hfloat

a C++ library for high precision computations

Jörg Arndt
arndt@jjj.de

this documentation[1] was LaTeX'd at September 29, 1997

[1]This document and hfloat are online at http://www.jjj.de/hfloat/

Chapter 1

Quick Start

1.1 What hfloat is (good for)

- hfloat (for 'huge floats') is a library package for doing calculations with floating point numbers of extreme precision. It is optimised for computations with 1000...several million digits. The computations can be done in (almost) arbitrary radix.
- The library contains routines for add, subtract, multiply, divide, n-th power, square root, n-th root, logarithm, exp, polynom, poly-root and many more.
- There are implementations of several superlinear converging algorithms for the computation of pi=3.14159265... (in src/pi/). On a PC or workstation the computation of 1 million decimal pi-digits takes less than 1h (1h on a i486/100, about 20min on an AMD K6/233).
- Code examples for the usage of the lib are in `simpleex/`.
- Code for a binary that collects the pi algorithms can be found in `examples/`.
- Included is the fxt-library, containing many FFT-implementations, code for convolution, correlation, spectrum and much more. It has its own documentation in `src/fxt/`.
- High precision computations test your systems reliability (hardware and compiler): every little error results in garbage digits (or coredump)
- Digits of appropriate constants can be used as high quality 'random' numbers eg. for cryptographic stuff.
- You may (and are encouraged to) use the fast multiplication in your own noncommercial bignum software.
- hfloat (and the accompanying texts) may help you to learn about the things mentioned above.

1.2 Compiling the library

hfloat is developed under Linux (versions 2.0.x) with GNU C (versions 2.7.x). Compilation under other UNIXes should be possible with few or no changes, especially if GNU C is used (compiled on a DEC alpha and on a HP without any changes). Basically you need a decent make-utility and a C-compiler, GNU C and GNU make are strongly recommended. To compile the library

- Type '`make dep`' (for dependency files)
- Type '`make lib`' to make the library

DOS users, you have to edit the makefiles:

- truncate many filenames
- replace '`g++`' with the name of your C++ compiler (e.g. djgpp)
- split lines with length > 120 chars (namely lists of *.o files)
- replace the commands rm, mv, ls by their equivalents
- work around the 'one statement per line' limitation

... and much more unpleasant tasks to work around the braindead limitations of that crappy OS wannabe. Let me cite: 'I tried to compile the hfloat package with Borland 4.52 and MS VC++ 4.0: a hopeless task.'

Chapter 2

More about hfloat

2.1 Functions of the hfloat library

Member functions of the `hfloat` class are declared in `src/include/hfloat.h`

Static member functions, they affect some property of the whole `hfloat` class:

- `max_prec()` returns the maximal precision that can be used (unit is `LIMB`s)[1];
  `max_prec(unsigned long m)` sets the maximal precision `m`;
  normally you will call it once at the beginning of your code
- `radix()` returns the radix, currently the same for all `hfloat`s;
  use the function `hfloat_set_radix(int r)` to set the radix.
  Note that after a radix change all `hfloat`s contain garbage data in their mantissa so they must be reassigned new values before they are read.

Nonstatic member functions, they affect one particular instance of an `hfloat`:

- the constructor `hfloat()` creates an `hfloat` with `max_precision()`
- the constructor `hfloat(unsigned long n)` creates an `hfloat` with `n` `LIMB`s
  it has the `explicit` modifier (cf. note below)
- the copy constructor `hfloat(const hfloat &h)` creates a copy of the `hfloat h`
- the(assignment) operator `=`
  for arguments (i.e. right hand side quantities) `int`, `long`, `unsigned long`, `double`, strings (i.e. `char *`) and `hfloat`,
- `size()` returns the size of the mantissa (in `LIMB`s)
  `size(unsigned long s)` resizes the mantissa to `s` `LIMB`s
- `prec()` returns the current working precision (in `LIMB`s)
  `prec(unsigned long p)` sets the working precision to `p` `LIMB`s
- `exp()` returns the exponent (with respect to `LIMB`s);
  `exp(long)` sets the exponent
- `sign()` returns $+1, 0, -1$ as usual;
  `sign(int s)` sets the sign

[1] for the notion of a `LIMB` see page 7

Always create `hfloat`s like `hfloat a;` (for default, i.e. maximal precision) or `hfloat a(1024);` (for a precision of 1024 `LIMB`s). Do not try to directly initialise `hfloat`s like `hfloat a = 1234;` this would create a hfloat with *precision* 1234 and value 0, surely not what was intended! To avoid mistakes like that the constructor `hfloat(unsigned long)` has the explicite modifier. Instead say: `hfloat a; a = 1234;` .

Functions and operators for `hfloat`s are declared in `src/include/hfloatfu.h`:

- the shortcut operators `+=, -=, *=, /=`
  for right arguments `long` and `hfloat`,
- comparison operators `==, !=, >=, <=, <, >`
  (for comparisons of `hfloat`s with `hfloat`s or `long`s)
- comparison operators `<, >`
  (for comparisons of `hfloat`s with `double`s)
- functions of the type fct(src,result)
  where `fct` ∈ `inv, sqr, sqrt, isqrt, cbrt, log`[2]`, exp, ...`
- functions of the type f(src1,src2,result)
  where `fct` ∈ `add, sub, mul, div, pow, root, iroot, log, ...`

The function names should be self-explanatory.
Print `src/include/hfloatfu.h` and `src/include/hfloat.h` now!

There are also the binary operators `+, -, *, /` and functions that return a `hfloat`, like `fct(src)` where `fct` ∈ `inv, sqrt, ...` but do not use them if you are after performance: they create temporary `hfloat`s and are there only to allow lazy coding.

E.g. instead of
```
x = a + b;
```
use
```
x = a;   x += b;
```
or
```
add(a, b, x);
```
Instead of
```
x = exp(a);
```
use
```
exp(a,x);
```
Note that the result is always the rightmost argument.

2.2 Using hfloats in your own code

In order to write your own code that uses hfloats

- you must `#include src/include/hfloatfu.h` to get the functions of the hfloat lib.
- use `hfloat::max_prec(n)` where `n` is the precision in `LIMB`s (use a power of two)
- use `hfloat::radix(rx)` where `rx` is the radix (use 10000 for decimal numbers or 65536 for hex numbers)
- use `hfverbosity::tell_all()` if you like to have many operations echoed, `hfverbosity::hush_all()` for silent operations

[2]logarithm to base 2.71828...

- `#include src/include/mybuiltin.h` for timing (`start_timer()` and `return_elapsed_time()`).
- when compiling `yoursrc.cc` that uses `hfloat`s give the library `libhflt.a` (or whatever you renamed it to).
- for extreme precisions increase the maximal workspace size, as described on page 7.

Look into `src/include/hfloatfu.h` for the functions available.

Cf. `simpleex/ex?.cc` for some simple examples of how to use hfloats. Here is `ex2.c`:

```
//
// simple example 2:
// compute pi with 2 different algorithms
//

#include  "../src/include/hfloatfu.h"
#include  "../src/include/mybuiltin.h"  // for timing

int main()
{
    // precision in LIMBs, use o power of two:
    hfloat::max_prec(2048);

    // radix, use 10000 (decimal) or 65536 (hex numbers):
    hfloat::radix(10000);

    hfloat pi1, pi2, d;
    double dt1, dt2;     // for timing

    // first pi computation:
    start_timer();
    pi_4th_order(pi1);  // compute pi using borweins quartic algorithm
    dt1 = return_elapsed_time();
    print("\n pi1=\n", pi1, 44); // print the first 44 LIMBs of pi1
    print_last("\n last digits are \n", pi1, 16); // print the last 16 LIMBs
    save("pi1.dat",pi1);  // save pi1 to file "pi1.dat"

    // second pi computation:
    start_timer();
    pi_agm(pi2);  // compute pi using the agm algorithm
    dt2 = return_elapsed_time();

    /* // you can choose from these, cf. src/pi/pi*.cc:
    pi_4th_order(pi2, var);  // var = 0,1
    pi_5th_order(pi2);
    pi_agm(pi2,var);         // var = 0,1
    pi_agm3(pi2,var);        // var = +1,-1,+4,-4
    pi_2nd_order(pi2);
    pi_derived_agm(pi2);
    pi_3rd_order(pi2);
    pi_9th_order(pi2);
    pi_cubic_agm(pi2);
    pi_arctan(pi2, fno);     // fno = 2,3,4,5,6,7,11
    */

    print("\n pi2=\n", pi2, 44);
    print_last("\n last digits are \n", pi2, 16);
    save("pi2.dat",pi2);  // save pi2 to file "pi2.dat"
```

```
    cout << "\n pi1 took " << dt1 << " seconds ";
    cout << "\n pi2 took " << dt2 << " seconds ";

    d =  pi1;
    d -= pi2;

    cout << "\n\n decimal precision of the results is "
         << pi1.dec_prec() << " digits "
         << endl;

    print("\n difference of the results: p1-p2= \n",d,8);

    long dl = pi1.prec()-ABS(cmp_limbs(pi1,pi2));
    cout << "\n i.e. the last " << dl << " LIMBs disagree " << endl;

    return 0;
}
```

When in the directory `simpleex/` type `make ex2` to create the executable `ex2`. When run it creates an output[3] similar to this:

```
 ----====== HUGE_FLOAT ver 20-august-1997 ======----
    author: Joerg Arndt, email: arndt@jjj.de
 compiled using GNU C++ version "2.7.2.1"
 at Sep  1 1997, 19:53:31
    hfloat is online at http://www.jjj.de/hfloat/
    (check my homepage for the latest version of hfloat)
 radix set to 10000

 size of workspace is 32768 bytes  (maxsize= 4194304 bytes)
 max precision set to 1024 LIMBs =4096 decimal digits   =13606 bits
 set mul_convolution to FHT_CNVL
 set sqr_convolution to FHT_CNVL
 fxt multiplies ARE checked via sum of digit test
 iterations for inverse n-th root are NOT checked

 pi1=
+.31415926535897932384626433832795028841971693993751058209749445923078164062
8620899862803482534211706798214808651328230664709384460955058223172535940812
8481117450284102701938 5*10^1

 last digits are
+. <...1008LIMBs+> 2065310989652691862056476931257058635662018558100729360632570537

 pi2=
+.31415926535897932384626433832795028841971693993751058209749445923078164062
8620899862803482534211706798214808651328230664709384460955058223172535940812
8481117450284102701938 5*10^1

 last digits are
+. <...1008LIMBs+> 2065310989652691862056476931257058635662018558100729360659786498

 pi1 took 0.88 seconds
 pi2 took 0.82 seconds

 decimal precision of the results is 4096 digits
```

[3] the data of the first π computation can be found in the file `pi1.dat`

```
 difference of the results: p1-p2=
-.272159610000000000000000000000000*10^-4084

 i.e. the last 2 LIMBs disagree
```

To repeat the computations with a precision $> 250,000$ decimal digits change the line with the statement `hfloat::max_prec(2048);` to `hfloat::max_prec(65536);` [4]. If you want more π-computation and are tired of recompiling than goto section 2.5, there is a canned π example (though the code doesn't look too nice).

Also check the other code in `simleex/` ! For 'real world' examples that use more functions of the hfloat lib see `src/pi/pi*.cc`, start there with `src/pi/pi4th.cc`.

2.3 Computations with extreme precision

In order to do computations with the maximal possible precision on your computer you have to manually set the maximal workspace size: With the default maximum[5] of 4MB you can work with precisions of (radix 10,000 assumed) up to 1 million decimal digits. To use bigger sizes than the default set the environment variable `HFLOAT_MAX_WORKSPACE`. No swap must occur when using the workspace, i.e. it has to be unused RAM. Set `HFLOAT_MAX_WORKSPACE` to the amount of physical RAM minus the bytes used by operating system and other running programs. Typically set to RAMsize minus 8MB or so. Do *not* give the total amount installed RAM, your machine will swap to death if you try really use it for hfloat. Currently if you don't give a power of 2 then the size is truncated to the next smaller power of 2. You may append 'k' for kilobyte or 'M' for megabyte.

You might have to say something like `export HFLOAT_MAX_WORKSPACE=20M` to set it to 20MB. This should work fine on a computer with 32MB RAM installed.

2.4 Precision and radix

For the quick readers I begin with the resumee:

- for decimal digits and precisions up to 2^{23} `LIMB`s[6] use radix 10,000 (for greater precisions choose radix 1,000)
- for hexadecimal digits and precisions up to 2^{18} `LIMB`s[7] use radix 65,536 (for greater precisions choose radix 4,096)

The mantissa of a `double` consists of (typically) 53 bits[8], its radix (base of representation) is 2. The analogous quantity in the mantissa of a hfloat is a `LIMB` (typedef'd to a 16 bit unsigned int). So the radix of a hfloat can be in the range 2...65536($= 2^{16}$).

If one wants to get decimal numbers one would *not* use radix 10 but the greatest power of 10 that is $\leq 2^{16}$, which is 10000. Due to the implementation of the convolution there is a second restriction to the radix: The cumulative sums c_k have to be represented exactly enough to distinguish every (integer) quantity from the next bigger (or smaller) value. The highest possible value for a c_k, c_m, must not jump to $c_m \pm 1$ due to numerical errors. For radix R and a precision of N `LIMB`s

$$c_m \quad = \quad N\,(R-1)^2 \tag{2.1}$$

[4] radix is 10,000, one gets 4 decimal digits per `LIMB` or a total of 262,144
[5] see `src/dt/workspace.cc`
[6] corresponding to more than 32 million decimal digits
[7] corresponding to more than 1 million hexadecimal digits
[8] Of which only the 52 least significant bits are physically present, the most signicant bit is implied to be always set.

This is N times the product of two 'nines' $(R-1)$, which can appear in the middle term of the convolution. c_m needs

$$\log_2(N\,(R-1)^2) \quad = \quad \log_2 N + 2\,\log_2(R-1) \tag{2.2}$$

bits to be represented exactly.

Due to numerical noise there must be a few more bits for safety. The c_k are computed using `doubles`. We need to have

$$M \quad \geq \quad \log_2 N + 2\,\log_2(R-1) + S \tag{2.3}$$

where S :=safetybits and M :=mantissabits. With $\log_2(R-1) < \log_2(R)$ we have equivalently

$$N_{max}(R) \quad = \quad 2^{M-S-2\,\log_2(R)} \tag{2.4}$$

Thus if we want base 2 numbers we would first choose radix $R = 2^{16}$ but this can (if we have $M = 53$ mantissabits and require $S = 3$ safetybits) only be used for precisions up to

$$N_{max}(\text{radix} = 65,536) \quad = \quad 2^{53-3-2\cdot 16} = 2^{18} = 256 kilo\texttt{LIMB}\text{s} \tag{2.5}$$

(corresponding to $4096 kilo$ bits $= 1024 kilo$ hex digits).

$$N_{max}(\text{radix} = 32,768) \quad = \quad 2^{20} = 1 Mega\texttt{LIMB}\text{s} \tag{2.6}$$

(corresponding to $15360 kilo$ bits $= 3840 kilo$ hex digits).

$$N_{max}(\text{radix} = 16,384) \quad = \quad 2^{22} = 4 Mega\texttt{LIMB}\text{s} \tag{2.7}$$

(corresponding to $57344 kilo$ bits $= 14336 kilo$ hex digits).

For decimal numbers

$$N_{max}(\text{radix} = 10,000) \quad = \quad 2^{53-3-2\cdot 13.29} = 2^{23.42} > 8 Mega\texttt{LIMB}\text{s} \tag{2.8}$$

($N = 23$ corresponding to $32 Mega$ decimal digits).

$$N_{max}(\text{radix} = 1,000) \quad = \quad 2^{30.07} > 1 Giga\texttt{LIMB}\text{s} \tag{2.9}$$

($N = 30$ corresponding to $3 Giga$ decimal digits).

If the `LIMB`s weren't restricted to 16 bits:

$$N_{max}(\text{radix} = 100,000) \quad = \quad 2^{16.78} > 16 kilo\texttt{LIMB}\text{s} \tag{2.10}$$

($N = 16$ corresponding to $80 kilo$ decimal digits).

$$N_{max}(\text{radix} = 1,000,000) \quad = \quad 2^{10.13} > 1 kilo\texttt{LIMB}\text{s} \tag{2.11}$$

($N = 10$ corresponding to $6 kilo$ decimal digits).

2.5 Compiling & running the π-example code

To compile the code in the directory `examples/`

- Type '`make example`'.
- cd to the `bin` directory, find the example binary `pi` there.

- Type '`pi --help`' for usage information (the help text can also be found in the file `doc/pihelp.txt`).
- Type '`pi 8`' to run the program. It will compute 1024 decimal digits of π (in less than 1 second on a i486/100).
- See the file `result.txt` for the digits of π. All but the last few digits (10 or so) of the output should be correct. Check correctness like this:
- Type '`pi 8 1`' to run the program again using another algorithm. Only the last few digits should differ from the last result.
- Type '`pi 8 0 65536`' to get 1024 hex digits of π.

2.6 Limitations of hfloat

- lousy performance for precisions <200 decimal digits
- no trigonometric functions
- no type complex
- not good for integer computations
- after a radix change one has to reset all hfloats
- no mass storage code, i.e. precision is limited by the size of your physical RAM.
- not thoroughly tested !
- code uglyness:
  - The code in `examples/` is messy
  - The code in `testing/` and `timing/` is horrible
  - The code in the iterations has to be cleaned up
- not very good for 'real' work: hfloat, together with the included documents, is intended mainly for learning about hight precision computations.

2.7 Structure of hfloat

The hfloat code is divided into three layers:

1.) The class hfloat (top, user interface) layer. A hfloat consists of a unique id number (for internal use only) and a pointer to class hfguts. Currently there is a one-to-one correspondence between hfloats and hfguts. The arithmetical operators and functions like `mul(src1,src2,dest)` are here. The iterations (for root extraction, inversion, ...) are also in this layer.
Declarations are in `src/include/hfloat.h` (class), `src/include/hfloatop.h` (operators) and `src/include/hfloatfu.h` (functions). Source files are in `src/hf/`.

2.) The class hfguts (arithmetical) layer. A hfguts consists of an exponent, a sign, a flag whether it is infinite, a flag whether the mantissa is read only, a precision flag and a pointer to class hfdata (its mantissa data). Here the exponent, sign, etc. stuff happens, the operations with the mantissa are handled by hfdata. Typical function names are `gt_something(...)`.
Declarations are in `src/include/hfguts.h` (class) and `src/include/hfgutsfu.h` (functions). Source files are in `src/gt/`.

3.) The class hfdata (mantissa operations, number crunching) layer. A hfdata consists of a pointer to the digit field, a flag for the allocated size of mantissa data (as opposed to: precision in hfguts), a flag whether the data was allocated by this object, a counter for the links to this mantissa (other numbers can link read only to the mantissa data), static radix stuff (the radix is in `rx`) and counters for the work done so far (for statistics). Typical function names are `dt_something(...)`,
Declarations are in `src/include/hfdata.h` (class) and `src/include/hfdatafu.h` (functions). Source files are in `src/dt/`.

Auxiliary functions that work on `LIMB`s or `double`s typically have names like `i_something(...)`, `d_something(...)`, `id_something(...)` or `di_something(...)`.
Declarations are in `src/include/auxarith.h`.

2.8 Organisation of the files

directory	which files are there
`bin/`	binaries and libs
`doc/`	the main documentation
`doc/e271828/`	stuff about the computation of e
`simpleex/`	simple example code, look there!
`examples/`	example application(s)
`src/`	(source directory)
`src/fxt/`	source for the fxt-librarry
`src/include/`	header files for the hfloat-librarry
`src/tz/`	source for the transcendental functions
`src/hf/`	source for the hfloat-layer
`src/gt/`	source for the hfguts-layer
`src/dt/`	source for the hfdata-layer
`src/pi/`	source for the π-library
`testing/`	cryptic test routines, better ignore
`timing/`	timing routines.
`*/bucket/`	snippets, temporarily unused files & garbage

Selected files in some directories:

- `src/include/`:
  - `hfloatfu.h`: hfloat functions
  - `hfloat.h`: the hfloat class
  - `hfverbosity.h`: adjust how talkative the hfloat operations are
  - `hfgutsfu.h`: hfguts functions
  - `hfguts.h`: the hfguts class
  - `hfdatafu.h`: hfdata functions
  - `hfdata.h`: the hfdata class
  - `convolut.h`: functions and defines for long multiplications
- `src/hf/`:
  - `hfloat.cc`: hfloat class member functions
  - `hfloatop.cc`: hfloat class operators
  - `init.cc`: magic initialiser for the hfloat class

  - `it*.cc`: iterations for root extraction etc.

- `src/gt/`:

  - `hfguts.cc`: hfguts class member functions
  - `gt*.cc`: hfguts functions

- `src/dt/`:

  - `hfdata.cc`: hfdata class member functions
  - `dt*.cc`: hfdata functions

Appendix A

Distribution policy & no warranty

——— legal stuff ———

- The hfloat package and code is freeware, you may use it at no cost for noncommercial purposes.
- The copyright remains by the author (Jörg Arndt, `arndt@jjj.de`). Obvious exceptions are the included pieces of code by other authors.
- You may (and are encouraged to) give hfloat away free of charge. Always give a pointer to the original hfloat if you redistribute pieces of it.
- You are not allowed to make money with hfloat (or parts of it) in any way.
- Before putting hfloat on a CD or similar get my agreement.
- WARNING: hfloat is distributed 'as is', you are using hfloat at your own risk. I will take no responsibility for potential damage that might be caused.

——— end of legal stuff ———

- Please be nice and give me credits if you use hfloat for anything interesting/noticable/scientific. I am interested in hearing about your project.
- If you use hfloat for scientific purposes then cite it in your publications as you would cite an ordinary publication.
- Please check for newer versions before passing hfloat on. I tend to upload new versions even for small changes.
- If you notice errors in the code or the documentation please take the time and drop me an email.

Appendix B

Other high precision libraries

The following is an extract of the documentation that comes with the hfloat *package.*

Mentioned here are only packages that I have seen myself. For more info see `bignums.txt` from `ftp://ripem.msu.edu/pub/bignum/` and the software there. Note that the list doesn't seem to be maintained anymore. Also check `http://www.jjj.de/hfloat/`.

Libraries for extreme precisions:

- apfloat:
  lib similar to hfloat, uses number theoretic transforms (NTTs) with Chinese Remainder Theorem (CRT) technique, has type complex and trigonometric functions and mass storage multiplication.
  author: Mikko Tommila (`mikko.tommila@hut.fi`)
  site: `http://www.hut.fi/~mtommila/apfloat/`
- cln:
  ('Class Library for Numbers') big- int, rational and float, complex and modint C++ library, designed to be fast, memory efficient, complete and easy to use. Includes gmp. Has nice features like binary splitting algorithm and Schönhage-Strassen multiplication (with Fermat number transforms): The π-computation in CLN uses binsplit and the Chudnovsky formula and is about 3 times (!) faster than hfloat. CLN is a killer: be sure to fasten your seat belt before looking into the code !
  author: Bruno Haible
  site: `ftp://ma2s2.mathematik.uni-karlsruhe.de/pub/gnu/cln*`
- mpfun:
  'a multiple precision floating point computation package' FORTRAN code and a bit vector machine orientated. From the documentation (TeX) in `mpfun_tex_papers.tar.Z` you can learn a lot about the topic (see it!).
  author: David H. Bailey (`dbailey@nas.nasa.gov`)
  site: send email with subject '`send help`' to `mp-request@nas.nasa.gov` or `http://www.nas.nasa.gov/NAS/RNRreports/dbailey/dbailey.html`

Libraries for moderate ... medium precisions:

- pari/gp:
  arbitrary prec int and floats (does also operations in field extensions) optimised for approximately 50...1000 (?) digits *lots* of functions, many from numbertheory, callable from C programs, see it ! C code (plus asm for i386 and 68000), used in the freeware computer algebra system MuPAD (see below)
  author: C. Batut, D. Bernardi, H. Cohen and M. Olivier (`pari@math.u-bordeaux.fr`)
  sites: `ftp://megrez.math.u-bordeaux.fr/pub/pari/` or `ftp://math.ucla.edu`

- lidia:
  C++ library for computational number theory; 'highly optimized implementations'. Includes types int, rational, float, complex, modint, number field, vector, matrices, ...
  author: Buchmann et al.
  site: `ftp://crypt1.cs.uni-sb.de/pub/systems/LiDIA/` or `ftp://ftp.cs.uni-sb.de/pub/LiDIA/`

- bigflt:
  multiple precision floats, C code (plus asm for intel) used in fractint versions from 19.0 for huge magnifications
  author: Wesley Loewer (`loewer@tenet.edu`)
  sites: where you get fractint

- integer.h and rational.h:
  abitrary prec int/rational with GNU C++ compiler
  sites: where you get GCC, e.g. `ftp://prep.ai.mit.edu/pub/gnu/`

- gmp: (the GNU mp library)
  general emphasis on speed, asm for many processors included
  author: Torbjörn Granlund (`tege@zevs.sics.se`)
  sites: `ftp://prep.ai.mit.edu/pub/gnu/gmp*`

General purpose:

- MuPAD: a freeware full featured computer algebra system !
  author: Benno Fuchssteiner et al.
  sites: `ftp://ftp.uni-paderborn.de/MuPAD/`