)xyz\dots$, where `xyz...` are the digits of `s` before the radix character `(.)` in its decimal representation.

xfix - Obtain the integer part of a number (2nd method).

```
struct xpr xfix (struct xpr s)
```

- **s** = structure containing argument.

Remark:

xfix() is provided as an alternative to **xtrunc()**. **xfix()** tries to take into account possible rounding errors and cancel their effects. The use of **xfix()** is strongly suggested whenever the argument is presumed to be an integer number, but it is reasonable to expect that some rounding errors make its actual value a bit different from that one it should be. For instance, when **sizeof(struct xpr) == 64**, on my machine I obtain

```
xtrunc (xdiv (inttox(100), inttox(100))) == xZero
```

while the expect result is **xOne**. This happens since **xdiv (inttox(100), inttox(100))** returns a number a bit lower than 1. On the other hand

```
xfix (xdiv (inttox(100), inttox(100))) == xOne .
```

Since **xfix()** introduces another type of rounding error to give the correct answer in the cases similar to the previous one, it is not always the right choice.

xround - Round an extended precision number to its nearest integer value (halfway cases are rounded away from zero).

```
struct xpr xround (struct xpr s)
```

- **s** = structure containing argument.

The return value is the integer value nearest to **s**.

xceil - Round an extended precision number to the smallest integral value not less than it.

```
struct xpr xceil (struct xpr s)
```

- **s** = structure containing argument.

The return value is the smallest integral value not less than the argument.

xfloor - Round an extended precision number to the largest integral value not greater than it.

```
struct xpr xfloor (struct xpr s)
```

- **s** = structure containing argument.

The return value is the largest integral value not greater than the argument.

xpr_print - Print an extended precision number in scientific or floating point format to a given file.

```
void xpr_print (FILE* stream, struct xpr u, int sc_not,
               int sign, int lim)
```

- **stream** = file where the number must be printed;
- **u** = structure containing number to print;
- **sc_not** = zero to mean floating point format, not zero to mean scientific format;
- **sign** = not zero to put a plus sign (+) before the number if it is non-negative (in case of negative number a minus sign (-) is printed even if the parameter **sign** is zero);
- **lim** = number of decimal digits to the right of the decimal point (**lim+1** = total digits displayed) in case of scientific format, otherwise number of significant digits - 1 (**lim+1** = total of significant digits).

xpr_asprint - Convert an extended precision number to a string.

```
char* xpr_asprint (struct xpr u, int sc_not, int sign, int lim)
```

- **u** = structure containing number to print;
- **sc_not** = zero to mean floating point format, not zero to mean scientific format;
- **sign** = not zero to put a plus sign (+) before the number if it is non-negative (in case of negative number a minus sign (-) is printed even if the parameter **sign** is zero);
- **lim** = number of decimal digits to the right of the decimal point (**lim+1** = total digits displayed) in case of scientific format, otherwise number of significant digits - 1 (**lim+1** = total of significant digits).

xpr_asprint() returns the string with the converted number. The memory for this string is calloc'ed inside the function.

xprxpr - Print an extended precision number in scientific format.

```
void xprxpr(struct xpr u,int lim)
```

- **u** = structure containing number to print;
- **lim** = number of decimal digits to the right of the decimal point (**lim+1** = total digits displayed).

Remark:

```
xprxpr(u, lim)
```

is equivalent to

```
xpr_print(stdout, u, 1, 0, lim)
```

xtoa - This function converts an extended precision number to a string. Scientific format is always used.

```
char* xtoa (struct xpr u,int lim)
```

- **u** = structure containing number to print;
- **lim** = number of decimal digits to the right of the decimal point (**lim+1** = total digits displayed).

Remark:

```
xtoa(u, lim)
```

is equivalent to

```
xpr_asprint(u, 1, 0, lim)
```

xbprint - Print an extended precision number in binary format.

```
void xbprint (FILE* stream, struct xpr u)
```

- **stream** = file where the number must be printed;
- **u** = structure containing number to print.

The **xbprint()** function supports a bit oriented analysis of rounding error effects. It always prints a newline (**\n**) at the end of the binary string.

xprint - Print an extended precision number as a string of hexadecimal numbers.

```
void xprint(FILE* stream, struct xpr u)
```

- **stream** = file where the number must be printed;
- **u** = structure containing number to print.

The **xprint()** function supports a bit oriented analysis of rounding error effects. It always prints a newline (**\n**) at the end.

xfout - Print an extended precision number on file according to a given set of I/O flags.


```
int xfout (FILE * stream, struct xoutflags ofs, struct xpr x)
```

- **stream** = file where the number must be printed;
- **ofs** = structure containing all I/O flags;
- **x** = structure containing number to print.

The return value is 0 in case of success, -1 to mean a failure.

Remark: For the definition of **struct xoutflags** and the meaning of its fields see [section "Real Arithmetic"](#). **xfout()** does not add any newline at the end of the printed number.

xout - Print an extended precision number on **stdout** according to a given set of I/O flags.

```
int xout (struct xoutflags ofs, struct xpr x)
```

- **ofs** = structure containing all I/O flags;
- **x** = structure containing number to print.

The return value is 0 in case of success, -1 to mean a failure.

Remark: For the definition of **struct xoutflags** and the meaning of its fields see [section "Real Arithmetic"](#). **xout()** does not add any newline at the end of the printed number.

xsout - Write an extended precision number on a string according to a given set of I/O flags.

```
int xsout (char* s, unsigned long n, struct xoutflags ofs,  
          struct xpr x)
```

- **s** = pointer to a buffer of characters (char);
- **n** = size of the buffer;
- **ofs** = structure containing all I/O flags;
- **x** = structure containing number to print.

The return value is the number of non-null characters written to the buffer or, if it is greater or equal than **n**, which would have been written to the buffer if enough space had been available.

Remarks: For the definition of **struct xoutflags** and the meaning of its fields see [section "Real Arithmetic"](#). **xsout()** always adds a null character ('\0') at the end of the written number. **xsout()** does not write more than **n** bytes (including the trailing '\0'). Thus, a return value of **n** or more means that the output was truncated. In this case, the contents of the buffer pointed to by the first argument of **xsout()** are completely unreliable.

7.3 Extended Precision Math Library

The Extended Precision Math Library provides the elementary functions normally supported in a C math library. They are designed to provide full precision accuracy.

xsqrt - Compute the square root of an extended precision number.

```
struct xpr xsqrt(struct xpr x)
```

- **x** = structure containing the input number.

The return value is the square root of the input number. A negative argument results in a domain error.

xexp - Compute the exponential function.

```
struct xpr xexp(struct xpr x)
```

- **x** = structure containing function argument.

The return value is **e** (the base of natural logarithms) raised to **x**.

xexp2 - Compute the base-2 exponential function.

```
struct xpr xexp2(struct xpr x)
```

- **x** = structure containing function argument.

The return value is 2 raised to **x**.

xexp10 - Compute the base-10 exponential function.

```
struct xpr xexp10(struct xpr x)
```

- **x** = structure containing function argument.

The return value is 10 raised to **x**.

xlog - Compute natural (base **e**) logarithms.

```
struct xpr xlog(struct xpr x)
```

- **x** = structure containing function argument.

This function returns the natural logarithm of its argument. A null or negative argument results in a domain error.

xlog2 - Compute base-2 logarithms.

`struct xpr xlog2(struct xpr x)`

- `x` = structure containing function argument.

This function returns the base-2 logarithm of its argument. A null or negative argument results in a domain error.

xlog10 - Compute base-10 logarithms.

`struct xpr xlog10(struct xpr x)`

- `x` = structure containing function argument.

This function returns the base-10 logarithm of its argument. A null or negative argument results in a domain error.

xtan - Tangent function.

`struct xpr xtan(struct xpr x)`

- `x` = structure containing function argument.

The return value is the tangent of `x`, where `x` is given in radians. **xtan(x)** returns **xPinf** if `x` is equal to **xPi2** (up to an integer multiple of **xPi**), **xMinf** if `x` is equal to **-xPi2** (up to an integer multiple of **xPi**). In both cases a domain error is produced.

xcos - Cosine function.

`struct xpr xcos(struct xpr x)`

- `x` = structure containing function argument.

The return value is the cosine of `x`, where `x` is given in radians.

xsin - Sine function.

`struct xpr xsin(struct xpr x)`

- `x` = structure containing function argument.

The return value is the sine of `x`, where `x` is given in radians.

xatan - Arc tangent function.

`struct xpr xatan(struct xpr x)`

- `x` = structure containing function argument.

This function returns the arc tangent of **x** in radians and the value is mathematically defined to be between $-\pi/2$ and $\pi/2$ (inclusive).

xasin - Arc sine function.

struct xpr xasin(struct xpr x)

- **x** = structure containing function argument.

This function returns the arc sine of **x** in radians and the value is mathematically defined to be between $-\pi/2$ and $\pi/2$ (inclusive). If **x** falls outside the range -1 to 1, a domain error is produced.

xacos - Arc cosine function.

struct xpr xacos(struct xpr x)

- **x** = structure containing function argument.

This function returns the arc cosine of **x** in radians and the value is mathematically defined to be between zero and π (inclusive). If **x** falls outside the range -1 to 1, a domain error is produced.

xatan2 - arc tangent function of two variables.

struct xpr xatan2 (struct xpr y, struct xpr x)

- **y** = structure containing first argument;
- **x** = structure containing second argument.

This function returns the principal value of the arc tangent of **y/x**, using the signs of the two arguments to determine the quadrant of the result, a real number in the range $[-\pi, \pi)$ ($-\pi$ is included, π is excluded). If **x** and **y** are both zero, a domain error is produced and the dummy value zero is returned. If **x** is zero and **y** is positive, then $\pi/2$ (i.e. $\pi/2$) is returned. If **x** is zero and **y** is negative, the function returns $-\pi/2$ (i.e. $-\pi/2$).

xtanh - Hyperbolic tangent function.

struct xpr xtanh(struct xpr x)

- **x** = structure containing function argument.

The return value is the hyperbolic tangent of **x**.

xcosh - Hyperbolic cosine function.

struct xpr xcosh(struct xpr x)

- **x** = structure containing function argument.

The return value is the hyperbolic cosine of **x**.

xsinh - Hyperbolic sine function.

`struct xpr xsinh(struct xpr x)`

- **x** = structure containing function argument.

The return value is the hyperbolic sine of **x**.

xatanh - Hyperbolic arc tangent function.

`struct xpr xatanh(struct xpr x)`

- **x** = structure containing function argument.

This function returns the hyperbolic arc tangent of **x**. If the absolute value of **x** is greater than 1, then a domain error is produced.

xasinh - Hyperbolic arc sine function.

`struct xpr xasinh(struct xpr x)`

- **x** = structure containing function argument.

This function returns the hyperbolic arc sine of **x**.

xacosh - Hyperbolic arc cosine function.

`struct xpr xacosh(struct xpr x)`

- **x** = structure containing function argument.

This function returns the hyperbolic arc cosine of **x**. If **x** is less than 1, a domain error is produced.

7.4 Applications of Extended Precision Arithmetic

The Tchebycheff expansion supplied with the library can be used to compute the Tchebycheff expansion coefficients of a function to an accuracy of 32 digits at least. This ability is useful in developing high accuracy function approximations, since the effect of rounding error on coefficients used in double precision can effectively be eliminated with these inputs. The functions provided to this purpose are `xchcof()` and `xvtrch()`.

xchcof - Compute the Tchebycheff expansion coefficients of a specified function $f(x)$.

```
struct xpr* xchcof(int m, struct xpr (*xfunc)(struct xpr))
```

- m = index of the last coefficient (the computed coefficients will be $m+1$, indexed from 0 to m);
- $xfunc$ = pointer to user defined function returning extended precision values of the function $f()$;

The return value is the array of the computed coefficients. The equality

$$f(x) = c[0]/2 + \text{Sum}(k=1 \text{ to } m) c[k]*T_k(x)$$

holds, where T_k is the k th Tchebycheff polynomial.

Remarks: The memory needed by the returned array is malloc'ed inside the function **xchcof()**. To avoid memory leaks it should be explicitly freed through a call to **free()**. In case of insufficient memory **xchcof()** will return NULL.

If $m \leq XMAX_DEGREE$ ($= 50$), then the array returned by **xchcof()** will have exactly $m+1$ elements, indexed from 0 to m . If $m > XMAX_DEGREE$, then **xchcof()** will behave as if m were equal to $XMAX_DEGREE$, namely, the array returned by **xchcof()** will have only $XMAX_DEGREE + 1$ elements, indexed from 0 to $XMAX_DEGREE$. In other words, a value of m greater than $XMAX_DEGREE$ is ignored and replaced by $XMAX_DEGREE$. $XMAX_DEGREE$ is a macro declared inside the header file **xpre.h**.

xevtch - Evaluate an extended precision Tchebycheff expansion.

```
struct xpr xevtch(struct xpr x, struct xpr *a, int m)
```

- x = structure containing function argument;
- a = structure array containing expansion coefficients;
- m = maximum index of coefficient array (dimension= $m+1$).

The return value is the number given by the formula

$$f(x) = \text{Sum}(k=0 \text{ to } m) a[k]*T_k(x),$$

where T_k is the k th Tchebycheff polynomial.

8 Complex Arithmetic

The second module of the HPA library is formed by functions for Extended Precision Complex Arithmetic and functions of the Extended Precision Complex Math Library. They are all declared in the file **cxpre.h** together with some macros and numerical constants.

8.1 Complex constants

The header file `cxpre.h` defines the constants `cxZero` ($= 0$), `cxOne` ($= 1$) and `cxIU` ($=$ imaginary unit):

```
extern const struct cxpr cxZero;
extern const struct cxpr cxOne;
extern const struct cxpr cxIU;
```

which require no explanation.

8.2 Extended Precision Complex Arithmetic

The functions for complex arithmetic support the basic computations and input/output operations with extended precision complex values. Some of the functions supply capabilities designed to enhance the computational efficiency of the complex arithmetic (e.g., `cxpwr` and `cxpow`). What follows is the complete list of the functions with related synopses.

cxreset - Make a new complex number from its real and imaginary parts.

```
struct cxpr cxreset (struct xpr re, struct xpr im)
```

- `re` = structure containing real part;
- `im` = structure containing imaginary part.

The value returned by `cxreset()` is the complex number having `re` as its real part, `im` as its imaginary part.

Remark: `cxreset()` is also available in the form of a macro:

```
#define CXRESET(re, im) (struct cxpr){re, im}
```

cxconv - Convert a real number into a complex one.

```
struct cxpr cxconv (struct xpr x)
```

- `re` = structure containing real part.

The value returned by `cxconv()` is the complex number having `x` as its real part, zero as its imaginary part.

Remark: `cxconv()` is also available in the form of a macro:

```
#define CXCONV(x) (struct cxpr){x, xZero}
```

cxre - Obtain the real part of a complex number.

```
struct xpr cxre (struct cxpr z)
```

- **z** = structure containing function argument.

The value returned by **cxre(z)** is the real part of **z**.

Remark: **cxre()** is also available in the form of a macro:

```
#define CXRE(z) (z).re
```

cxim - Obtain the imaginary part of a complex number.

```
struct xpr cxim (struct cxpr z)
```

- **z** = structure containing function argument.

The value returned by **cxim(z)** is the imaginary part of **z**.

Remark: **cxim()** is also available in the form of a macro:

```
#define CXIM(z) (z).im
```

cxswap - Swap the real and the imaginary part of a complex number.

```
struct cxpr cxswap (struct cxpr z)
```

- **z** = structure containing function argument.

The value returned by **cxswap(z)** is the complex number $\{\text{cxim}(z), \text{cxre}(z)\}$.

Remark: **cxswap()** is also available in the form of a macro:

```
#define CXSWAP(z) (struct cxpr){(z).im, (z).re}
```

cxconj - Calculate the complex conjugate of a complex number.

```
struct cxpr cxconj (struct cxpr z)
```

- **z** = structure containing function argument.

The **cxconj()** function returns the complex conjugate value of its argument, i.e. the value obtained by changing the sign of the imaginary part.

cxneg - Change sign (unary minus)

```
struct cxpr cxneg (struct cxpr z)
```


- **z** = structure containing function argument.

cxneg(z) returns the complex number **-z**: if **z** = **a+ib**, then **cxneg(z)** returns **-a-ib**.

cxinv - Obtain the reciprocal of a complex number.

struct cxpr cxinv (struct cxpr z)

- **z** = structure containing function argument.

The value returned by **cxinv()** is the reciprocal of its argument. If **z** is zero, then a division-by-zero error is produced.

cxabs - Calculate the absolute value of a complex number.

struct xpr cxabs (struct cxpr z)

- **z** = structure containing function argument.

cxabs(z) returns the absolute value of the complex number **z**. The result is a real number.

cxarg - Calculate the argument of a complex number.

struct xpr cxarg (struct cxpr z)

- **z** = structure containing function argument.

cxarg(z) returns the argument or phase angle of the complex number **z**. The result is a real number in the range **[-xPi,xPi]** (**-xPi** is included, **xPi** is excluded). **cxarg(z)** is equivalent to **xatan2(z.im, z.re)**.

cxadd - Add/subtract two extended precision complex numbers.

struct cxpr cxadd (struct cxpr z1, struct cxpr z2, int k)

- **z1** = structure containing first number;
- **z2** = structure containing second number;
- **k** = control flag: if 0, then **z1** and **z2** are added, else they are subtracted (**z1-z2**).

The value returned by **cxadd()** is the result of the addition/subtraction.

cxsum - Add two extended precision complex numbers.

struct cxpr cxsum (struct cxpr z1, struct cxpr z2)

- **z1** = structure containing first number;
- **z2** = structure containing second number.

The value returned by **cxsum()** is the result of the addition **z1 + z2**.

cxsub - Subtract two extended precision complex numbers.

```
struct cxpr cxsub (struct cxpr z1, struct cxpr z2)
```

- **z1** = structure containing first number;
- **z2** = structure containing second number.

The value returned by **cxsub()** is the result of the subtraction **z1 - z2**.

cxmul - Multiply two extended precision complex numbers.

```
struct cxpr cxmul (struct cxpr z1, struct cxpr z2)
```

- **z1** = structure containing first number;
- **z2** = structure containing second number.

The value returned by **cxmul()** is the product **z1 * z2**.

cxrmul - Multiply a complex number by a real one.

```
struct cxpr cxrmul (struct xpr c, struct cxpr z)
```

- **c** = structure containing a real number;
- **z** = structure containing a complex number.

The value returned by **cxrmul()** is the product **c * z**.

cxdrot - Multiply a complex number by the imaginary unit.

```
struct cxpr cxdrot (struct cxpr z)
```

- **z** = structure containing function argument.

The value returned by **cxdrot(z)** is the product of **z** times the imaginary unit.

cxrrot - Multiply a complex number by **-1i**, where **1i** is the imaginary unit.

```
struct cxpr cxrrot (struct cxpr z)
```

- **z** = structure containing function argument.

The value returned by `cxrrot(z)` is the product of `z` times the negative imaginary unit.

cxdiv - Divide two extended precision complex numbers.

`struct cxpr cxdiv (struct cxpr z1, struct cxpr z2)`

- `z1` = structure containing first number;
- `z2` = structure containing second number.

The value returned by `cxdiv(z1, z2)` is the quotient `z1 / z2`. If `z2` is zero, then a division-by-zero error is produced.

cxgdiv - Gaussian division between complex numbers having both real and imaginary part integer.

`struct cxpr cxgdiv (struct cxpr z1, struct cxpr z2)`

- `z1` = structure containing first number;
- `z2` = structure containing second number.

After eventually rounding `z1` and `z2` by means of `cxround()` (see below), `cxgdiv(z1, z2)` returns the quotient of the gaussian division of `z1` by `z2`. If `z2` is zero, then a division-by-zero error is produced.

If you do not know what gaussian division means, probably you will never need this function :)

cxgmod - Remainder of the Gaussian division.

`struct cxpr cxgmod (struct cxpr z1, struct cxpr z2)`

- `z1` = structure containing first number;
- `z2` = structure containing second number.

After eventually rounding `z1` and `z2` by means of `cxround()` (see below), `cxgmod(z1, z2)` returns the remainder of the gaussian division of `z1` by `z2`. If `z2` is zero, then a division-by-zero error is produced.

If you do not know what gaussian division means, probably you will never need this function :)

cxidiv - Integer division.

`struct cxpr cxidiv (struct cxpr z1, struct cxpr z2)`

- `z1` = structure containing first number;
- `z2` = structure containing second number.

After eventually rounding **z1** and **z2** by means of **cxround()** (see below), **cxidiv(z1, z2)** returns the quotient of the integer division of **z1** by **z2**. If **z2** is zero, then a division-by-zero error is produced. **cxidiv()** is a *smooth* extension of the integer division between real numbers.

cxmod - Remainder of the integer division.

```
struct cxpr cxmod (struct cxpr z1, struct cxpr z2)
```

- **z1** = structure containing first number;
- **z2** = structure containing second number.

After eventually rounding **z1** and **z2** by means of **cxround()** (see below), **cxmod(z1, z2)** returns the remainder of the integer division of **z1** by **z2**. If **z2** is zero, then a division-by-zero error is produced.

cxpwr - Raise to integer powers.

```
struct cxpr cxpwr (struct cxpr z, int n)
```

- **z** = structure containing input number;
- **n** = exponent.

The return value is the **nth** power of the first argument.

cxpow - Power function.

```
struct cxpr cxpow (struct cxpr z1, struct cxpr z2)
```

- **z1** = base;
- **z2** = exponent.

The return value is the power of the first argument raised to the second one. Note that the modulus of the first argument must be greater than zero, if the real part of **z2** is less or equal than zero, otherwise a bad-exponent error is produced.

cxsq - Square of a number.

```
struct cxpr cxsq (struct cxpr z)
```

- **z** = structure containing function argument.

This function returns the square of its argument.

cxroot - **nth** root of a complex number.

`struct cxpr cxroot (struct cxpr z, int i, int n)`

`cxroot(z,i,n)` returns the `i`th branch of the `n`th root of `z`. If `n` is zero or negative and the modulus of `z` is zero, then a bad-exponent error is produced.

cxsqrt - Principal branch of the square root of a complex number.

`struct cxpr cxsqrt (struct cxpr z)`

- `z` = structure containing function argument.

This function returns the principal branch of the square root of its argument.

cxprcmp - Compare two extended precision complex numbers.

`struct cxprcmp_res cxprcmp (const struct cxpr* z1,
 const struct cxpr* z2)`

- `z1` = pointer to first number;
- `z2` = pointer to second number.

The value returned by `cxprcmp()` is a structure formed by two comparison flags:

```
struct cxprcmp_res
{
    int re, im;
};
```

If the `.re` field of the returned structure is:

- `+1`, then `z1->re` is greater than `z2->re`,
- `0`, then `z1->re` is equal to `z2->re`,
- `-1`, then `z1->re` is less than `z2->re`.

The meaning of the `.im` field is the same but refers to `z1->im` and `z2->im`.
Note that the input numbers are not altered by `cxprcmp()`.

cxis0 - Compare a complex number with zero.

`int cxis0 (const struct cxpr* z)`

- `z` = pointer to an extended precision complex number.

The return value is 0 if `*z` is not zero, else a non-zero value.

cxnot0 - Compare a complex number with zero.

`int cxnot0 (const struct cxpr* z)`

- `z` = pointer to an extended precision complex number.

The return value is 0 if `*z` is zero, else a non-zero value.

cxeq - Check if two complex numbers are or are not equal.

`int cxeq (struct cxpr z1, struct cxpr z2)`

- `z1` = first number;
- `z2` = second number.

The return value is 0 if `z1` and `z2` are different, else a non-null value.

cxneq - Check if two complex numbers are or are not equal.

`int cxneq (struct cxpr z1, struct cxpr z2)`

- `z1` = first number;
- `z2` = second number.

The return value is 0 if `z1` and `z2` are equal, else a non-null value.

cxgt - Check if a complex number is greater than another one.

`int cxgt (struct cxpr z1, struct cxpr z2)`

- `z1` = first number;
- `z2` = second number.

The return value is 0 if `z1` is not greater than `z2`, else a non-null value.

cxge - Check if a complex number is greater or equal to another one.

`int cxge (struct cxpr z1, struct cxpr z2)`

- `z1` = first number;
- `z2` = second number.

The return value is 0 if `z1` is not greater or equal to `z2`, else a non-null value.

cxlt - Check if a complex number is less than another one.

`int cxlt (struct cxpr z1, struct cxpr z2)`

- `z1` = first number;

- **z2** = second number.

The return value is 0 if **z1** is not less than **z2**, else a non-null value.

cxle - Check if a complex number is less or equal to another one.

```
int cxle (struct cxpr z1, struct cxpr z2)
```

- **z1** = first number;
- **z2** = second number.

The return value is 0 if **z1** is not less or equal to **z2**, else a non-null value.

dctocx - Convert a double precision complex number to an extended precision number.

```
struct cxpr dctocx (double re, double im)
```

- **re** = real part of the double precision complex number;
- **im** = imaginary part of the double precision complex number.

The value returned by **dctocx(re,im)** is the extended precision equivalent of the complex number (**re**, **im**).

xtodc - Convert an extended precision complex number to a double precision complex number.

```
void xtodc (const struct cxpr *z, double *re, double *im)
```

- **z** = pointer to an extended precision complex number;
- **re** = pointer to a double precision number;
- **im** = pointer to a double precision number.

xtodc() stores in its second and last argument respectively the real and the imaginary part of the number pointed to by its first argument.

ftocx - Convert a single precision complex number to an extended precision number.

```
struct cxpr ftocx (float re, float im)
```

- **re** = real part of the single precision complex number;
- **im** = imaginary part of the single precision complex number.

The value returned by `ftocx(re,im)` is the extended precision equivalent of the complex number `(re, im)`.

cxtofc - Convert an extended precision complex number to a single precision complex number.

```
void cxtofc (const struct cxpr *z, float *re, float *im)
```

`z` = pointer to an extended precision complex number;

- `re` = pointer to a single precision number;
- `im` = pointer to a single precision number.

`cxtofc()` stores in its second and last argument respectively the real and the imaginary part of the number pointed to by its first argument.

ictocx - Convert an integer complex number to an extended precision number.

```
struct cxpr ictocx (long re, long im)
```

- `re` = real part of the integer complex number;
- `im` = imaginary part of the integer complex number.

The value returned by `ictocx(re,im)` is the extended precision equivalent of the complex number `(re, im)`.

uctocx - Convert an integer complex number to an extended precision number.

```
struct cxpr uctocx (unsigned long re, unsigned long im)
```

- `re` = real part of the integer complex number;
- `im` = imaginary part of the integer complex number.

The value returned by `uctocx(re,im)` is the extended precision equivalent of the complex number `(re, im)`.

strtocx - Convert a floating point complex number, expressed as a decimal ASCII string in a form consistent with C, into the extended precision format.

```
struct cxpr strtocx (const char *s, char **endptr)
```

- `s` = pointer to a null terminated ASCII string expressing a complex number;
- `endptr` = NULL or address of a pointer defined outside `strtocx()`.

The value returned by `strtocx()` is the input number in extended precision format.

Remarks: The `strtocx()` function converts the initial portion of the string pointed to by `s` to its extended precision representation.

The expected form of the (initial portion of the) string is optional leading white space as recognized by the standard library function `isspace()`, an optional plus (+) or minus sign (-) and then a decimal number. A decimal number consists of a nonempty sequence of decimal digits possibly containing a radix character (decimal point, i.e. '.'), optionally followed by a decimal exponent. A decimal exponent consists of an `E` or `e`, followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits, and indicates multiplication by a power of 10.

After this decimal number there can be an `i` character or, alternatively, some optional white spaces, an optional plus (+) or minus sign (-) and then another decimal number followed by an `i` character. Examples of valid representations of complex numbers are:

"12", "34.56", ".7895i", "-34.56-7.23i", "-45.7 +23.4i".

This function returns the converted value, if any. If the correct value for the real or/and the imaginary part would cause overflow, then `xPinf` or `xMinf` is returned in the corresponding field, according to the sign of the value. If the correct value would cause underflow, `xZero` is returned. If no conversion is performed, `xNaN` is returned.

If `endptr` is not `NULL`, a pointer to the character after the last character used in the conversion is stored in the location referenced by `endptr`. If no conversion is performed, the value of `s` is stored in the location referenced by `endptr`.

atocx - Convert a floating point complex number, expressed as a decimal ASCII string in a form consistent with C, into the extended precision format.

```
struct cxpr atocx (const char *s)
```

- `s` = pointer to a null terminated ASCII string expressing a complex number.

The return value is the input number converted to the extended precision format.

Remark: The call `atocx(s)` is equivalent to `strtocx(s, NULL)`.

cxpr_asprint - Convert an extended precision complex number to a string.

```
char *cxpr_asprint (struct cxpr z, int sc_not, int sign,
                   int lim)
```

- `z` = structure containing number to print;
- `sc_not` = zero to mean floating point notation, not zero to mean scientific notation;

- **sign** = not zero to put a plus sign (+) before the real part of the number if it is non-negative (in case of negative real part a minus sign (-) is printed even if the parameter **sign** is zero);
- **lim** = number of decimal digits to the right of the decimal point (**lim+1** = total digits displayed) in case of scientific notation, else number of significant digits - 1 (**lim+1** = total of significant digits).

cxpr_asprint() returns the string with the converted number. The memory for this string is calloc'ed inside the function. **cxpr_asprint()** uses always the format "**a+bi**" in the conversion.

cxtoa - This function converts an extended precision complex number to a string. Scientific notation is always used for both real and imaginary part.

```
char *cxtoa (struct cxpr z, int lim)
```

- **z** = structure containing number to print;
- **lim** = number of decimal digits to the right of the decimal point (**lim+1** = total digits displayed).

Remark:

```
cxtoa(z, lim)
```

is equivalent to

```
cxpr_asprint(z, 1, 0, lim)
```

cxfrac - Fractional part of both real and imaginary part.

```
struct cxpr cxfrac (struct cxpr z)
```

- **z** = structure containing function argument.
- cxfrac(z)** returns {**xfrac(z.re)**, **xfrac(z.im)**}.

cxtrunc - Integer part of both real and imaginary part.

```
struct cxpr cxtrunc (struct cxpr z)
```

- **z** = structure containing function argument.
- cxtrunc(z)** returns {**xtrunc(z.re)**, **xtrunc(z.im)**}.

cxfix - Integer part of both real and imaginary part (2nd method).

```
struct cxpr cxfix (struct cxpr z)
```

- `z` = structure containing function argument.

`cxfix(z)` returns `{xfix(z.re), xfix(z.im)}`.

cxround - Rounding real and imaginary part to the nearest integer values (halfway cases are rounded away from zero).

`struct cxpr cxround (struct cxpr z)`

- `z` = structure containing function argument.

`cxround(z)` returns `{xround(z.re), xround(z.im)}`.

cxfloor - Rounding real and imaginary part to the largest integral values not greater than them.

`struct cxpr cxfloor (struct cxpr z)`

- `z` = structure containing function argument.

`cxfloor(z)` returns `{xfloor(z.re), xfloor(z.im)}`.

cxceil - Rounding real and imaginary part to the smallest integral values not less than them.

`struct cxpr cxceil (struct cxpr z)`

- `z` = structure containing function argument.

`cxceil(z)` returns `{xceil(z.re), xceil(z.im)}`.

cxpr_print - Print an extended precision complex number in scientific or floating point notation to a given file.

`void cxpr_print (FILE * stream, struct cxpr z, int sc_not,
int sign, int lim)`

- `stream` = file where the number must be printed;
- `z` = structure containing number to print;
- `sc_not` = zero to mean floating point notation, not zero to mean scientific notation;
- `sign` = not zero to put a plus sign (+) before the real part of the number if it is non-negative (in case of negative real part a minus sign (-) is printed even if the parameter `sign` is zero);
- `lim` = number of decimal digits to the right of the decimal point (`lim+1` = total digits displayed) in case of scientific notation, else number of significant digits - 1 (`lim+1` = total of significant digits).

cxprcxpr - Print an extended precision complex number in scientific notation.

```
void cxprcxpr (struct cxpr z, int m)
```

- **z** = structure containing number to print;
- **lim** = number of decimal digits to the right of the decimal point (**lim+1** = total digits displayed).

Remark:

```
cxprcxpr(z, lim)
```

is equivalent to

```
cxpr_print(stdout, z, 1, 0, lim)
```

cxprint - Print an extended precision complex number as a couple of strings of hexadecimal numbers.

```
void cxprint (FILE * stream, struct cxpr z)
```

- **stream** = file where the number must be printed;
- **z** = structure containing number to print.

The **cxprint()** function supports a bit oriented analysis of rounding error effects. It always prints a newline (**\n**) at the end.

cxfout - Print an extended precision complex number to a file according to a given set of I/O flags.

```
int cxfout (FILE * stream, struct xoutflags ofs, struct cxpr z)
```

- **stream** = file where the number must be printed;
- **ofs** = structure containing all I/O flags;
- **z** = structure containing number to print.

The return value is 0 in case of success, -1 to mean a failure.

Remark: For the definition of **struct xoutflags** and the meaning of its fields see [section "Real Arithmetic"](#). **cxfout()** does not add any newline at the end of the printed number.

cxout - Print an extended precision complex number on **stdout** according to a given set of I/O flags.

```
int cxout (struct xoutflags ofs, struct cxpr z)
```

- **ofs** = structure containing all I/O flags;
- **z** = structure containing number to print.

The return value is 0 in case of success, -1 to mean a failure.

Remark: For the definition of **struct xoutflags** and the meaning of its fields see [section "Real Arithmetic"](#). **cxout()** does not add any newline at the end of the printed number.

cxcout - Write an extended precision complex number on a string according to a given set of I/O flags.

```
unsigned long cxcout (char *s, unsigned long n,  
                     struct xoutflags ofs, struct cxpr z)
```

- **s** = pointer to a buffer of characters (char);
- **n** = size of the buffer;
- **ofs** = structure containing all I/O flags;
- **z** = structure containing number to print.

The return value is the number of the non-null characters written to the buffer or, if it is greater or equal than **n**, which would have been written to the buffer if enough space had been available.

Remark: For the definition of **struct xoutflags** and the meaning of its fields see [section "Real Arithmetic"](#). **cxcout()** always adds a null character ('\0') at the end of the written number. **cxcout()** does not write more than **n** bytes (including the trailing '\0'). Thus, a return value of **n** or more means that the output was truncated. In this case, the contents of the buffer pointed to by the first argument of **cxcout()** are completely unreliable.

8.3 Extended Precision Complex Math Library

The Extended Precision Complex Math Library provides the elementary functions normally supported in a complex math library. They are designed to provide full precision accuracy.

cxexp - Compute the complex exponential function.

```
struct cxpr cxexp (struct cxpr z)
```

- **z** = structure containing function argument.

The return value is e (the base of natural logarithms) raised to z .

cxexp2 - Compute the base-2 complex exponential function.

```
struct cxpr cxexp2 (struct cxpr z)
```

- z = structure containing function argument.

The return value is 2 raised to z .

cxexp10 - Compute the base-10 complex exponential function.

```
struct cxpr cxexp10 (struct cxpr z)
```

- z = structure containing function argument.

The return value is 10 raised to z .

cxlog - Compute natural (base e) logarithm of a complex number.

```
struct cxpr cxlog (struct cxpr z)
```

- z = structure containing function argument.

This function returns the natural logarithm of its argument. A null argument results in a domain error. The imaginary part of the result lies in the interval $[-x\pi, x\pi)$ ($-x\pi$ is included, $x\pi$ is excluded).

cxlog2 - Compute base-2 logarithm of a complex number.

```
struct cxpr cxlog2 (struct cxpr z)
```

- z = structure containing function argument.

This function returns the base-2 logarithm of its argument. A null argument results in a domain error.

cxlog10 - Compute base-10 logarithm of a complex number.

```
struct cxpr cxlog10 (struct cxpr z)
```

- z = structure containing function argument.

This function returns the base-10 logarithm of its argument. A null argument results in a domain error.

cxlog_sqrt - Compute natural (base e) logarithm of the principal branch of the square root of a complex number.

`struct cxpr cxlog_sqrt (struct cxpr z)`

- **z** = structure containing function argument.

This function returns the natural logarithm of the principal branch of the square root of its argument. A null argument results in a domain error. The imaginary part of the result lies in the range $[-x\text{Pi}2, x\text{Pi}2)$ ($-x\text{Pi}2$ is included, $x\text{Pi}2$ is excluded).

cxtan - Complex tangent function.

`struct cxpr cxtan (struct cxpr z)`

- **z** = structure containing function argument.

The return value is the tangent of the complex number **z**. **cxtan(z)** yields a domain error if the imaginary part of **z** is null and the real part is equal, up to an integer multiple of $x\text{Pi}$, to $x\text{Pi}2$.

cxcos - Complex cosine function.

`struct cxpr cxcos (struct cxpr z)`

- **z** = structure containing function argument.

The return value is the cosine of the complex number **z**.

cxsin - Complex sine function.

`struct cxpr cxsin (struct cxpr z)`

- **z** = structure containing function argument.

The return value is the sine of the complex number **z**.

cxatan - Complex arc tangent function.

`struct cxpr cxatan (struct cxpr z)`

- **z** = structure containing function argument.

This function returns the arc tangent of the complex number **z**, i.e. a number **w** such that **z** = **tan(w)**. If the real part of **z** is null and the imaginary part is equal to $+1$ or -1 , then a domain error is produced.

cxasin - Complex arc sine function.

`struct cxpr cxasin (struct cxpr z)`

- **z** = structure containing function argument.

This function returns the arc sine of the complex number **z**, i.e. a number **w** such that **z** = **sin(w)**.

cxacos - Complex arc cosine function.

struct cxpr cxacos (struct cxpr z)

- **z** = structure containing function argument.

This function returns the arc cosine of the complex number **z**, i.e. a number **w** such that **z** = **cos(w)**.

cxtanh - Complex hyperbolic tangent function.

struct cxpr cxtanh (struct cxpr z)

- **z** = structure containing function argument.

The return value is the hyperbolic tangent of the complex number **z**. **cxtanh(z)** yields a domain error if the real part of **z** is null and the imaginary part is equal, up to an integer multiple of **xPi**, to **xPi2**.

xcosh - Complex hyperbolic cosine function.

struct cxpr xcosh (struct cxpr z)

- **z** = structure containing function argument.

The return value is the hyperbolic cosine of the complex number **z**.

xsinh - Complex hyperbolic sine function.

struct cxpr xsinh (struct cxpr z)

- **z** = structure containing function argument.

The return value is the hyperbolic sine of the complex number **z**.

xatanh - Complex hyperbolic arc tangent function.

struct cxpr xatanh (struct cxpr z)

- **z** = structure containing function argument.

This function returns the hyperbolic arc tangent of the complex number **z**, i.e. a number **w** such that **z** = **tanh(w)**. If the imaginary part of **z** is null and the real part is equal to **+1** or **-1**, then a domain error is produced.

xasinh - Complex hyperbolic arc sine function.


```
struct cxpr cxasinh (struct cxpr z)
```

- **z** = structure containing function argument.

This function returns the hyperbolic arc sine of the complex number **z**, i.e. a number **w** such that **z** = **sinh(w)**.

cxacosh - Complex hyperbolic arc cosine function.

```
struct cxpr cxacosh (struct cxpr z)
```

- **z** = structure containing function argument.

This function returns the hyperbolic arc cosine of the complex number **z**, i.e. a number **w** such that **z** = **cosh(w)**.

FINAL REMARK: The header file **cxpre.h** also defines the macros **cxconvert**, **cxdiff**, **cxprod** and **cxipow**:

```
#define cxconvert cxconv
#define cxdiff cxsub
#define cxprod cxmul
#define cxipow cxpwr
```

This allows to use **cxconvert** as synonym of **cxconv**, **cxdiff** as synonym of **cxsub**, **cxprod** in place of **cxmul**, and **cxipow** for **cxpwr**.

9 The C++ interface

The HPA library supplies a C++ wrapper allowing to perform high precision computations with the same syntax of the normal code. Using the C++ wrapper permits you to write things like:

```
// Compute the factorial of the integer n
```

```
xreal factorial(xreal n)
{
    xreal i;
    xreal product = 1;
    for (i=2; i <= n; i++)
        product *= i;
    return product;
}
```

Technically, the C++ wrapper is contained in a separate library. However, this module is distributed together with the HPA library as an extension to it. This section of the manual describes the C++ interface of this extension. This one is formed by two classes, called `xreal` and `xcomplex` respectively. The type `xreal` can be used to declare or define real variables. In addition, it defines the mathematical operations and functions which can be used with them. The same is for the type `xcomplex` with respect to complex variables and their manipulation. Once you have variables of `xreal` and `xcomplex` type, you can use the usual C++ syntax, but all computations will be performed with the high precision math library. Of course, this possibility greatly simplifies the writing of code. Note the difference between

```
struct xpr x, y;
char buffer[256];

while ( (fgets (buffer, 256, stdin)) )
{
    x = atox (buffer);
    printf ("The square root of 2 * %s + 1 is ", buffer);
    y = xadd (xpr2(x, 1), xOne, 0);
    xprxpr (xsqrt (y), 30);
    putchar ('\n');
}
```

and its C++ version:

```
xreal x;

while ( x.getfrom(cin) > 0 )
    cout << "The square root of 2 * " << x << " + 1 is "
         << sqrt(2 * x + 1) << endl;
```

which is much more compact and easier to understand.

The use of the C++ wrapper requires however a recent and ANSI-compliant C++ compiler (for instance g++ 3.x). Moreover, before declaring or defining variables of `xreal` type and before using anyone of the functions or operators declared in the header file `xreal.h`, you have to insert the line

```
#include <xreal.h>
```

in your source code file.

Similarly, before declaring or defining variables of `xcomplex` type and before using anyone of the functions or operators declared in the header file `xcomplex.h`, you have to add the line

```
#include <xcomplex.h>
```

to your source code file.

After that, it is recommendable to add the directive

```
using namespace HPA;
```

since all objects of the C++ wrapper are declared or defined within the namespace `HPA`. Alternatively, you should always use the prefix `HPA::` in front of any identifiers coming from the files `xreal.h` or `xcomplex.h`. One has to tell the compiler indeed, where it has to look for the classes `xreal`, `xcomplex` and the related stuff.

This code comes from a real program:

```
#include<iostream>
#include<xreal.h>
using namespace HPA;

int main (void)
{
    xreal x;

    while ( x.getfrom(cin) > 0 )
        cout << "The square root of 2 * " << x << " + 1 is "
              << sqrt(2 * x + 1) << endl;
    return 0;
}
```

The header files `xreal.h` and `xcomplex.h` make also directly available the standard classes `ostream`, `istream` and `string` (see [sections "The xreal class"](#) and ["The xcomplex class"](#)).

10 Compiling and linking with the C++ wrapper

Whenever you have to compile and build a program making use of the C++ wrapper for the HPA library, you can do it by following the same instructions given in the [section "Compiling and linking"](#), just take care to use `hpaxxconf` in place of `hpaconf`, as in

```
c++ -c $(hpaxxconf -c) example.cc
```

```
c++ -c 'hpaxxconf -c' example.cc
```

to compile the file `example.cc` and obtain the object file `example.o`, or in

```
c++ example.o $(hpaxxconf -l) -o example
```

```
c++ example.o 'hpaxxconf -l' -o example
```

to do the linkage. If you want, you may also compile and build at the same time by using

```
c++ example.cc $(hpaxxconf -c -l) -o example
```

```
c++ example.cc 'hpaxxconf -c -l' -o example
```

All these examples assume that you are working with bash or with another shell sh-compatible. In any case, the synopsis of `hpaxxconf` is the same of `hpaconf`.

Warning: The command `hpaxxconf` is available only if the C++ wrapper was also built when the HPA library was installed on the system where you are working. It is possible to install the HPA library without its C++ wrapper. In this case, if you try to launch the command `hpaxxconf`, the shell will print the error message "Command not found" or something similar.

11 The xreal class

The interface of the `xreal` class is contained in the header file `xreal.h`, whose contents you can find here together with the necessary explanations.

```
#ifndef _XREAL_H_
#define _XREAL_H_

#include <xpre.h>
#include <iostream>
#include <string>
#include <cstdlib>

using std::ostream;
using std::istream;
using std::string;

namespace HPA {

class xreal {
    // << and >> are used respectively for the output and the
    // input of extended precision numbers.
    // The input operator >> reads a double precision
    // number and then converts it to an extended precision
    // number. This can have undesirable rounding effects.
    // To avoid them, use the input function
    // xreal::getfrom() (see below).
    friend ostream& operator<< (ostream& os, const xreal& x);
```

```

friend istream& operator>> (istream& is, xreal& x);

// +, -, *, / are the usual arithmetic operators
friend xreal operator+ (const xreal& x1, const xreal& x2);
friend xreal operator- (const xreal& x1, const xreal& x2);
friend xreal operator* (const xreal& x1, const xreal& x2);
friend xreal operator/ (const xreal& x1, const xreal& x2);

// x % n is equal to x * pow (2,n)
friend xreal operator% (const xreal& x1, int n);

// ==, !=, <=, >=, <, > are the usual comparison operators
friend int operator== (const xreal& x1, const xreal& x2);
friend int operator!= (const xreal& x1, const xreal& x2);
friend int operator<= (const xreal& x1, const xreal& x2);
friend int operator>= (const xreal& x1, const xreal& x2);
friend int operator< (const xreal& x1, const xreal& x2);
friend int operator> (const xreal& x1, const xreal& x2);

// sget (s, n, x) tries to read an extended precision
// number from the string 's' starting from the position
// 'n'. The retrieved number is converted and stored in
// 'x'. The return value is the number of characters
// composing the decimal representation of this number
// as read from 's'. For example, if s == "12.34dog" and
// n == 0, then 'x' is set to 12.34 and the return value
// is 5.
// If the portion of 's' starting from the position 'n'
// can not be converted to a number, then 'x' is set to
// xNAN and 0 is returned.
// If the exactly converted value would cause overflow,
// then xINF or x_INF is returned, according to the sign
// of the value.
// If 'n' is greater or equal to the length of 's', then 0
// is returned and 'x' is set to xZERO.
friend unsigned long sget (string s, unsigned long startptr,
                           xreal& x);

// bget (buff, x) tries to read an extended precision
// number from the buffer pointed to by 'buff'.
// The retrieved number is converted and stored in 'x'.
// The return value is a pointer to the character after
// the last character used in the conversion.
// For example, if 'buff' is a pointer to the buffer
// "12.34dog", then 'x' is set to 12.34 and the return
// value is a pointer to "dog" (i.e., a pointer

```

```

// to the character 'd').
// If the initial portion of the string pointed to by 'buff'
// can not be converted to a number, then 'x' is set to xNAN
// and 'buff' is returned.
// If the exactly converted value would cause overflow,
// then xINF or x_INF is returned, according to the sign
// of the value.
// If 'buff' is NULL (0), then an error message is printed
// on 'cerr' (standard error device).
friend const char* bget (const char* buff, xreal& x);

// compare (x1, x2) returns
// +1 to mean x1 > x2
// 0 to mean x1 == x2
// -1 to mean x1 < x2
friend int compare (const xreal& x1, const xreal& x2);

// isNaN (x) returns 1 when x == xNAN, else 0
friend int isNaN (const xreal& x);

// The following functions do not need a particular comment:
// each of them is defined as the corresponding function
// of the standard math library, that is to say the function
// from <cmath> having the same name.
// However qfmod(), sfmod(), frac() and fix() do not have
// counterparts in the standard math library.
// With respect to fmod(), qfmod() requires one more
// argument, where the quotient of the division of
// the first argument by the second one is stored.
// sfmod (x,&n) stores in the integer variable
// 'n' the integer part of 'x' and, at the same time,
// returns the fractional part of 'x'.
// The usage of sfmod() is strongly discouraged.
// frac() returns the fractional part of its argument.
// Finally, fix() is a frontend to the xfix()
// function (see section "Extended Precision Floating
// Point Arithmetic").
friend xreal abs (const xreal& s);
friend xreal frexp (const xreal& s, int *p);
friend xreal qfmod (const xreal& s, const xreal& t, xreal& q);
friend xreal fmod (const xreal& s, const xreal& t);
friend xreal sfmod (const xreal& s, int *p);
friend xreal frac (const xreal& x);
friend xreal trunc (const xreal& x);
friend xreal round (const xreal& x);
friend xreal ceil (const xreal& x);

```

```

friend xreal floor (const xreal& x);
friend xreal fix (const xreal& x);
friend xreal tan (const xreal& x);
friend xreal sin (const xreal& x);
friend xreal cos (const xreal& x);
friend xreal atan (const xreal& a);
friend xreal atan2 (const xreal& y, const xreal& x);
friend xreal asin (const xreal& a);
friend xreal acos (const xreal& a);
friend xreal sqrt (const xreal& u);
friend xreal exp (const xreal& u);
friend xreal exp2 (const xreal& u);
friend xreal exp10 (const xreal& u);
friend xreal log (const xreal& u);
friend xreal log2 (const xreal& u);
friend xreal log10 (const xreal& u);
friend xreal tanh (const xreal& v);
friend xreal sinh (const xreal& v);
friend xreal cosh (const xreal& v);
friend xreal atanh (const xreal& v);
friend xreal asinh (const xreal& v);
friend xreal acosh (const xreal& v);
friend xreal pow (const xreal& x, const xreal& y);

```

public:

```

// Various constructors. They allow to define
// an extended precision number in several ways.
// In addition, they allow for conversions from other
// numeric types.
xreal (const struct xpr* px = &xZero);
xreal (struct xpr x);
xreal (double x);
xreal (float x);
xreal (int n);
xreal (long n);
xreal (unsigned int u);
xreal (unsigned long u);

// This constructor requires a special comment. If
// only the first argument is present, the initial portion
// of the string pointed to by this argument is converted
// into an extended precision number, if a conversion is
// possible. If no conversion is possible, then the
// returned number is xNAN. If the second argument is
// present and is not null, it must be the address of a
// valid pointer to 'char'.

```

```

// Before returning, the constructor will set this pointer
// so that it points to the character of the string 'str' after the last
// character used in the conversion.
xreal (const char* str, char** endptr = 0);
xreal (string str);
xreal (const xreal& x);

// Assignment operators. They do not require
// any explanation with the only exception of '%=',
// which combines a '%' operation with an assignment.
// So, x %= n is equivalent to x *= pow(2,n) .
xreal& operator= (const xreal& x);
xreal& operator+= (const xreal& x);
xreal& operator-= (const xreal& x);
xreal& operator*= (const xreal& x);
xreal& operator/= (const xreal& x);
xreal& operator%= (int n);

// Increment and decrement operators. Both prefixed
// and postfix versions are defined.
xreal& operator++ ();
xreal& operator-- ();
xreal& operator++ (int dummy);
xreal& operator-- (int dummy);

// Destructor. You will never have to recall it
// explicitly in your code.
~xreal (void);

// Integer exponent power. For any extended precision
// number 'x', x(n) is equal to 'x' raised to 'n'.
xreal operator() (int n) const;

// This is the usual unary minus.
xreal operator-() const;

// For any extended precision number 'x', !x evaluates to 1
// when 'x' is null, else it evaluates to 0.
int operator!() const;

// x.isneg() returns 1 if 'x' is negative, else it
// returns 0.
int isneg() const;

// x.exp() returns the exponent part
// of the binary representation of 'x'.

```



```

int exp() const;

// Functions for conversions. x._2double(), x._2float(),
// x._2xpr() and x._2string() convert the extended precision
// number 'x' in a double precision number, in a single
// precision number, in a structure of
// type 'xpr', and in a string, respectively.
double _2double () const;
float _2float() const;
struct xpr _2xpr() const;
string _2string() const;

// The member function xreal::getfrom() can be used to
// recover an extended precision number from an input
// stream. The input stream is passed as argument to the
// function.
// The return value is 0 in case of input error (in case
// of End-Of-File, for example).
// When it starts to process its input, this function drops
// all the eventual leading white spaces.
// After reading the first non space character, it
// continues to read from the input stream until it finds
// a white space or reaches the End-Of-File.
// Then it tries to convert into an extended
// precision number the (initial portion of the) string just
// read.
// If no conversion can be performed, then x.getfrom(is)
// sets 'x' to the value xNAN.
// If the exactly converted value would cause overflow,
// then 'x' is set to xINF or x-INF, according to the sign
// of the correct value.
int getfrom (istream& is);

// The member function xreal::print() can be used to write
// an extended precision number to an output stream.
// The output stream is passed to the function as first
// argument. The next three arguments have the same meanings
// of the fields 'notat', 'sf' and 'lim' of the
// structure 'xoutflags', respectively (see section "Real Arithmetic").
int print (ostream& os, int sc_not, int sign, int lim) const;

// The function call x.asprint(sc_not, sign, lim) returns
// a buffer of characters with the representation, in form
// of a decimal ASCII string, of the extended precision
// number 'x'. The arguments 'sc_not', 'sign' and 'lim' are
// used to format the string.

```

```

// They have the same meanings of the fields 'notat', 'sf'
// and 'lim' of the structure 'xoutflags', respectively (see section
// "Real Arithmetic").
// The buffer returned by this function is malloc'ed inside
// the function. In case of insufficient memory, the null
// pointer is returned.
char* asprint (int sc_not, int sign, int lim) const;

// The following static functions are used to set
// or get the values of the fields of the structure
// 'xreal::ioflags'. This structure is a static member
// variable of the class 'xreal' and it is used by the
// output operator << to know how to format its second
// argument. The meaning of the fields of the structure
// 'xreal::ioflags' is explained in the section
// "Real arithmetic".

// xreal::set_notation (which) sets to 'which' the value
// of 'xreal::ioflags.notat' .
static void set_notation (short notat);

// xreal::set_signflag (which) sets to 'which' the value
// of 'xreal::ioflags.sf' .
static void set_signflag (short onoff);

// xreal::set_mfwd (which) sets to 'which' the value
// of 'xreal::ioflags.mfwd' .
static void set_mfwd (short wd);

// xreal::set_lim (which) sets to 'which' the value
// of 'xreal::ioflags.lim' .
static void set_lim (short lim);

// xreal::set_padding (which) sets to 'which' the value
// of 'xreal::ioflags.padding' .
static void set_padding (signed char ch);

// xreal::get_notation () returns the current value
// of 'xreal::ioflags.notat' .
static short get_notation (void);

// xreal::get_signflag () returns the current value
// of 'xreal::ioflags.sf' .
static short get_signflag (void);

// xreal::get_mfwd () returns the current value

```

```

// of 'xreal::ioflags.mfwd' .
static short get_mfwd (void);

// xreal::get_lim () returns the current value
// of 'xreal::ioflags.lim' .
static short get_lim (void);

// xreal::get_padding () returns the current value
// of 'xreal::ioflags.padding' .
static signed char get_padding (void);
private:
    struct xpr br; /* binary representation */
    static struct xoutflags ioflags; /* output flags */
};

// xmatherrcode() returns the current value of the global
// variable 'xErrNo' (see section
// "Dealing with runtime errors"), if
// this variable is defined.
// Otherwise xmatherrcode() returns -1.
int xmatherrcode ();

// clear_xmatherr() resets to 0 the value of the global
// variable 'xErrNo' (see section "Dealing with runtime
// errors"), if this variable is defined.
// Otherwise, clear_xmatherr() prints a suitable warning
// on 'cerr' (standard error device).
void clear_xmatherr ();

// Some useful constants:
// xZERO    == 0
// xONE     == 1
// xTWO     == 2
// xTEN     == 10
// xINF     == +INF
// x_INF    == -INF
// xNAN     == Not-A-Number
// xPI      == Pi Greek
// xPI2     == Pi / 2
// xPI4     == Pi / 4
// xEE      == e (base of natural logarithms)
// xSQRT2   == square root of 2
// xLN2     == natural logarithm of 2
// xLN10    == natural logarithm of 10
// xLOG2_E  == base-2 logarithm of e
// xLOG2_10 == base-2 logarithm of 10

```

```

// xLOG10_E == base-10 logarithm of e
extern const xreal xZERO, xONE, xTWO, xTEN;
extern const xreal xINF, x_INF, xNAN;
extern const xreal xPI, xPI2, xPI4, xEE, xSQRT2;
extern const xreal xLN2, xLN10, xLOG2_E, xLOG2_10, xLOG10_E;

} /* End namespace HPA */

#endif /* _XREAL_H_ */

```

12 The xcomplex class

The interface of the `xcomplex` class is contained in the header file `xcomplex.h`, whose contents you can find here together with the necessary explanations.

```

#ifndef _XCOMPLEX_H_
#define _XCOMPLEX_H_

#include <cxpre.h>
#include <iostream>
#include <string>
#include "xreal.h"

using std::istream;
using std::ostream;
using std::string;

namespace HPA {

    struct double_complex {
        double re, im;
    };

    struct float_complex {
        float re, im;
    };

    class xcomplex {
        // << and >> are used respectively for the output and the
        // input of extended precision complex numbers.
        // The input operator >> reads a couple of double
        // precision numbers and then converts it into
        // an extended precision complex number. This can have
        // undesirable rounding effects. To avoid them, use the

```

```

// input function xcomplex::getfrom() (see below).
friend ostream& operator<< (ostream& os, const xcomplex& z);
friend istream& operator>> (istream& is, xcomplex& z);

// +, -, *, / are the usual arithmetic operators
friend xcomplex
    operator+ (const xcomplex& z1, const xcomplex& z2);
friend xcomplex
    operator- (const xcomplex& z1, const xcomplex& z2);
friend xcomplex
    operator* (const xcomplex& z1, const xcomplex& z2);
friend xcomplex
    operator/ (const xcomplex& z1, const xcomplex& z2);

// z % n is equal to z * pow (2,n)
friend xcomplex
    operator% (const xcomplex& z, int n);

// ==, !=, <=, >=, <, > are the usual comparison operators
friend int
    operator== (const xcomplex& z1, const xcomplex& z2);
friend int
    operator!= (const xcomplex& z1, const xcomplex& z2);
friend int
    operator<= (const xcomplex& z1, const xcomplex& z2);
friend int
    operator>= (const xcomplex& z1, const xcomplex& z2);
friend int
    operator< (const xcomplex& z1, const xcomplex& z2);
friend int
    operator> (const xcomplex& z1, const xcomplex& z2);

// sget (s, n, z) tries to read an extended precision
// complex number from the string 's' starting from the
// position 'n'. The retrieved number is converted and
// stored in 'z'. The return value is the number of
// characters composing the decimal representation of this
// number as read from 's'.
// For example, if s == "12.34+6.7idog" and n == 0,
// then 'z' is set to 12.34+6.7i and the return value
// is 10.
// If the portion of 's' starting from the position 'n' can
// not be converted to a number, then 'z' is set to
// xNAN + xNANi and 0 is returned.
// If the exactly converted value would cause overflow in
// the real or/and imaginary part, then the real or/and the

```

```

// imaginary part of 'z' are set to xINF or x_INF, according
// to the signs of the correct value.
// If 'n' is greater or equal to the length of 's', then 0
// is returned and 'z' is set to cxZERO.
friend unsigned long sget (string s, unsigned long startptr,
                           xcomplex& z);

// bget (buff, z) tries to read an extended precision
// complex number from the buffer pointed to by 'buff'.
// The retrieved number is converted and stored in 'z'.
// The return value is a pointer to the character after
// the last character used in the conversion.
// For example, if 'buff' is a pointer to the buffer
// "12.34+6.7idog", then 'z' is set to 12.34+6.7i and
// the return value is a pointer to "dog" (i.e.,
// a pointer to the character 'd').
// If the initial portion of the string pointed to by 'buff'
// can not be converted to a number, then 'z' is set to
// xNAN + xNANi and 'buff' is returned.
// If the exactly converted value would cause overflow
// in the real or/and imaginary part, then the real or/and
// the imaginary part of 'z' are set to xINF or x_INF,
// according to the signs of the correct value.
// If 'buff' is NULL (0), then an error message is printed
// on 'cerr' (standard error device).
friend const char* bget (const char* buff, xcomplex& z);

// rmul (x,z) (here 'x' is a real number) returns the
// product x * z.
// It is faster than the * operator.
friend xcomplex rmul (const xreal& x, const xcomplex& z);

// After eventually rounding 'z1' and 'z2' by recalling
// round() (see below) on them, gdiv(z1, z2) returns
// the quotient of the gaussian division of 'z1' by 'z2'.
// If you do not know what gaussian division means, probably
// you will never need this function :)
friend xcomplex
    gdiv (const xcomplex& z1, const xcomplex& z2);

// After eventually rounding 'z1' and 'z2' by recalling
// round() (see below) on them, gmod(z1, z2) returns
// the remainder of the gaussian division of 'z1' by 'z2'.
// If you do not know what gaussian division means, probably
// you will never need this function :)
friend xcomplex

```

```

    gmod (const xcomplex& z1, const xcomplex& z2);

// idiv() is a wrapper to cxidiv() (see section
// "Extended Precision Complex Arithmetic").
friend xcomplex
    idiv (const xcomplex& z1, const xcomplex& z2);

// mod() is a wrapper to cxmod() (see section
// "Extended Precision Complex Arithmetic").
friend xcomplex
    mod (const xcomplex& z1, const xcomplex& z2);

// conj() returns the complex conjugate of its argument.
friend xcomplex conj (const xcomplex& z);

// inv() returns the complex reciprocal of its argument:
// inv(z) == 1/z .
friend xcomplex inv (const xcomplex& z);

// swap(z) returns the complex number {z.im, z.re}.
friend xcomplex swap (const xcomplex& z);

// Multiplication by 1i (imaginary unit).
friend xcomplex drot (const xcomplex& z);

// Multiplication by -1i
friend xcomplex rrot (const xcomplex& z);

// abs() returns the absolute value (or modulus) of its
// argument.
// The return value of abs() is then an 'xreal' number.
friend xreal abs (const xcomplex& z);

// arg(z) returns the phase angle (or argument)
// of the complex number 'z'.
// The return value of arg() is an 'xreal' number
// in the range [-xPI, xPI) (-xPI is included, xPI is excluded).
// If 'z' is null, then a domain-error is produced.
friend xreal arg (const xcomplex& z);

// The next six functions have the same
// meanings of the corresponding real functions,
// but they affect both the real
// and the imaginary part of their argument.
friend xcomplex frac (const xcomplex& z);
friend xcomplex trunc (const xcomplex& z);

```

```

friend xcomplex round (const xcomplex& z);
friend xcomplex ceil (const xcomplex& z);
friend xcomplex floor (const xcomplex& z);
friend xcomplex fix (const xcomplex& z);

// sqr() returns the square of its argument.
friend xcomplex sqr (const xcomplex& z);

// sqrt() returns the principal branch of the square root
// of its argument.
friend xcomplex sqrt (const xcomplex& z);

// root (z,i,n) returns the 'i'th branch of the 'n'th root
// of 'z'. If 'n' is zero or negative and 'z' is
// zero, then a bad-exponent error is produced.
friend xcomplex root (const xcomplex& z, int i, int n);

// These functions do not require any comment, except that
// tan() and tanh() yield a domain-error in the same cases
// as cxtan() and cxtanh(), respectively.
friend xcomplex exp (const xcomplex& z);
friend xcomplex exp2 (const xcomplex& z);
friend xcomplex exp10 (const xcomplex& z);
friend xcomplex tan (const xcomplex& z);
friend xcomplex sin (const xcomplex& z);
friend xcomplex cos (const xcomplex& z);
friend xcomplex tanh (const xcomplex& z);
friend xcomplex sinh (const xcomplex& z);
friend xcomplex cosh (const xcomplex& z);

// Natural, base-2 and base-10 logarithm of a complex
// number.
// A null argument results in a domain-error.
// The imaginary part of the return value of log() is always
// in the interval [-xPI,xPI) (-xPI is included, xPI is excluded).
friend xcomplex log (const xcomplex& z);
friend xcomplex log2 (const xcomplex& z);
friend xcomplex log10 (const xcomplex& z);

// log_sqrt(z) returns the natural logarithm of the
// principal branch of the square root of 'z'.
// A null argument results in a domain-error.
// The imaginary part of the return value of log_sqrt()
// is always in the interval [-xPI2,xPI2) (-xPI2 is included, xPI2 is excluded).
friend xcomplex log_sqrt (const xcomplex& z);

```



```

// These functions are self-explanatory. atan(z)
// yields a domain-error if the real part of 'z' is null and
// the imaginary part is equal to '+1' or '-1'.
// Similarly, atanh(z) yields a domain-error if the
// imaginary part of 'z' is null and the
// real part is equal to '+1' or '-1'.
friend xcomplex atan (const xcomplex& z);
friend xcomplex asin (const xcomplex& z);
friend xcomplex acos (const xcomplex& z);
friend xcomplex atanh (const xcomplex& z);
friend xcomplex asinh (const xcomplex& z);
friend xcomplex acosh (const xcomplex& z);

// The return value of pow() is the power of the first
// argument raised to the second one.
// Note that the modulus of the first argument must be
// greater than zero, if the real part of
// the second argument is less or equal than zero, otherwise
// a bad-exponent error is produced.
friend xcomplex pow (const xcomplex& z1, const xcomplex& z2);

public:
// Various constructors. They allow to define
// an extended precision complex number in several ways.
// In addition, they allow for conversions from other
// numeric types.
xcomplex (const struct cxpr* pz = &cxZero);
xcomplex (struct cxpr z);
xcomplex (struct xpr x, struct xpr y = xZero);
xcomplex (xreal x, xreal y = xZERO);
xcomplex (double x, double y = 0.0);
xcomplex (float x, float y = 0.0);
xcomplex (int m, int n = 0);
xcomplex (long m, long n = 0);
xcomplex (unsigned int u, unsigned int v = 0U);
xcomplex (unsigned long u, unsigned long v = 0U);

// This constructor requires a special comment. If
// only the first argument is present, the initial portion
// of the string pointed to by this argument is converted
// into an extended precision complex number, if a
// conversion is possible. If no conversion is possible,
// then the returned number is xNAN + xNANi.
// If the second argument is present and is not null,
// it must be the address of a valid pointer to 'char'.
// Before returning, the constructor will set this pointer

```

```

// so that it points to the character after the last
// character used in the conversion.
xcomplex (const char* str, char** endptr = 0);

xcomplex (string str);
xcomplex (const xcomplex& z);

// Assignment operators. They do not require
// any explanation with the only exception of '%=',
// which combines a '%' operation with an assignment.
// So, x %n is equivalent to x *= pow(2,n) .
xcomplex& operator= (const xcomplex& z);
xcomplex& operator+= (const xcomplex& z);
xcomplex& operator-= (const xcomplex& z);
xcomplex& operator*= (const xcomplex& z);
xcomplex& operator*= (const xreal& x);
xcomplex& operator/= (const xcomplex& z);
xcomplex& operator%=(int n);

// Increment and decrement operators. Both prefixed
// and postfix versions are defined. These operators
// only act on the real part of their argument.
xcomplex& operator++ ();
xcomplex& operator-- ();
xcomplex& operator++ (int dummy);
xcomplex& operator-- (int dummy);

// Destructor. You will never have to recall it
// explicitly in your code.
~xcomplex (void);

// Integer exponent power. For any extended precision
// complex number 'z', z(n) is equal to 'z' raised to 'n'.
xcomplex operator() (int n) const;

// This is the usual unary minus.
xcomplex operator-() const;

// For any extended precision complex number 'z',
// !z evaluates to 1 when
// 'z' is null, else it evaluates to 0.
int operator!() const;

// Functions for conversions. z._2dcomplex(), z._2fcomplex(),
// z._2cxpr() and z._2string() convert the extended
// precision complex number 'z' in a double precision

```

```

// complex number, in a single precision complex
// number, in a structure of type 'cxpr', and in a
// string, respectively.
double_complex _2dcomplex () const;
float_complex _2fcomplex() const;
struct cxpr _2cxpr() const;
string _2string() const;

// For any extended precision complex number 'z',
// z.real() and z.imag() return, respectively, the
// real and the imaginary part of 'z' in the form
// of an extended precision number.
xreal real () const;
xreal imag () const;

// For any extended precision complex number 'z',
// z._real() and z._imag() return, respectively, the
// real and the imaginary part of 'z' in the form
// of a structure of 'xpr' type.
struct xpr _real () const;
struct xpr _imag () const;

// For any extended precision complex number 'z',
// z.dreal() and z.dimag() return, respectively, the
// real and the imaginary part of 'z' in the form
// of a double precision number.
double dreal () const;
double dimag () const;

// For any extended precision complex number 'z',
// z.freal() and z.fimag() return, respectively, the
// real and the imaginary part of 'z' in the form
// of a single precision number.
double freal () const;
double fimag () const;

// For any extended precision complex number 'z',
// z.sreal() and z.simag() return, respectively, the
// real and the imaginary part of 'z' in the form
// of a string.
string sreal () const;
string simag () const;

// The next functions allow to set (or reset)
// the real and the imaginary part of a complex number.
void real (const xreal& x);

```

```

void imag (const xreal& x);
void real (struct xpr x);
void imag (struct xpr x);
void real (const struct xpr* px);
void imag (const struct xpr* px);
void real (double x);
void imag (double x);
void real (float x);
void imag (float x);
void real (int n);
void imag (int n);
void real (long n);
void imag (long n);
void real (unsigned int u);
void imag (unsigned int u);
void real (unsigned long u);
void imag (unsigned long u);
void real (const char* str, char** endptr = 0);
void imag (const char* str, char** endptr = 0);
void real (string str);
void imag (string str);

// The member function xcomplex::getfrom() can be used
// to recover an extended precision complex number from an
// input stream. The input stream is passed as argument to
// the function.
// The return value is 0 in case of input error (in case of
// End-Of-File, for example).
// When it starts to process its input, this function drops
// all eventual leading white spaces.
// After reading the first non space character, it continues
// to read from the input stream until it finds a white
// space or reaches the End-Of-File.
// Then it tries to convert into an extended
// precision complex number the (initial portion of the)
// string which has just been read from the input stream.
// If no conversion can be performed, then z.getfrom(is)
// sets 'z' to the value xNAN + xNANi.
// If the exactly converted value would cause overflow in
// the real or/and in the imaginary part, then the real part
// or/and the imaginary part of 'z' are set to xINF or x_INF,
// according to the signs of the correct value.
int getfrom (istream& is);

// The member function xcomplex::print() can be used to
// write an extended precision complex number to an output

```

```

// stream. The output stream is passed to the function as
// first argument. The next three arguments have the same
// meanings of the fields 'notat', 'sf'
// and 'lim' of the structure 'xoutflags', respectively (see section
// "Real Arithmetic").
int print (ostream& os, int sc_not, int sign, int lim) const;

// The function call z.asprint(sc_not, sign, lim) returns
// a buffer of characters with the representation,
// in form of a decimal ASCII string,
// of the extended precision complex number 'z'.
// The arguments 'sc_not', 'sign' and 'lim' are used
// to format the string.
// They have the same meanings of the fields 'notat', 'sf'
// and 'lim' of the structure 'xoutflags', respectively (see section
// "Real Arithmetic").
// The buffer returned by this function is malloc'ed inside
// the function. In case of insufficient memory, the null
// pointer is returned.
char* asprint (int sc_not, int sign, int lim) const;

// The following static functions are used to set
// or get the values of the fields of the structure
// 'xcomplex::ioflags'. This structure is a static member
// variable of the class 'xcomplex' and it is used by
// the output operator << to know how to format its second
// argument. The meaning of the
// fields of the structure 'xcomplex::ioflags' is explained
// in the section "Real arithmetic".

// xcomplex::set_fmt (which) sets to 'which' the value
// of 'xcomplex::ioflags.fmt' .
static void set_fmt (short format);

// xcomplex::set_notation (which) sets to 'which' the value
// of 'xcomplex::ioflags.notat' .
static void set_notation (short notat);

// xcomplex::set_signflag (which) sets to 'which' the value
// of 'xcomplex::ioflags.sf' .
static void set_signflag (short onoff);

// xcomplex::set_mfwd (which) sets to 'which' the value
// of 'xcomplex::ioflags.mfwd' .
static void set_mfwd (short wd);

```

```

// xcomplex::set_lim (which) sets to 'which' the value
// of 'xcomplex::ioflags.lim' .
static void set_lim (short lim);

// xcomplex::set_padding (which) sets to 'which' the value
// of 'xcomplex::ioflags.padding' .
static void set_padding (signed char ch);

// xcomplex::set_ldelim (ch) sets to 'ch' the value
// of 'xcomplex::ioflags.ldel' .
static void set_ldelim (signed char ch);

// xcomplex::set_rdelim (ch) sets to 'ch' the value
// of 'xcomplex::ioflags.rdel' .
static void set_rdelim (signed char ch);

// xcomplex::get_fmt () returns the current value
// of 'xcomplex::ioflags.fmt' .
static short get_fmt (void);

// xcomplex::get_notation () returns the current value
// of 'xcomplex::ioflags.notat' .
static short get_notation (void);

// xcomplex::get_signflag () returns the current value
// of 'xcomplex::ioflags.sf' .
static short get_signflag (void);

// xcomplex::get_mfwd () returns the current value
// of 'xcomplex::ioflags.mfwd' .
static short get_mfwd (void);

// xcomplex::get_lim () returns the current value
// of 'xcomplex::ioflags.lim' .
static short get_lim (void);

// xcomplex::get_padding () returns the current value
// of 'xcomplex::ioflags.padding' .
static signed char get_padding (void);

// xcomplex::get_ldelim () returns the current value
// of 'xcomplex::ioflags.ldel' .
static signed char get_ldelim (void);

// xcomplex::get_rdelim () returns the current value
// of 'xcomplex::ioflags.rdel' .

```

```

    static signed char get_rdelim (void);
private:
    struct cxpr br; /* binary representation */
    static struct xoutflags ioflags; /* output flags */
};

// Some useful constants:
// cxZERO    == 0
// cxONE     == 1
// cxI       == 1i
extern const xcomplex cxZERO, cxONE, cxI;

} /* End namespace HPA */

#define xi cxI
#define xj cxI
#define _i cxI
#define _j cxI

#endif /* _XCOMPLEX_H_ */

```

13 Acknowledgments

A big thanks to Daniel A. Atkinson, since without his work the HPA library would not exist. I have also to thank Aurelio Marinho Jargas <verde (at) aurelio (dot) net>, author of txt2tags (<http://txt2tags.sf.net>), a free (GPL'ed) and wonderful text formatting and conversion tool, which I used extensively in writing this manual and the web page of the HPA library.

Last but not least, I want to thank all the people till now involved in the Free Software community, starting from those ones directly involved in the GNU project (<http://www.gnu.org>).

14 GNU Free Documentation License

GNU Free Documentation License
Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
<<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall

directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material

this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on

the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all

of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements",

and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include

translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the

GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.