Richard M. Heiberger • Burt Holland

Statistical Analysis and Data Display

An Intermediate Course with Examples in R

Second Edition

Springer

Richard M. Heiberger
Department of Statistics
Temple University
Philadelphia, PA, USA

Burt Holland
Department of Statistics
Temple University
Philadelphia, PA, USA

ISSN 1431-875X ISSN 2197-4136 (electronic)
Springer Texts in Statistics
ISBN 978-1-4939-2121-8 ISBN 978-1-4939-2122-5 (eBook)
DOI 10.1007/978-1-4939-2122-5

Library of Congress Control Number: 2015945945

Springer New York Heidelberg Dordrecht London

Printed on acid-free paper

Springer Science+Business Media LLC New York is part of Springer Science+Business Media (www.springer.com)

Appendix G
Computational Precision and Floating-Point Arithmetic

Computers use *floating point* arithmetic. The floating point system is not identical to the real-number system that we (teachers and students) know well, having studied it from kindergarten onward. In this section we show several examples to illustrate and emphasize the distinction.

The principal characteristic of real numbers is that we can have as many digits as we wish. The principal characteristic of floating point numbers is that we are limited in the number of digits that we can work with. In double-precision IEEE 754 arithmetic, we are limited to exactly 53 binary digits (approximately 16 decimal digits)

The consequences of the use of floating point numbers are pervasive, and present even with numbers we normally think of as far from the boundaries. For detailed information please see FAQ 7.31 in file

```
system.file("../../doc/FAQ")
```

The help menus in **Rgui** in Windows and **R.app** on Macintosh have direct links to the FAQ file.

G.1 Examples

Let us start by looking at two simple examples that require basic familiarity with floating point arithmetic.

1. Why is .9 not recognized to be the same as (.3 + .6)?
   Table G.1 shows that .9 is not perceived to have the same value as .3 + .6, when calculated in floating point (that is, when calculated by a computer). The difference between the two values is not 0, but is instead a number on the order of

machine epsilon (the smallest number ϵ such that $1 + \epsilon > 1$). In R, the standard mathematical comparison operators recognize the difference. There is a function `all.equal` which tests for *near equality*. See `?all.equal` for details.

Table G.1 Calculations showing that the floating point numbers .9 and .3 + .6 are not stored the same inside the computer. R comparison operators recognize the numbers as different.

```
> c(.9, (.3 + .6))
[1] 0.9 0.9

> .9 == (.3 + .6)
[1] FALSE

> .9 - (.3 + .6)
[1] 1.11e-16

> identical(.9, (.3 + .6))
[1] FALSE

> all.equal(.9, (.3 + .6))
[1] TRUE
```

Table G.2 $\left(\sqrt{2}\right)^2 \neq 2$ in floating point arithmetic inside the computer.

```
> c(2, sqrt(2)^2)
[1] 2 2

> sqrt(2)^2
[1] 2

> 2 == sqrt(2)^2
[1] FALSE

> 2 - sqrt(2)^2
[1] -4.441e-16

> identical(2, sqrt(2)^2)
[1] FALSE

> all.equal(2, sqrt(2)^2)
[1] TRUE
```

2. Why is $\left(\sqrt{2}\right)^2$ not recognized to be the same as 2?

   Table G.2 shows that the difference between the two values $\left(\sqrt{2}\right)^2$ and 2 is not 0, but is instead a number on the order of machine epsilon (the smallest number ϵ such that $1 + \epsilon > 1$).

We will pursue these examples further in Section G.7, but first we need to introduce floating point numbers—the number system used inside the computer.

G.2 Floating Point Numbers in the IEEE 754 Floating-Point Standard

The number system we are most familiar with is the infinite-precision base-10 system. Any number can be represented as the infinite sum

$$\pm\,(a_0 \times 10^0 + a_1 \times 10^{-1} + a_2 \times 10^{-2} + \ldots) \times 10^p$$

where p can be any positive integer, and the values a_i are digits selected from decimal digits $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. For example, the decimal number 3.3125 is expressed as

$$\begin{aligned}3.3125 &= (3 \times 10^0 + 3 \times 10^{-1} + 1 \times 10^{-2} + 2 \times 10^{-3} + 5 \times 10^{-4}) \times 10^0\\ &= 3.3125 \times 1\end{aligned}$$

In this example, there are 4 decimal digits after the radix point. There is no limit to the number of digits that could have been specified. For decimal numbers the term *decimal point* is usually used in preference to the more general term *radix point*.

Floating point arithmetic in computers uses a finite-precision base-2 (binary) system for representation of numbers. Most computers today use the 53-bit IEEE 754 system, with numbers represented by the finite sum

$$\pm\,(a_0 \times 2^0 + a_1 \times 2^{-1} + a_2 \times 2^{-2} + \ldots + a_{52} \times 2^{\bar{}}52) \times 2^p$$

where p is an integer in the range -1022 to 1023 (expressed as decimal numbers), the values a_i are digits selected from $\{0, 1\}$, and the subscripts and powers i are decimal numbers selected from $\{0, 1, \ldots, 52\}$. The decimal number 3.125_{10} is 11.0101_2 in binary.

$$\begin{aligned}3.3125_{10} = 11.0101_2 &= (1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} +\\ &\qquad 1 \times 2^{-5}) \times 2_{10}\\ &= 1.10101_2 \times 2_{10}\end{aligned}$$

This example (in the normalized form $1.10101_2 \times 2_{10}$) has five binary digits (bits) after the radix point (binary point in this case). There is a maximum of 52 binary positions after the binary point.

Strings of 0 and 1 are hard for people to read. They are usually collected into units of 4 bits (called a byte).

The IEEE 754 standard requires the base $\beta = 2$ number system with $p = 53$ base-2 digits. Except for 0, the numbers in internal representation are always *normalized* with the leading bit always 1. Since it is always 1, there is no need to store it and only 52 bits are actually needed for 53-bit precision. A string of `0` and `1` is difficult for humans to read. Therefore every set of 4 bits is represented as a single hexadecimal digit, from the set {`0 1 2 3 4 5 6 7 8 9 a b c d e f`}, representing the decimal values {0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15}. The 52 stored bits can be displayed with 13 hex digits. Since the base is $\beta = 2$, the exponent of an IEEE 754 floating point number must be a power of 2. The double-precision computer numbers contain 64 bits, allocated 52 for the significant, 1 for the sign, and 11 for the exponent. The 11 bits for the exponent can express $2^{11} = 2048$ unique values. These are assigned to range from -2^{-1022} to 2^{1023}, with the remaining 2 exponent values used for special cases (zero and the *special quantities* NaN and ∞).

The number 3.3125_{10} is represented in hexadecimal (base-16) notation as

$$\begin{aligned} 3.3125_{10} = 3.50_{16} &= (1 \times 16^0 + a_{16} \times 16^{-1} + 8_{16} \times 16^{-2}) \times 2_{10} \\ &= 1.a8_{16} \times 2_{10} \\ &= 1.10101000_2 \times 2_{10} \end{aligned}$$

There are two hex digits after the radix point (binary point, not hex point because the normalization is by powers of 2_{10} not powers of 16_{10}).

The R function `sprintf` is used to specify the printed format of numbers. The letter `a` in format `sprintf("%+13.13a", x)` tells R to print the numbers in hexadecimal notation. The "13"s say to use 13 hexadecimal digits after the binary point. See `?sprintf` for more detail on the formatting specifications used by the `sprintf` function. Several numbers, simple decimals, and simple multiples of powers of 1/2 are shown in Table G.3 in both decimal and binary notation.

G.3 Multiple Precision Floating Point

The R package **Rmpfr** allows the construction and use of arbitrary precision floating point numbers. It was designed, and is usually used, for higher-precision arithmetic—situations where the 53-bit double-precision numbers are not precise

Table G.3 Numbers, some simple integers divided by 10, and some fractions constructed as multiples of powers of 1/2. The $i/10$ decimal numbers are stored as repeating binaries in the hexadecimal notation until they run out of digits. There are only 52 bits (binary digits) after the binary point. For decimal input 0.1 we see that the repeating hex digit is "9" until the last position where it is rounded up to "*a*".

```
> nums <- c(0, .0625, .1, .3, .3125, .5, .6, (.3 + .6), .9,  1)

> data.frame("decimal-2"=nums,
+            "decimal-17"=format(nums, digits=17),
+            hexadecimal=sprintf("%+13.13a", nums))
   decimal.2          decimal.17          hexadecimal
1     0.0000 0.00000000000000000 +0x0.0000000000000p+0
2     0.0625 0.06250000000000000 +0x1.0000000000000p-4
3     0.1000 0.10000000000000001 +0x1.999999999999ap-4
4     0.3000 0.29999999999999999 +0x1.3333333333333p-2
5     0.3125 0.31250000000000000 +0x1.4000000000000p-2
6     0.5000 0.50000000000000000 +0x1.0000000000000p-1
7     0.6000 0.59999999999999998 +0x1.3333333333333p-1
8     0.9000 0.89999999999999991 +0x1.ccccccccccccccp-1
9     0.9000 0.90000000000000002 +0x1.cccccccccccccdp-1
10    1.0000 1.00000000000000000 +0x1.0000000000000p+0
```

enough. In this Appendix we use it for lower-precision arithmetic—four or five significant digits. In this way it will be much easier to illustrate how the behavior of floating point numbers differs from the behavior of real numbers.

G.4 Binary Format

It is often easier to see the details of the numerical behavior when numbers are displayed in binary, not in the hex format of `sprintf("%+13.13a", x)`. The **Rmpfr** package includes a binary display format for numbers. The `formatBin` function uses `sprintf` to construct a hex display format and then modifies it by replacing each hex character with its 4-bit expansion as shown in Table G.4.

Optionally (with argument `scientific=FALSE`), all binary numbers can be formatted to show aligned radix points. There is also a `formatHex` function which is essentially a wrapper for `sprintf`. Both functions are used in the examples in this Appendix. Table G.5 illustrates both functions, including the optional `scientific` argument, with a 4-bit arithmetic example.

Table G.4 Four-bit expansions for the sixteen hex digits. We show both lowercase [a:f] and uppercase [A:F] for the hex digits.

```
> Rmpfr:::HextoBin
            1      2      3      4      5      6      7
"0000" "0001" "0010" "0011" "0100" "0101" "0110" "0111"
     8      9      A      B      C      D      E      F
"1000" "1001" "1010" "1011" "1100" "1101" "1110" "1111"
     a      b      c      d      e      f
"1010" "1011" "1100" "1101" "1110" "1111"
```

G.5 Round to Even

The IEEE 754 standard calls for "rounding ties to even". The explanation here is from `help(mpfr, package="Rmpfr")`:

> The *round to nearest* (`"N"`) mode, the default here, works as in the IEEE 754 standard: in case the number to be rounded lies exactly in the middle of two representable numbers, it is rounded to the one with the least significant bit set to zero. For example, the number 5/2, which is represented by (10.1) in binary, is rounded to (10.0)=2 with a precision of two bits, and not to (11.0)=3. This rule avoids the *drift* phenomenon mentioned by Knuth in volume 2 of *The Art of Computer Programming* (Section 4.2.2).

G.6 Base-10, 2-Digit Arithmetic

Hex numbers are hard to fathom the first time they are seen. We therefore look at a simple example of finite-precision arithmetic with 2 significant decimal digits.

Calculate the sum of squares of three numbers in 2-digit base-10 arithmetic. For concreteness, use the example

$$2^2 + 11^2 + 15^2$$

This requires rounding to 2 significant digits at *every* intermediate step. The steps are easy. Putting your head around the steps is hard.

We rewrite the expression as a fully parenthesized algebraic expression, so we don't need to worry about precedence of operators at this step.

$$((2^2) + (11^2)) + (15^2)$$

Now we can evaluate the parenthesized groups from the inside out.

Table G.5 Integers from 0 to 39 stored as 4-bit **mpfr** numbers. The numbers from 17 to 39 are rounded to four significant bits. All numbers in the "16" and "24" columns are multiples of 2, and all numbers in the "32" columns are multiples of 4. The numbers are displayed in decimal, hex, binary, and binary with aligned radix points. To interpret the aligned binary numbers, replace the "_" placeholder character with a zero.

```
> library(Rmpfr)

> FourBits <- mpfr(matrix(0:39, 8, 5), precBits=4)

> dimnames(FourBits) <- list(0:7, c(0,8,16,24,32))

> FourBits
'mpfrMatrix' of dim(.) =  (8, 5) of precision  4   bits
     0    8   16   24   32
0 0.00 8.00 16.0 24.0 32.0
1 1.00 9.00 16.0 24.0 32.0
2 2.00 10.0 18.0 26.0 32.0
3 3.00 11.0 20.0 28.0 36.0
4 4.00 12.0 20.0 28.0 36.0
5 5.00 13.0 20.0 28.0 36.0
6 6.00 14.0 22.0 30.0 40.0
7 7.00 15.0 24.0 32.0 40.0

> formatHex(FourBits)
  0         8         16        24        32
0 +0x0.0p+0 +0x1.0p+3 +0x1.0p+4 +0x1.8p+4 +0x1.0p+5
1 +0x1.0p+0 +0x1.2p+3 +0x1.0p+4 +0x1.8p+4 +0x1.0p+5
2 +0x1.0p+1 +0x1.4p+3 +0x1.2p+4 +0x1.ap+4 +0x1.0p+5
3 +0x1.8p+1 +0x1.6p+3 +0x1.4p+4 +0x1.cp+4 +0x1.2p+5
4 +0x1.0p+2 +0x1.8p+3 +0x1.4p+4 +0x1.cp+4 +0x1.2p+5
5 +0x1.4p+2 +0x1.ap+3 +0x1.4p+4 +0x1.cp+4 +0x1.2p+5
6 +0x1.8p+2 +0x1.cp+3 +0x1.6p+4 +0x1.ep+4 +0x1.4p+5
7 +0x1.cp+2 +0x1.ep+3 +0x1.8p+4 +0x1.0p+5 +0x1.4p+5

> formatBin(FourBits)
  0           8           16          24          32
0 +0b0.000p+0 +0b1.000p+3 +0b1.000p+4 +0b1.100p+4 +0b1.000p+5
1 +0b1.000p+0 +0b1.001p+3 +0b1.000p+4 +0b1.100p+4 +0b1.000p+5
2 +0b1.000p+1 +0b1.010p+3 +0b1.001p+4 +0b1.101p+4 +0b1.000p+5
3 +0b1.100p+1 +0b1.011p+3 +0b1.010p+4 +0b1.110p+4 +0b1.001p+5
4 +0b1.000p+2 +0b1.100p+3 +0b1.010p+4 +0b1.110p+4 +0b1.001p+5
5 +0b1.010p+2 +0b1.101p+3 +0b1.010p+4 +0b1.110p+4 +0b1.001p+5
6 +0b1.100p+2 +0b1.110p+3 +0b1.011p+4 +0b1.111p+4 +0b1.010p+5
7 +0b1.110p+2 +0b1.111p+3 +0b1.100p+4 +0b1.000p+5 +0b1.010p+5

> formatBin(FourBits, scientific=FALSE)
  0             8             16            24            32
0 +0b_____0.000 +0b__1000.___ +0b_1000_.___ +0b_1100_.___ +0b1000__.___
1 +0b_____1.000 +0b__1001.___ +0b_1000_.___ +0b_1100_.___ +0b1000__.___
2 +0b____10.00_ +0b__1010.___ +0b_1001_.___ +0b_1101_.___ +0b1000__.___
3 +0b____11.00_ +0b__1011.___ +0b_1010_.___ +0b_1110_.___ +0b1001__.___
4 +0b___100.0__ +0b__1100.___ +0b_1010_.___ +0b_1110_.___ +0b1001__.___
5 +0b___101.0__ +0b__1101.___ +0b_1010_.___ +0b_1110_.___ +0b1001__.___
6 +0b___110.0__ +0b__1110.___ +0b_1011_.___ +0b_1111_.___ +0b1010__.___
7 +0b___111.0__ +0b__1111.___ +0b_1100_.___ +0b1000__.___ +0b1010__.___
```

$$\begin{array}{cccccccl}
(& (2^2) & + & (11^2) &) & + & (15^2) & \text{\#\# parenthesized expression} \\
(& (4) & + & (121) &) & + & (225) & \text{\#\# square each term} \\
(& 4 & + & 120 &) & + & 220 & \text{\#\# round each term to two significant decimal digits} \\
(& & 124 & &) & + & 220 & \text{\#\# calculate the intermediate sum} \\
(& & 120 & &) & + & 220 & \text{\#\# round the intermediate sum to two decimal digits} \\
 & & & & & 340 & & \text{\#\# sum the terms}
\end{array}$$

Compare this to the full precision arithmetic

$$\begin{array}{cccccccl}
(& (2^2) & + & (11^2) &) & + & (15^2) & \text{\#\# parenthesized expression} \\
(& 4 & + & 121 &) & + & 225 & \text{\#\# square each term} \\
(& & 125 & &) & + & 225 & \text{\#\# calculate the intermediate sum} \\
 & & & & & 350 & & \text{\#\# sum the terms}
\end{array}$$

We see immediately that two-decimal-digit rounding at each stage gives an answer that is not the same as the one from familiar arithmetic with real numbers.

G.7 Why Is .9 Not Recognized to Be the Same as (.3 + .6)?

We can now continue with the first example from Section G.1. The floating point binary representation of 0.3 and the floating point representation of 0.6 must be aligned on the binary point before the addition. When the numbers are aligned by shifting the smaller number right one position, the last bit of the smaller number has nowhere to go and is lost. The sum is therefore one bit too small compared to the floating point binary representation of 0.9. Details are in Table G.6.

G.8 Why Is $\left(\sqrt{2}\right)^2$ Not Recognized to Be the Same as 2?

We continue with the second example from Section G.1. The binary representation inside the machine of the two numbers $\left(\sqrt{2}\right)^2$ and 2 is not identical. We see in Table G.7 that they differ by one bit in the 53rd binary digit.

G.9 `zapsmall` to Round Small Values to Zero for Display

R provides a function that rounds small values (those close to the machine epsilon) to zero. We use this function for printing of many tables where we wish to interpret numbers close to machine epsilon as if they were zero. See Table G.8 for an example.

Table G.6 Now let's add 0.3 and 0.6 in hex:

```
0.3  +0x1.3333333333333p-2 = +0x0.9999999999999p-1  aligned binary (see below)
0.6  +0x1.3333333333333p-1 = +0x1.3333333333333p-1
--------------------------   ---------------------
0.9  add of aligned binary   +0x1.cccccccccccccp-1
0.9  convert from decimal    +0x1.ccccccccccccdp-1
```

We need to align binary points for addition. The shift is calculated by converting hex to binary, shifting one bit to the right to get the same `p-1` exponent, regrouping four bits into hex characters, and allowing the last bit to fall off:
`1.0011 0011 0011 ... 0011` $\times 2^{-2} \rightarrow$ `.1001 1001 1001 ... 1001 |` $\cancel{1} \times 2^{-1}$

```
> nums369 <- c(.3, .6, .3+.6, 9)

> nums369df <-
+ data.frame("decimal-2"=nums369,
+            "decimal-17"=format(nums369, digits=17),
+            hexadecimal=sprintf("%+13.13a", nums369))

> nums369df[3,1] <- "0.3 + 0.6"

> nums369df
  decimal.2          decimal.17          hexadecimal
1       0.3 0.29999999999999999 +0x1.3333333333333p-2
2       0.6 0.59999999999999998 +0x1.3333333333333p-1
3 0.3 + 0.6 0.89999999999999991 +0x1.cccccccccccccp-1
4         9 9.00000000000000000 +0x1.2000000000000p+3
```

Table G.7 The binary representation of the two numbers $\left(\sqrt{2}\right)^2$ and 2 is not identical. They differ by one bit in the 53rd binary digit.

```
> sprintf("%+13.13a", c(2, sqrt(2)^2))
[1] "+0x1.0000000000000p+1" "+0x1.0000000000001p+1"
```

Table G.8 We frequently wish to interpret numbers that are very different in magnitude as if the smaller one is effectively zero. The display function `zapsmall` provides that capability.

```
> c(100, 1e-10)
[1] 1e+02 1e-10

> zapsmall(c(100, 1e-10))
[1] 100   0
```

G.10 Apparent Violation of Elementary Factoring

We show a simple example of disastrous cancellation (loss of high-order digits), where the floating point statement

$$a^2 - b^2 \neq (a+b) \times (a-b)$$

is an inequality, not an equation, for some surprising values of a and b. Table G.9 shows two examples, a decimal example for which the equality holds so we can use our intuition to see what is happening, and a hex example at the boundary of rounding so we can see precisely how the equality fails.

Table G.9 Two examples comparing $a^2 - b^2$ to $(a + b) \times (a - b)$. On the top, the numbers are decimal $a = 101$ and b=102 and the equality holds on a machine using IEEE 754 floating point arithmetic. On the bottom, the numbers are hexadecimal $a =$ `0x8000001` and $b =$ `0x8000002` and the equality fails to hold on a machine using IEEE 754 floating point arithmetic. The outlined $\boxed{0}$ in the decimal column for `a^2` with `x=+0x8000000` would have been a $\boxed{1}$ if we had 54-bit arithmetic. Since we have only 53 bits available to store numbers, the 54th bit was rounded to 0 by the Round to Even rule (see Section G.5). The marker $\updownarrow$ in the hex column for `a^2` with `x=+0x8000000` shows that one more hex digit would be needed to indicate the squared value precisely.

	Decimal	Hex
	100 =	+0x64
x	100	+0x1.9000000000000p+6
a <- x+1	101	+0x1.9400000000000p+6
b <- x+2	102	+0x1.9800000000000p+6
a^2	10201	+0x1.3ec8000000000p+13
b^2	10404	+0x1.4520000000000p+13
b^2 - a^2	203	+0x1.9600000000000p+7
(b+a) * (b-a)	203	+0x1.9600000000000p+7

	Decimal	Hex
	134217728 =	+0x8000000
x	134217728	+0x1.0000000000000p+27
a <- x+1	134217729	+0x1.0000002000000p+27
b <- x+2	134217730	+0x1.0000004000000p+27
a^2	1801439877791744$\boxed{0}$	+0x1.000000400000$\overset{\updownarrow}{0}$p+54
b^2	18014399046352900	+0x1.0000008000001p+54
b^2 - a^2	268435460	+0x1.0000004000000p+28
(b+a) * (b-a)	268435459	+0x1.0000003000000p+28

G.11 Variance Calculations

Once we understand disastrous cancellation, we can study algorithms for the calculation of variance. Compare the two common formulas for calculating sample variance, the two-pass formula and the disastrous one-pass formula.

Two-pass formula	One-pass formula
$\left(\sum_{i=1}^{n}(x_i - \bar{x})^2\right)/(n-1)$	$\left(\left(\sum_{i=1}^{n} x_i^2\right) - n\bar{x}^2\right)/(n-1)$

Table G.10 shows the calculation of the variance by both formulas. For $x = (1, 2, 3)$, $\text{var}(x) = 1$ by both formulas. For $x = (k+1, k+2, k+3)$, $\text{var}(x) = 1$ by both formulas for $k \le 10^7$. For $k = 10^8$, the one-pass formula gives 0. The one-pass formula is often shown in introductory books with the name "machine formula". The "machine" it is referring to is the desk calculator, not the digital computer. The one-pass formula gives valid answers for numbers with only a few significant figures (about half the number of digits for machine precision), and therefore does not belong in a general algorithm. The name "one-pass" is reflective of the older computation technology where scalars, not the vector, were the fundamental data unit. See Section G.14 where we show the one-pass formula written with an explicit loop on scalars.

We can show what is happening in these two algorithms by looking at the binary display of the numbers. We do so in Section G.12 with presentations in Tables G.11 and G.12. Table G.11 shows what happens for double precision arithmetic (53 significant bits, approximately 16 significant decimal digits). Table G.12 shows the same behavior with 5-bit arithmetic (approximately 1.5 significant decimal digits).

G.12 Variance Calculations at the Precision Boundary

Table G.11 shows the calculation of the sample variance for three sequential numbers at the boundary of precision of 53-bit floating point numbers. The numbers in column "15" fit within 53 bits and their variance is the variance of $k + (1, 2, 3)$ which is 1. The numbers $10^{16} + (1, 2, 3)$ in column "16" in Table G.11 require 54 bits for precise representation. They are therefore rounded to $10^{16} + c(0, 2, 4)$ to fit within the capabilities of 53-bit floating point numbers. The variance of the numbers in column "16" is calculated as the variance of $k + (0, 2, 4)$ which is 4. When we place the numbers into a 54-bit representation (not possible with the standard 53-bit floating point), the calculated variance is the anticipated 1.

Table G.12 shows the calculation of the sample variance for three sequential numbers at the boundary of precision of 5-bit floating point numbers. The numbers {33, 34, 35} on the left side need six significant bits to be represented precisely.

Table G.10 The one-pass formula fails at $x = (10^8 + 1, 10^8 + 2, 10^8 + 3)$ (about half as many significant digits as machine precision). The two-pass formula is stable to the limit of machine precision. The calculated value at the boundary of machine precision for the two-pass formula is the correctly calculated floating point value. Please see Section G.12 and Tables G.11 and G.12 for the explanation.

```
> varone <- function(x) {
+   n <- length(x)
+   xbar <- mean(x)
+   (sum(x^2) - n*xbar^2) / (n-1)
+ }

> x <- 1:3

> varone(x)
[1] 1

> varone(x+10^7)
[1] 1

> ## half machine precision
> varone(x+10^8)
[1] 0

> varone(x+10^15)
[1] 0

> ## boundary of machine precision
> ##
> varone(x+10^16)
[1] 0

> varone(x+10^17)
[1] 0
```

```
> vartwo <- function(x) {
+   n <- length(x)
+   xbar <- mean(x)
+   sum((x-xbar)^2) / (n-1)
+ }

> x <- 1:3

> vartwo(x)
[1] 1

> vartwo(x+10^7)
[1] 1

> ## half machine precision
> vartwo(x+10^8)
[1] 1

> vartwo(x+10^15)
[1] 1

> ## boundary of machine precision
> ## See next table.
> vartwo(x+10^16)
[1] 4

> vartwo(x+10^17)
[1] 0
```

Following the Round to Even rule, they are rounded to {32, 34, 36} in the five-bit representation on the right side. The easiest way to see the rounding is in the `scientific=FALSE` binary presentation (the last section on both sides of the Table G.12, repeated in Table G.13).

There is also a third formula, with additional protection against cancellation, called the "corrected two-pass algorithm".

$$y = x - \bar{x}$$
$$\sum(y - \bar{y})^2/(n - 1)$$

We define a function in Table G.14 and illustrate its use in a very tight boundary case (the 54-bit column "16" of Table G.11) in Table G.15.

Table G.11 Variance of numbers at the boundary of 53-bit double precision arithmetic. Column "15" fits within 53 bits and the variance is calculated as 1. Column "16" requires 54 bits for precise representation. With 53-bit floating point arithmetic the variance is calculated as 4. With the extended precision to 54 bits, the variance is calculated as 1.

```
> x <- 1:3; p <- 15:16

> xx <- t(outer(10^p, x, ‘+‘)); dimnames(xx) <- list(x, p)

> print(xx, digits=17)
                15                16
1 1000000000000001 10000000000000000
2 1000000000000002 10000000000000002
3 1000000000000003 10000000000000004

> formatHex(xx)
  15                     16
1 +0x1.c6bf526340008p+49 +0x1.1c37937e08000p+53
2 +0x1.c6bf526340010p+49 +0x1.1c37937e08001p+53
3 +0x1.c6bf526340018p+49 +0x1.1c37937e08002p+53

> var(xx[,"15"])
[1] 1

> var(xx[,"16"])
[1] 4

> x54 <- mpfr(1:3, 54)

> xx54 <- t(outer(10^p, x54, ‘+‘)); dimnames(xx54) <- list(x, p)

> xx54
'mpfrMatrix' of dim(.) =  (3, 2) of precision  54   bits
                   15                   16
1 1000000000000001.00 10000000000000001.0
2 1000000000000002.00 10000000000000002.0
3 1000000000000003.00 10000000000000003.0

> vartwo(xx54[,"16"] - mean(xx54[,"16"]))
1 'mpfr' number of precision  54   bits
[1] 1

> ## hex for 54-bit numbers is not currently available from R.
>

> ## We manually constructed it here.
> formatHex(xx54)  ## We manually constructed this
  15                      16
1 +0x1.c6bf5263400080p+49 +0x1.1c37937e080008p+53
2 +0x1.c6bf5263400100p+49 +0x1.1c37937e080010p+53
3 +0x1.c6bf5263400180p+49 +0x1.1c37937e080018p+53
```

Table G.12 Numbers with six and five significant bits. The six-bit integers 33 and 35 on the left side cannot be expressed precisely with only five significant bits. They are rounded to 32 and 36 in the five-bit representation on the right side. The variance of the numbers {33, 34, 35} is 1. The variance of the numbers {32, 34, 36} is 4. Please see Section G.12 for the discussion of this table. Please see Table G.13 for additional details.

```
> y <- 1:3; q <- 3:5; yy <- t(outer(2^q, y, '+'))

> yy6 <- mpfr(yy, 6); dimnames(yy6) <- list(y, q)

> yy6
'mpfrMatrix' of dim(.) =  (3, 3) of precision  6   bits
     3    4    5
1 9.00 17.0 33.0
2 10.0 18.0 34.0
3 11.0 19.0 35.0

> vartwo(yy6[,"5"])
1 'mpfr' number of precision  53   bits
[1] 1

> formatBin(yy6)
  3             4             5
1 +0b1.00100p+3 +0b1.00010p+4 +0b1.00001p+5
2 +0b1.01000p+3 +0b1.00100p+4 +0b1.00010p+5
3 +0b1.01100p+3 +0b1.00110p+4 +0b1.00011p+5

> formatBin(yy6, scientific=FALSE)
  3            4            5
1 +0b__1001.00 +0b_10001.0_ +0b100001.__
2 +0b__1010.00 +0b_10010.0_ +0b100010.__
3 +0b__1011.00 +0b_10011.0_ +0b100011.__
```

```
> y <- 1:3; q <- 3:5; yy <- t(outer(2^q, y, '+'))

> yy5 <- mpfr(yy, 5); dimnames(yy5) <- list(y, q)

> yy5
'mpfrMatrix' of dim(.) =  (3, 3) of precision  5   bits
     3    4    5
1 9.00 17.0 32.0
2 10.0 18.0 34.0
3 11.0 19.0 36.0

> vartwo(yy5[,"5"])
1 'mpfr' number of precision  53   bits
[1] 4

> formatBin(yy5)
  3            4            5
1 +0b1.0010p+3 +0b1.0001p+4 +0b1.0000p+5
2 +0b1.0100p+3 +0b1.0010p+4 +0b1.0001p+5
3 +0b1.0110p+3 +0b1.0011p+4 +0b1.0010p+5

> formatBin(yy5, scientific=FALSE)
  3           4           5
1 +0b__1001.0 +0b_10001._ +0b10000_._
2 +0b__1010.0 +0b_10010._ +0b10001_._
3 +0b__1011.0 +0b_10011._ +0b10010_._
```

Table G.13 This table focuses on the last displays in Table G.12. The last three bits in the 6-bit display of 33 (001) show a 1 in the "1" position. The number is rounded to the nearest even number (000) in the "2" digit (with a 0 in the "1" position) and truncated to (00_) in the 5-bit display. The last three bits (011) of 35 are rounded to the nearest even number (100) in the "2" digit and truncated to (10_) In both values, the resulting "2" digit is (0). The last three bits (010) of 34 already have a (0) in the "1" digit and therefore no rounding is needed. The sample variance of {33, 34, 35} is 1. When those numbers are rounded to five-bit binary, they become {32, 34, 36}. The sample variance of {32, 34, 36} is 4.

6-bit binary		rounded to 5-bit binary		
decimal	binary	displayed as 6-bit	truncated to 5-bit	equivalent decimal
33	+0b100001.	+0b100000.	+0b10000_.	32
34	+0b100010.	+0b100010.	+0b10001_.	34
35	+0b100011.	+0b100100.	+0b10010_.	36

Table G.14 The corrected two-pass algorithm centers the data by subtracting the mean, and then uses the two-pass algorithm on the centered data. It helps in some boundary conditions, for example the one shown in Table G.15.

```
> vartwoC <- function(x) {
+   vartwo(x-mean(x))
+ }

> x <- 1:3

> vartwoC(x)
[1] 1

> vartwoC(x+10^7)
[1] 1

> ## half machine precision
> vartwoC(x+10^8)
[1] 1

> vartwoC(x+10^15)
[1] 1

> ## boundary of machine precision
> ##
> vartwoC(x+10^16)
[1] 4

> vartwoC(x+10^17)
[1] 0
```

Table G.15 `vartwo` doesn't work for some problems on the boundary, such as this example at the boundary of 54-bit arithmetic. Summing the numbers effectively required one more significant binary digit. Since there are no more digits available, the data was rounded and the variance is not what our real-number intuition led us to expect. `vartwoC` does work.

```
> vartwo(xx54[,"15"])
1 'mpfr' number of precision  54   bits
[1] 1

> ## wrong answer.  numbers were shifted one binary position.
> vartwo(xx54[,"16"])
1 'mpfr' number of precision  54   bits
[1] 2.5

> ## vartwoC protects against that problem and gets the right answer.
> vartwoC(xx54[,"16"])
1 'mpfr' number of precision  54   bits
[1] 1

> sum(xx54[1:2,"16"])
1 'mpfr' number of precision  54   bits
[1] 20000000000000004

> ## Adding the first two numbers effectively doubled the numbers which
> ## means the significant bits were shifted one more place to the left.
> ## The first value was rounded up. Looking at just the last three bytes
> ## (where the last three bits are guaranteed 0):
> ##    +0x008p53 + 0x010p53 -> +0x010p53 + 0x010p53 -> +0x020p53
>
> sum((xx54)[1:3,"16"])      ## too high
1 'mpfr' number of precision  54   bits
[1] 30000000000000008

> sum((xx54)[3:1,"16"])      ## too low
1 'mpfr' number of precision  54   bits
[1] 30000000000000004

> sum((xx54)[c(1,3,2),"16"]) ## just right
1 'mpfr' number of precision  54   bits
[1] 30000000000000006
```

G.13 Can the Answer to the Calculation be Represented?

Chan et al. (1983) discuss various strategies needed to make sure that the fundamental goal of numerical analysis is achieved:

> If the input values can be represented by the computer, and if the answer can be represented by the computer, then the calculation should get the right answer.

It is very easy to construct an easy sum-of-squares problem for which naive calculations cannot get the right answer. The programmer's task is to get the right answer.

The Pythagorean Theorem tells us that $z = \sqrt{x^2 + y^2}$ will be an integer for several well known sets of triples $\{x, y, z\}$. The triple $\{3, 4, 5\}$ is probably the best known. The triple $\{3k, 4k, 5k\}$ for any k is also a triple which works. Table G.16 shows an example of k for which naive calculation fails, and for which `Mod` (modulus), one of R's `base` function, works. The goal is to understand how `Mod` is written.

Table G.16 The naive square-root-of-the-sum-of-squares algorithm gets the right answer in two of the three cases shown here. `Mod` gets the right answer in all three cases.

```
> x <- 3; y <- 4

> sqrt(x^2 + y^2)
[1] 5

> Mod(x + 1i*y)
[1] 5

> x <- 3e100; y <- 4e100

> sqrt(x^2 + y^2)
[1] 5e+100

> Mod(x + y*1i)
[1] 5e+100

> x <- 3e305; y <- 4e305

> sqrt(x^2 + y^2)
[1] Inf

> Mod(x + y*1i)
[1] 5e+305
```

The problem is that squaring arguments with a large exponent causes floating point overflow, which R interprets as infinite. The repair, shown in the `MyMod` function in Table G.17, is to rescale the numbers to a smaller exponent and then take the square root of the sum of squares. At the end the exponent is restored.

Table G.17 The `MyMod` function rescales the numbers and gets the right answer in all three cases. Only the third case is shown here.

```
> MyMod <- function(x, y) {
+   XYmax <- max(abs(c(x, y)))
+   xx <- x/XYmax
+   yy <- y/XYmax
+
+   result <- sqrt(xx^2 + yy^2)
+
+   result * XYmax
+ }

> x^2
[1] Inf

> y^2
[1] Inf

> MyMod(x, y)
[1] 5e+305
```

G.14 Explicit Loops

Desk calculator technology had different technical goals than digital computer technology. Entering the data manually multiple times was expensive and to be avoided. Table G.18 shows a scalar version of the one-pass algorithm for calculating sample variance. The explicit loop uses each entered value `x[i]` twice before going on to the next value. The term *one-pass* refers to the single entry of the data value for both accumulations $\left(\sum(x_i) \text{ and } \sum(x_i^2)\right)$ on the scalars in the vector x. Explicit scalar loops are exceedingly slow in R and are normally avoided. When we use vectorized operations (as in statements such as `sum(x^2)`) the looping is implicit at the user level and is done at machine speeds inside R.

We timed the scalar version of the one-pass algorithm along with the vectorized one-pass and two-pass algorithms. The explicitly looped one-pass algorithm `varoneScalar` is much slower than the vectorized algorithms. The vectorized `onepass` and `twopass` algorithms are about equally fast. Only the `twopass` algorithm gives the correct calculation to the precision of the computer.

Table G.18 The one-pass formula written as a scalar loop. This made sense for desk calculators because the user keyed in each number exactly once. It is about the most inefficient way to write code for R. In this example it is about 60 times slower than the vectorized algorithms.

```
> varoneScalar <- function(x) {
+   ## This is a pedagogical example.
+   ## Do not use this as a model for writing code.
+   n <- length(x)
+   sumx <- 0
+   sumx2 <- 0
+   for (i in 1:n) {
+     sumx <- sumx + x[i]
+     sumx2 <- sumx2 + x[i]^2
+   }
+     (sumx2 - (sumx^2)/n) / (n-1)
+   }

> x <- 1:3

> varoneScalar(x)
[1] 1

> varoneScalar(x+10^7)
[1] 1

> ## half machine precision
> varoneScalar(x+10^8)
[1] 0

> xx <- rnorm(1000)

> ## explicit loops are much slower in R
> system.time(for (j in 1:1000) varoneScalar(xx))
   user  system elapsed
  1.249   0.309   1.483

> system.time(for (j in 1:1000) varone(xx))
   user  system elapsed
  0.020   0.002   0.021

> system.time(for (j in 1:1000) vartwo(xx))
   user  system elapsed
  0.021   0.001   0.022
```