

Chapter 2

Basic Computational Algorithms

John F. Monahan

2.1 Computer Arithmetic

Numbers are the lifeblood of statistics, and computational statistics relies heavily on how numbers are represented and manipulated on a computer. Computer hardware and statistical software handle numbers well, and the methodology of computer arithmetic is rarely a concern. However, whenever we push hardware and software to their limits with difficult problems, we can see signs of the mechanics of floating point arithmetic around the frayed edges. To work on difficult problems with confidence and explore the frontiers of statistical methods and software, we need to have a sound understanding of the foundations of computer arithmetic. We need to know how arithmetic works and why things are designed the way they are.

As scientific computation began to rely heavily on computers, a monumental decision was made during the 1960s to change from base ten arithmetic to base two. Humans had been doing base ten arithmetic for only a few hundred years, during which time great advances were possible in science in a short period of time. Consequently, the resistance to this change was strong and understandable. The motivation behind the change to base two arithmetic is merely that it is so very easy to do addition (and subtraction) and multiplication in base two arithmetic. The steps are easy enough that a machine can be designed – wire a board of relays – or design a silicon chip – to do base two arithmetic. Base ten arithmetic is comparatively quite difficult, as its recent mathematical creation would suggest. However two big problems arise in changing from base ten to base two: (1) we need to constantly convert numbers written in base ten by humans to base two number system and then back again to base ten for humans to read the results, and (2) we need to understand the limits of arithmetic in a different number system.

J.F. Monahan (✉)

Department of Statistics, North Carolina State University, Raleigh, NC, USA

e-mail: monahan@ncsu.edu

2.1.1 Integer Arithmetic

Computers use two basic ways of writing numbers: fixed point (for integers) and floating point (for real numbers). Numbers are written on a computer following base two positional notation. The *positional number system* is a convention for expressing a number as a list of integers (digits), representing a number x in base B by a list of digits $a_m, a_{m-1}, \dots, a_1, a_0$ whose mathematical meaning is

$$x = a_{m-1}B^{m-1} + \dots + a_2B^2 + a_1B + a_0 \quad (2.1)$$

where the digits a_j are integers in $\{0, \dots, B-1\}$. We are accustomed to what is known in the West as the Arabic numbers, $0, 1, 2, \dots, 9$ representing those digits for writing for humans to read. For base two arithmetic, only two digits are needed $\{0, 1\}$. For base sixteen, although often viewed as just a collection of four binary digits (1 *byte* = 4 *bits*), the Arabic numbers are augmented with letters, as $\{0, 1, 2, \dots, 9, a, b, c, d, e, f\}$, so that $f_{\text{sixteen}} = 15_{\text{ten}}$.

The system based on (2.1), known as *fixed point arithmetic*, is useful for writing integers. The choice of $m = 32$ dominates current computer hardware, although smaller ($m = 16$) choices are available via software and larger ($m = 48$) hardware had been common in high performance computing. Recent advances in computer architecture may soon lead to the standard changing to $m = 64$. While the writing of a number in base two requires only the listing of its binary digits, a convention is necessary for expression of negative numbers. The survivor of many years of intellectual competition is the *two's complement* convention. Here the first (leftmost) bit is reserved for the sign, using the convention that 0 means positive and 1 means negative. Negative numbers are written by complementing each bit (replace 1 with 0, 0 with 1) and adding one to the result. For $m = 16$ (easier to display), this means that 22_{ten} and its negative are written as

$$(0\ 001\ 0110) = 22_{\text{ten}}$$

and

$$(1\ 110\ 1010) = -22_{\text{ten}}.$$

Following the two's complement convention with m bits, the smallest (negative) number that can be written is -2^{m-1} and the largest positive number is $2^{m-1} - 1$; zero has a unique representation of $(0\ 000\ \dots\ 0000)$. Basic arithmetic (addition and multiplication) using two's complement is easy to code, essentially taking the form of mod 2^{m-1} arithmetic, with special tools for overflow and sign changes. See, for example, [Knuth \(1997\)](#) for history and details, as well as algorithms for base conversions.

The great advantage of fixed point (integer) arithmetic is that it is so very fast. For many applications, integer arithmetic suffices, and most nonscientific computer software only uses fixed point arithmetic. Its second advantage is that it does not

suffer from the rounding error inherent in its competitor, floating point arithmetic, whose discussion follows.

2.1.2 Floating Point Arithmetic

To handle a larger subset of the real numbers, the positional notation system includes an exponent to express the location of the radix point (generalization of the decimal point), so that the usual format is a triple (sign, exponent, fraction) to represent a number as

$$x = (-1)^{\text{sign}} B^{\text{exponent}} (a_1 B^{-1} + a_2 B^{-2} + \dots + a_d B^{-d}) , \quad (2.2)$$

where the fraction is expressed by its list of base B digits $0.a_1 a_2 a_3 \dots a_d$. To preserve as much information as possible with the limited d digits to represent the fraction, *normalization* is usually enforced, that is, the leading/most significant digit a_1 is nonzero – except for the special case $x = 0$. The mathematical curiosity of an infinite series expansion of a number has no place here where only d digits are available. Moreover, a critical issue is what to do when only d digits are available. *Rounding* to the nearest number is preferred to the alternative *chopping*; in the case of representing $\pi = 3.14159265\dots$ to $d = 5$ decimal ($B = \text{ten}$) digits leads to the more accurate $(+, +1, 0.31416)$ in the case of rounding, rather than $(+, +1, 0.31415)$ for the chopping alternative. Notice that normalization and the use of this positional notation reflects a goal of preserving relative accuracy, or reducing the relative error in the approximation. The expression of a real number x in floating point arithmetic can be expressed mathematically in terms of a function $fl : R \rightarrow \mathcal{F}$ where \mathcal{F} is the set of numbers that can be represented using this notation, the set of floating point numbers. The relative accuracy of this rounding operation can be expressed as

$$fl(x) = (1 + u)x , \quad (2.3)$$

where $|u| \leq U$ where U is known as the *machine unit*. Seen in terms of the relative error of $fl(x)$ in approximating x , the expression above can be rewritten as

$$|x - fl(x)|/|x| \leq U \quad \text{for } x \neq 0 .$$

For base B arithmetic with d digits and chopping, $U = B^{1-d}$; rounding reduces U by a factor of 2.

An important conceptual leap is the understanding that most numbers are represented only approximately in floating point arithmetic. This extends beyond the usual irrational numbers such as π or e that cannot be represented with a finite number of digits. A novice user may enter a familiar assignment such as $x = 8.6$ and, observing that the computer prints out 8.6000004, may consider this an error.

When the “8.6” was entered, the computer had to first parse the text “8.6” and recognize the decimal point and arabic numbers as a representation, for humans, of a real number with the value $8 + 6 \times 10^{-1}$. The second step is to convert this real number to a base two floating point number – approximating this base ten number with the closest base two number – this is the function $fl(\cdot)$. Just as $1/3$ produces the repeating decimal $0.33333 \dots$ in base 10, the number 8.6 produces a repeating binary representation $1000.100110011 \dots_{two}$, and is chopped or rounded to the nearest floating point number $fl(8.6)$. Later, in printing this same number out, a second conversion produces the closest base 10 number to $fl(8.6)$ with few digits; in this case 8.6000004, not an error at all. Common practice is to employ numbers that are integers divided by powers of two, since they are exactly represented. For example, distributing 1,024 equally spaced points makes more sense than the usual 1,000, since $j/1024$ can be exactly represented for any integer j .

A breakthrough in hardware for scientific computing came with the adoption and implementation of the IEEE 754 binary floating point arithmetic standard, which has standards for two levels of precision, *single precision* and *double precision* (IEEE 1985). The single precision standard uses 32 bits to represent a number: a single bit for the sign, 8 bits for the exponent and 23 bits for the fraction. The double precision standard requires 64 bits, using 3 more bits for the exponent and adds 29 to the fraction for a total of 52. Since the leading digit of a normalized number is nonzero, in base two the leading digit must be one. As a result, the floating point form (2.2) above takes a slightly modified form:

$$x = (-1)^{\text{sign}} B^{\text{exponent} - \text{excess}} (1 + a_1 B^{-1} + a_2 B^{-2} + \dots + a_d B^{-d}) \quad (2.4)$$

as the fraction is expressed by its list of binary digits $1.a_1a_2a_3 \dots a_d$. As a result, while only 23 bits are stored, it works as if one more bit were stored. The exponent using 8 bits can range from 0 to 255; however, using an excess of 127, the range of the difference (*exponent* – *excess*) goes from –126 to 127. The finite number of bits available for storing numbers means that the set of floating point numbers \mathcal{F} is a finite, discrete set. Although well-ordered, it does have a largest number, smallest number, and smallest positive number. As a result, this IEEE Standard expresses positive numbers from approximately 1.4×10^{-45} to 3.4×10^{38} with a machine unit $U = 2^{-24} \approx 10^{-7}$ using only 31 bits. The remaining 32nd bit is reserved for the sign. Double precision expands the range to roughly $10^{\pm 300}$ with $U = 2^{-53} \approx 10^{-16}$, so the number of accurate digits is more than doubled.

The two extreme values of the exponent are employed for special features. At the high end, the case *exponent* = 255 signals two infinities ($\pm\infty$) with the largest possible fraction. These values arise as the result of an *overflow* operation. The most common causes are adding or multiplying two very large numbers, or from a function call that produces a result that is larger than any floating point number. For example, the value of $\exp(x)$ is larger than any finite number in \mathcal{F} for $x > 88.73$ in single precision. Before adoption of the standard, $\exp(89.9)$ would cause the program to cease operation due to this “exception”. Including $\pm\infty$ as members of \mathcal{F} permits the computations to continue, since a sensible result is now available.

As a result, further computations involving the value $\pm\infty$ can proceed naturally, such as $1/\infty = 0$. Again using the exponent = 255, but with any other fraction represents *not-a-number*, usually written as “NaN”, and used to express the result of *invalid operations*, such as $0/0$, $\infty - \infty$, $0 \times \infty$, and square roots of negative numbers. For statistical purposes, another important use of NaN is to designate missing values in data. The use of infinities and NaN permit continued execution in the case of anomalous arithmetic operations, instead of causing computation to cease when such anomalies occur. The other extreme exponent = 0 signals a denormalized number with the net exponent of -126 and an unnormalized fraction, with the representation following (2.2), rather than the usual (2.4) with the unstated and unstored 1. The *denormalized* numbers further expand the available numbers in \mathcal{F} , and permit a *soft underflow*. Underflow, in contrast to overflow, arises when the result of an arithmetic operation is smaller in magnitude than the smallest representable positive number, usually caused by multiplying two small numbers together. These denormalized numbers begin approximately 10^{-38} near the reciprocal of the largest positive number. The denormalized numbers provide even smaller numbers, down to 10^{-45} . Below that, the next number in \mathcal{F} is the floating point zero: the smallest exponent and zero fraction – all bits zero.

Most statistical software employs only double precision arithmetic, and some users become familiar with apparent aberrant behavior such as a sum of residuals of 10^{-16} instead of zero. While many areas of science function quite well using single precision, some problems, especially nonlinear optimization, nevertheless require double precision. The use of single precision requires a sound understand of rounding error. However, the same rounding effects remain in double precision, but because their effects are so often obscured from view, double precision may promote a naive view that computers are perfectly accurate.

The machine unit expresses a relative accuracy in storing a real number as a floating point number. Another similar quantity, the *machine epsilon*, denoted by ϵ_m , is defined as the smallest positive number that, when added to one, gives a result that is different from one. Mathematically, this can be written as

$$fl(1 + x) = 1 \text{ for } 0 < x < \epsilon_m . \quad (2.5)$$

Due to the limited precision in floating point arithmetic, adding a number that is much smaller in magnitude than the machine epsilon will not change the result. For example, in single precision, the closest floating point number to $1 + 2^{-26}$ is 1. Typically, both the machine unit and machine epsilon are nearly the same size, and these terms are often used interchangeably without grave consequences.

2.1.3 Cancellation

Often one of the more surprising aspects of floating point arithmetic is that some of the more familiar laws of algebra are occasionally violated: in particular, the

associative and distributive laws. While most occurrences are just disconcerting to those unfamiliar to computer arithmetic, one serious concern is cancellation. For a simple example, consider the case of base ten arithmetic with $d = 6$ digits, and take $x = 123.456$ and $y = 123.332$, and note that both x and y may have been rounded, perhaps x was 123.456478 or 123.456000 or 123.455998 . Now x would be stored as $(+, 3, 0.123456)$ and y would be written as $(+, 3, 0.123332)$, and when these two numbers are subtracted, we have the unnormalized difference $(+, 3, 0.000124)$. Normalization would lead to $(+, 0, .124???)$ where merely “?” represents that some digits need to take their place. The simplistic option is to put zeros, but 0.124478 is just as good an estimate of the true difference between x and y as 0.124000 , or 0.123998 , for that matter. The problem with cancellation is that the relative accuracy that floating point arithmetic aims to protect has been corrupted by the loss of the leading significant digits. Instead of a small error in the sixth digit, we now have that error in the third digit; the relative error has effectively been magnified by a factor of 1,000 due to the cancellation of the first 3 digits.

The best way to deal with the potential problem caused by catastrophic cancellation is to avoid them. In many cases, the cancellation may be avoided by reworking the computations analytically to handle the cancellation:

$$1 - (1 - 2t)^{-1} = \frac{1 - 2t - 1}{1 - 2t} = \frac{-2t}{1 - 2t}.$$

In this case, there is significant cancellation when t is small, and catastrophic cancellation whenever t drops below the machine epsilon. Using six digit decimal arithmetic to illustrate, at $t = 0.001$, the left hand expression, $1 - (1 - 2t)^{-1}$, gives

$$1.00000 - 1.00200 = 0.200000 \times 10^{-2}$$

while the right hand expression, $-2t/(1 - 2t)$, gives

$$0.200401 \times 10^{-2},$$

the correct (rounded) result. The relative error in using the left hand expression is an unacceptable 0.002. At $t = 10^{-7}$, the left hand expression leads to a complete cancellation yielding zero and a relative error of one. Just a little algebra here avoids the most of the effect of cancellation. When the expressions involve functions, cases where cancellation occurs can often be handled by approximations. In the case of $1 - e^{-t}$, serious cancellation will occur whenever t is very small. The cancellation can be avoided for this case by using a power series expansion:

$$1 - e^{-t} = 1 - (1 - t + t^2/2 - \dots) \approx t - t^2/2 = t(1 - t/2).$$

When $t = 0.0001$, the expression $1 - e^{-t}$ leads to the steps

$$1.00000 - 0.99990 = 0.100000 \times 10^{-4},$$

while the approximation gives

$$(0.0001)(0.999950) = 0.999950 \times 10^{-4}$$

which properly approximates the result to six decimal digits. At $t = 10^{-5}$ and 10^{-6} , similar results occur, with complete cancellation at 10^{-7} . Often the approximation will be accurate just when cancellation must be avoided.

One application where rounding error must be understood and cancellation cannot be avoided is numerical differentiation, where calls to a function are used to approximate a derivative from a first difference:

$$f'(x) \approx [f(x+h) - f(x)]/h. \quad (2.6)$$

Mathematically, the accuracy of this approximation is improved by taking h very small; following a quadratic Taylor's approximation, we can estimate the error as

$$[f(x+h) - f(x)]/h \approx f'(x) + \frac{1}{2}hf''(x).$$

However, when the function calls $f(x)$ and $f(x+h)$ are available only to limited precision – a relative error of ϵ_m , taking h smaller leads to more cancellation. The cancellation appears as a random rounding error in the numerator of (2.6) which becomes magnified by dividing by a small h . Taking h larger incurs more bias from the approximation; taking h smaller incurs larger variance from the rounding error. Prudence dictates balancing bias and variance. [Dennis and Schnabel \(1983\)](#) recommend using $h \approx \epsilon_m^{1/2}$ for first differences, but see also [Bodily \(2002\)](#).

The second approach for avoiding the effects of cancellation is to develop different methods. A common cancellation problem in statistics arises from using the formula

$$\sum_{i=1}^n y_i^2 - n\bar{y}^2 \quad (2.7)$$

for computing the sum of squares around the mean. Cancellation can be avoided by following the more familiar two-pass method

$$\sum_{i=1}^n (y_i - \bar{y})^2 \quad (2.8)$$

but this algorithm requires all of the observations to be stored and repeated updates are difficult. A simple adjustment to avoid cancellation, requiring only a single pass and little storage, uses the first observation to center:

$$\sum_{i=1}^n (y_i - \bar{y})^2 = \sum_{i=1}^n (y_i - y_1)^2 - n(y_1 - \bar{y})^2. \quad (2.9)$$

An orthogonalization method from regression using Givens rotations (see [Chan et al. 1983](#)) can do even better to find $s_n = \sum_{i=1}^n (y_i - \bar{y})^2$:

$$t_i = t_{i-1} + y_i \quad (2.10)$$

$$s_i = s_{i-1} + (iy_i - t_i)^2 / (i(i-1)) . \quad (2.11)$$

To illustrate the effect of cancellation, take the simple problem of $n = 5$ observations, $y_i = 4,152 + i$ so that $y_1 = 4,153$ through $y_5 = 4,157$. Again using six decimal digits, the computations of the sum and mean encounter no problems, and we easily get $\bar{y} = 4,155$ or 0.415500×10^4 , and $\sum y_i = 20,775$ or 0.207750×10^5 . However, each square loses some precision in rounding:

$$\begin{aligned} y_1 &= 4,153 , & y_1^2 &= 4,153^2 = 17,247,409 & \text{rounded to} & 0.172474 \times 10^8 \\ y_2 &= 4,154 , & y_2^2 &= 4,154^2 = 17,255,716 & \text{rounded to} & 0.172557 \times 10^8 \\ y_3 &= 4,155 , & y_3^2 &= 4,155^2 = 17,264,025 & \text{rounded to} & 0.172640 \times 10^8 \\ y_4 &= 4,156 , & y_4^2 &= 4,156^2 = 17,272,336 & \text{rounded to} & 0.172723 \times 10^8 \\ y_5 &= 4,157 , & y_5^2 &= 4,157^2 = 17,280,649 & \text{rounded to} & 0.172806 \times 10^8 . \end{aligned}$$

Summing the squares encounters no further rounding on its way to 0.863200×10^8 , and we compute the corrected sum of squares as

$$\begin{aligned} &0.863200 \times 10^8 - (0.207750 \times 10^5) \times 4,155 \\ &0.863200 \times 10^8 - 0.863201 \times 10^8 = -100 . \end{aligned}$$

The other three algorithms, following (2.8), (2.9), (2.10), and (2.11), each give the perfect result of 10 in this case.

Admittedly, while this example is contrived to show an absurd result, a negative sum of squares, the equally absurd value of zero is hardly unusual. Similar computations – differences of sum of squares – are routine, especially in regression and in the computation of eigenvalues and eigenvectors. In regression, the orthogonalization method (2.10) and (2.11) is more commonly seen in its general form. In all these cases, simply centering can improve the computational difficulty and reduce the effect of limited precision arithmetic.

2.1.4 Accumulated Roundoff Error

Another problem with floating point arithmetic is the sheer accumulation of rounding error. While many applications run well in spite of a large number of calculations, some approaches encounter surprising problems. An enlightening example is just to add up many ones: $1 + 1 + 1 + \dots$. Astonishingly, this infinite

series appears to converge – the partial sums stop increasing as soon as the ratio of the new number to be added, in this case, one, to the current sum (n) drops below the machine epsilon. Following (2.5), we have $fl(n + 1) = fl(n)$, from which we find

$$1/n \approx \epsilon_m \quad \text{or} \quad n \approx 1/\epsilon_m .$$

So you will find the infinite series of ones converging to $1/\epsilon_m$. Moving to double precision arithmetic pushes this limit of accuracy sufficiently far to avoid most problems – but it does not eliminate them. A good mnemonic for assessing the effect of accumulated rounding error is that doing m additions amplifies the rounding error by a factor of m . For single precision, adding 1,000 numbers would look like a relative error of 10^{-4} which is often unacceptable, while moving to double precision would lead to an error of 10^{-13} . Avoidance strategies, such as adding smallest to largest and nested partial sums, are discussed in detail in Monahan, (2001, Chap. 2).

2.1.5 Interval Arithmetic

One of the more interesting methods for dealing with the inaccuracies of floating point arithmetic is interval arithmetic. The key is that a computer can only do arithmetic operations: addition, subtraction, multiplication, and division. The novel idea, though, is that instead of storing the number x , its lower and upper bounds (\underline{x}, \bar{x}) are stored, designating an interval for x . Bounds for each of these arithmetic operations can be then established as functions of the input. For addition, the relationship can be written as:

$$\underline{x} + \underline{y} < x + y < \bar{x} + \bar{y} .$$

Similar bounds for the other three operations can be established. The propagation of rounding error through each step is then captured by successive upper and lower bounds on intermediate quantities. This is especially effective in probability calculations using series or continued fraction expansions. The final result is an interval that we can confidently claim contains the desired calculation. The hope is always that interval is small. Software for performing interval arithmetic has been implemented in a practical fashion by modifying a Fortran compiler. See, for example, Hayes (2003) for an introductory survey, and Kearfott and Kreinovich (1996) for articles on applications.

2.2 Algorithms

An algorithm is a list of directed actions to accomplish a designated task. Cooking recipes are the best examples of algorithms in everyday life. The level of a cookbook reflects the skills of the cook: a gourmet cookbook may include the instruction

“saute the onion until transparent” while a beginner’s cookbook would describe how to choose and slice the onion, what kind of pan, the level of heat, etc. Since computers are inanimate objects incapable of thought, instructions for a computer algorithm must go much, much further to be completely clear and unambiguous, and include all details.

Most cooking recipes would be called *single pass* algorithms, since they are a list of commands to be completed in consecutive order. Repeating the execution of the same tasks, as in baking batches of cookies, would be described in algorithmic terms as *looping*. Looping is the most common feature in mathematical algorithms, where a specific task, or similar tasks, are to be repeated many times. The computation of an inner product is commonly implemented using a loop:

$$\mathbf{a}^\top \mathbf{b} = a_1 b_1 + a_2 b_2 + \dots + a_n b_n ,$$

implemented as

```
s = 0
do i = 1 to n
    s = s + ai × bi
end do
```

where the range of the loop includes the single statement with a multiplication and addition. In an *iterative* algorithm, the number of times the loop is repeated is not known in advance, but determined by some monitoring mechanism. For mathematical algorithms, the focus is most often monitoring convergence of a sequence or series. Care must be taken in implementing iterative algorithms to insure that, at some point, the loop will be terminated, otherwise an improperly coded procedure may proceed indefinitely in an *infinite loop*. Surprises occur when the convergence of an algorithm can be proven analytically, but, because of the discrete nature of floating point arithmetic, the procedure implementing that algorithm may not converge. For example, in a square-root problem to be examined further momentarily, we cannot find $x \in \mathcal{F}$ so that $x \times x$ is exactly equal to 2. The square of one number may be just below two, and the square of the next largest number in \mathcal{F} may be larger than 2. When monitoring convergence, common practice is to convert any test for equality of two floating point numbers or expressions to tests of closeness:

$$\text{if } (\text{abs}(x^*x - 2) < \text{eps}) \text{ then exit.} \quad (2.12)$$

Most mathematical algorithms have more sophisticated features. Some algorithms are *recursive*, employing relationships such as the gamma function: $\Gamma(x + 1) = x\Gamma(x)$ so that new values can be computed using previous values. Powerful recursive algorithms, such as the Fast Fourier Transform (FFT) and sorting algorithms, follow a *divide-and-conquer* paradigm: to solve a big problem, break it into little problems and use the solutions to the little problems to solve the big problem. In the case of sorting, the algorithm may look something like:

```

algorithm sort(list)
  break list into two pieces: first and second
  sort (first)
  sort (second)
  put sorted lists first and second together to form
    one sorted list
end algorithm sort

```

Implemented recursively, a big problem is quickly broken into tiny pieces and the key to the performance of divide-and-conquer algorithms is in combining the solutions to lots of little problems to address the big problem. In cases where these solutions can be easily combined, these recursive algorithms can achieve remarkable breakthroughs in performance. In the case of sorting, the standard algorithm, known as bubblesort, takes $O(n^2)$ work to sort a problem of size n – if the size of the problem is doubled, the work goes up by factor of 4. The Discrete Fourier Transform, when written as the multiplication of an $n \times n$ matrix and a vector, involves n^2 multiplications and additions. In both cases, the problem is broken into two subproblems, and the mathematics of divide and conquer follows a simple recursive relationship, that the time/work $T(n)$ to solve a problem of size n is the twice the time/work to solve two subproblem with half the size, plus the time/work $C(n)$, to put the solutions together:

$$T(n) = 2T(n/2) + C(n) . \quad (2.13)$$

In both sorting and the Discrete Fourier Transform, $C(n) \approx cn + d$, which leads to $T(n) = cn \log(n) + O(n)$. A function growing at the rate $O(n \log n)$ grows so much slower than $O(n^2)$, that the moniker “Fast” in Fast Fourier Transform is well deserved. While some computer languages preclude the use of recursion, recursive algorithms can often be implemented without explicit recursion through clever programming.

The performance of an algorithm may be measured in many ways, depending on the characteristics of the problems the it may be intended to solve. The sample variance problem above provides an example. The simple algorithm using (2.7) requires minimal storage and computation, but may lose accuracy when the variance is much smaller than the mean: the common test problem for exhibiting catastrophic cancellation employs $y_i = 2^{12} + i$ for single precision. The two-pass method (2.8) requires all of the observations to be stored, but provides the most accuracy and least computation. Centering using the first observation (2.9) is nearly as fast, requires no extra storage, and its accuracy only suffers when the first observation is unlike the others. The last method, arising from the use of Givens transformations (2.10) and (2.11), also requires no extra storage, gives sound accuracy, but requires more computation. As commonly seen in the marketplace of ideas, the inferior methods have not survived, and the remaining competitors all have tradeoffs with speed, storage, and numerical stability.

2.2.1 Iterative Algorithms

The most common difficult numerical problems in statistics involve optimization, or root-finding: maximum likelihood, nonlinear least squares, M-estimation, solving the likelihood equations or generalized estimating equations. And the algorithms for solving these problems are typically iterative algorithms, using the results from the current step to direct the next step.

To illustrate, consider the problem of computing the square root of a real number y . Following from the previous discussion of floating point arithmetic, we can restrict y to the interval $(1, 2)$. One approach is to view the problem as a root-finding problem, that is, we seek x such that $f(x) = x^2 - y = 0$. The bisection algorithm is a simple, stable method for finding a root. In this case, we may start with an interval known to contain the root, say (x_1, x_2) , with $x_1 = 1$ and $x_2 = 2$. Then bisection tries $x_3 = 1.5$, the midpoint of the current interval. If $f(x_3) < 0$, then $x_3 < \sqrt{y} < x_2$, and the root is known to belong in the new interval (x_3, x_2) . The algorithm continues by testing the midpoint of the current interval, and eliminating half of the interval. The rate of convergence of this algorithm is *linear*, since the interval of uncertainty, in this case, is cut by a constant $(1/2)$ with each step. For other algorithms, we may measure the rate at which the distance from the root decreases. Adapting Newton's method to this root-finding problem yields Heron's iteration

$$x_{n+1} = \frac{1}{2}(x_n + y/x_n) .$$

Denoting the solution as $x^* = \sqrt{y}$, the error at step n can be defined as $\epsilon_n = x_n - x^*$, leading to the relationship

$$\epsilon_{n+1} = \frac{1}{2} \frac{\epsilon_n^2}{x_n} . \quad (2.14)$$

This relationship of the errors is usually called *quadratic convergence*, since the new error is proportional to the square of the error at the previous step. The relative error $\delta_n = (x_n - x^*)/x^*$ follows a similar relationship,

$$\delta_n = \frac{1}{2} \delta_n^2 / (1 + \delta_n) . \quad (2.15)$$

Here, the number of accurate digits is doubled with each iteration. For the secant algorithm, analysis of the error often leads to a relationship similar to (2.14), but $|\epsilon_{n+1}| \approx C|\epsilon_n|^p$, with $1 < p < 2$, achieving a rate of convergence known as *superlinear*. For some well-defined problems, as the square root problem above, the number of iterations needed to reduce the error or relative error below some criterion can be determined in advance.

While we can stop this algorithm when $f(x_n) = 0$, as discussed previously, there may not be any floating point number that will give a zero to the function, hence the stopping rule (2.12). Often in root-finding problems, we stop when $|f(x_n)|$ is small

enough. In some problems, the appropriate “small enough” quantity to ensure the desired accuracy may depend on parameters of the problem, as in this case, the value of y . As a result, termination criterion for the algorithm is changed to: stop when the relative change in x is small

$$|x_{n+1} - x_n|/|x_n| < \delta .$$

While this condition may cause premature stopping in rare cases, it will prevent infinite looping in other cases. Many optimization algorithms permit the iteration to be terminated using any combination – and “small enough” is within the user’s control. Nevertheless, unless the user learns a lot about the nature of the problem at hand, an unrealistic demand for accuracy can lead to unachievable termination criteria, and an endless search.

As discussed previously, rounding error with floating point computation affects the level of accuracy that is possible with iterative algorithms for root-finding. In general, the relative error in the root is at the same relative level as the computation of the function. While optimization problems have many of the same characteristics as root-finding problems, the effect of computational error is a bit more substantial: k digits of accuracy in the function to be optimization can produce but $k/2$ digits in the root/solution.

2.2.2 *Iterative Algorithms for Optimization and Nonlinear Equations*

In the multidimensional case, the common problems are solving a system of nonlinear equations or optimizing a function of several variables. The most common tools for these problems are Newton’s method or secant-like variations. Given the appropriate regularity conditions, again we can achieve quadratic convergence with Newton’s method, and superlinear convergence with secant-like variations. In the case of optimization, we seek to minimize $f(x)$, and Newton’s method is based on minimizing the quadratic approximation:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^\top \nabla f(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^\top \nabla^2 f(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) .$$

This leads to the iteration step

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - [\nabla^2 f(\mathbf{x}^{(n)})]^{-1} \nabla f(\mathbf{x}^{(n)}) .$$

In the case of solving a system of nonlinear equations, $\mathbf{g}(\mathbf{x}) = \mathbf{0}$, Newton’s method arises from solving the affine (linear) approximation

$$\mathbf{g}(\mathbf{x}) \approx \mathbf{g}(\mathbf{x}_0) + \mathbf{J}_g(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) ,$$

leading to a similar iteration step

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - [\mathbf{J}_g(\mathbf{x}^{(n)})]^{-1} \mathbf{g}(\mathbf{x}^{(n)}) .$$

In both cases, under suitable smoothness conditions, the Newton iteration will achieve quadratic convergence – using norms to measure the error at each step:

$$\|\mathbf{x}^{(n+1)} - \mathbf{x}^*\| \approx C \|\mathbf{x}^{(n)} - \mathbf{x}^*\|^2 .$$

For both problems, Newton's method requires the computation of lots of derivatives, either the gradient $\nabla f(\mathbf{x}_0)$ and Hessian $\nabla^2 f(\mathbf{x}_0)$, or the Jacobian matrix $\mathbf{J}_g(\mathbf{x}^{(n)})$. In the univariate root-finding problem, the secant method arises by approximating the derivative with the first difference using the previous evaluation of the function. Secant analogues can be constructed for both the optimization and nonlinear equations problems, with similar reduction in the convergence rate: from quadratic to superlinear.

In both problems, the scaling of the parameters is quite important, as measuring the error with the Euclidean norm presupposes that errors in each component are equally weighted. Most software for optimization includes a parameter vector for suitably scaling the parameters, so that one larger parameter does not dominate the convergence decision. In solving nonlinear equations, the condition of the problem is given by

$$\|\mathbf{J}_g(\mathbf{x}^{(n)})\| \left\| [\mathbf{J}_g(\mathbf{x}^{(n)})]^{-1} \right\|$$

(as in solving linear equations) and the problem of scaling involves the components of $\mathbf{g}(\mathbf{x})$. In many statistical problems, such as robust regression, the normal parameter scaling issues arise with the covariates and their coefficients. However, one component of $\mathbf{g}(\mathbf{x})$, associated with the error scale parameter may be orders of magnitude larger or smaller than the other equations. As with parameter scaling, this is often best done by the user and is not easily overcome automatically.

With the optimization problem, there is a natural scaling with $\nabla f(\mathbf{x}_0)$ in contrast with the Jacobian matrix. Here, the eigenvectors of the Hessian matrix $\nabla^2 f(\mathbf{x}_0)$ dictate the condition of the problem; see, for example, [Gill et al. \(1981\)](#) and [Dennis and Schnabel \(1983\)](#). Again, parameter scaling remains one of the most important tools.

References

- Bodily, C.H.: Numerical Differentiation Using Statistical Design. Ph.D. Thesis, NC State University (2002)
- Chan, T.F., Golub, G.H., LeVeque, R.J.: Algorithms for computing the sample variance. *Am. Stat.* **37**, 242–247 (1983)
- Dennis, J.E. Jr., Schnabel, R.B.: Numerical Methods for Unconstrained Optimization. Prentice-Hall, Englewood Cliffs, NJ (1983)

- Gill, P.E., Murray, W., Wright, M.H.: Practical Optimisation. Academic Press, London (1981)
- Goldberg, D. (1991) *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys 23(1):5–48.
- Hayes, B.: A lucid interval. Am. Sci. **91**, 484–488 (2003)
- Institute of Electrical and Electronics Engineers: A Proposed IEEE-CS Standard for Binary Floating Point Arithmetic. Standard, 754–1985, IEEE, New York (1985)
- Kearfott, R.B., Kreinovich, V. (ed.): Applications of Interval Computations. Boston, Kluwer (1996)
- Knuth, D.E.: The Art of Computer Programming, Seminumerical Algorithms, (3rd edn.), Vol. 2, Addison-Wesley, Reading MA (1997)
- Monahan, J.F.: Numerical Methods of Statistics. Cambridge University Press, Cambridge (2001)
- Overton, M.L.: Numerical Computing with IEEE Floating Point Arithmetic. Philadelphia, SIAM (2001)

Handbook of Computational Statistics

Concepts and Methods

Gentle, J.E.; Härdle, W.K.; Mori, Y. (Eds.)

2012, XII, 1192 p. 297 illus., 96 illus. in color. In 2

volumes, not available separately., Hardcover

ISBN: 978-3-642-21550-6