Aug. 01 2006

Multiple-Precision Arithmetic Library exflib (C++)

FUJIWARA Hiroshi
fujiwara@acs.i.kyoto-u.ac.jp

exflib

Exflib (extended precision floating-point arithmetic library) is a simple software for multiple-precision arithmetic in scientific numerical computation. Multiple-precision arithmetic is a method for representation and calculation of real numbers with arbitrary accuracy.

We shall abbreviate binary digit as “bit”, and decimal digit as “digit.” And we use the term “decimal” as the same mean of radix-10.

1. Introduction

- Exflib consists of two files: a library file (`libexfloat.a`) and a header file (`exfloat.h`). Though file names are same on all architectures, you must use suitable files for your architecture.
- At the beginning of your program, precision is defined, and the header file `exfloat.h` is included.

```
#define PRECISION 1000
#include "exfloat.h"
```

- The name of multiple-precision type is `exfloat`. A declaration of variables is the following form:

```
exfloat x;
```

You must link the file `libexfloat.a` finally. When you have program `sample.cpp`, `exfloat.h`, and `libexfloat.a` in the same directory, a compile instruction is follows:

- Sun Studio10, Studio11 (AMD64, EM64T / UltraSPARC)

```
% CC -xarch=amd64 -xO2 sample.cpp libexfloat.a
% CC -xarch=v9 -xO2 sample.cpp libexfloat.a
```

- GCC (64-bit / 32-bit)

```
% g++ -m64 -O2 sample.cpp libexfloat.a
% g++ -m32 -O2 sample.cpp libexfloat.a
```

- Intel Compiler (Ver. 9)

```
% icc -O2 sample.cpp libexfloat.a
```

- PGI Compiler

```
% pgCC -O2 sample.cpp libexfloat.a
```

- FCC on HPC2500

```
% FCC -KV9 -O2 sample.cpp libexfloat.a
```

Of course, you can link the library with `-lexfloat` option, instead of `libexfloat.a`.

% *C++* *options* `sample.cpp -lexfloat`

You can define precision by the pre-processor macro `PRECISION`.

% *C++* *options* `-DPRECISION=100 sample.cpp -lexfloat`

2. Substitution

- You can set an `exfloat` type value with an integer.

```
exfloat x;
int i;

x = 1;                          // valid
x = -10;                        // valid
x = i;                          // valid
```

- Literal constants must have a C-style string form (within double quotations) in a substitution sentence.

```
exfloat x;

x = 0.001;                      // not valid

x = "0.001";                    // valid
x = "1/100";                    // valid
x = "#PI/2";                    // valid
x = "1e-10";                    // valid

x = "12.34*(5/6)*(7e-8)*#E";    // valid
```

In a C-style string, you can put decimal numbers (with fixed point format or scientific format), mathematical constants (π `#PI`, the natural logarithm base e `#E`, or Euler's constant γ `#G`), their arithmetics, and parenthesis(). Strings are parsed and calculated at run time. Syntax errors in the strings are detected at run time.

- You can not substitute a sigle-precision type (`float`) or a double precision type (`double`) to `exfloat` type. You need an explicit type conversion.

```
float f;
double d;
exfloat x;

x = f;                          // invalid
x = static_cast<exfloat>(f);    // valid

x = d;                          // invalid
x = static_cast<exfloat>(d);    // valid

x = 0.1;                        // invalid
x = static_cast<exfloat>(0.1);  // valid, but not accurate
```

Values set with a single- or double-precision type have the same accuracy as single- or double-precision type respectively. (`float` has about 7 digits, and `double` has about 15 digits)

```
x = "0.1";                      // valid, accurate
x = "1e-10";                    // valid, accurate
x = "1/10";                     // valid, accurate
```

3. Output

There are two ways to output an `exfloat` value in decimal:

1. Convert `exfloat` to built-in floating-point value, then output the built-in value. (fast, low-accurate)
2. Directly output the `exfloat` type (slow, high- and arbitrary accurate)

- Convert `exfloat` to built-in floating-point type, then output the built-in floating-point type.

```
exfloat x;
double tmp;

tmp = static_cast<double>(x);
std::cout << tmp;
```

or

```
exfloat x;

std::cout << static_cast<double>(x);
```

- Directly output; convert to a string then output

  The stream output operator `<<` of `exfloat` is overloaded, then the next is valid.

```
exfloat x;

std::cout << x;
```

  You can put an `exfloat` type to `ofstream` and the standard output `std::cout` in the same way.

An output format and precision are controlled with a manipulator (`iomanip`) as same as built-in types.

- Fixed-Point Format (Decimal)

```
std::cout << std::setiosflags(std::ios::fixed) << x;
```

- Floating-Point Format (Scientific Format)

```
std::cout << std::setiosflags(std::ios::scientific) << x;
```

- Implementation (internal representation) of `exfloat`

```
std::cout << std::hex << x;
```

- Specification of precision

```
std::cout << std::setprecision(100) << x;
```

4. Four Basic Rules

An `exfloat` type has the four basic rules (`+`, `-`, `*`, `/`) with `exfloat` type, and with built-in integers (`int`, `long`).

```
exfloat x, y, z;
int i;

z = x + y;                          // valid
z = x / y;                          // valid

z = x + i;                          // valid
z = x * i;                          // valid

z = (x + y) * i;                    // valid
```

The four rules with built-in floating-point types (`float`, `double`) is not supported. If you need, first you convert the bulit-in type to a temporary `exfloat` type with an explicit type conversion, then you can operate with the temporary `exfloat` type instead of the built-in types.

```
exfloat x, y, z;
double f;

z = x + f;                          // invalid

y = static_cast<exfloat>(f);        // valid
z = x + y;                          // valid

z = x * 1.2;                        // invalid
z = (x * 12) / 10;                  // valid
```

5. Comparison

Comparison between an `exfloat` type with an `exfloat` type, a built-in floating-point type, or an integer are defined.

```
exfloat x, y;
double f;
int i;

if ( x == y )                       // valid
  ....;

if ( x != 1 )                       // valid

if ( x < f )                        // valid

if ( x <= 1e-3 )                    // valid

if ( x > y + 1 )                    // valid

if ( x > y + i )                    // valid

if ( x >= y + 1e-3 )                // invalid
```

6. Auxiliary Utility Functions and Member of the Class `exfloat`

1. library version
   `const char* exflib_version()`
   - Return the date and time when the `libexfloat.a` was compiled.
2. precision of multiple-precision type
   `double exfloat::precision`
   - Return precision in digits actually used in computation.

   `double exfloat::precision10`
   - Return the user request precision in digits.

   `long exfloat::byte`
   - Return the memory quantity in byte a `exfloat` spends. The sum of sign, exponent and fractional part.

   `long exfloat::size`
   - Return the size of elements in the internal array `num` which holds the multiple-precision floating-point value.
3. rounding control
   `void *exfloat::round`
   - Set to `exflib::near` if ronding to the nearest (`exfloat::round = exflib::near;`), to `exflib::plus` if rounding towards $+\infty$, to `exflib::minus` if rounding towards $-\infty$, and to `exflib::chop` if chopping mode. Default is `exflib::near`. You can change the mode dynamically in the program.

   `void setround(long mode)`
   - Changing the rounding mode. Rounding to the nearest is `setround(0)`, rounding towards $+\infty$ is `setround(1)`, rounding towards $-\infty$ is `setround(-1)`. You can not set chopping mode. Same as setting the value `exfloat::round`.
   - Exflib does not have the feature to display the current rounding mode (feature work).
4. constructor
   ```
   exfloat::exfloat()
   exfloat::exfloat(const char *op)
   exfloat::exfloat(long op)
   exfloat::exfloat(int op)
   exfloat::explicit exfloat(const double op)
   ```
5. substitution
   ```
   exfloat exfloat::operator=(long op2)
   exfloat exfloat::operator=(int op2)
   exfloat exfloat::operator=(const char *op2)
   ```
   - Default is bit copy.

   `exfloat exfloat::`*assign-op*`(const exfloat& op2)`
   `exfloat exfloat::`*assign-op*`(long op2)`
   `exfloat exfloat::`*assign-op*`(int op2)`
   - *assign-op* is one of the following operators:
     - `operator+=`
     - `operator-=`

  - · `operator*=`
  - · `operator/=`

6. type conversion (cast)

   ```
   operator double() const
   ```

7. operator

   ```
   exfloat exfloat::operator+() const
   exfloat exfloat::operator-() const
   ```

   - unaray operator.

   ```
   exfloat exfloat::binary-op(const exfloat& op2) const
   exfloat exfloat::binary-op(long op2) const
   exfloat exfloat::binary-op(int op2) const
   exfloat binary-op(long op, const exfloat& op2)
   exfloat binary-op(int op, const exfloat& op2)
   ```

   - *binary-op* is one of the following operators:
     - · `operator+`
     - · `operator-`
     - · `operator/`
     - · `operator*`

   ```
   bool exfloat::comparison(const exfloat& op2) const
   bool exfloat::comparison(double op2) const
   bool exfloat::comparison(long op2) const
   bool exfloat::comparison(int op2) const
   bool comparison(double op, const exfloat& op2)
   bool comparison(long op, const exfloat& op2)
   bool comparison(int op, const exfloat& op2)
   ```

   - *comparison* is one of the following operators:
     - · `operator==`
     - · `operator!=`
     - · `operator<`
     - · `operator<=`
     - · `operator>`
     - · `operator>=`

8. Output

   ```
   std::ostream& operator<<(std::ostream& strm, const exfloat& op)
   ```

   - Return `strm`.

Example of class members and auxiliary functions.

```
cout << "Required precision: " << exfloat::precision10 << endl;
cout << "Computation precision: " << exfloat::precision << endl;
cout << "An exfloat number consists of " << exfloat::size
     << " unsigned long long elements." << endl;
cout << "Entity is x.num[" << exfloat::size << ']' << endl;
cout << "The size of exfloat is " << exfloat::byte << " byte" << endl;
cout << "Current libexfloat.a is " << exflib_version() << endl;
```

The example returns the following, for example:

```
Required precision: 100
Computation precision: 115.596
An exfloat number consists of 7 unsigned long long elements.
Entity is x.num[7]
The size of exfloat is 56 byte
Current libexfloat.a is compiled at Apr 17 2006 13:40:52
```

7. Built-in Mathematical Functions

`exfloat abs(const exfloat& x)`	absolute value $\|x\|$
`exfloat fabs(const exfloat& x)`	absolute value $\|x\|$
`exfloat sqrt(const exfloat& x)`	$\sqrt{x}, x \geq 0$
`exfloat sin(const exfloat& x)`	sine of x
`exfloat cos(const exfloat& x)`	cosine of x
`exfloat tan(const exfloat& x)`	tangent of x
`exfloat exp(const exfloat& x)`	exponential function e^x

```
exfloat pow(const exfloat& x, long y)
exfloat pow(const exfloat& x, const exfloat& y)
exfloat pow(long x, const exfloat& y)
```

x^y, A domain error occurs if $x = 0$ and $y \leq 0$, or $x < 0$ and y is not an integer.

`exfloat log(const exfloat& x)`	natural logarithm $\ln(x), x > 0$
`exfloat log10(const exfloat& x)`	base 10 logarithm $\log_{10}(x), x > 0$
`exfloat asin(const exfloat& x)`	$\sin^{-1}(x)$ in range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$
`exfloat acos(const exfloat& x)`	$\cos^{-1}(x)$ in range $[0, \pi]$, $x \in [-1, 1]$
`exfloat atan(const exfloat& x)`	$\tan^{-1}(x)$ in range $[-\pi/2, \pi/2]$
`exfloat sinh(const exfloat& x)`	hyperbolic sine of x
`exfloat cosh(const exfloat& x)`	hyperbolic cosine of x
`exfloat tanh(const exfloat& x)`	hyperbolic tangent of x
`exfloat ceil(const exfloat& x)`	smallest integer not less than x
`exfloat floor(const exfloat& x)`	largest integer not greater than x

```
exfloat modf(const exfloat& x, exfloat *ip)
```

splits x into integral and fractional parts, each with the same sign as x. It stores the integral part in `*ip`, and returns the fractional part.

```
exfloat fmod(const exfloat& x, const exfloat& y)
```

floating-point remainder of x/y, with the same sign as x. A domain error occurs if y is zero.

(`atan2`, `ldexp`, degree trigonometric functions like `sind` are not implemented; future work)

8. Multiple-Precision Programming with exflib – How to Convert your Program

1. Change the declaration of real valued variables (`float` and `double`).

   (Original)

```
float x;                          // single-precision floating-point type
double x;                         // double-precision floating-point type
```

   (Revised)

```
exfloat x;                       // multiple-precision floating-point type
```

2. Add double quotations to all literal constants in substitution to `exfloat` variables.

```
x = 0.1;       →      x = "0.1";
```

3. Changing all statements which contain arithmetic or built-in functions of literal floating-point constants.

   (Original)

```
z = w + 3.4;
y = x*1.2;
```

   (Revised)

```
exfloat tmp;
tmp = "3.4";
z = w + tmp;

y = x * 12 / 10;
```

If you do not to output an `exfloat` value with high-accuracy, the following changes enable fast output.

(Original)

```
std::cout << x;
```

(Revised)

```
std::cout << static_cast<double>(x);
```

- Most templete libraries work with exflib.

  For example, multiple-precision complex type is available with the C++ standard `complex` class. JAMA/C++ and TNT (template numerical tookit) works under exflib with some changes in literarl `float` and `double` substitutions.

- A library which strongly depends on floating-point types, like LAPACK, does not work with exflib.

- OpenMP works under exflib with limitations in the use of REDUCTION directives.

- MPI works under exflib, with some changes of floating-point data transfer sentences.

- The entity of `exfloat` multiple-precision type is an array of unsigned 8-byte long integers (64-bit width integer, `unsigned long long` or `unsigned __int64`). The name of the array is `num`, the number of elements is `exfloat::size`, and the index of the array starts from 0.

  More concretely, the entity of an `exfloat` multiple-precision type `x` is

  ```
  unsigned long long x.num[exfloat::size]
  ```

  In parallel computations you send and receive the array to exchange `exfloat` type values.

- The following identifiers are reserved for use as keywords.

  ```
  PRECISION exfloat exflib exflib_* EXFLIB_*
  ```

9. Remarks and Limitations

- You must specify precision more than or equal to 40 digits. Some computations fail because of implementation when you requirement precision under 40 digits.

- You can not use the built-in floating-point types and `exfloat` in one arithmetic sentence togeter. You need an explicit type conversion.

- Multiple-precision interval arithmetic is now under developped.

- Rounding to the nearest in the built-in mathematical functions are sometimes not exact.

- Precision of `exfloat` must be defined at a compiling time. You can not change precision in user programs. (future work)

- Actual computation precision is differ from user request precision. Actual precision is higher than request precision. The member `exfloat::precision` has actual precision, and the member `exfloat::precision10` has request precision in digits. You do not care the difference in usual.

- For example, actual computation precision is same for requests `#define PRECISION 100` and `#define PRECISION 110`. This means that the member `exfloat::precision` is same for these two descriptions.

  The next table shows user request precision and actual computation precision.

User Request Precision `exfloat::precision10`	Actual Computation Precision `exfloat::precision`
$39 \sim 57$	57.80
$58 \sim 77$	77.06
$78 \sim 96$	96.33
$97 \sim 115$	115.60
$116 \sim 134$	134.87
$135 \sim 154$	154.13
$155 \sim 173$	173.39
$174 \sim 192$	192.66
$\vdots$	$\vdots$

  A step of actual computation precision is about 19.3, more precisely, it is $64 \times \log_{10} 2 \approx 19.2659$. The step comes from the reason that `exflib` holds a multiple-precision value as an array of `unsigned long long` types (radix-2^{64} integer), and $2^{64} \approx 10^{19.3}$.

- Limiting to four basic rules, there are no limitations of computation precision in used algorithms.

- Precision of built-in constants $(\pi, \log_{10} 2)$ is about $19,700$ digits. If you need more precision, the library `libexfloat.a` must be compiled again.

  This means that the upper limit precision of built-in functions and decimal output is about $19,700$ digits. This corresponds to 1023 elements in an array of `unsigned long long` integers. ($2^{64\times 1023} \approx 10^{19709.035}$)

10. Copyright and Disclaimer