Tests for floating point arithmetic in **R**

Richard M. Heiberger and John C. Nash

2017-04-24

Abstract

Floating point calculations in any computing language may give results that are very different from results humans expect from arithmetic on real numbers. While in most instances these differences are inconsequential, there are several cases where they could be important:

- results are changed in ways that will affect outcomes or decisions;
- differences or changes in output from new versions of software may point to errors in that software;
- the differences cause upset and wasted effort in either explaining or addressing them.

This article and the accompanying **R** package are intended to provide a structure for building programs that test and report such issues. It is anticipated that this will be open-ended, that is, the development of such test codes will be ongoing and react to innovations in software and hardware.

JN: NOTE: We should figure out how we can both TEST and SHOW. I think that for each of the elements we show – which is where RMH work is strong – we want to have a test. Similarly for each test, we should be able to set `show=TRUE` to present the results in lots of detail. Does that seem the right kind of objective?

Motivations

This package and vignette arose from a number of considerations:

- There have been many queries on R-help and elsewhere about "unexpected" computational results. An example posting is https://stat.ethz.ch/pipermail/r-help/2017-April/446375.html, which initiated this particular work.
- The R FAQ section 7.31 (R Core Team (2017)) is the essential subject of the most frequent R-help postings. ??Can we get numbers?? I suspect that is an interesting R exercise. JN
- It would be helpful to detect changes that could affect results of any computations with **R**. In particular, we would like to detecth changes between
  - versions of **R**
  - versions of compilers used in **R** or packages (**Fortran**, **C**, **C++**, **Java**, or others)
  - package versions or packages claiming to do the same computations
- As an adjunct to FAQ section 7.31, we would like to provide guidance about numerical computation, with examples available for user experimentation. In this some objectives are to answer questions such as
  - What is the answer, correct to the working precision?
  - What is the best answer available via a prescribed calculation (assuming all steps are carried out to the best limits)?
  - How different is a computed answer from a "correct" result? From the result of the "best available" computational process?
- We would like to suggest "best practice" if we can.

While this article focuses on **R**, we anticipate that some tests will be ported (if they are not already available) to computing languages that **R** may call and which will therefore affect users' results.

Background

- Heiberger and Holland (2015), particularly Appendix G.
- FAQ 7.31 R Core Team (2017) of the **R** documentation
- Gnumeric test xls
- PARANOIA Karpinski (1985)
- Goldberg (1991)
- Rounding Errors in Algebraic Processes, Wilkinson (1963)

Scope

Because arithmetic affects a large range of computations, it is very easy to allow the scope of tests to become too wide. Here we will categorize the possibilities. However, we will defer the decision of where to draw the line around our tests in the expectation that other users of **R** and its packages will raise issues that as yet have not attracted our attentio.

Core arithmetic

Tests of core arithmetic concern the basic arithmetic capabilities of

- fundamental operations of add, subtract, multiply, and divide on the numeric data types in the language.
- the square root (sqrt) function, which is part of the IEEE standard. ?? should we mention ?Arithmetic IEC standard?
- the input of numeric quantities
- the output of numeric quantities
  - for display or printing
  - for reuse in other computations
- those special functions that are "built-in" to some hardware but not part of the floating-point arithmetic standards,
  - some form of logarithm
  - some form of exponential function
  - some form of trigonometric function(s)

Note that a^b may not be supported directly. For example (Palmer (1980) and http://www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/x87.html), the Intel 80x87 family of floating point instructions has the instruction F2XM1 that computes $2^x - 1$, and the functions FYL2X and FYL2XP1 which compute $y * log_2(x)$ and $y * log_2(x + 1)$ respectively. We can combine these instructions to compute a^b, but need to do so carefully for best results. We may also wish to distinguish the case where b is integer.

Note that we consider logical tests on numerical quantities separately below, as these raise some issues of programming that need to be addressed.

The numeric data types in **R** that we shall consider are: – numeric (same as double in R) – integer (??R seems to have just 32 bit integers??) – ??any others: pracma, gsl, ...

Specialized arithmetic

There are a number of specialized tools to make use of features of computing hardware that may be available. These complicate our testing, and may require a large amount of extra effort, so for the moment, this

project currently has no tests relating to the computational tools mentioned in this section. However, to the extent that we can do so easily, our tests should be capable of at least recognizing the existence of such features, rendering them amenable to extension. The following are three categories of specialized arithmetic capabilities:

- parallel processing where the order of computation is potentially uncertain
- graphical processing unit (GPUs), which are increasingly used to perform computations e.g., http://www.cs.unc.edu/~ibr/projects/paranoia/
- specialized processing capabilities (database machines, quantum computing, random number devices, others?)

Many **R** users are interested in parallel operations, so the first of these categories is likely the most important to address first. However, the use of GPUs (so-called Graphical Processing Units) is becoming more common. These often have limited precision for storage and arithmetic. Very particular tools in the third category will only be considered in the more distant future, if at all, though others may wish to attempt to port our tests.

Special functions within the language

We have already noted trigonometric, log and power functions above. Trigonometric functions are part of the **R** language, and are worth testing as errors can have important implications in real life, as they are critical to computing navigational information. (??do we want refs, such as Kahan's "Arithmetic written in sand")

What kinds of tests should we consider? In no particular order (??may want to organize later)

- tests of principal range of inverse trigonometric functions. These should be testable fairly easily within **R**.

```
xx <- (-40:40)/40
#   xx
   ss <- asin(xx) # check if in -pi/2, pi/2
   if (all(ss >= -pi/2) && all(ss <= pi/2)) cat("asin OK") else stop("asin BAD")
```

```
## asin OK
```

```
   cc <- acos(xx) # check if in 0, pi
   if (all(cc >= 0) && all(cc <= pi)) cat("acos OK") else stop("acos BAD")
```

```
## acos OK
```

```
   tt <- atan(xx) # check if in -pi/2, pi/2
   if (all(tt >= -pi/2) && all(tt<= pi/2)) cat("atan OK") else stop("atan BAD")
```

```
## atan OK
```

- ??similar for hyperbolic functions
- ?? Do we want to consider implications of arithmetic (storage modes) for random number generation. JN suggests no, but keeping a list of potential issues for later consideration.

??notes: R: ?Arithmetic ?groupGeneric

Special functions in packages

?? What packages generate special functions? Which are worth testing?

Sums and products

Sums and products are used for many purposes in computation. They are also prone to a number of possible failures when the physical limits of the computational system are exceeded. Such limits include the precision of numbers, the size of index quantities and the available memory.

A particular case of sums that is important for statistical computation is that of finding the variance of a set of numbers. As that example is long enough to require considerable discussion, we have placed it in a separate section.

- sums and products
  - ordering
  - other strategies
  - good test cases??

Logical tests – `if` statements

- program flow – convergence and termination tests
- sorting
- ?? others

Concerns: - optimizing compilers - setting of tolerances - "Fuzz" or equivalent (get ref to Tektronix e.g. Nash 1978) - other?

R structures for floating-point operations and display

- "+" "-" "*" "/" basic ops
- "<-"
- "as." for integer and (double / numeric), noting the equivalence of latter two
- dput(), dget()
- print, sprintf, ??
- features of Rmpfr (especially formatBin, formatDec, formatHex)

How tests and reports should be presented

Tests against stored `standard` results

Numerical results

Error condition results

Some computations should raise error conditions. Therefore it is sensible for us to include some tests of this nature. The Gnome Office project has a spreadsheet processor **Gnumeric** and one of us (JCN) prepared a test spreadsheet `trig.xls` (https://projects-old.gnome.org/gnumeric/func-tests/trig.xls) that presents many "bad" inputs.

Verifying internal nature of R objects

- str()

- need to show bit pattern / hex pattern ??
- all.equal()
- identical()
- Rmpfr functions ??

Other useful features of R for floating-point testing

Known issues ??

Tests using identities

There are a number of tests we can create that use mathematical identities that may or may not be satisfied in floating point arithmetic.

Square of square root

The test script `isqrt.R` uses the identity

```
$(sqrt(x)) ^ 2 == x$
```

to test this identity for the sequence of integers 0:49. These are established using integer input, and verified using the **R** function `str()` to display the object structure.

Computation of means and variances

JN to RMH: Do you think I should do a separate vignette and just extract a summary here?

Means and variances are good test cases for arithmetic.

- they only involve the four basic arithmetic operations. Even if the standard deviation is desired, the only extra function required is the square root.
- it is quite easy to devise cases for which the answer can be computed analytically, that is, by formula NOT requiring a summation
- such cases can relatively easily be devised to give rise to errors in standard computational methods
- there are many possible algorithms for computing variance.

Definitions

We will define the mean of a set of numbers $x(i), i = 1, ..., n$ as

$$mean(x) = \sum_{i=1}^{n} (x_i)/n$$

and variance as

$$var(x) = \sum_{i=1}^{n} ((x_i - mean(x))^2)/(n-1)$$

As far as we can tell, **R** uses essentially this definition (see the code in R-3.4.0/src/library/stats/src/cov.c). Note that the definition is of the so-called **sample** variance where the divisor is

$$n - 1$$

. A simpler **population** variance uses the divisor n and will be called $var_p(x)$. Clearly

$$var(x) = (n/(n-1)) * var_p(x)$$

Other choices

It is easy to show algebraically that

$$var_p(x) = \sum_{i=1}^{n}((x_i - mean(x))^2)/n = mean(x^2) - (mean(x))^2$$

This approach is attractive in that we can do our sum of x and x^2 at the same time, which is convenient for long data series or for data arriving as a **stream**. The formula, in words "the mean of the square minus the square of the mean", is called the **textbook formula**. It has, unfortunately, serious computational disadvantages, since we are often forced to subtract nearly equal quantities.

Note, however, that subtracting the shift s from all the numbers in our set gives a shifted mean

$$mean(x - s) = mean(x) - s$$

but the variance is unaltered. If we can choose s so that it is close to $mean(x)$, then we avoid digit cancellation. The awkward issue is choosing s well.

The accumulations for both the mean and variance are subject to digit loss errors if the current accumulated sum is very large relative to the next element of the data. A simple example of this is the addition of 1 cent to a million dollars in an 8-digit decimal calculator. $1,000,000.00 needs **9** digits for precision to 1 cent. The addition (without guard digits) loses the penny.

A better way to do the accumulation is to sort the data and sum from smallest to largest. Sorting is a slow operation relative to the summation, and requires that our data be stored. As we have noted, some data is received in a stream and the whole series is never saved, though we must be able to keep a count of the number of elements so we have n.

Issues to test

From the above, we clearly have three matters of concern:

- that we have sufficient integer or index precision to hold n
- that we can avoid digit loss that may occur when accumulating a relatively small data element into a partial sum
- that we can avoid digit cancellation in computing the variance in a one-pass method.

The first two concerns here can only be addressed by having data storage structures of sufficient precision. For the mean, this may imply an extended precision accumulator rather than a universally large number of digits for numbers.

Variance algorithms

Many methods for computing the variance - 2 pass method as per the definition - variants on 2-pass to improve the accuracy of the result (??explain) - 1 pass using textbook formula - shifted 1 pass method - Kahan - other fixes - a pairwise or stack approach

Mean and variance test data

In Appendix A, mathematical formulas for the mean and variance of simple sequences of numbers and of geometric progressions are given. Clearly we can multiply this data by a scale q or shift it by s to force size or input errors. Moreover, the **R** function `rev()` allows the order of the data to be reversed so we can test for digit loss in accumulation.

How to extend the tests

Discussion and open issues

Index of tests by name

isqrt: squares of square roots of integers

Appendix A

Some special tests for variance testing.

I. Sequences of integers

The sequence $1 : n$ has a sum of

$$n * (n + 1)/2$$

and a sum of squares of

$$n * (n + 1) * (2 * n + 1)/6$$

The mean is therefore $ (n + 1) / 2 $ and the variance is

$$(n/(n - 1)) * ((n + 1) * (2 * n + 1)/6 - (n + 1) * (n + 1)/4)$$

which simplifies to

$$(n/(n - 1)) * (2n^2 + 3n + 1)/6 - (n^2 + 2n + 1)/4)$$

$$= (n/(n - 1)) * (4n^2 + 6n + 2 - 3n^2 - 6n - 3)/12$$

$$= (n/(n - 1)) * (n^2 - 1)/12$$

$$= (n/(n - 1) * (n + 1) * (n - 1)/12)$$

$$= n * (n + 1)/12$$

Since the variance is unaffected by a $shift$, the sequence $(shift + 1) : (shift + n)$ has mean $shift + (n + 1)/2$ and the same variance.

II. Geometric progressions

A geometric progression is the sequence of n numbers $a, a * r, a * r^2, ..., a * r^(n - 1)$.

The sum is

$$S = a * (1 - r^n)/(1 - r) = a * (r^n - 1)/(r - 1)$$

We note that the squares of the elements of a geometric progression form a geometric progression also. Thus we have an analytic formula for both the sum and the sum of squares and can apply algebraically rather than computationally the "textbook" formula for the variance, namely $ n/(n - 1)$ times the "average of the squares minus the square of the average". Since we are doing the sum exactly, the textbook formula is not problematic, though we should be careful to evaluate the final formula in a stable manner. One form of the result is

$$(a^2) * (r^(2 * n) * ((n - 1) * r - (n + 1)) + r^n * 2 * (r + 1) - (n + 1) * r + (n - 1))/(n * (n - 1) * (r - 1) * (r^2 - 1))$$

References

Goldberg, David. 1991. "What Every Computer Scientist Should Know About Floating-Point Arithmetic." *ACM Computing Surveys*, 5–48. http://www.validlab.com/goldberg/paper.pdf.

Heiberger, Richard M., and Burt Holland. 2015. *Statistical Analysis and Data Display: An Intermediate Course with Examples in R*. Second. Springer-Verlag, New York. http://www.springer.com/978-1-4939-2121-8.

Karpinski, Richard. 1985. "Paranoia: A Floating-Point Benchmark." *Byte Magazine* 10 (2): 223–35.

Palmer, John F. 1980. "The Intel&Reg; 8087 Numeric Data Processor." In *Proceedings of the May 19-22, 1980, National Computer Conference*, 887–93. AFIPS '80. New York, NY, USA: ACM. doi:10.1145/1500518.1500674.

R Core Team. 2017. Vienna, Austria: R Foundation for Statistical Computing. https://cran.r-project.org/doc/FAQ/R-FAQ.html#Why-doesn_0027t-R-think-these-numbers-are-equal_003f.

Wilkinson, James H. 1963. *Rounding Errors in Algebraic Processes*. London: Her Majesty's Stationery Office.