

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

September 19, 2022

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 9.99999999999831.¹

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.

¹Some 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 requested³. 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>

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

⁴Precision 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 $\text{sqrt}(2)$) or transcendental (as $\text{cos}(1)$), problem. or the read-print issue. These are reviewed in the alternative decimal design section. Not that it solves all issues, but 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-precision 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 `decimalftrim(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 multiplication⁵

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 significand 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.

⁵That is, satisfying the “ring axioms” of abstract algebra.


```
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 `decfloat` 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), 0.999999999 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>.