Quad Double Arithmetic in Lisp

Richard Fateman
Computer Science
University of California
Berkeley, CA, USA

October 17, 2007

Abstract

In a numerical calculation sometimes we need higher-than double-precision floating-point arithmetic to allow us to be confident of a result. One alternative is to rewrite the program to use a software package implementing *arbitrary-precision* extended floating-point arithmetic such as ARPREC or MPFR, and try to choose a suitable precision.

Such an arithmetic scheme, in spite of helpful tools, may be inconvenient to write. There are also facilities in computer algebra systems (CAS) for such software-implemented "bigfloats." These facilities are convenient if one is already using the CAS. In any of these situations the bigfloats may be rather slow, a cost of its generality.

There are possibilities intermediate between the largest hardware floating-point format and the general arbitrary-precision software which combine a considerable (but not arbitrary) amount of extra precision with a (relatively speaking) modest factor loss in speed. Sometimes merely doubling the number of bits in a double-floating-point fraction is enough, in which case arithmetic on double-double (DD) operands would suffice. Another possibility is to go for yet another doubling to quad-double (QD) arithmetic: instead of using the machine double-floats to give about 16 decimal digits of precision, QD supplies about 64 digits. DD and QD as used here provide the same exponent range as ordinary double.

Here we describe how we incorporated QD arithmetic implemented in a library into a Common Lisp system, providing a smooth interface while adding only modest overhead to the run-time costs (compared to accessing the library from C or C++). One advantage is that we keep the program text almost untouched while switching from double to quad-double. Another is that the programs can be written, debugged, and run in an interactive environment. Most of the lessons from QD can be used for other versions of arithmetic which can be embedded in Lisp, including MPFR, for indefinite (arbitrary) precision, should QD provide inadequate precision or range.

1 Introduction

Quad-double (QD) arithmetic is described in Hida, Li, Bailey [1], and programs implementing QD are available free. Although one can think about QD numbers as indivisible 256-bit quantities storing one number, the storage mechanism is that of four double-float numbers in an array. Each of the numbers contains its own exponent. In principle the number represented could be computed by adding together these numbers into a very wide accumulator.

The documentation [1] assumes that access to the library will be from traditional batch languages such as C++, with wrappers supplied for C and Fortran 90. While these languages may represent the most common numerical environments, this choice inhibits experimentation: programs must be carefully composed and compiled before any numbers can be produced. In our experience, an interactive shell for prototyping computations can be much more productive in the programming and debugging stage.

Lisp is of course not the only interactive computation system; in fact others that are more widely known in the mathematical computation community include computer algebra systems like Maple, Mathematica, Macsyma, Axiom, or numerical systems like Matlab, Octave, MathCad.

Lisp has the advantage of having a rather simple programming model, and it has a number of packages available for which it can be used as a scripting level. One feature of most Lisp implementations is that an optimizing compiler is always available, and so one can define and test even a one-line function; debug it interactively with full "symbolic" debugging tools, and then by using a one-line command, compile it and make it run much faster. In this way a QD arithmetic module may be easily transferred from interactive to compiled (even batch) production mode. In Lisp a compiled module can still be accessed interactively and will cooperate with most debugging tools. In some sense the highly-touted Java notion of JIT or "just in time" compiling has been supported in Lisp for several decades.

2 If you write in Lisp, here's what you do

These instructions pertain specifically to Allegro Common Lisp running on Windows/Intel computers. Other implementations should have similar setups.

If you are not especially concerned about run-time speed, but want the extra precision, or perhaps are just testing an algorithm, obtain our program directory and try this:

```
(load "packs")   ;; load the file declaring packages
(load "ga.fasl") ;; load generic arithmetic
(load "qd.fasl") ;; load the quad-double interface
(load "qd.dll")  ;; load the quad-double [C coded] library
(load "my-own-system")
(in-package :ms)
```

Here we assume that the package for `my-own-system` is called `:ms`. A template, which can be copied into your own file, for doing this is shown below. The file `my-own-system.lisp` is this:

```
(defpackage :ms
  (:use :ga :cl)
    (:shadowing-import-from  :ga
   "+" "-" "/" "*"  = > <
   sin cos tan atan asin acos sinh cosh tanh asinh acosh atanh
   expt log exp sqrt  1- 1+ abs ))

(in-package :ms)
(eval-when (compile) (load "qd"))

;; Your programs go here. For example, a naive factorial program

(defun fact(x)(if (= x 0) 1 (* x (fact (1- x)))))
```

This factorial program can be used for ordinary Lisp numbers or qd numbers. It looks like this:

```
 ms(196): (fact 50) ;; ordinary integer 50
30414093201713378043612608166064768844377641568960512000000000000
 ms(197): (fact (qd:into 50))  ;; qd input: in answer, an exponent follows the Q
0.30414093201713378043612608166064768844377641568960512Q65
```

Note: another way to input a qd number 50 is to write `#q50q0`.

If you want extra speed, you can do this:

```
ms(197): (compile 'fact)
```

If you want even more speed, it may pay to put the function definition in a file with the package (etc) header and compile the whole file. That way you can be sure that the environment of the compilation process is correct. Note that it is entirely possible to run previously-compiled programs in a Lisp system that knows nothing about the equivalent in-line expansions of code; once the in-line expansion macros are loaded, further compilations will use them. For programs with loops or recursive calls, using `with-temps` or `dsetv`, discussed below, may be helpful in getting the best efficiency. The effect of using these functions on expressions computing QD numbers is that run-time type-testing and most of the storage re-allocation is bypased.

What is bypassed? Ordinarily some checking is needed to discriminate among the various types that can be fed at run-time into a function like "+". If we know that all arguments to certain numeric functions are QD numbers, all functions used in the enclosed expressions pass their arguments directly to the c-coded library. If such functions are mistakenly passed values that are not QD numbers, or cannot be converted to QD numbers *at compile time*, then the results will probably be some uninformative error message such as "Segmentation violation".

3 How it is done

We define a "package" called "ga" for generic arithmetic. For a program that runs "using" the ga package, each of the function operators like +, *, sin, cos, etc. is overloaded so that if any of their operands is a QD number, the operation is done in QD arithmetic.

The simplest way of producing a QD number from an ordinary Lisp number is `(qd:into X)`. The original X can be a single- or double-float, an integer of any number of digits (yes, Lisp allows this), or a ratio of two arbitrary-precision integers. QD constants can be denoted syntactically by prefixing them with `#q` as for example `#q3.1415926q0`.

Lisp normally uses a prefix notation so that addition would look like `(+ 1/4 1/4)` to evaluate to `1/2`. In QD, `(+ (qd:into 1/4) 1/4)` returns `0.5Q0`. In this last expression we show both the use of the `qd:into` program to convert an ordinary lisp number to a QD number, as well as the result showing the notation for a QD answer: a string of characters with the exponent following a `Q`. Naturally it is possible to extract the exponent and fraction part of a result using the same programs used by the `print` routine, in the program listings.

Anticipating that readers unfamiliar with Lisp will object to the prefix notation, let us point out that there are numerous parsers written in Lisp that will allow infix notation, e.g. `a+4*c` or `1/4+into(1/4)` for such expressions. We supply a parser with this collection of generic arithmetic systems. Nevertheless we will continue to use prefix notation for clarity and consistency with ordinary Lisp.

A complete listing of the QD functionality and further examples are available in the directory with the files.

Other overloadings for Lisp, for example MPFR, the (arbitrary-length) floating-point with rounding package associated with the GMP package, can be similarly accessed by `(mpfr:into X)` and are also documented further in that directory.

4 For Lisp fans

A data structure and type for a QD number, which we call `aqd` is defined and QD methods are essentially refactored into the "normal" arithmetic associated with numbers native to Lisp. Then the arithmetic associated with new operations on QD or mixed types are overlaid. These are actually built upon programs

with names like `two-arg-+` out of which an n-ary "+" is built. For unary functions like sin or cos, the situation is even simpler. The similarities among the Lisp-to-C interfaces for `sin, asin, sinh, asinh` are exploited by repeating a macroexpansion of the piece of code, only changing the name "sin" to "asin" etc. via a parameter.

The Lisp n-ary functions like "+" are redefined as macros which expand forms like `(+ a b c)` into `(two-arg-+ a (two-arg-+ b c))` so as to allow the compiler to compile specific versions of `two-arg-+` depending on the declared or deduced types of `a, b,` or `c`. They need not all be QD numbers. If they are all QD numbers, and the computation is done inside a loop, it should run faster by using `with-temps` mentioned below.

Printing of QD numbers is implemented by conversion of each of the component double-precision numbers to exact Lisp (arbitrary-precision) rational numbers, exactly adding them together, and then taking the integer part after multiplication by a power (possibly negative) of the print radix. (The default radix is 10). This is vastly simpler than having to compute in a language that does not have arbitrary-precision integers.

Many implementations of Common Lisp use a compacting garbage collector (GC). This means that data structures, once allocated, may be moved in memory at some later time. This has the unwanted side effect of making pointers into those objects (from C programs) invalid, if it is possible for the GC to be run during one of the QD programs. This would happen perhaps if the Lisp were allowed to run another multiprocessing thread during execution of a C program, and that other thread provoked a GC. While this cannot happen in the current Lisp implementation, it seems prudent to be aware of this possibility, especially as multiple processor systems become less expensive. The current Allegro implementation allows arrays to be allocated in memory such that they are not relocated with a GC, and so as a precaution we have specified that QD arrays are allocated in this "lisp-static" space.

5 Functional vs. State-Transition

It may not be obvious at first glance, but there is a conceptual mismatch between the programming model supported by the QD library and the model used in Lisp and other functional-style languages. This mismatch can be overcome, but first let us explain the issue.

If we chose a uniform functional notation for arithmetic, we would say `add(A,B)`; If we used an "infix" notation, then $A + B$. In Lisp, we would use parenthesized prefix notation `(+ A B)`. In all these cases the expression means "return the memory location of a *new* qd-containing array M containing the sum of A and B." The numbers A and B are unchanged.

This notation extends so that to compute $A + BC$ write `(+ A (* B C))`

By contrast, the state-transition version uses a different set of fundamental operations. We would say `add(A,B,M)`. This means "destroy what was in the target memory M, and overwrite it with the sum of A and B." Thus to compute A+B*C we need to identify (or perhaps newly-allocate) two temporaries, T1 and T2, and compute `mul(B,C,T1)`, `add(A,T1,T2)`; The answer is in T2.

The obvious clumsiness of the latter approach is a trade-off related to the associated freedom from storage issues. In fact, if the computation is done repeatedly in a loop, the same T1 and T2 can be used, and this will use a fixed amount of storage and perhaps save time too. Compared this to the functional version, where new pieces of memory would be allocated: eventually the functional version requires that something be done to reclaim storage used for variables whose values are no longer used.

We can, however, use either paradigm in Lisp by (essentially) changing the requirement "allocate a new array M" to "find a (perhaps previously used, but no longer needed) array, like T1 or T2."

Since Lisp is perfectly able to figure out how many temporaries are needed for this at compile time, and then re-use them as needed, we have programmed `with-temps`, a macro expansion procedure to allow this:
`(define foo (A B C) (qd:with-temps (+ A (* B C))))`
which essentially associates with `foo` two temporaries as needed. One for the product and one for the sum. If `foo` is called twice and the first result is not to be overwritten by the second, it is important to make a

fresh copy this way `(copy (foo x y z))`. Otherwise the return value from `foo`, a pointer to one of those temporary locations, will be aliased between the two calls.

Perhaps more often than not in scientific computing you really DO want to overwrite a location with a new value, as for example, updating an array.

Instead of using the Lisp idiom `(setf (aref G i j) (+ ...))` we implemented a (new) idiom `(qd:dsetv (aref G i j) (+ ...))` which destroys the previous value in $G_{i,j}$ rather than creating a new object and pointing $G_{i,j}$ to it. The name `dsetv` comes from Destructively SET the Value. The `qd` package prefix indicates that all components must be QD numbers for this to work properly: the compiler macro-expansion will enforce that assumption on all arguments of its known functions; all explicit constants encountered will be converted to QDs at compile time, and all variables are assumed to be QDs. The expansion of `with-temps`, which can have any nested arithmetic expressions supported by QD arithmetic as well as `setq` makes extensive use of `dsetv`. As is usual, the Lisp function `macroexpand` can be used to look at the generated code that the compiler actually uses. The programs `dsetv` and `with-temps` can be used together; a good example of this use can be seen in the QD-FFT code. The FFT is computed in-place in the input array, and no additional storage is allocated, even though the program nominally looks like it is using functional programming. While `dsetv` could be elaborated to include more of Lisp, for example, capturing additional control structures, our current examples do not seem to need this. As an example of the code generation, consider the macro-expansion illustrated below.

```
;; macroexpansion of (with-temps (setq (aref x 10) (+ y (sin z))))))

;; We have edited this code, adding comments for clarity.
;; gg is really a gensym.
;; reg1,reg2, reg3 .... are "registers" .. allocated QD numbers
;; at compile time. They can be overwritten.
;; Type declarations have been removed for brevity.
;; Notice direct calls to qd_sin and qd_add.

(let ((gg
       (let ((a1 (aqd-q y))                 ;grab qd value in y
             (a2 (aqd-q (let ((a1 (aqd-q z));grab qd value in z
                              (tt (aqd-q reg3))) ;reg3 workspace
                          (qd_sin a1 tt)    ;put sin(z) in reg3
                          reg3)))
             (tt (aqd-q reg2)))
         (qd_add a1 a2 tt)                  ;put y+sin(x) in reg2
         reg2)))                            ;gg holds val in reg2
  (qd_copy_into (aqd-q gg) (aqd-q (aref x 10))) ;copy gg into x[10]
  gg)
```

6 How fast is this?

The amount of overhead in Common Lisp for the wrapping may depend critically on compilation optimization flags and making sure the compiler knows as much as possible about the arguments to functions. Allegro Common Lisp is able to compile a direct function call from the Lisp invocation `(sin x)` to the C language program `c_qd_sin` in about 3 instructions on an Intel X86 processor. This same implementation requires that the floating-point control word be set at the interface; this is a consequence of decisions related to saving state at process switching, and that QD arithmetic depends on rounding to double, not extended. Other

lisp systems (and non-X86 versions of Allegro) do not need to reset this control word.

One might be concerned about the overhead of type discrimination needed to determine which of the (perhaps many) overloaded methods will be used for the argument types given to such general programs as "+". The typechecking and its attendant overhead can be eliminated by `qd:dsetv` or `qd:with-temps`.

As an example, consider evaluating by Horner's rule a polynomial t with 999 terms, beginning $x^{998} - x^{997} + x^{996} - \cdots$. That is, the coefficients are alternating ± 1. We evaluate this polynomial at 2. The value is about 1.78D300.

We timed it on an Intel 2.5Ghz Pentium 4, using Allegro Common Lisp 7.0. We averaged the times over 10000 trials.

- The time to evaluate exactly as a 300 decimal digit integer is about 1.3ms.
- The time to evaluate approximately as a 64-digit quad-double using Horner's rule written in Lisp, is about 1.6ms.
- The time to evaluate approximately as a 64-digit quad-double written in C++ is about .42 ms.
- The (best) time to evaluate approximately as a 16-digit machine double-float is about 0.017 ms.

The Lisp and C programs look like this:

```
(defun polyeval (qdlist x)  ;;qdlist is a list of qd objects, x is a qd object
      (let ((sum (qd:into 0)))
      (dolist (i qdlist sum)
        (setf sum (with-temps (+ i (* x sum)))))))  ;; 10% slower without "with-temps"

(defun polyevald(dlist x) ;; dlist is a list of doubles, x is a double
  (let ((sum 0.0d0))
    (dolist (i dlist sum)
      (setf sum (+ i (* x sum))))))

;; To speed this up, we need to insert some declares:

(defun polyevald (dlist x)
  (let ((sum 0.0d0))
    (declare (double-float sum x) (optimize (speed 3)(debug 0)))
    (dolist (i dlist sum)
      (declare (double-float i))
      (setf sum (the double-float
                  (cl::+ i (the double-float
                             (cl::* x sum))))))))

/* the C++ program, operating on a ``flattened'' polynomial */

void c_qd_polyevalflat( double *c, int n, const double *x, double *result) {
  double *p = c + 4*n;  /* pointer to current coefficient */
  fpu_fix_start(NULL);  /* set rounding to 'double' not double-extended on x86 */
  qd_real r = qd_real(p);
  qd_real xx = qd_real(x);
    for (int i = n-1; i >= 0; i--)
```

```
    { p -= 4;
      r *= xx;
      r += qd_real(p);
     }
TO_DOUBLE_PTR(r, result); }
```

Observing the final result: If we send the QD numbers flattened out as an array of 3996 double-floats to a C++ program, we remove almost all the overhead attributable to using Lisp. Compare this to the time to execute the same polynomial evaluation making individual Lisp calls to qd-addition and qd-multiplication: There is a factor of 4 slowdown in using Lisp to script the Horner's rule[1].

On the Pentium 4, then, if we compute with QD numbers but they are not needed–we can use double-precision machine floats instead–then we are paying a penalty of about a factor of 25 (writing in C++) or a factor of 94 (writing in Lisp). Of course double-floats should be used when the extra precision is not needed; considering that multiplying a pair of QDs uses 16 double-float machine multiplications, this seems reasonable.

These figures can vary substantially depending on the compilers, both Lisp and C++ involved. We used optimization setting `/O2` for the Microsoft Visual Studio (version 8) 2005 C++ compiler.

A final note on timing of Horner's rule: this algorithm uses an equal number of QD adds and multiplies. Profiling the QD Horner benchmark suggests that the cost for a QD multiply is about 1.7 times the cost of an addition.

Another test, computing the fast Fourier transform suggests a similar ratio (of about 30) between double-float and QD.

We also programmed an alternative version of the FFT, still written in Lisp, but removing all tags–that is, the QD numbers cannot be distinguished in any way from other length-4 double-float arrays. This change, removing type-checking for QD numbers, provides a modest 15-25 percent boost. This version looks essentially like assembly language with lines like `(qd_mul a c t1)`.

In addition to the FFT, two other applications which uses these tools are included in the `qd.lisp file`. These are a simple zero-finder based on Newton's method, and a Gaussian quadrature program.

7 What about other kinds of arithmetic?

If you like the QD version of overloading of Lisp, the same ideas are available for MPFR [2], a library in which computations can be done for much longer fractions and larger exponents. If QD is insufficiently precise or has inadequate range (which, recall, is the same as the double-precision exponent range) then MPFR, built on the GMP library, is an alternative. In fact, the code base described here was, in part, originally written for use with MPFR.

Yet other kinds of arithmetic, including symbolic and polynomial arithmetic, can be supported. For these domains there is much less concern about linkage overhead: the time to multiply two polynomials is usually many times greater than the time needed to determine the types of the inputs and the allocation of the space for the result.

Raymond Toy has used the model of QD implementation and written the programs entirely in Common Lisp. This system, called OCT[2], is therefore available in any conforming Common Lisp. There are several routes to implementation; Toy initially wrote the system for CMU-CL, relying heavily for its efficiency on a compiler that expanded inline function calls. This mechanism is not necessarily implemented in a CL

[1]The timings also depend on the computer: the slowdown factor for a Pentium III between Lisp and C++ using QD arithmetic is only 1.7, not 4. We expect that cache size is an issue, since the polynomial itself is 1000 qd numbers, each of which is 32 bytes. The Pentium III L1 data cache is only 16k bytes, and so can hold only half the polynomial. The Pentium 4 has 512k bytes, so all the data can fit.

[2]downloadable from commonlisp.net

compiler, but it now runs in several other Lisps[3]. We have adopted a few ideas from OCT, in particular the input syntax for quad-double constants. In fact, with careful compilation, the pure Lisp solution in Allegro CL runs at about the same speed (within 10 percent) of the qd.dll. It is, however, key to re-use storage (see `dsetv`) to be competitive with code written in C.

Another possibility in some architectures (and in some lisps) is to use not double, but double-extended as the building blocks for the quad system. The result of using 80 rather than 64 bits per unit can provide perhaps 25 more decimal digits, although the subtleties of the arithmetic may cause difficulties.

8 Pieces left out, limitations

There are several difficulties in the design of Common Lisp's arithmetic support that require delicate treatment. These include parameter-count variations and the transition from real to complex numbers.

Two built-in Common Lisp functions of interest to QD arithmetic, namely `atan` and `log` can take one or two arguments. In the case of `atan`, the two-argument version is mapped to `atan2`, a well-known function. This mapping requires some scuffling around at compile time, noticing whether the call is to `one-arg-atanh` or `two-arg-atanh` and expanding to the right form as necessary. It is also necessary to convert one or the other of the arguments to QD form if it not already in the right form. The `log` function of two args is logarithm with respect to a particular base, the second argument. The processing of `log` is provided with a special recognition of `log10` which is available from the qd library. Otherwise `(two-arg-log a b)` is simply /tt(/ (log a)(log b)).

There are other one- or two-argument functions, namely `floor` and `ceil` we ignore, at least for now. They merge notions of division (by the second argument) and truncation.

A more serious issue is the transition from real to complex, which is not treated consistently in the QD library with Common Lisp. Consider the `sqrt` function, which can return a complex result even if given a real argument. This behavior is required of ANSI Standard Common Lisp. The QD library does not support, at the moment, complex numbers, and a square-root of a negative number returns a "Not A Number" (`NaN`). It is not difficult to check for a negative argument to square-root and as we add complex to our qd library for Lisp, return a mathematically correct value. Unfortunately this kind of result (namely: surprise, the answer is a different type from what you expected!) is problematical in a tightly compiled situation in which types and hence storage formats "matter". If the answer must fit in a "real qd" space, returning `NaN` or signalling an exception is then preferable. Given either a `NaN` or an exception, it is easy to build a (light-weight) wrapper in Lisp around functions that sometimes return complex values, so that in the case of an otherwise unanticipated non-real result, the function can be recomputed as complex. This would be similar to automatically recomputing sqrt(-5) as though it had been `complex_sqrt(-5+0*i)`. A Lisp function may not be inconvenienced by such a return value; a Lisp function using carefully declared type assumptions, as with our `with-temps` macro features, would by-pass these wrappers and consider the unexpected introduction of a complex value as an error[4]. Additionally, the QD number format is rather generous in size, and given that only one component, the first one, is needed to signal a `NaN`, the other three words can be used as an encoding to inform the consumer of the cause of the `NaN`, including perhaps a tag for the program counter, and one or more of the arguments.

Additional efficiencies can be obtained by hand-coding calls to specific functions, for example one can multiply a QD by a double-precision number s with much less work than converting s to QD. These alternatives can be presented through the same CLOS interface, merely requiring more uninspired coding of each possible function interface. Less obviously, multiplying or dividing a QD by a power of 2 can be accomplished

[3]We have written an alternative version of the underlying code that compiles to more efficient code in Allegro CL, and possibly other CL implementations. Allegro CL version 8 does not make use of user-defined inlining unless they are macro expansions.

[4]Note that small errors, perhaps caused by roundings, can change the argument to `sqrt` from zero or a small positive number to some small negative quantity. In such a case it may be prudent to allow square-root of a small negative number to be 0.

by floating point scaling operations, and this can be generalized by (say) changing a multiplication by 3 to a shift and add. Other possibilities include using double-double (DD) numbers as an intermediate precision at lower cost than QD, when appropriate.

9 Acknowledgments

Yozo Hida helped get qd running in a Windows dll for use by Lisp. A newsgroup posting by Ingvar Mattsson provoked us into thinking, once again, about generic arithmetic in Lisp. We especially thank Duane Rettig who patiently advised us how to coax substantially better performance from Allegro Common Lisp's foreign function interface. The quadrature program benefitted by the example by Bailey, Jeyabalan and Li [**?**]. I also thank Raymond Toy and Duane Rettig for discussions regarding OCT, various Lisp implementations, and rounding bits in different hardware implementations of IEEE arithmetic and its extensions.

References

[1] David H. Bailey, Karthik Jeyabalan and Xiaoye S. Li. "A Comparison of Three High-Precision Quadrature Schemes," Experimental Mathematics, 14:3, 2005, pp. 317-329, `http://crd.lbl.gov/~xiaoye/quadrature.pdf`

[2] Yozo Hida, Xiaoye S. Li and David H. Bailey. "Quad-Double Arithmetic: Algorithms, Implementation, and Application," October 30, 2000 Report LBL-46996 (see `http://www.cs.berkeley.edu/~yozo` for links and related information

[3] Multiple Precision Floating-Point Reliable Library, `http://www.mpfr.org/`.

Appendix

Lisp source code for the QD interface, including comments and examples, can be found in the directory// `http://www.cs.berkeley.edu/~fateman/generic`, as well as a compiled "qd.dll" suitable for windows x86 linkages.