Importing Pre-packaged Software into Lisp: Experience with Arbitrary-Precision Floating-Point Numbers

Richard J. Fateman
University of California at Berkeley

Abstract

We advocate the use of Common Lisp as a powerful glue for building scientific computing environments. Naturally one then has to address mixing pre-existing (non Lisp) code into this system. We provide a specific example as an elaborate FORTRAN system written by David Bailey for arbitrary-precision floating-point numeric calculation. We discuss the advantages and disadvantages of wholesale importing into Lisp. A major advantage is being able to use state-of-the art packaged software sooner, while overcoming the disadvantages caused by FORTRAN's traditional batch orientation and weak storage model. In this paper we emphasize in particular how effective use of imported systems may require one to address the contrast between the functional (Lisp-like) versus state-transition-based (Fortran-like) approaches to dealing with compound objects. While our example is high-precision floats, other highly useful packages including those for simulation, PDE solutions, signal processing, statistical computation, etc. may also benefit by similar consideration. Most of the work described in this paper was completed in 1992.

1 Introduction

One of the traditional advantages of Lisp as a programming language is that it is relatively supportive of language extensions to data abstractions, even those not specifically anticipated. The building of problem solving environments [10] is just such a situation.

An additional selling point for Lisp is that it has an interactive basis, and a thorough support for object-oriented programming. CLOS (the Common Lisp Object System) is a standard that has reinforced the ability of programmers to extend operations by allowing the use of the same generic function names for dealing with different but related abstractions.

Thus one might consider building models of data using the same operators PLUS and TIMES, extending their meaning to depend on what is being added or multiplied[1]. In this way, various investigators have, without particularly great violence to the language, extended Lisp to incorporate new operations on polynomials, intervals, or other abstractions while preserving the same general mathematical or algebraic structure.

One such extension that has repeatedly appeared as user-implemented programs is that of a multiple-precision floating-point number system. A design for such a system in Fortran goes back at least as far as Wyatt [17]. A well-distributed version is the one by Brent [5]. An early implementation in Lisp is described by Fateman [8]. Other implementations have been written through the last several decades. We note a few more: Sasaki [14] wrote programs in Lisp for the Reduce system; for the SAC-I system, Pinkert wrote a system [12]; most other programs seem to have been written in C such as the desk-calculator (`dc`) in the UNIX system [15], and components of Maple [6], Mathematica [16], PARI [4] and Saclib [11].

Viewed from the perspective of a subroutine library for applications, it is clear that the authors of the Fortran-extension packages were sensitive to usability issues, and that a "raw library" interface was thought to be too hard to use. Therefore many of the Fortran-based system providers (e.g. Wyatt, Brent, and Bailey) include as an option, some kind of pre-processor or interpreter to make access to the subroutines "easier". By contrast, the environment of a computer algebra system written in Lisp generally provides a ready-made interface, one familiar to Lisp programmers. For those already engaged in writing large systems in Lisp who need long floating-point routines, this direct access from Lisp is ideal. For programmers ignorant of Lisp, this is of course yet another burden.

2 Strategies for Implementation

Because Lisp systems today include built-in arbitrary precision *integers*, it would seem that implementation of arbitrary-precision *floating-point* numbers would be easy. The main question, and it appears to be a thorny one, for the implementor is usually whether to use a decimal (10 or 10^n) or binary (2 or 2^n) radix. Decimal radix[2] makes input and output easier, but may waste space or time on internal computations, compared to speed in base 2 (or a power of 2). Using decimal radix, which therefore eliminates binary-to-

[1] This kind of facility is familiar to users of new-comer languages such as C++ and Java, which have implemented the "easy parts" of object inheritance.

[2] or sometimes, as used in Maple, radix = largest power of 10 that fits in a word or half-word size

Submitted to 1997 International Symposium on Symbolic and Algebraic Computation, Maui, HI (USA), July, 1997

decimal conversion, the system provides more of a what-you-see-is-what-you-computed view of the bigfloat regime, making certain kinds of explanations unnecessary. These include translations between binary "unit in the last place (ulp)" and decimal ulp, rounding, truncation, etc. The Reduce system [14] uses decimal. In our bigfloat package, we wrote two versions for Macsyma (the differences are limited to a few places in the code), with the more recent one being binary; Bailey's MPFUN went through two versions, settling on a binary version.

But all such Lisp implementations (including our own earlier work) suffer a major (although non-technical) flaw: There are too few programmers to maintain such a package in the face of continually improving algorithms, changing Lisp environments, and even changing computer architectures. For efficiency, it is generally much better to re-use a state-of-the-art package, written by another person who is an expert on the latest algorithmic advances and perhaps polished for the latest hardware wrinkles, than to implement a an elaborate algorithm in your own language and system. Fortunately, David H. Bailey has done an excellent job of building a multiple-precision package in a Fortran[3]. To quote from Bailey's abstract[1] describing MPFUN:

> This package features (1) virtually universal portability, (2) high performance, especially on vector supercomputers, (3) advanced algorithms, including FFT-based multiplication and quadratically convergent algorithms for π, exp and log, and (4) extensive self-checking and debug facilities that permit the package to be used as a rigorous system integrity test.
>
> This paper describes the routines in the MPFUN package and includes discussion of the algorithms employed, the implementation techniques, performance results and some applications. Notable among the performance results is that the MPFUN package runs up to 14 times faster than another widely used package on a RISC workstation, and it runs up to 153 times faster than the other package on a Cray supercomputer.

Bailey's paper includes discussions of applications to testing of the transcendence of certain mathematical constants, the computation of π to high precision, and the role of the FFT (fast Fourier transform) in real and complex arithmetic.

In any case, given such a thorough package and excellent documentation, the more appealing approach for using MPFUN programs from Lisp is to set up a collection of communication functions, and access the subroutine library via "foreign-function" calls. Since MPFUN naturally uses the Fortran data model, some interfaces are necessary to make it convenient to use from Lisp.

3 State-Transition vs. Functional Programming

Lisp and Fortran programmers generally prefer different styles, as emphasized in their the programming environments. Crudely speaking, the idiomatic Lisp programmer views most programs as compositions of functions. The Fortran programmer views most programs as incremental alterations of some global state.

[3] Could Bailey's suite of routines be completely rewritten in Lisp? Yes, see Appendix A. But we would like to build on Bailey's shoulders, not his toes.

In using MPFUN from Lisp, some bridging of these two models may be helpful, and so we explain the distinction in greater detail.

3.1 The State-Transition / Fortran Model

MPFUN code provides subroutines to be used typically from a FORTRAN main program in a sequences of `CALL` statements. These affect the contents of memory (the computer's "state") in various ways. For example, the form `CALL MPADD(A,B,C)` adds MP (multiple-precision) numbers A and B and stores the resulting MP sum into location C. Another example is `CALL MPCSSN(A,PI,X,Y)` which computes both the cosine and sine of the MP number A and returns the two MP results in X and Y, respectively. PI is the MP value of π computed by a previous call to `MPPI`.

Note that this code uses the call-by-reference convention that allows values to be returned through arguments.

3.2 The Functional / Lisp Model

Using the *functional style* of programming in Lisp means that one computes `r:=a*x+b` by writing `(setf r (+ (* a x) b))`. Let us say that `a`, `b`, and `x` are all MP numbers. Then one functional-style interface to MPFUN would be to extract from the 3-argument state-changing functions like `MPADD` and `MPMUL`, corresponding functions `mp+` and `mp+`, each of which takes two arguments, and *returns* the MP result. Then we utter the computation as:

```
{\tt (setf r (mp+ (mp+ a x) b))}                (1)
```

Of course, even in Lisp we could hew more closely to MPFUN's conventions, and use a sequence in Lisp. The three statements multiply, add and store results, and *use two temporaries*. Then the FORTRAN program

```
CALL MPMUL(A,X,TEMP1)
CALL MPADD(TEMP1,B,R)
```

would be rendered in Lisp as

```
(progn  (mpmul a x temp1)
        (mpadd temp1 b r))                                    (2)
```

In the program (1) the system must implicitly allocate some temporary space (comparable to `temp1` in version (2), for the temporary storage of $a \cdot x$. This may be both inconvenient and a major blow to efficiency if in fact what happens is that Lisp allocates a (perhaps substantial) section of memory for the single-time temporary use of this computation. This section of memory may have to be de-allocated later. Version (2) also seems to have that problem. But we can allocate a collection of temporary locations – comparable to floating point registers, or a stack of floating point locations, for use by these programs. We recycle the temporary location such as `temp1`. In a pure-functional approach, such a dependence on such a recyclable resource may be difficult or impossible to handle. (One could, of course "compile" the functional program into the sequential version, and that would avoid some of the inefficiencies, as long as one had enough registers and a clever enough compiler. This argument is commonly heard among Lisp aficionados, but compilation is not always available).

So we see one problem with functional programming if it were the sole paradigm of dealing with large structures. Creating and discarding objects seems to be a potentially expensive requirement.

It is not the only problem that shows up here. A strict functional approach forbids global data. In such an environment it is necessary to pass along every consideration that might affect the result as a parameter, and to return as part of the result every change in the environment. Given such orthodoxy with respect to MPFUN, it would be necessary to pass into programs like `MPADD` or its functional correspondent, the following information (and more) that is set up in `MPFUN`'s `common`: `idb`, a debug flag normally set to 0.If `idb` is set to other values up to 10, various information will be printed. `lbd` is the "logical unit number" in Fortran lingo for output of debug and error messages. This is normally set to 6. Another flag, discussed later, is `ird` which controls the rounding. The action of a program in case of an error is affected by a "mask" `ker` – an array of integers which can inhibit messages and/or allow for conditional execution resumption for each of 72 routines.

The result of each computation is theoretically a composite of the MP numbers in the returned fields and `ier`, an integer error flag which is changed from its initial value of 0 if any unmasked errors occur.

Thus we would see `(mpplus x y idb lbd ird)` returning a pair `(result . ier)`.

This kind of programming without side effects or global variables is generally painful for sufficiently large systems. So much so that most people programming in Lisp, a language that is naturally supportive of functional programming, do not adhere to a *strictly* functional regime.

Since MPFUN supports the state-based model, it seems reasonable to pass all this functionality on to the user, at least as one option, but to provide the functional "Lisp-like" model for those who find it a convenience, and for whom the inefficiencies are unimportant.

The user can program in either style. Programming in a mixture of the two styles is also possible, and even convenient sometimes, but the programmer must guard against conflicts in the use of MP registers.

4 The Implementation

Here we provide a brief survey of Bailey's design; a full description is given by Bailey [3] [1].

4.1 MP Data Types

This description is taken from Bailey's documentation [3] with only minor change. An MP number is represented by a single precision floating point array. The sign of the first word is the sign of the MP number. The magnitude of the first word is the number of mantissa words. The second word of the MP array contains the exponent, which represents the power of the radix `bdx` $= 2^{24}$. The succeeding words beginning with the third word in the array contain the mantissa. Mantissa words are floating point whole numbers between 0 and `bdx - 1`. For MP numbers with zero exponent, the "decimal" point is assumed after the first mantissa word. For example, the number 3 can be represented exactly in the MP format by the three-long array `#(1.0 0.0 3.0)`. An MP zero is represented by the two-long array `#(0.0 0.0)`. The number 2^{24} is `#(1.0 1.0 1.0)`. The number 10^{12} is `#(2.0 1.0 59604.0 1.0817536e+7)` because `(+ (* (expt 2 24) 59604) 10817536) = 1.0e+12`. If the number of mantissa words is 16, then these values may be store in larger arrays. For example, the number three would then be the length-18 array `#(16.0 0.0 3.0 0.0 ...)`.

In Lisp it is easy to add some sugar to this structure: for example a header that says "this is a multiple-precision number" so that printing routines (etc.) can be automatically invoked, and so that the Lisp system can unequivocally determine that the object being examined is of type MP, and not, say, some other array of single-floats. We have refrained from doing so for the moment. If our universe of numbers is sufficiently constrained so that arrays of single-floats are not used for anything else, we can consider this as "manifestly typed" to indicate MP's type signature. As for printing MPs, it is unclear how anyone really wants to see very long integers. Common Lisp would return a result that looks like an array, but would provide an option (especially if used with CLOS) for other kinds of default or specific printing.

The MPC data type (multiple-precision complex) is represented in MPFUN by an array of length $2n+4$ (or possibly more) where the first $n+2$ words are a real MP number (composed of n mantissa words plus the length/exponent words). Next there are two empty words (used by MPFUN internally for rounding), followed by $n+2$ words for the imaginary part. Because there are various ways of breaking up this array if the numbers are shorter than the maximum, it is necessary to keep track of the inter-component spacing. Thus in Lisp it is natural (and here we have succumbed to what is natural in Lisp), to use a structure of two parts, L, the spacing which is usually $n+4$ and the array of length $2n+4$ with the components inserted in the appropriate places.

4.2 Setting Parameters in Common

The Lisp package provides the opportunity to set any of the parameter in Fortran Common directly, as well as to read them. The one that is of primary interest is `nw` or the maximum number of mantissa words used for a newly created MP number. Not all MP numbers will use this number of words: if the value can be represented exactly with fewer words, we will normally use only that number. `nw` should be set in the main calling program to `1+nd/7.225` where `nd` is the desired maximum precision level in decimal digits. This number can be changed, and will affect the results of future computations, but will not affect the previously generated results. There is a practical maximum `nw` that is unfortunately set at Fortran-compile time. It has to do with the allocation of scratch space for some of the routines. For the basic (non-advanced) routines, a scratch space pre-allocated at 1024 words allows for over 7000 decimal digits. For routines such as sine and cosine, the need for extra scratch space restricts the number of decimal digits to about 780. This can be expanded by allocating more space in a `Common` area; such inflexibility is a hallmark of Fortran's storage mechanism and an alternative (having Lisp allocate more storage and passing it to Fortran) could alleviate this problem; it would require changing the Fortran code in substantial ways. Rewriting the code in C, with C's flexibility for allocation and deallocation is another solution. This opens up enormous possibilities for programming errors compared to automatic storage control with garbage collection, but also provides a level of efficiency that may be welcome.

A model of how one can bridge the gap to Fortran is represented by our approach. We have provided Lisp programs to read and write in `common`: `setcom` sets one value in common `mpcom1` e.g. by `(setcom nw 16)`, and returns the old value. If `nw` is increased, then the MP "registers" `mpreg1` to `mpreg4` and `mpregx` in Lisp are also enlarged if necessary. The `mpregx` register is used by "internal" programs; unless otherwise documented, the programmer is free to use the others. We keep track of their "high water mark" `mpreghwm` and never shrink the registers. We also provide registers `mpl2` for $\log_e 2$ and `mppi` for π. For the complex number system we have a set of (distinct) registers `mpcreg1` to `mpcreg4` and `mpcregx` for temporary computations.

It is also possible (and probably more stylish in Lisp) to use the idiom `(setf (mpcom1 nw) 16)` for changing `nw`, and if the complex registers are going to be used, executing `(enlarge-mpcregs (mpcom1 nw))` is advisable.

We adopt the same treatment as MPFUN for dealing with errors: "Common block MPCOM2 contains an integer array `ker` of length 72. This array controls the action taken when one of the MP routines detects an error condition, such as an attempted divide by zero. If the entry corresponding to a particular error number is zero, the error is ignored and execution resumes, although the routine that detects the condition is often exited. If the entry is one, then a message is output on unit `ldb`, `ier` is set to the error number (an integer between 1 and 72), and execution resumes, although subsequent calls to MP routines are immediately exited (i.e. control is quickly returned to the user program). If the entry is two, a message is output and execution is immediately terminated. All entries of `ker` are initially 2." [3] And yes, messages output to the default unit `ldb` appear as Lisp standard output.

5 Additional Utilities for MP in Lisp

5.1 Input

A single Lisp program reads any normal Lisp number and converts to a MP form. The conversion from usual types provides input as an integer (in which case enough precision is provided to record all the digits correctly), any preexisting floating-point format, and any ratio (Lisp supports numbers like 1/3). Numbers that are not exactly representable (like 1/3) have `NW` "mantissa" words of precision. Reading of some other formats can be handled by combinations of these conversions. For example `(mp* (into-mp 123) (mppower mpten 40))` computes a representation for 123.0e40.

5.2 Output

Several programs are provided. The simplest, `mpprint` will take a single MP argument and print it to the default user output stream in the form exactly as produced by MPFUN, a form which is practical but may seem rather stilted for people used to using Lisp. For example, MPPI is

```
10 ^          0 x  3.14159265358979323846264338327950\
288419716939937510582097494459230781640628620899862\
8034825342117067982148086l2566
```

The programmer can use obtain this kind of information as a string by `(mp2string mppi)`. A more conventional version `(mppr mppi)` provides a printout of

```
 3.1415926535897932384626433832795028841971693993751\
0582097494459230781640628620899862803482534211706798\
214808612566x10^0
```

and we provide a more flexible program for reformatting or printing numbers at your leisure: `(mp2triple mp)` takes an MP number and returns 3 *integer* values: the sign (either 1 or -1), an integer representing the (base 10) exponent, and an integer representing the fraction. The fraction will have an implied decimal point to the left of its first digit.

All outputs are computed from the results from the MPFUN output conversion subroutine `mpoutc` which computes how many decimal digits are meaningful, assuming the given number is precise to the currently set `nw` number of mantissa words. It then rounds to the nearest decimal number, and omits trailing 0's (or rounds up 9's) when these can be seen as artifacts of the conversion from binary to decimal.

If MP numbers are not printed with such programs, they will look like vectors of single-precision floats. For example, `mppi` is

```
MPFUN(76): mppi
#(16.0 0.0 3.0 2375530.0 8947107.0 578323.0 1673774.0
225395.0 4498441.0 3678761.0 ...)
```

Since the source code for the printing functions (as well as other programs) is available, it is possible to make special formats as needed. Especially long integers are rarely useful as normally printed, since the human eye needs more help than is provided by a stream of digits.

5.3 Conversions

Converting an MP number `m` to the usual Lisp form is done by `(mp2rational m)`. If it happens to be an integer then the expression `(mp2integer m)` will evaluate to the nearest (rounded) integer equivalent. You may not want to actually see the result of either of these programs since it is visually hard to understand large strings of digits or rational fractions; it is probably preferable to print `m` as a decimal number instead, using `(mppr m)`. If you want to be more specific about the format, use `mpstringdecom` as described above.

We provide a Lisp function that returns a (new) freshly rounded result `(mpround m)`. It is based on the MPFUN routine `(mproun m)` which destructively rounds the MP number to the current precision, using the current setting of the rounding flag `ird`. `mproun` can be used directly only if `m` is already an MP array of length `nw+4` or more (or precision `nw+2`). The extra two words are needed in-place for guard digits in the rounding arithmetic.

The function `(mpcopy m)` creates a fresh copy of an MP number of the current size. `(mpmove m n)` moves `m` into `n` destroying `n`'s previous contents.

Changing the number of mantissa words should be done by `(setf (mpcom1 nw) k)` where `k` is an integer. The equivalent number of decimal digits is about 7.225 times `k`.

6 Sample Code

Here's how we wrote some simple routines to call MPFUN.

```
;;(mp+ a b :target c)  does  c:=a+b
;;(mp+ a b)            returns a+b

(defun mp+ (x y &key (target mpreg1 supplied) )
  "add 2 MP numbers and optionally store in target"

  (unless
    (and *typecheckmp*
         ;;is typechecking turned on? if so, check ..
         (mpp x)(mpp y) (mpp target)
         ;;check types of input
         (<= (mpcom1 nw) (+ 2 (length target))))
```

```
          ;; is target large enough?
     (error
      "MPFUN mp+ called with incorrect
       arguments ~s ~s ~s"
       x y target))

   (mpadd x y target)
   ;;call MPFUN Fortran subroutine MPADD

   (cond ((zerop (mpcom1 ier))
         ;; if no error happened in the Fortran part,
  (if supplied target
              ;; and if target supplied, return it else
    ;;copy out the result and return new object
    (subseq target 0
             (+ 2 (truncate (abs (aref mpreg1 0)))))))

;; otherwise, if there was an error in mpadd
         ;; it will already be reported by MPFUN;
         ;; we add our message too
(t (error "MPFUN ier= ~s from mpadd" (setcom ier 0))
    ;; return the value of ier and reset it to 0
   )))

(defun mpev (a x)
"Evaluate a polynomial a[0]*x^n+ ...+a[n-1]
 with real multiple precision coefficients
 at a real mp argument."

   (unless (mpp x) (setf x (into-mp x)))
   (let* ((p (mpbuf))
  (n (1- (length a))))
     (mpadd (aref a 0) p p)
     (do ((j 1 (1+ j)))
   ((> j n) p)
;; use a register model rather than
         ;; functional, to save mem. alloc
(mpmul x p mpreg1) ; x*p -> r1
(mpadd (aref a j) mpreg1 p)))) ; r1+a[j]-> p
```

A simple modification to this code would always convert the inputs regardless of their original (presumably numerical) type, to MF numbers: Just insert the line `(setf x (into-mp x) y (into-mp y))` as the first executable statement.

7 About Accuracy

Most programmers prefer a rather unsophisticated approach to numerical analysis. That is, use as little as possible, Often one hopes that a wrong answer will somehow look suspicious, in which case re-doing the calculation in double-precision may fix the result. This does not always work however. For an alternative approach using ever-increasing precision, see Richman's paper [13].

Even a good understanding of fixed-precision floating-point numbers does not necessarily imply an understanding of the subtleties of variable-precision floating-point errors. Especially considering that with a package such as MPFUN it is possible to ask for computations over an absurdly large exponent range (at least $10^{\pm 14{,}000{,}000}$) and well as potentially high accuracy (about 16 million decimal digits).

Consider how you would deal with a request for a certain accuracy over an arbitrary input. Usually, if you want an absolute error of d in computing $\sin x$, you will need at least $\log_{10}(|x|) + \log_{10}(|1/d|)$ decimal digits of π. That is, to get $\sin(10000)$ right to 0.0001, you will need π to 7 digits.

More commonly you might expect a routine to be described as having a *relative error* of d. Then it needs to do more work. Consider $\sin(31415926)$. If you only knew 8 digits of π, you might reduce the argument modulo π to 0, then compute $\sin(0) = 0.0000$. But the correct answer is about-0.5106. So the relative error in the answer 0 is quite high, and it is certainly not good to 8 decimal places.

8 For details on MPFUN

We do not attempt to reproduce the details given in the excellent documentation of MPFUN [1]. In fact, we encourage users except of the most casual sort, to obtain full documentation for reference. The principal support is for arithmetic, but also log, exponential, sin/cos, sinh/cosh, angle (related to arctan and arcsin), square-root, cube-root, polynomial zero-finding, searching for an integer relation (A typical application of this routine is to "determine if a given computed real number r is the root of any algebraic equation of degree $n-1$ with integer coefficients." [1].

9 Various details

The program is loaded by typing `:ld mpfunext` into Lisp in the appropriate environment.

The number of words used in the mantissa of MP numbers is initially 16. This can be changed to, say, 20 by `(setf(mpcom1 nw) 20)`.

Converting another number known to Lisp, say an ordinary floating-point number, an integer, or a rational number x is done by `(into-mp x)`. The conversion is done exactly for integers, even if it requires more than `nw` words. The conversion for rationals may be inexact. For sufficient MP precision, the conversion from single or double floats is exact.

Arithmetic routines `mp+, mp*, mp-, mp/` all work about the same way: they each take two required input arguments. a third keyword `:target` is optional. The common usage would be something like `(setf x (mp+ a b))` but a "more efficient" one would be `(mp+ a b :target x)`. Each of these usages "returns" the same value, but in the second case, the inside of the value of `x` have been changed. It is, in that case, necessary that `x` already be an MP representation of the appropriate length (mantissa length nw). If `x` is some other quantity, say `NIL` or a normal Lisp number, using this latter form is an error.

If you wish to use `:target` arguments to avoid memory allocation costs, but dont have any special places allocated for these temporaries, you may use `mpreg1` through `mpreg4` for your purposes. These will always be the right length for the current value of nw, since they are stretched automatically if nw is increased.

The predicate `(mpp m)` returns `t` if `m` is an MP. The predicate `(mpzerop m)` returns `t` if `m` is an MP zero.

The program `(enlarge-mppi n)` computes a higher precision value for π unless it already has stored sufficient precision for `n` words of mantissa. The value `mppi` is the current value of π. Thus `(mpprint mppi)` will show you an approximation to π in decimal. You may wish to have a higher-precision value for π than for other numbers because this value is used in range-reduction for the trigonometry programs.

The program (enlarge-mpl2 n) computes a higher precision value for $\log_e 2$ unless it already has stored sufficient precision for n words of mantissa. The value mpl2 is the current value of $\log_e 2$. Thus (mpprint mpl2) will show you an approximation to $\log_e 2$ in decimal. You may wish to have a higher-precision value for $\log_e 2$ than for other numbers because this value is used in range-reduction for exponential and logarithm programs.

The value of mpeps is the value of one ulp (unit in the last place) for the current MP precision. It is the smallest positive number that when added to 1 gives a number different from 1. This value is useful for computing error bounds or stopping criteria for iterations.

The functions for log and exponential, as well as the trigonometry functions are supplied in a someone more primitive state, corresponding to the MPFUN routines: (mpexp a log2 b) has the effect of computing e^a and storing it in b which had better be an MP number representation, and of the appropriate length. If you have already computed an appropriate value of log 2 as mpl2 you could do this: (mpexp a mpl2 mpreg1) and then copy the resulting value (if you so desire) from mpreg1 to some other place. This can be done by (setf ea (mpcopy mpreg1)). Or the same result can be computed by (setf ea (mpbuf))(mpexp a mpl2 ea).

If you do not have an appropriate value for log 2, you can do this:

```
(enlarge-mpl2 n) ;; where n>= nw, most likely
(setf ea (mpbuf))
(mpexp a mpl2 ea)}
```

The program (mpcssh a mpl2 x y) stores $\cosh a$ in x and $\sinh a$ in y.

The program (mpcssn a mppi x y) stores $\cosh a$ in x and $\sinh a$ in y.

If you do not have an appropriate value for π, and you need only sine for further work (and thus do not wish to save the cosine and can leave it in a register) do this:

```
(enlarge-mppi n) ;; where n>= nw, most likely
(setf si (mpbuf))
(mpssn a mppi si mpreg1)}
```

Notice that the alternatives to providing a value of π as input to the sine/cosine routine are not entirely satisfactory. For example, one could (perhaps iteratively) find a good-enough value of π to provide the requested precision with full accuracy (done, for example, by Macsyma [8]) or refuse to provide a result for an argument of very large size (loss of precision in range reduction). Many systems implement this latter option. Perhaps it is unreasonable request $\sin(10^{100})$ if you are only carrying 100 digits of precision.

The program (mpang x y mppi a) computes the MP angle A subtended by the MP pair (X, Y) considered as a point in the x-y plane. This is more useful than an arctan or arcsin routine, since it places the result correctly in the full circle, i.e. $-\pi < a \leq \pi$.

The polynomial evaluation program given above is not part of the Fortran MPFUN package. The program (mpev a x) evaluates a real polynomial in MP at the real MP (or other Lisp number) point, x. The argument a must be an array of coefficients, each of which is an MP number. This array be created in a variety of ways, but the simplest may be to use a program we provide. For example (make-mpfa 10 20 3 0) creates an appropriate data structure for the polynomial $10x^3 + 20x^2 + 3x$. The program mpmpeval takes the same arguments as mpev but returns 4 values. The first is the polynomial value (same as mpev), the second is a bound on the error in that value. The next two values are useful in iterating to a root of the polynomial using Laguerre iteration. For details, see the code.

(mpcompare a b) returns -1, 0, or 1 in ic as $a < b$, $a = b$ or $a > b$. Built upon this are the programs mp>, mp<, mp= with obvious semantics.

(mpabs m) is the absolute-value function. If a target location is preferred in case m is negative, there is an optional second argument: (mpabs m r) in which case r will be altered to contain m or a copy of m with a sign change. If r is not supplied and m is negative, a new (sign-changed) copy of m is returned, otherwise m itself is returned, not a copy.

(mpsqrt a b) is the MPFUN square-root. We haven't encumbered it (yet) with Lisp: Normally, the MP b is changed to $\sqrt{a}$, but if the argument is negative, this is what happens:

```
*** MPSQRT: Argument is negative.
*** MPABRT: MPFUN cannot continue, err code =  71
```

and all subsequent calls to MPFUN routines will be "no-ops". To correct this, one must reset the error code by (setf (mpcom1 ier) 0) as illustrated in the sample code for mp+ above.

9.1 Extra Speed for High Precision

Bailey has provided a suite of routines for extra-high precision using FFT-based algorithms. These generally require that the number of mantissa words be rounded up to a power of two, and that certain initializations are performed by mpinix Rather than provide the details, we direct the interested reader to the source code provided for this as well as Bailey's documentation.

9.2 Complex MP subroutines

A suite of complex MP routines is also part of the MPFUN package. We have provided analogous routines such as into-mpc, mpc+, mpc* etc. Their usage is described in the program documentation.

10 Extensions

It is natural to use MP numbers as the end-points of intervals. Although Lisp provides exact rational numbers, and these can be used as rational endpoints bounding any kinds of real operations, the sizes (lengths of numerators and denominators) tend to grow rather dramatically (e.g. doubling on many operations). As their sizes grow, the cost of the operations grow, and therefore a complicated computation may result in intervals whose endpoints are needlessly elaborate. One can sometimes periodically widen intervals with rational endpoints in order to take advantage of nearby "easy-to-represent" rational numbers (e.g. those with denominators that are powers of 2 [11]) but in fact this is approximately equivalent to conservative-rounding of floating-point numbers.

Building a real-interval package on top of an existing arithmetic is another example [7] of a task easily undertaken in Lisp. In fact, MPFUN has provided a handle, although not a complete solution for one of the essential controls: (setf (mpcom1 ird) n) sets the rounding mode in MPFUN. When ird is 0 the last mantissa word is the same as the result of truncating a higher precision result. When ird is 1 the last word is rounded up if the next ("guard") word would have been 2^{23} or more. (This is assuming that the word size of 24 bits is used, and that therefore this represents the half-way point for numbers up to $2^{24}-1$) When `ird` is 2, the last word is rounded up if the guard word is non-zero. The default is `ird=1`. This is tested in only one place (Fortran subroutine `mfroun`) and could be replaced or augmented by additional settings for `ird` such as round toward $\pm\infty$ or round toward/away from zero, or round to nearest even.

There is no guarantee for most of the routines that all the digits produced are correct, however the detailed documentation for MPFUN gives an approximate bound on each of the non-trivial routines. However, carrying additional words in the mantissa can provide, with extremely high probability, a result correct to within rounding error, for most of the routines if the results are computed to one or two extra mantissa words of precision and then rounded before further use. The answers to addition and multiplication are accurate to all digits given, subject to the rounding mode.

An extension to MP that might also be useful is to provide a closer model to that of IEEE-754 binary floating-point arithmetic. Such possible modifications include encoding of not-a-number forms, infinities, and signed zeros. Perhaps a natural way of encoding this is to use the exponent word in the MP format: a reserved operand in that position could be used as a key to a reserved MP. A negative zero could be signalled by a negatively signed zero in that same word. Most of this could be done in the Lisp interface, although small changes in the Fortran would be the easiest way to include new rounding modes (e.g. round toward/away from zero).

We expect that the treatment of roundings would be less critical to most MPFUN applications since (many) additional guard bits are relatively easy to specify: `(setf (mpcom1 nw) (1+ nw))` gives you 24 more bits.

Based on our experience with Lisp, we would encourage the design a treatment of arithmetic in the language that includes a smooth transition from hardware floating-point to software arbitrary-precision, and that integrates smoothly with existing Lisp types (like rationals, complex integers, etc). There is a potential for using CLOS to put together extensions to intervals, polynomials, matrices, etc., for a more uniform approach to numeric objects.

Another direction for extension is to provide a more friendly (i.e. algebraic language) user interface. For this purpose, we have instrumented a parser (written in Lisp) that accepts an infix language based to interpret simple arithmetic expressions and directives for changed precision, as commands to MPFUN. This system can be contrasted with computer algebra interactive systems in terms of speed, accuracy, or other utility.

For a sample speed comparison: we computed 725 decimal digits of π The time[4] to square this and store the result in a pre-allocated MP register was 1.15 milliseconds (ms). Storing the result in a newly-allocated array raised the apparent cost to 1.35 ms, a figure which excludes garbage collection. We repeated the test until we provoked two garbage collections, each of which cost 500 ms., and observed that they occurred about every 600 multiplies. Amortizing the GC cost over each multiplication raises the total cost to about 2.5 ms.

For Pari 1.35.01 the time for a comparable multiplication and storage of the result was about 15 ms., for Mathematica 2.2 about 29 ms., for Maple V about 83.5 ms. For Macsyma, we observed two timings, one for a local version of the system[5] which took 85 ms. and a system compiled in Ibuki/Kyoto Common Lisp (Macsyma Inc.'s Macsyma 417.100) which took about 302 ms. We believe that the faster underlying built-in implementation of "bignums" in Allegro accounts for the 3.5 factor speedup.

11 Acknowledgments

Thanks to D. H. Bailey and NASA Ames Research Center for providing an interesting, efficient and well-documented program, as well as for comments along the way; thanks to D. Margolis of Franz Inc. for help in using the foreign-function interface in Allegro Common Lisp.

This work was supported in part by NSF Infrastructure Grant number CDA-8722788 and by NSF Grant number CCR-9214963.

12 Appendix: If MPFUN were written for or in Lisp

It is natural to ask how the design for MPFUN would differ if it were written from scratch (or re-written) for use from Lisp or written with Lisp as its source language.

In fact, the source code could have been written in Lisp, since it requires only facilities available in any Common Lisp: floating-point arrays. However, for high speed, the Lisp would have to be compiled into code as efficient as that from FORTRAN. Most Lisp compilers are not as sophisticated as FORTRAN in manipulating arrays; some FORTRAN compilers are even capable of taking advantage of the substantial parallel or pipe-lined components of MPFUN.

If we grant for the moment that the code will be in FORTRAN or C, how could we change it to make it "better" for our purposes in accessing it from Lisp?

1. Memory management. The primary difference would be in memory allocation for scratch space. Scratch space for calculations in Fortran MPFUN is allocated in a shared, named, statically allocated Fortran `common` module. It is therefore fixed in size at compile-time, and cannot be extended except by re-compilation. Attempting to exceed the available space produces an "allocation error" message from MPFUN. While it is possible to continue computing in Lisp, or even to make further calls to MPFUN routines, it probably is necessary to reduce the number of mantissa words (`nw` being carried before continuing to use MPFUN. If the computation needs to be done to that precision the only alternative is to follow the instructions in the error message to change two parameters in the Fortran code, followed by re-compiling and re-loading MPFUN. In C, a larger allocation may be made at run time, but their may be an overhead of an extra memory indirection to find that space.

   Inflexibility is one of the prices one pays to write in portable Fortran, compared to Lisp, C, or even a modern dialect of Fortran (e.g. Fortran 90 [2]).

2. Complex numbers. MPCs in MPFUN are represented as single arrays with the real and imaginary parts placed in adjacent places with a given offset. That is one vector `V` with the real part starting at `V(1)` and the

[4] Times were computed over many trials, and results exclude the cost of the "empty loop". Times were on a Sun Microsystems SPARC1+ processor.

[5] based on William Schelter's "Maxima" code, but compiled in Allegro Common Lisp

imaginary part starting at `V(L)`. If the real part is "full length", then `L` is `NW+4` or more where `NW`, an integer stored in common, is the number of mantissa-words in the representation of an MP number. Thus the value of `L` must be passed around, and all arguments to complex functions must have the same spacing. In Lisp, a more normal approach would be to treat a complex MP as a pair of MP vectors, simplifying the programming. Although this would cost an extra word in the worst case, it could save considerable space where some of the arguments are short. This ordinarily happens when the arguments are exact—there is no need to use `NW` mantissa words to represent 1/2 or 2.

3. Functional vs. State-modification programming. Arguably, the functional versions would be provided as the fundamental routines, and the compiler would have to optimize a construction like `(setf z (mp+ x y))` into `(mpadd x y z)`. The compiler would have to be told that `x` and `y` were `mp` representations, and also assure that `z` was previously allocated an array of suitable length and type.

4. Better abstraction. Arguably, one would construct MP numbers (via `defstruct` or as a class in CLOS) so that the "type" of the MP data type would be part of the datum, to distinguish it from other vectors of single-floats; indeed, a "structure" could be established to identify type, sign, actual mantissa length M, exponent (an integer), and a vector of M single-floats. This would cost a few extra words for each number, but would make the programming simpler and less error-prone. In particular, the association of a printing function with the MP type could be done automatically.

5. Using "bignums" for fractions. Arguably, one might represent a mantissa as an arbitrary-precision integer, and hope that this data type is handled well by the Lisp system. Alternatively one might choose a larger (integer) radix than currently in use. The requirement of a full-precision fast unsigned 32 by 32 multiply to get a 64-bit result can be met on some machines, but even so it is likely that the speed of integer multiplication will be substantially lower than the speed of floating-point multiplication. Indeed, some RISC machines do not even have an integer multiply instruction, yet the floating-point multiplication is pipelined. While the packing of bits will save space and can in principle speed up computation in very long arithmetic, it will probably be difficult to see this advantage in practice.

6. Better error handling. It might be possible to use the idea of a "Not-a-Number" present in IEEE-754 floating-point standard, in MP. This could be done by returning a reserved operand of some sort rather than leaving an error flag `ier` set in common. The notion of having to check and reset `ier` after each operation is sufficiently much of a pain that programmers might easily neglect it. Unfortunately, the operation of the MPFUN routines is such that a non-zero `ier` effectively prevents any further computation in the MP-FUN package – all routines immediately return with their arguments unchanged.

References

[1] David H. Bailey. MPFUN: A Portable High Performance Multiprecision Package. NAS Applied Research Office, NASA Ames Research Center, Moffett Field, CA 94035. March, 1991. A report was published as "Algorithm 719, Multiprecision translation and execution of FORTRAN programs." *ACM Trans. Math. Softw. 19*, no 3 (Sept, 1993) 288—320.

[2] David H. Bailey. "A Fortran 90-based multiprecision system." *ACM Trans. Math. Softw., 21* no 4, (Dec. 1995) 379—87.

[3] David H. Bailey. Source Code for MPFUN. `dbailey@wk49.nas.nasa.gov`

[4] C. Batut, D. Bernardi, H. Cohen, M. Olivier. User's Guide to PARI-GP. Feb, 1991 (version 1.35).

[5] Richard P. Brent. A Fortran Multiple Precision Arithmetic Package, *ACM Trans. Math. Softw. 4* (March, 1978), p. 57 – 70.

[6] B. Char et al. *Maple Reference Manual*, Springer-Verlag, 1992.

[7] I. Emiris and R. Fateman. Interval arithmetic in Lisp. (working paper).

[8] R. Fateman. 'The MACSYMA 'Big-Floating-Point' Arithmetic System," in: R. D. Jenks, (ed.), *Proc. of the 1976 ACM Symp. on Symbolic and Algebraic Computation,* (SYMSAC-76), Yorktown Height, N.Y., August 1976, 209-213.

[9] R. Fateman, K. Broughan, D. Willcock, D. Rettig, Fast Floating-Point Computation in Lisp *ACM Trans. Math. Softw. 21* no 1. March 1996 26–62.

[10] E. Gallopoulos, E. Houstis and J. R. Rice. "Future Research Directions in Problem Solving Environments for Computational Science," Report of a Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Artificial Intelligence for Computational Science, April, 1991 Washington DC

[11] Krandick, Johnson. SACLIB. RISC-Linz

[12] J. R. Pinkert. "SAC-1 Variable Precision Floating Point Arithmetic," *Proc. ACM 75*, (1975) 274–276.

[13] P. L. Richman, "Automatic Error Analysis for Determining Precision," *Comm. ACM 15* (1972) 813–817.

[14] T. Sasaki. "An Arbitrary Precision Real Arithmetic Package in REDUCE" in *Proc. EUROSAM 1979, Lecture Notes in Computer Science, 72,* 1979, Springer-Verlag pp. 358–368.

[15] UNIX manual.

[16] Stephen Wolfram. *Mathematica – A System for doing Mathematics.* 2nd edition. Addison Wesley, 1990.

[17] W. T. Wyatt Jr., D. W. Lozier, and D. J. Orsen. A Portable Extended-Precision "Arithmetic Package and Library with Fortran Precompiler," Math. Software II, Purdue Univ., May 1974.