hfloat

a C++ library for high precision computations

Jörg Arndt
`arndt@jjj.de`

this documentation[1] was LaTeX'd at July 3, 2007

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

0.1 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 addition, subtraction, multiplication, division, n-th power, square root, n-th root, logarithm, exponentiation and many more.
- There are implementations of several superlinear converging algorithms for the computation of pi=3.14159265... (in src/pi/).
- Code examples for the usage of the library are in `examples/`.
- Code for a binary that collects the pi algorithms can be found in `calcpi/`.
- Included is the fxt-library, containing many FFT-implementations, code for convolution, correlation, spectrum and much more.
- High precision computations test your systems reliability (hardware and compiler): every little error results in garbage digits.
- 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 fast arithmetical algorithms.

0.2 Compiling the library

hfloat is developed under Linux with GNU C It has been compiled under Linux on intel (386, 486, 586 and compatibles and ia64), power pc, alpha and S/390. Compilation under other UNIXes should be possible with few or no changes, especially if GNU C is used. Basically you need a decent make-utility and a C-compiler[1], 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

0.3 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:

- `hfloat::default_prec()` returns the default precision that can be used (unit is `LIMB`s)[2];
  `hfloat::default_prec(unsigned long m)` sets the default precision `m`;
  call it once at the beginning of your code

[1] expect problems with crippled Microsoft compiler-oids

[2] for the notion of a `LIMB` see page 4

- `hfloat::radix()` returns the radix, which is the same for all `hfloat`s;
  use `hfloat::radix(unsigned long 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.
  normally you will use `hfloat::radix(unsigned long r)` once at the beginning of your code

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

- the constructor `hfloat()` creates an `hfloat` with `default_precision()`
- the constructor `hfloat(unsigned long n)` creates an `hfloat` with `n` `LIMB`s
  Create `hfloat`s like `hfloat a;` (for default precision) or `hfloat a(1024);` (for a precision of 1024 `LIMB`s). Direct initialization like `hfloat a = 1234;` is not possible because the constructor `hfloat(unsigned long)` has the explicit modifier. Instead say: `hfloat a; a = 1234;` .
- 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, resizes if necessary.
- `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

Operators for `hfloat`s are declared in `src/include/hfloatop.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 for `hfloat`s are declared in `src/include/hfloatfu.h`:

- functions of the type func(src,result)
  where `func` $\in$ `inv, sqr, sqrt, isqrt, cbrt, log`[3]`, exp, ...`
- functions of the type f(src1,src2,result)
  where `func` $\in$ `add, sub, mul, div, pow, root, iroot, log, ...`

The function names should be self-explanatory.
Print `hfloat.h`, `hfloatfu.h` and `hfloatop.h` now!

[3]logarithm to base 2.71828...

There are also the binary operators `+, -, *, /` and functions that return a `hfloat`, like `func(src)` where `func` ∈ `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.

0.4 Using hfloats in your own code

In order to write your own code that uses hfloats

- you must `#include src/include/hfloat.h` to get the functions of the `hfloat` lib.
- use `hfloat::default_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
- when compiling `yoursrc.cc` that uses `hfloat`s link it against the library `libhflt.a`
- for extreme precisions increase the maximal workspace size, as described on page 3.

Cf. `examples/ex*.cc` for some simple examples of how to use hfloats. Look into `examples/index.txt` for what's there.

0.5 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: Set the environment variable `NOSWAP_BYTES` according to the size of physical RAM where no swapping will occur, e.g. with bash say:
`export NOSWAP_BYTES=32M`
See `src/fxt/auxil/workspace.cc` for the default.

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 increased to the next bigger power of 2. You may append 'k' for kilobyte or 'M' for megabyte.

0.6 Precision and radix

For the quick readers I begin with the resumee:

- for decimal digits and precisions up to 2^{23} `LIMB`s[4] use radix 10,000 (for even greater precisions choose radix 1,000)
- for hexadecimal digits and precisions up to 2^{18} `LIMB`s[5] use radix 65,536 (for even greater precisions choose radix 4,096)

The mantissa of a `double` consists of (typically) 53 bits[6], 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

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

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

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

bits to be represented exactly.

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

$$M \geq \log_2 N + 2\,\log_2(R-1) + S \tag{3}$$

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

$$N_{max}(R) = 2^{M-S-2\,\log_2(R)} \tag{4}$$

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

$$N_{max}(\text{radix} = 65,536) = 2^{53-3-2\cdot 16} = 2^{18} = 256kilo\texttt{LIMBs} \tag{5}$$

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

$$N_{max}(\text{radix} = 32,768) = 2^{20} = 1Mega\texttt{LIMBs} \tag{6}$$

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

$$N_{max}(\text{radix} = 16,384) = 2^{22} = 4Mega\texttt{LIMBs} \tag{7}$$

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

For decimal numbers

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

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

$$N_{max}(\text{radix} = 1,000) = 2^{30.07} > 1Giga\texttt{LIMBs} \tag{9}$$

[4]corresponding to more than 32 million decimal digits
[5]corresponding to more than 1 million hexadecimal digits
[6]Of which only the 52 least significant bits are physically present, the most signicant bit is implied to be always set.

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

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

$$N_{max}(\text{radix} = 100,000) \quad = \quad 2^{16.78} > 16kilo\texttt{LIMBs} \tag{10}$$

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

$$N_{max}(\text{radix} = 1,000,000) \quad = \quad 2^{10.13} > 1kilo\texttt{LIMBs} \tag{11}$$

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

0.7 Compiling & running the π-example code

To compile the code in the directory `calcpi/`

- Type '`make pi`'.
- 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 10`' to run the program. It will compute 4096 decimal digits of π (in about 1 second on a not very ancient computer).
- 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 10 314`' to run the program selftest which uses several algorithms and outputs how many LIMBs differ in the results.
- Type '`pi 8 0 65536`' to get 1024 hexadecimal digits of π.
- Type '`pi 18 16`' to get $>$ 1 million decimal digits of π.

0.8 Limitations of hfloat

Let me emphasize: hfloat is *experimental* code, it is **not** thoroughly tested !

It is not very good for 'real' work, it was and is intended mainly for learning about high precision computations.

Other limitations are:

- lousy performance for small precisions like <200 decimal digits
- no trigonometric functions
- no type complex
- general code uglyness, if you'd like to learn programming: please look elsewhere.

0.9 Structure of hfloat

The hfloat code is divided into two layers:

The class hfloat (top, user interface) layer. A hfloat consists of a unique id number (for internal use only), sign and exponent plus and a pointer to class hfdata. Currently there is a one-to-one correspondence between hfloats and hfdata (it possibly will remain like this). 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/`.

The class hfdata (mantissa operations, number crunching) layer. A hfdata consists of a pointer to the digit field, the size and precision of the mantissa data. 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/`.

0.10 Organisation of the files

directory	which files are there
`bin/`	binaries and libs
`doc/`	the main documentation
`examples/`	simple example code, look there!
`calcpi/`	π example
`src/`	(source directory)
`src/fxt/`	source for the fxt-librarry
`src/fxt/mult/`	source for the fft-multiplication
`src/include/`	header files for the hfloat-librarry
`src/tz/`	source for the transcendental functions
`src/hf/`	source for the hfloat-layer
`src/dt/`	source for the hfdata-layer
`src/pi/`	source for the π-library
`testing/`	cryptic test routines, better ignore
`*/bucket/`	garbage & trash

Selected files in some directories:

- `src/include/`: (declarations)
  - `hfloat.h`: the hfloat class
  - `hfloatfu.h`: hfloat functions
  - `hfloatop.h`: hfloat operators
  - `hfverbosity.h`: adjust how talkative the hfloat operations are
  - `hfdata.h`: the hfdata class
  - `hfdatafu.h`: hfdata functions
- `src/pi/`: implementations of the π-algorithms
- `src/hf/`:
  - `hfloat.cc`: hfloat class member functions
  - `hfloatop.cc`: hfloat class operators

  - `init.cc`: magic initialiser for the hfloat class
  - `it*.cc`: code for iterations (inverse, sqrt, inverse n-th root)

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

0.11 Distribution policy & no warranty

Verbatim copy of the included file `00legal.txt`

```
       author = Joerg Arndt       email: arndt@jjj.de

       permanent address:
         Huehlweg 37,  D-95448 Bayreuth,  Germany

       this software is online at   http://www.jjj.de/

---------------- *** LEGAL NOTICE: *** ------------------

hfloat is distributed under the GNU GENERAL PUBLIC LICENSE (GPL)
cf. the file COPYING.txt

------------ *** end of legal notice *** ------------------


1.) Please be nice and give me credits if you use this package for
    something interesting/noticable/scientific.

2.) If you use this software for scientific purposes then cite it in your
    publications as you would cite an ordinary publication.

3.) If you notice errors in the code or the documentation
    please take the time and drop me an email.
```