
QUAD-DOUBLE/DOUBLE-DOUBLE COMPUTATION PACKAGE

Revision date: 2010 June 14

Authors:

Yozo Hida	U.C. Berkeley	yozo@cs.berkeley.edu
Xiaoye S. Li	Lawrence Berkeley Natl Lab	xiaoye@nersc.gov
David H. Bailey	Lawrence Berkeley Natl Lab	dhbailey@lbl.gov

C++ usage guide:

Alex Kaiser	Lawrence Berkeley Natl Lab	adkaiser@lbl.gov

This work was supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231.

This work was supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract numbers DE-AC03-76SF00098 and DE-AC02-05CH11231.

*** IMPORTANT NOTES:

See the file COPYING for modified BSD license information.
See the file INSTALL for installation instructions.
See the file NEWS for recent revisions.

Outline:

I. Introduction

This package provides numeric types of twice the precision of IEEE double (106 mantissa bits, or approximately 32 decimal digits) and four times the precision of IEEE double (212 mantissa bits, or approximately 64 decimal digits). Due to features such as operator and function overloading, these facilities can be utilized with only minor modifications to conventional C++ and Fortran-90 programs.

In addition to the basic arithmetic operations (add, subtract, multiply, divide, square root), common transcendental functions such as the exponential, logarithm, trigonometric and hyperbolic functions are also included. A detailed description of the algorithms used is available in the docs subdirectory (see docs/qd.ps). An abridged version of this paper, which was presented at the ARITH-15 conference, is also available in this same directory (see docs/arith15.ps).

II. Directories and Files

There are six directories and several files in the main directory of this distribution, described below

src	This contains the source code of the quad-double and double-double library. This source code does not include inline functions, which are found in the header files in the include directory.
include	This directory contains the header files.
fortran	This directory contains Fortran-90 files.
tests	This directory contains some simple (not comprehensive) tests.
docs	This directory contains two papers describing the algorithms.
config	This directory contains various scripts used by the configure script and the Makefile.

C++ Usage:

Please note that all commands refer to a Unix-type environment such as Mac OSX or Ubuntu Linux using the bash shell.

A. Building

To build the library, first run the included configure script by typing

```
./configure
```

This script automatically generates makefiles for building the library and selects compilers and necessary flags and libraries to include. If the user wishes to specify compilers or flags they may use the following options.

CXX	C++ compiler to use
CXXFLAGS	C++ compiler flags to use
CC	C compiler to use (for C demo program)
CFLAGS	C compiler flags to use (for C demo program)
FC	Fortran 90 compiler
FCFLAGS	Fortran 90 compiler flags to use
FCLIBS	Fortran 90 libraries needed to link with C++ code.

For example, if one is using GNU compilers, configure with:

```
./configure CXX=g++ FC=gfortran
```

The Fortran and C++ compilers must produce compatible binaries. On some systems additional flags must be included to ensure that portions of the library are not built with 32 and 64 bit object files. For example, on 64-Bit Mac OSX 10.6 (Snow Leopard) the correct configure line using GNU compilers is:

```
./configure CXX=g++ FC=gfortran FCFLAGS=-m64
```

To build the library, simply type

```
make
```

and the automatically generated makefiles will build the library including archive files.

To allow for easy linking to the library, the user may also wish to install the archive files to a standard place. To do this type:

```
make install
```

This will also build the library if it has not already been built. Many systems, including Mac and Ubuntu Linux systems, require administrator privileges to install the library at such standard places. On such systems, one may type:

```
sudo make install
```

instead if one has sufficient access.

The directory 'tests' contains programs for high precision quadrature and integer-relation detection. To build such programs, type:

```
make demo
```

in the 'tests' directory.

B. Linking

The simplest way to link to the library is to install it to a standard place as described above, and use the –l option. For example

```
g++ compileExample.cpp -o compileExample -l qd
```

One can also use this method to build with make. A file called "compileExample.cpp" and the associated makefile "makeCompileExample" illustrate the process.

A third alternative is to use a link script. If one types "make demo" in the test directory, the output produced gives guidance as to how to build the files. By following the structure of the compiling commands one may copy the appropriate portions, perhaps replacing the filename with an argument that the user can include at link time. An example of such a script is as follows:

```
g++ -DHAVE_CONFIG_H   -I.. -I../include -I../include   -O2  -MT $1.o -MD -MP -MF
.deps/qd_test.Tpo -c -o $1.o $1.cpp
mv -f .deps/$1.Tpo .deps/$1.Po
g++  -O2    -o $1 $1.o ../src/libqd.a –lm
```

To use it, make the link script executable and type:

```
./link.scr compileExample
```

Note that the file extension is not included because the script handles all extensions, expecting the source file to have the extension '.cpp' .

C. Programming techniques

As much as possible, operator overloading is included to make basic programming as much like using standard typed floating-point arithmetic. Changing many codes should be as simple as changing type statements and a few other lines.

i. Constructors

To create dd_real and qd_real variables calculated to the proper precision, one must use care to use the included constructors properly. Many computations in which variables are not explicitly typed to multiple-precision may be evaluated with double-precision arithmetic. The user must take care to ensure that this does not cause errors. In particular, an expression such as 1.0/3.0 will be evaluated to double precision before assignment or further arithmetic. Upon assignment to a multi-precision variable, the value will be zero padded. This problem is serious and potentially difficult to debug. To avoid this, use the included constructors to force arithmetic to be performed in the full precision requested. Here is a list of the included constructors with brief descriptions:

Type dd_real, with text of inline constructors included:

Constructor	Description
dd_real(double hi, double lo) { x[0] = hi; x[1] = lo; }	Initializes from two double precision values.
dd_real() {x[0] = 0.0; x[1] = 0.0; }	Default constructor initializes to zero.
dd_real(double h) { x[0] = h; x[1] = 0.0; }	Initializes from a double precision value, setting the trailing part to zero. Use care to ensure that the trailing part should actually be set to zero.
dd_real(int h) {	Initializes from an integer value, setting the

x[0] = (static_cast<double>(h));
x[1] = 0.0;
}
	trailing part to zero. Use care to ensure that the trailing part should actually be set to zero.

dd_real (const char *s);	Initializes from a string.

explicit dd_real (const double *d) {
x[0] = d[0]; x[1] = d[1];
}
	Initializes from a length two array of double precision values.

Type qd_real, with their functions included inline:

Constructor	Description

inline qd_real::qd_real
(double x0, double x1, double x2, double x3)
{
x[0] = x0;
x[1] = x1;
x[2] = x2;
x[3] = x3;
}
	Initializes from four double precision values.

inline qd_real::qd_real(const double *xx) {
x[0] = xx[0];
x[1] = xx[1];
x[2] = xx[2];
x[3] = xx[3];
}
	Initializes from a length four array of double precision values.

inline qd_real::qd_real(double x0) {
x[0] = x0;
x[1] = x[2] = x[3] = 0.0;
}
	Initializes from a double precision value, setting the trailing part to zero. Use care to ensure that the trailing part should actually be set to zero.

inline qd_real::qd_real() {
x[0] = 0.0;
x[1] = 0.0;
x[2] = 0.0;
x[3] = 0.0;
}
	Default constructor initializes to zero.

inline qd_real::qd_real(const dd_real &a) {
x[0] = a._hi();
x[1] = a._lo();
x[2] = x[3] = 0.0;
}
	Initializes from a double-double value, setting the trailing part to zero.

inline qd_real::qd_real(int i) {	Initializes from an integer value, setting the

x[0] = static_cast<double>(i);
x[1] = x[2] = x[3] = 0.0;
}
	trailing part to zero. Use care to ensure that the trailing part should actually be set to zero.

Some examples of initialization are as follows

```
qd_real x = "1.0" ;
x /= 3.0 ;
```

or

```
qd_real x = qd_real(1.0) / 3.0 ;
```

ii. Included functions and Constants

Supported functions include assignment operators, comparisons, arithmetic and assignment operators, and increments for integer types. Standard C math functions such as exponentiation, trigonometric, logarithmic, hyperbolic, exponential and rounding functions are included. As in assignment statements, one must be careful with implied typing of constants when using these functions. Many codes need particular conversion for the power function, which is frequently used with constants that must be explicitly typed for multi-precision codes.

Many constants are included, which are global and calculated upon initialization. The following list of constants is calculated for both the dd_real and qd_real classes separately. Use care to select the correct value. The variables, with type signatures, are:

Variable Name	Explanation
static const qd_real _2pi;	Two pi.
static const qd_real _pi;	Pi.
static const qd_real _3pi4;	Three pi over four.
static const qd_real _pi2;	Pi over two.
static const qd_real _pi4;	Pi over four
static const qd_real _e;	e, the base of the natural logarithm.
static const qd_real _log2;	Natural logarithm of two.
static const qd_real _log10;	Natural logarithm of ten.
static const qd_real _nan;	Not a number. Behaves like a double-precision nan.
static const qd_real _inf;	Infinity. Behaves like a double-precision inf.
static const double _eps;	Estimated precision for dd_real or qd_real data type.
static const double _min_normalized;	Minimum absolute value represent able without denormalization.
static const qd_real _max;	Maximum representable value.
static const qd_real _safe_max;	Maximum safe value. Slightly smaller than maximum representable value.
static const int _ndigits;	Number of digits available for dd_real or

	qd_real datatypes.

ii. Conversion of types

Static casts may be used to convert constants between types. One may also use constructors to return temporary multi-precision types within expressions, but should be careful, as this will waste memory if done repeatedly. For example:

```
qd_real y ;
y = sin( qd_real(4.0) / 3.0 ) ;
```

C–style casts may be used, but are not recommended. Dynamic and reinterpret casts are not supported and should be considered unreliable. Casting between multi-precision and standard precision types can be dangerous, and care must be taken to ensure that programs are working properly and accuracy has not degraded by use of a misplaced type-conversion.

D. Available precision, Control of Precision Levels,

The library provides greatly extended accuracy when compared to standard double precision. The type dd_real provides for 106 mantissa bits, or about 32 decimal digits. The type qd_real provides for 212 mantissa bits, or about 64 decimal digits.

Both the dd_real and qd_real values use the exponent from the highest double-precision word for arithmetic, and as such do not extend the total range of values available. That means that the maximum absolute value for either data type is the same as that of double-precision, or approximately 10^308. The precision near this range, however, is greatly increased.

To ensure that arithmetic is carried out with proper precision and accuracy, one must call the function “fpu_fix_start” before performing any double-double or quad-double arithmetic. This forces all arithmetic to be carried out in 64-bit double precision, not the 80-bit precision that is found on certain compilers and interferes with the existing library.

```
unsigned int old_cw;
fpu_fix_start(&old_cw);
```

To return standard settings for arithmetic on one’s system, call the function “fpu_fix_end”. For example:

```
fpu_fix_end(&old_cw);
```

E. I/O

The standard I/O stream routines have been overloaded to be fully compatible with all included data types. One may need to manually reset the precision of the stream to obtain full output. For example, if 60 digits are desired, use:

```
cout.precision(60) ;
```

When reading values using cin, each input numerical value must start on a separate line. Two formats are acceptable:

1. Write the full constant
3. Mantissa e exponent

Here are three valid examples:

```
1.1
3.14159 26535 89793
123.123123e50
```

When read using cin, these constants will be converted using full multi-precision accuracy.

IV. Fortran-90 Usage

NEW (2007-01-10): The Fortran translation modules now support the complex datatypes "dd_complex" and "qd_complex".

Since the quad-double library is written in C++, it must be linked in with a C++ compiler (so that C++ specific things such as static initializations are correctly handled). Thus the main program must be written in C/C++ and call the Fortran 90 subroutine. The Fortran 90 subroutine should be called f_main.

Here is a sample Fortran-90 program, equivalent to the above C++ program:

```
subroutine f_main
    use qdmodule
    implicit none
    type (qd_real) a, b
    integer*4 old_cw

    call f_fpu_fix_start(old_cw)
    a = 1.d0
    b = cos(a)**2 + sin(a)**2 - 1.d0
    call qdwrite(6, b)
    stop
end subroutine
```

This verifies that cos^2(1) + sin^2(1) = 1 to 64 digit accuracy.

Most operators and generic function references, including many mixed-mode type combinations with double-precision (ie real*8), have been overloaded (extended) to work with double-double and quad-double data. It is important, however, that users keep in mind the fact that expressions are evaluated strictly according to conventional Fortran operator precedence rules. Thus some subexpressions may be evaluated only to 15-digit accuracy. For example, with the code

```
  real*8 d1
  type (dd_real) t1, t2
  ...
  t1 = cos (t2) + d1/3.d0
```

the expression d1/3.d0 is computed to real*8 accuracy only (about 15 digits), since both d1 and 3.d0 have type real*8. This result is then converted to dd_real by zero extension before being added to cos(t2). So, for example, if d1 held the value 1.d0, then the quotient d1/3.d0 would only be accurate to 15 digits. If a fully accurate double-double quotient is required, this should be written:

```
 real*8 d1
 type (dd_real) t1, t2
  ...
 t1 = cos (t2) + ddreal (d1) / 3.d0
```

which forces all operations to be performed with double-double arithmetic.

Along this line, a constant such as 1.1 appearing in an expression is evaluated only to real*4 accuracy, and a constant such as 1.1d0 is evaluated only to real*8 accuracy (this is according to standard Fortran conventions). If full quad-double accuracy is required, for instance, one should write

```
  type (qd_real) t1
  ...
  t1 = '1.1'
```

The quotes enclosing 1.1 specify to the compiler that the constant is to be converted to binary using quad-double arithmetic, before assignment to t1. Quoted constants may only appear in assignment statements such as this.

To link a Fortran-90 program with the C++ qd library, it is recommended to link with the C++ compiler used to generate the library. The Fortran 90 interface (along with a C-style main function calling f_main) is found in qdmod library. The qd-config script installed during "make install" can be used to determine which flags to pass to compile and link your programs:

```
 "qd-config --fcflags"  displays compiler flags needed to compile your Fortran files.
 "qd-config --fclibs"   displays linker flags needed by the C++ linker to link in all the
                        necessary libraries.
```

A sample Makefile that can be used as a template for compiling Fortran programs using quad-double library is found in fortran/Makefile.sample.

F90 functions defined with dd_real arguments:
 Arithmetic: + - * / **
 Comparison tests: == < > <= >= /=
 Others: abs, acos, aint, anint, asin, atan, atan2, cos, cosh, dble, erf,
 erfc, exp, int, log, log10, max, min, mod, ddcsshf (cosh and sinh),

ddcssnf (cos and sin), ddranf (random number generator in (0,1)),
ddnrtf (n-th root), sign, sin, sinh, sqr, sqrt, tan, tanh
Similar functions are provided for qd_real arguments (with function names qdcsshf, qdcssnf, qdranf and qdnrtf instead of the names in the list above).

Input and output of double-double and quad-double data is done using the special subroutines ddread, ddwrite, qdread and qdwrite. The first argument of these subroutines is the Fortran I/O unit number, while additional arguments (as many as needed, up to 9 arguments) are scalar variables or array elements of the appropriate type. Example:

```
integer n
type (qd_real) qda, qdb, qdc(n)
...
call qdwrite (6, qda, qdb)
do j = 1, n
        call qdwrite (6, qdc(j))
enddo
```

Each input values must be on a separate line, and may include D or E exponents. Double-double and quad-double constants may also be specified in assignment statements by enclosing them in quotes, as in

```
 ...
 type (qd_real) pi
 ...
 pi =
"3.14159265358979323846264338327950288419716939937510582097494459230"
 ...
```

Sample Fortran-90 programs illustrating some of these features are provided in the f90 subdirectory.

V. Note on x86-Based Processors (MOST systems in use today)

The algorithms in this library assume IEEE double precision floating point arithmetic. Since Intel x86 processors have extended (80-bit) floating point registers, the round-to-double flag must be enabled in the control word of the FPU for this library to function properly under x86 processors. The following functions contains appropriate code to facilitate manipulation of this flag. For non-x86 systems these functions do nothing (but still exist).

fpu_fix_start	This turns on the round-to-double bit in the control word.
fpu_fix_end	This restores the control flag.

These functions must be called by the main program, as follows:

```
int main() {
  unsigned int old_cw;
  fpu_fix_start(&old_cw);
  ... user code using quad-double library ...

  fpu_fix_end(&old_cw);
}
```

A Fortran-90 example is the following:

```
subroutine f_main
use qdmodule
implicit none
integer*4 old_cw

call f_fpu_fix_start(old_cw)

  ... user code using quad-double library ...

call f_fpu_fix_end(old_cw)
end subroutine
```