ECON 950 — Winter 2020

Prof. James MacKinnon

13. Floating-Point Arithmetic

Estimates and test statistics are rarely integers. Therefore, any computer program for econometrics will need to use floating-point arithmetic.

The rules of floating-point arithmetic are not at all the same as the rules followed by the real numbers that we try to approximate using it.

Floating-point arithmetic is needed because econometricians need to be able to deal with numbers that may be very large or very small.

Suppose we wished to deal with numbers between 2^{-40} and 2^{40} without using floating-point arithmetic. This is actually quite a small range, since $2^{-40} = 0.90949 \times 10^{-12}$ and $2^{40} = 1.0995 \times 10^{12}$.

It would take 80 bits to represent all of these numbers, or 81 bits if we added a sign bit to allow them to be either positive or negative.

The smallest number we could deal with, 2^{-40}, would be coded as 79 0s followed by a 1, and the largest, $2^{40}-1$, would be coded as 80 1s. This is quite a lot of memory.

Despite the large number of bits, many of these numbers would not be stored accurately.

The relative error for small numbers would be greater than for large numbers. Small numbers would consist of a long string of 0s followed by a few meaningful bits.

On most modern computers, only 32 (sometimes 64) bits are devoted to storing each integer, and either 32 or 64 bits (sometimes 128) are devoted to storing each floating-point number.

13.1. Floating-point numbers

Floating-point arithmetic stores numbers in the form of a signed **mantissa** (or **significand**) times a fixed **base** (or **radix**) raised to the power of a signed **exponent**.

That is exactly what we did above to write the decimal equivalents of 2^{40} and 2^{-40}.

In general, we can write a floating-point number as

$$m \times b^c = \pm\, .d_1\, d_2\, d_3 \ldots d_p \times b^c, \tag{1}$$

where m is the mantissa, b is the base, and c is the exponent.

The mantissa has p digits, d_1 through d_p, and one sign bit. The number of digits in the mantissa is the principal (but not the only) factor that determines the **precision** of any floating-point system.

The choice of base and the number of bits devoted to storing the exponent jointly determine its **range**.

All commercially significant computer architectures designed since about 1980 have used $b = 2$. Older architectures used 8 and 16.

Any particular system of floating-point arithmetic can only represent a finite number of rational numbers, and these numbers are not equally spaced.

Imagine a 12-bit format, which is too small to be of practical use. Suppose that 7 of the 12 bits are used for the mantissa, including one sign bit, and the remaining 5 bits are used for the exponent, again including the sign bit.

Thus the smallest exponent would be -15 and the largest would be 15.

It is possible to normalize all allowable floating-point numbers (except 0) so that the leading digit is a 1. Thus there is no need to store this leading digit.

By using this **hidden bit**, we effectively obtain a 7-bit mantissa, which can take on any value between .1000000 and .1111111 (binary).

The spacing between adjacent floating-point numbers depends on their magnitude.

Consider the floating-point numbers between 1 and 2. In our 12-bit format, there are only 63 of these (not counting the end points), and they are equally spaced. The next number after 1 is 1.015625, then 1.03125, and so on.

There are also 63 numbers between 2 and 4, also equally spaced, but twice as far apart as the numbers between 1 and 2. The next number after 2 is 2.03125, then 2.0625, and so on.

Likewise, there are 63 equally-spaced numbers between 0.5 and 1.0, which are half as far apart as the numbers between 1 and 2.

Real systems of floating-point numbers work in exactly the same way.

If the base is 2, there is always a fixed number of floating-point numbers between each two adjacent powers of 2.

For IEEE 754 single precision, which has a 24-bit mantissa, this fixed number is 8,388,607.

It is possible to represent only certain rational numbers exactly. For example, the decimal number 0.1 cannot be represented exactly in any floating-point arithmetic that uses base 2.

In our simple example, the best we could do is .1100110 (binary) $\times 2^{-3} \cong 0.0996094$, which is not really a very good approximation.

Thus, even if we were to start a computation with decimal numbers that were completely accurate, simply storing them as floating-point numbers would introduce errors.

There are two, widely-used IEEE 754 standard floating-point formats.

Single precision has a 24-bit mantissa (counting the hidden bit but not the sign bit), and an 8-bit exponent (counting the sign bit). Its range is, approximately, $10^{\pm 38}$, and it is accurate to about 7 decimal digits.

Double precision, has a 53-bit mantissa and an 11-bit exponent. Its range is, approximately, $10^{\pm 308}$, and it is accurate to almost 16 decimal digits.

If two numbers, say x_1 and x_2, are just below and just above a power of 2, then

the maximum possible (absolute) error in storing x_1 will be half as large as the maximum possible error in storing x_2.

The maximum absolute error doubles every time the exponent increases by 1.

In contrast, the maximum relative error **wobbles**. It is highest just after the exponent increases and lowest just before it does so.

This is the main reason why $b = 2$ is the best choice of exponent and $b = 16$ is a bad choice. With $b = 16$, the maximum relative error increases by a factor of 16 every time the exponent increases by 1.

13.2. Properties of floating-point arithmetic

The set of floating-point numbers available to a computer program is *not* the same as the set of real (or even rational) numbers. Therefore, arithmetic on floating-point numbers does not have the same properties as arithmetic on the real numbers.

Storing a number as a floating-point number often introduces an error, although not in all cases. Subsequent arithmetic operations then add more errors.

When floating-point numbers are added or subtracted, their exponents have to be adjusted to be the same. This means that some of the digits of the mantissa of the number with the smaller exponent are lost.

For example, suppose we wanted to add 32.5 and 0.2 using the 12-bit floating-point arithmetic discussed above. In this arithmetic, 32.5 is represented (exactly) as $.1000001 \times 2^6$, and 0.2 is represented (inexactly) as $.1100110 \times 2^{-2}$.

Adjusting the second number so that it has the same exponent as the first, we get $.0000000 \times 2^6$, which has zero bits of precision. Thus adding the two numbers simply yields 32.5.

By itself, this sort of error is perhaps tolerable. After all, in the floating-point system we are using, the next number after 32.5 is 33.0, so 32.5 is actually the closest we can get to the correct answer of 32.7.

When successive arithmetic operations are carried out, errors build up. The answer may be accurate to zero digits when numbers of different sizes and signs are added.

For example, consider the expression

$$69,393,121 - 1.0235 - 69,393,120, \tag{2}$$

which is equal to -0.0235.

Suppose we attempt to evaluate this expression using IEEE 754 single precision arithmetic. If we evaluate it as written, we obtain the answer 0, because 69,393,121 $-$ 1.0235 $\cong$ 69,393,120, where "$\cong$" denotes equality in floating-point arithmetic.

Alternatively, we could change the order of evaluation, first subtracting the last number from the first, and then subtracting the second. If we do that, we get the answer -1.0235, because $69,393,121 - 69,393,120 \cong 0$.

Of course, if we had used double precision, we would have obtained an accurate answer. But, by making the first and third numbers sufficiently large relative to the second, we could make even double-precision arithmetic fail.

Precisely the type of calculation that is exemplified by expression (2) occurs all the time in econometrics. The inner product of two n-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is

$$\boldsymbol{x}^\top \boldsymbol{y} = \sum_{i=1}^{n} x_i \, y_i. \tag{3}$$

The terms in the summation here will often be of both signs and widely varying magnitudes. Thus inner products can be computed quite inaccurately.

The above example illustrates a major problem of floating-point arithmetic. The order of operations matters. It is emphatically not the case that

$$x + (y + z) = (x + y) + z. \tag{4}$$

Unfortunately, compilers and computer programs sometimes assume that (4) holds.

The same program may yield different answers if compiled with different compilers, or with different compiler options.

In most cases, the answers will differ only in the least significant bits but, as example (2) shows, the differences can be extremely large.

Using single precision to compute an inner product is madness, even if the elements of the vectors are stored in single precision.

Even using double precision may work very badly for some vectors.

The IEEE 754 standard contains an 80-bit floating-point format that compilers can access but programmers usually cannot. It is designed to be used for intermediate calculations in cases like (3).

Unfortunately, modern Intel-based computers have more than one floating-point format. The newer (and faster) ones are not fully IEEE 754 compliant and do not have the 80-bit format.

13.3. A numerical example

The deficiencies of floating-point arithmetic are important in practice.

Suppose that we wish to calculate the sample mean and variance of a sequence of numbers y_i, $i = 1, \ldots, n$. Every student of statistics knows that

$$\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i \tag{5}$$

and that an unbiased estimate of the variance of the y_i is

$$\frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y})^2 = \frac{1}{n-1} \left(\sum_{i=1}^{n} y_i^2 - n\bar{y}^2 \right). \tag{6}$$

The equality here is true algebraically, but it is *not* true when calculations are done using floating-point arithmetic.

The first expression in (6) can be evaluated reasonably accurately, so long as the y_i are not greatly different in magnitude from $\bar{y}$.

But the second expression involves subtracting $n\bar{y}^2$ from $\sum y_i^2$. When $\bar{y}$ is large relative to $\mathrm{Var}(y_i)$, both these quantities can be very large relative to their difference.

In this situation, the second expression may calculate the variance very inaccurately.

Such expressions are **numerically unstable**, because they are prone to error when evaluated using floating-point arithmetic.

To illustrate the magnitude of potential numerical problems, I generated 1000 pseudo-random numbers from the normal distribution and then normalized them so that the sample mean was exactly μ and the sample variance was exactly unity.

Actually, these were not exact. I used **quadruple-precision** arithmetic, which is about twice as accurate as double-precision arithmetic, but it is often not available and tends to be quite slow when it is available.

I then calculated the sample variance for various values of μ using both single and double precision and using both formulas in (6).

Table 1. Absolute Errors in Calculating the Sample Variance

μ	stable single	stable double	unstable single	unstable double
0	0.298×10^{-6}	0.444×10^{-15}	0.298×10^{-6}	0.444×10^{-15}
10	0.358×10^{-6}	0.666×10^{-15}	0.677×10^{-4}	0.129×10^{-12}
10^2	0.596×10^{-6}	0.444×10^{-15}	0.853×10^{-2}	0.949×10^{-11}
10^3	0.119×10^{-6}	0.511×10^{-14}	0.139×10^{1}	0.239×10^{-9}
10^4	0.556×10^{-4}	0.326×10^{-13}	0.175×10^{3}	0.350×10^{-6}
10^5	0.108×10^{-3}	0.904×10^{-13}	0.145×10^{5}	0.397×10^{-5}
10^6	0.283×10^{-2}	0.120×10^{-11}	0.720×10^{7}	0.582×10^{-3}

All calculations were done on an IBM RS/6000 workstation, which uses IEEE 754 but does not have an 80-bit intermediate format.

This example illustrates several important points:

- Except when $\mu = 0$, so that no serious numerical problems arise for either formula, the stable formula, on the left side of (6), yields much more accurate

results than the unstable one.

Thus it is important to use numerically stable formulae rather than numerically unstable ones if the data may be poorly conditioned.

- Double-precision arithmetic yields very much more accurate results than single-precision arithmetic. One is actually better off to evaluate the numerically unstable expression using double precision than to evaluate the numerically stable expression using single precision.

  The best results are, of course, achieved by evaluating the stable expression using double-precision arithmetic.

- In the best case, the error is not much larger than the wobble in the floating-point system used. For IEEE single precision, this is $2^{-23} = 0.119 \times 10^{-6}$, and for IEEE double precision, it is $2^{-52} = 0.222 \times 10^{-15}$.

  The errors for $\mu = 0$ (which for this example are the same in relative as absolute terms) using the stable expression are less than three times the single-precision wobble and just twice the double-precision wobble.

- The nature of the data may determine whether a calculation gives valid results.

All the results deteriorate as μ increases, and even the stable expression with double-precision data will eventually yield seriously inaccurate results.

In the case of regression models and many machine-learning methods, it is highly desirable for all variables to have similar magnitudes, for their means not to be too large relative to their variances, and for them not to be too highly correlated.

This is especially important for nonlinear models, where simple reparametrizations can dramatically affect numerical stability, since these will change the matrices of derivatives used by artificial regressions, Newton's Method, and other procedures for determining what direction to search in.

Unlike programming languages such as Fortran and C++, most matrix packages use double precision exclusively. This reduces, but does not eliminate, the risk of numerical problems.

Even when double-precision arithmetic is used, severe numerical problems can arise when data are ill-conditioned or when inappropriate algorithms are used.

13.4. Numerical Derivatives

Numerical derivatives can be very useful.

We can use them with functions that are difficult to differentiate analytically.

We can use them to check the accuracy of analytic derivatives, or of a computer program that implements analytic derivatives.

We can compute them for smooth functions that we cannot write down analytically.

Unfortunately, the characteristics of floating-point arithmetic make it particularly hard to take derivatives numerically.

Suppose that x is a scalar and that we wish to take the derivative of $f(x)$. By definition,

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{7}$$

This suggests that we might use the formula

$$f'(x) \cong \frac{f(x+h) - f(x)}{h} \tag{8}$$

for some h that is chosen to be small. Unfortunately, this **one-sided formula** does not work very well.

There are two sources of error in any formula for numerical derivatives.

Truncation error arises because $h > 0$, so that the formula (8) is not quite correct.

Roundoff error arises from the limitations of floating-point arithmetic.

We could eliminate truncation error by making h arbitrarily small, but that would make roundoff error very severe.

Consider the truncation error associated with (8). A Taylor-series approximation of the numerator around $h = 0$ is

$$f(x+h) - f(x) = hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + O(h^4). \tag{9}$$

Dividing the right-hand side by h, we find that

$$\frac{f(x+h) - f(x)}{h} = f'(x) + \frac{1}{2}hf''(x) + O(h^2). \tag{10}$$

Thus the truncation error is of order h.

A much better formula is the **symmetric formula**

$$f'(x) \cong \frac{f(x+h) - f(x-h)}{2h}. \tag{11}$$

We now evaluate f on both sides of x, instead of on just one side.

A Taylor series approximation of the numerator yields

$$f(x+h) - f(x-h) = h\big(f'(x) + f'(x)\big) + \tfrac{1}{6}h^3\big(f'''(x) + f'''(x)\big) + O(h^5). \tag{12}$$

All even-order terms are zero, since they all involve factors of $f^{(m)}(x) - f^{(m)}(x)$, where $f^{(m)}$ is the m^{th} derivative of f, m being an even number.

Dividing (12) by $2h$, we find that

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \tfrac{1}{6}h^2 f'''(x) + O(h^4). \tag{13}$$

Thus the truncation error is $O(h^2)$.

Since h will normally be very small, the truncation error in (11) will normally be very much smaller than the truncation error in (8).

We should use a larger value of h in (11) than in (8). This allows us to reduce both roundoff error and total error, at the expense of more truncation error than we would get with a very small value of h.

Suppose that f is evaluated with a proportional error of ε. This might conceivably be as small as the wobble in the floating-point arithmetic, or it might be a lot larger.

Whatever the value of ε, the roundoff error is roughly $\varepsilon|f(x)/h|$. For the one-sided formula (9), the truncation error is given by (10).

The sum of the absolute values of these two errors is

$$v = \varepsilon|f(x)/h| + \left|\tfrac{1}{2}hf''(x)\right|. \tag{14}$$

Differentiating this with respect to h and solving yields

$$h \cong \sqrt{2\varepsilon f/f''}. \tag{15}$$

Thus, for the one-sided formula (9), the optimal choice of h will be proportional to the square root of ε. From (14), this implies that the total error will also be $O(\sqrt{\varepsilon})$.

Thus, the numerical derivative will be much less accurate than f itself. For IEEE 754 double-precision arithmetic, if f/f'' is roughly unity, we would probably want to choose h to be somewhere between 10^{-6} and 10^{-8}.

We can use the same sort of analysis for the symmetric formula (11). The result is

$$h \cong (3\varepsilon f/f''')^{1/3}. \tag{16}$$

The optimal choice of h is now proportional to the cube root of ε.

For IEEE 754 double-precision arithmetic, we would probably want to choose h to be in the neighborhood of 10^{-5} or 10^{-4}.

With h chosen optimally according to (16), the total error will be $O(\varepsilon^{2/3})$.

There are more accurate formulae that can be used to to reduce the order of the truncation error even further. One example is

$$f'(x) \cong \frac{1}{12h}\Big(8\big(f(x+h) - f(x-h)\big) + \big(f(x-2h) - f(x+2h)\big)\Big). \tag{17}$$

It can be shown that the truncation error for this **four-point formula** is $O(h^4)$ and that the optimal h is $O(\varepsilon^{1/5})$. This is a good exercise.

Of course, from (15) and (16), we see that the choice of h depends on the ratio of f to its second or third derivative. These may depend on how the data are scaled.

Numerical second derivatives are more unstable than numerical first derivatives.

Suppose that $\boldsymbol{x}$ is a k-vector and that $\boldsymbol{e}_i$ denotes a k-vector with 1 in the i^{th} place and 0 everywhere else. Using a symmetric formula, the second derivative of $f(\boldsymbol{x})$ with respect to x_i and x_j can be approximated by

$$\begin{aligned}\frac{1}{4h^2}\Big(&f\big(\boldsymbol{x}+h(\boldsymbol{e}_i+\boldsymbol{e}_j)\big)+f\big(\boldsymbol{x}-h(\boldsymbol{e}_i+\boldsymbol{e}_j)\big)\\&-f\big(\boldsymbol{x}+h(\boldsymbol{e}_i-\boldsymbol{e}_j)\big)-f\big(\boldsymbol{x}-h(\boldsymbol{e}_i-\boldsymbol{e}_j)\big)\Big).\end{aligned}\tag{18}$$

When $i=j$, this simplifies to

$$\frac{1}{4h^2}\big(f(\boldsymbol{x}+2h\boldsymbol{e}_i)+f(\boldsymbol{x}-2h\boldsymbol{e}_i)-2f(\boldsymbol{x})\big).\tag{19}$$

Both these formulae involve $4h^2$ rather than $2h$ in the denominator. This suggests that the optimal h will be much larger than it was in the case of first derivatives.

Formulae like (18) and (19) may work badly if the function is not scaled properly. Adding h to each element of $\boldsymbol{x}$ should cause f to change by a similar amount.

Some general advice for using numerical derivatives:

- Always use symmetric formulae, like (11), (18), and (19), or even four-point formulae like (17), rather than one-sided formulae.
- Choose h or the h_i very carefully, making sure that the function is scaled properly if only one h is used.
- Bear in mind that h needs to be larger for second than for first derivatives, larger for four-point than for symmetric formulae, and larger as the error in the underlying function evaluation increases.
- Use numerical second derivatives only if there is no alternative.
- If possible, compute first derivatives analytically. Then compute their derivatives numerically. The second derivatives are really numerical first derivatives.