Arbitrary-precision Decimal Floating-Point Numbers: More than You Expected, Less than You Asked For?

Richard J. Fateman

Electrical Engineering and Computer Sciences Dept.
University of California, Berkeley, 94720-1776, USA

May 16, 2020

Abstract

Some people expect that computers perform arithmetic the same way they learned in grade-school in essentially all respects, except much faster and without error. Must they be disappointed?

1 Introduction

Some people are surprised to see a computer make "obvious mistakes". For example, they notice that 0.01 added to itself 1000 times prints the absurd result 9.999999999999831.[1]

When users report this as a bug in whatever computer system X they are using, the reactions from the "experts" tend to point out that this is a feature of "floating-point numbers" and in any case is not unique to system X. The more didactic responses may direct the user to the excellent survey of binary floating-point numbers in David Goldberg's "What Every Computer Scientist Should Know About Floating-Point Arithmetic" `https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html` or something similar.

For such users, the system design fails one key idea in user interfaces, namely the "Principle of Least Astonishment".

There are computer arithmetic systems in which 0.1 is precisely 1/10, including some hand-held calculators, hardware systems specifically supporting decimal rather than binary arithmetic, and libraries for decimal or rational arithmetic. To make this generalize properly we require arbitrary-precision arithmetic, which rules out finite-precision hardware, hence software support. In our current context for specificity, we'll use the open-source computer algebra system Maxima. In the Appendix we provide URL links to some of the alternatives.

Here we discuss a model and implementation that differs in one respect or another from any of these other systems and is intended to be have some useful features and in particular should be less surprising.

2 Newbie expectations

What people expect varies from person to person. Here's a short list of possibilities. Each of the properties 1–7 below fails in some way for certain numbers in the usual computer's binary floating-point (IEEE 754 standard) system.

[1] Some systems are sloppy and print this number as 10. Mathematica does this, but if you set the number's precision to 16 or more, you see that it is not 10. Alternatively, subtracting 10, or comparing to 10 shows you the arithmetic result is different from 10.

1. Decimals are exact. 1/10 and 0.1 are the same.
2. Arithmetic (plus, times) is exact. $10.0^{100} + 1$ is a 101-decimal-digit number.
3. Arithmetic (plus, times) is associative: $(A + B) + C = A + (B + C)$.
4. The Distributive Law holds: $A * (B + C) = A * B + A * C$
5. There is no finite number that is too large to represent. You can just keep adding zero digits to the right (no "overflow")[2]
6. There is no smallest non-zero number. You can just keep adding zero digits to the right of the decimal point.
7. If you print a number N to a character string S and read S back into the computer, you get exactly the same number N.

There are other ideas sometimes requested[3]. The precision of numbers is indicated by how many digits are typed in or out.[4]

Some observations.

- We can sometimes do exact decimal division, as 10.0/2.0 is 5.0.
- Sometimes we cannot, as 1.0/3.0.
- Every exact binary float number can be expressed as a decimal float number since (1) any binary integer is trivially a decimal integer. (2) Non-zero digits following a binary point form numbers that are exactly representable in decimal also: 0.5, 0.25, 0.125, 0.6725 etc... and therefore sums of them are clearly decimal.
- The reverse is not true. Even the decimal 0.1 (meaning 1/10) cannot be expressed as a binary float exactly.
- Exact representation in decimal follows from the factorization of $10 = 2 \times 5$. If we used base 30030, we could exactly represent many other fractions, but not all. $30030 = 2 \times 3 \times 5 \times 7 \times 11$. Unfortunately this doesn't help if you are wedded to the notion of using digits 0-9.

3 The design

In our framework for computing in the Maxima computer algebra system, a system which is hosted on any of a number of Common Lisp implementations, we have at our disposal a nearly perfect model of the integers. Lisp and hence Maxima itself has no preset limit on how long an integer can be, and the boundaries between 16, 32, 64 -bit, or longer storage locations for integers is not visible to the user or Maxima programmer. Presumably the underlying technology sets some limit: certainly no integer can exceed in length the size of computer memory. That is usually quite a remote limitation.

[2] http://politicalhumor.about.com/library/jokes/bljokebushbrazilian.htm

[3] from people who are either physicists or perhaps in the construction trades.

[4] Precision and accuracy are sometimes mistaken for each other. 3.1400000 is a precise but inaccurate value for π.

3.1 Bluffing the user – design 1

Here we briefly describe and then discard a design alternative that is essentially a "bluff the user" approach.

In this version we do not store floating-point numbers at all. All numbers are integer or rational numbers. We merely change the printing routine so that, to the extent possible, fractions are printed in decimal. Thus 1/4 is printed as 0.25. This scheme fails for 1/3, so we need a notation to convey this value as a float. It is possible to use one for repeated fractions, but that is unfamiliar and uncomfortable. While 1/3 is 0.[3] where the bracketed digits repeat, 1/17 is 0.[0588235294117647]

Another is approximation with a note. For example, 1/17 prints as 0.0588¡*¿.

This does not solve the irrational (as sqrt(2)) or transcendental (as cos(1)), problem. or the read-print issue. These are reviewed in the alternative decimal design section. Not that it solves all issues, it seems that the design of the next section is more promising.

3.2 Decimal arithmetic – design 2

In this design, closer in spirit to usual floating-point binary numbers, we represent decimal numbers are pairs of arbitrary-precison integers, a significand and an exponent: For example, (123,-2) is 1.23 exactly. To maintain uniqueness, the first number has no trailing zeros unless it is 0. Thus (1230, -3) would be reduced to (123, -2). Zero is uniquely (0, 0).

We can define programs for the obvious arithmetic for adding and multiplying these quantities, but we note the concern that the significand can grow exponentially in length as a function of the number of operations, since the product of numbers of length n is 2n. If this becomes an issue, a significand of a number q can be rounded down to k digits by setting `fpprec:k` and invoking an operation `decimalfptrim(q);` This of course means the result is no longer fully accurate.

Implicit in the calculation system is the need to deal with non-terminating decimal numbers, first appearing for the (in general) inexact quotient, as in 1/3. We bite the bullet and provide that quotients are rounded to some pre-specified `k` digits by default.

Furthermore, operations on decimal floats that do not have any particular claim to exactitude are transferred to binary bigfloats (as the exponential, logarithmic, trigonometric functions). Calls to them are encoded as first a conversion to a binary bigfloat, then the application of the function, returning a binary bigfloat. The advantage here is that decades of programming have provided large libraries of (approximate, arbitrary-precision, carefully-rounded) binary-float routines, and we leverage off them when possible. Also, people who want exact decimals may be thinking only of addition and multiplication[5]

Given this translation into binary floats when needed, the user of Maxima can specify any operation on combinations of symbols, expressions, machine floats, integers, rationals, decimal and binary bigfloats. The arithmetic result of combining decimal and one of the other float types results in a binary bigfloat.

In some circumstances the appearance of exact decimal can be managed by (the user specifying the) rationalizing of binary bigfloats. The case of a decimal plus integer results in an exact decimal; combining a decimal with a ratio results in a decimal of precision `fpprec` decimal digits.

The input and output notation must distinguish decimal bigfloats from others, and so it differs in the signifier character introducing the exponent. The character L, for decimaL is used. The attempted exact conversion of a ratio into a decimal bigfloat is `decbfloat()` by analogy with the binary version `bfloat()`. If the ratio does not have an exact decimal representation (as 1/3), then the closest decimal version of the value given the globally specified `fpprec` is used.

[5]That is, satisfying the "ring axioms" of abstract algebra.

4 Implementation

The Maxima version of decimal bigfloats parallels the earlier implementation of binary bigfloats, with special circumstances for addition and multiplication which are allowed to grow in precision so as to be fully accurate unless specifically trimmed to the currently-set precision. Operations that cannot be done exactly are converted to (and left in) binary internal representation. Ordinarily the combination of integers with decimal bigfloats will result in decimals, but the presence of ratios or machine-floats or binary bigfloats will result in coercion to binary bigfloats. As is traditional in the Macsyma/Maxima code base, much of the interface is done in a data-directed fashion, in this case by associating "float programs" with the name of operators like sin, cos. Not everything could be done in such a systematic object-oriented fashion: Some of the components of the system including input, output, and the coercion simplification end up needing patching. While the complete package is about 438 lines of code (680 with comments), a more careful analysis is in order. Some of the code is actually unchanged, but copied to this file in order to insert small changes in built-in functions: 80 lines of `make-number` from the Maxima parser of which only 7 lines are added; 40 lines of the program `bfloat` of which 2 are added, etc.

The earliest version of (binary) bigfloats already provided limited. arithmetic facilities in decimal: sufficient to allow the reading and writing of bigfloat numbers in a carefully-rounded decimal form. Some of the prior code already checked for radix 2 or 10, but since we changed the semantics to have a growing "exact" fraction, this could not be used directly.

5 Speed

We are assuming that computation in decimal will be a small portion of any scientific computing task, since promotion to binary is so plausible in a mixed task. It is possible to perform some calculations of scientific interest exactly, even using only multiplication and addition. An exact dot product is probably the prime example that can be done in decimal. However, an exact dot product of (exactly represented) binary floats could also be done in binary by judicious choice of floating-point precision. There is an additional overhead in checking whether a bigfloat is decimal or binary at various simplification or evaluation functions, but this is a small percentage of the usual cost.

The cost of execution of arithmetic in decimal is probably higher than in binary because the shifting/alignment/rounding operations require multiplication or division by powers of ten, which is more expensive than binary shifting, the method used for multiplication or division by powers of 2. If performance becomes an important issue, a revision could fix this particular concern by encoding long decimals in radix 1000, using 10 bits at a time, thereby speeding up shifting. Each "digit" would be encoded by a group of 10 bits (good up to 1023). Using a large radix has other possible choices For example, 30030, somewhat less than 2^{15} or 32768. As mentioned previously, this gives us exact values for $1/p$ for p in the first 6 primes. (e.g. 1/11 is $2730 * (30030)^{-1}$). A consequence of this choice would be exact representation for numbers whose rational denominators can be represented by products of powers of prime numbers of 13 or less.

6 Examples

1.8L308+4.9L-324; returns a 634 decimal digit number. All representable double-float numbers fit within this range without round-off. That is, the numbers of the magnitude of the largest and the smallest double-floats as well as their sum can be represented exactly. In the Wxmaxima front end it displays as

`1.800000000000000000000000000[579 digits]00000000000000000000000049L308`

The computation `sum(0.01L0,i,1,1000)` returns `1.L1`
which we can compare to machine double precision
`sum(0.01,i,1,1000)` which is `9.999999999999831`

or binary bigfloat with precision set to 18 decimals (62 bits)
`fpprec:18;`
`sum(0.01b0,i,1,1000)` which is `1.0000000000000005b1`.

7 Availability

Maxima is a free open-source program which can be most easily downloaded as an executable binary from sourceforge, details depending on your host computer. The capabilities of decimal floats are in the share library and `load("decfp")` should load the package.

In addition to defining a small collection of programs, this file overwrites (and generalizes) a number of built-in functions already defined in Maxima, whose source code is in files `nparse, maxmin, trigi, numeric, float`.

8 Conclusion

Just as one can write a program to do arbitrary-precision exact integer arithmetic, one can also code arbitrary-precision exact decimal floating-point arithmetic. The exactness can be guaranteed for addition and multiplication, input, output. In Maxima the facility fits in to the general programming facility. The principal extra knowledge needed by users is to use the "L" exponent prefix or `decbfloat` to introduce such numbers into the system.

9 Acknowledgments

Some lively discussion on the Maxima users' electronic newsgroup helped consolidate decisions made in this implementation.

10 Appendix

As indicated earlier, software implementation makes it possible to construct floating-point systems with different properties.

Here is a selection of sources that are related to arbitrary- , or at least high- precision (possibly decimal) arithmetic.

Decimal radix:

- IEEE 854 standard `https://en.wikipedia.org/wiki/IEEE_854-1987` revision in progress.
- Maple `www.maplesoft.com`

Arbitrary (but prespecified) precision binary:

- Multiple-Precision (Brent) `https://maths-people.anu.edu.au/~brent/pub/pub043.html`
- GNU MPFR (Fousse, Hanrot, Lefevre, Pelissier, Zimmerman), `www.mpfr.org`
- ARPREC (Bailey) `http://crd-legacy.lbl.gov/ dhbailey/mpdist/`
- Unum (Gustafson),`https://www.crcpress.com/ The-End-of-Error-Unum-Computing/Gustafson/9781482239867`,
- interval arithmetic `http://www.cs.utep.edu/interval-comp/`

- Mathematica's significance arithmetic http://mathworld.wolfram.com/SignificanceArithmetic.html
- Macsyma Bigfloats (Fateman) https://sourceforge.net/projects/maxima/
- Reduce (Kanada, Sasaki),http://dl.acm.org/citation.cfm?doid=1089257.1089260
- Extended precision dot product (Kulisch) http://www.degruyter.com/view/product/178972

For some applications one can represent the bits of a bigfloat "sparsely" as the sum of a few machine-floats, leaving out sequences of consecutive zeros (or consecutive ones). This representation is not necessarily unique. For example (using decimal here), 0.9999999999 could be 1.0 - 1.0e-10. and 1.0000000001 could be 1.0 + 1.0e-10. For important applications requiring the resolution of geometric predicates, rapid and definitive results can be accomplished with such representation, results which might not be possible via naive computation with ordinary machine floats. See for example, Priest http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.3546 and Shewchuck http://dl.acm.org/citation.cfm?id=237337.