Statistics 580

Introduction to Numerical Computing

Number Systems

In the decimal system we use the 10 numeric symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 to represent numbers. The relative position of each symbol determines the magnitude or the value of the number.

`Example:`

6594 is expressible as

$6594 = 6 \cdot 10^3 + 5 \cdot 10^2 + 9 \cdot 10^1 + 4 \cdot 10^0$

`Example:`

436.578 is expressible as

$436.578 = 4 \cdot 10^2 + 3 \cdot 10^1 + 6 \cdot 10^0 + 5 \cdot 10^{-1} + 7 \cdot 10^{-2} + 8 \cdot 10^{-3}$

We call the number 10 the base of the system. In general we can represent numbers in any system in the polynomial form:

$$z = (\cdots a_k a_{k-1} \cdots a_1\, a_0 .\, b_0\, b_1 \cdots b_m \cdots)_B$$

where B is the *base* of the system and the period in the middle is the *radix* point. In computers, numbers are represented in the `binary system`. In the binary system, the base is 2 and the only numeric symbols we need to represent numbers in binary are 0 and 1.

`Example:`

0100110 is a binary number whose value in the decimal system is calculated as follows:

$$\begin{aligned} 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 &= 32 + 4 + 2 \\ &= (38)_{10} \end{aligned}$$

The `hexadecimal system` is another useful number system in computer work. In this system, the base is 16 and the 16 numeric and arabic symbols used to represent numbers are:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

`Example:`

Let 26 be a hexadecimal number. It's decimal value is:

$$(26)_{16} = 2 \cdot 16^1 + 6 \cdot 16^0 = 32 + 6 = (38)_{10}$$

`Example:`

$$\begin{aligned} (\text{DB7C})_{16} &= 13 \cdot 16^3 + 11 \cdot 16^2 + 7 \cdot 16^2 + 12 \cdot 16^0 \\ &= (56,188)_{10} \end{aligned}$$

`Example:`

$$\begin{aligned} (\text{2CA.B6})_{16} &= 2 \cdot 16^2 + 12 \cdot 16_1 + 10 \cdot 16^0 + 11 \cdot 16^{-1} + 6 \cdot 16^{-2} \\ &= 2 \cdot 256 + 12 \cdot 16 + 10 \cdot 1 + 11/16 + 6/256 \\ &= (714.7109275)_{10} \end{aligned}$$

It is easy to convert numbers from hexadecimal to binary and back since a maximum of four binary digits can represent one hex digit. To convert from hex to binary one replaces each hex digit by its binary equivalent.
`Example:`

$$(\text{E}7)_{16} = (1110 \quad 0111)_2 = (11100111)_2$$

`Example:`

$$\begin{aligned}(2\text{CA.B}6)_{16} &= (001011001010.10110110)_2 \\ (.828125)_{10} &= (.\text{D}4)_{16} = (.110101)_2 \\ (149.25)_{10} &= (95.4)_{16} = (10010101.0100)_2\end{aligned}$$

Fixed Point and Floating Point Numbers

The term fixed point implies that the radix point is always placed on the right end of the list of digits (usually implicitly), implying that the above representation, the digits $b_0\, b_1 \cdots$ are all zero. The set of numbers thus obtained is analogous to the set of integers. The floating-point representation of numbers is based on the scientific notation we are familiar with. For example, the numbers 635.837 and 0.0025361, respectively, are expressed in scientific notation in the form:

$$0.635837 \times 10^3 \text{ and } 0.2536 \times 10^{-2} \tag{1}$$

The general form for decimal numbers is $a \times 10^b$ where, in *normalized* form, a is determined such that $1 \leq a < 10$. Thus the above numbers in the normalized form are:

$$6.35837 \times 10^2 \text{ and } 2.5361 \times 10^{-3}, \tag{2}$$

respectively. The radix point *floats* to the right of the first nonzero digit and the exponent of the base adjusted accordingly; hence the name *floating-point*. In normalized form, the exponent b gives the number's order of magnitude. The convention is that floating-point numbers should always be normalized except during calculations. In this note, for simplicity the following convention is adopted when representing numbers in any floating-point system. A floating-point number in base β is written as an ordered pair:

$$(\pm\, d_0.d_1, \ldots, d_{t-1}, e)_\beta$$

and has the value $\pm\, d_0.d_1, \ldots, d_{t-1} \times \beta^e$. For example,

using $t = 4$	and decimal i.e., $\beta = 10$,	13.25	is represented as	$(+1.325, 1)_{10}$
using $t = 5$	and binary i.e., $\beta = 2$,	-38.0	is represented as	$(-1.0011, 5)_2$
using $t = 6$	and hexadecimal i.e., $\beta = 16$,	0.8	is represented as	$(+\text{C.CCCCC}, -1)_{16}$

The choice of β and the way the bits of a memory location are allocated for storing e and how many digits t are stored as the *significand*, varies among different computer architectures. Together, the base β and the number of digits t determine the *precision* of the numbers stored. Many computer systems provide for two different lengths in number of digits in base β of the significand (which is denoted by t), called `single precision` and `double precision`, respectively. The size of the exponent e, which is also limited by the number of digits available to store it, and in what form it is stored determines the *range* or magnitude of the numbers stored.

Representation of Integers on Computers

An integer is stored as a binary number in a string of m contiguous bits. Negative numbers are stored as their 2's complement, i.e., 2^m - the number. In the floating-point system available in many computers including those using , a bit string of length 32 bits is provided for storing integers. We shall discuss the integer representation using 32 bits; other representations are similar.

It is convenient to picture the 32 bit string as a sequence of binary positions arranged from left to right and to label them from 0 to 31 (as $b_0, b_1, \ldots, b_{31}$) for purpose of reference. When a positive integer is to be represented, its binary equivalent is placed in this bit string with the least significant digit in the rightmost bit position i.e., as b_{31}. For example, $(38)_{10} = (00100110)_2$ is stored as:

$$\boxed{000 \ldots\ldots\ldots\ldots 00100110}$$

The left-most bit is labelled b_0 is called the `sign bit` and for positive integers the sign bit will always be set to 0. Thus the largest positive integer that can be stored is

$$\boxed{0111 \ldots\ldots\ldots\ldots 11111} = 2^0 + 2^1 + \cdots + 2^{30} = 2^{31} - 1 = 2,147,483,647\,.$$

A negative integer is stored as its two's complement. The two's complement of the negative integer $-i$ is the 32 bit binary number formed by $2^{32} - i$. For example, $(-38)_{10}$ is stored as:

$$\boxed{111 \ldots\ldots\ldots\ldots 11011010}$$

Note that the sign bit is now automatically set to a 1 and therefore this pattern is identified as representing a negative number by the computer. Thus the smallest negative integer that can be stored is

$$\boxed{100 \ldots\ldots\ldots\ldots 00} = -2^{31} = -2,147,483,648\,.$$

From the above discussion it becomes clear that the set of numbers that can be stored as integers in 32 bits machines are the integers in the range -2^{31} to $2^{31} - 1$. The two's complement representation of negative numbers in the integer mode allows the treatment of subtraction as an addition in integer arithmetic. For e.g., consider the subtraction of the two numbers +48 and +21 stored in strings of 8 bits, i.e., m = 8 (for convenience and consider the leading bit as a sign bit).

$$\begin{array}{rl} +48 & 00110000 = 48 \\ \underline{-21} & \underline{11101011} = 235 = 2^8 - 21 \\ +27 & 00011011 = 283 = 256 + 27 \equiv 27 \bmod 2^8 \end{array}$$

Note that *overflow* occurred in computing 283 because it cannot be stored in 8 bits, but a *carry* was made into and out of the sign bit so the answer is deemed correct. A carry is made into the sign bit but not out of in the following so the answer is incorrect:

$$\begin{array}{rl} +48 & 00110000 \\ \underline{+96} & \underline{01100000} \\ +144 & 10010000 = 256 - 112 \equiv -112 \bmod 2^8 \end{array}$$

Floating-point Representation (FP)

The IEEE Standard for Binary Floating Point Arithmetic (ANSI/IEEE 754-1985) defines formats for representing floating-point numbers and a set operations for performing floating-point arithmetic in computers. It is the most widely-used standard for floating-point operations on modern computers including Intel-based PC's, Macintoshes, and most Unix platforms.

It says that hardware should be able to represent a subset F of real numbers called *floating-point numbers* using three basic components: s, the sign, e, the exponent, and m, the mantissa. The value of a number in F is calculated as

$$(-1)^s \times m \times 2^e$$

The exponent base (2) is implicit and need not be stored. Most computers store floating-point numbers in two formats: single (32-bit) and double (64-bit) precision. In the `single precision` format numbers are stored in 32-bit strings (i.e., 4 bytes) with the number of bits available partitioned as follows:

where

- s = 0 or 1, denotes sign of the number, the *sign bit* is 0 for positive and 1 for negative
- f = $d_1 d_2 \ldots d_{23}$ where $m = (1.d_1 d_2 \ldots d_{23})_2$ is called the `mantissa` or the significand
- E = $e + 127$ (i.e., the exponent is stored in binary with 127 added to it or in *biased form*)

When numbers are converted to *binary floating-point* format, the most significant bit of the mantissa will always be a 1 in the normalized form making the storage of this bit redundant. It is sometimes called the *hidden bit*. Hence it is not stored but implicitly assumed to be present and only the fraction part denoted as f above is stored. Thus, in effect, to represent a normalized 24-bit mantissa in binary, only 23 bits are needed. The bit thus saved is used to increase the space available to store the biased exponent E. Because E is stored in 8 bits (as shown above) possible values for E are in the range $0 \leq E \leq 255$

Since the exponent is stored in the biased form, both positive and negative exponents can be represented without using a separate sign bit. A *bias* (somtimes called the *excess*) is added to the actual exponent so that the result stored as E is always positive. For IEEE single-precision floats, the bias is 127. Thus, for example, to represent an exponent of zero, a value of $0 + 127$ in binary is stored in the exponent field. A stored value of 100 indicates implies an exponent of $(100 - 127)$, or -73. Exponents of -127 or E=0 (E field is all 0's) and $+128$ or E=255 (E field is all 1's) are reserved for representing results of special calculations.

In `double precision`, numbers are stored in 64-bit strings (i.e., 8 bytes) with the exponent biased with 1023, i.e., E = $e + 1023$ and stored in 11 bits and f, the fraction part of the mantissa, stored in 52 bits.

Some examples of numbers stored in single precision floating-point follow:

- $(38.0)_{10}$ is represented as $(1.0011, 5)_2$ and is stored in 32-bit single precision floating-point as:

0	10000100	00110000 ··· 0

- $(-149.25)_{10}$ is represented as $(-1.001010101, 7)_2$ and is stored in 32-bit single precision floating-point as:

1	10000110	0010101010 ··· 0

- $(1.0)_{10}$ is represented as $(1.0, 0)_2$ and is stored in 32-bit single precision floating-point as:

0	01111111	00000000 ··· 0

- $(0.022460938)_{10}$ is represented as $(1.01110, -6)_2$ and is stored in single precision 32-bit floating-point as:

0	01111001	01110000 ··· 0

The range of the positive numbers that can be stored in single precision is 2^{-126} to $(2 - 2^{-23}) \times 2^{127}$, or in decimal, $\approx 1.175494351 \times 10^{-38}$ to $\approx 3.4028235 \times 10^{38}$. In double precision, the range is 2^{-1022} to $(2-2^{-52})\times 2^{1023}$ or, in decimal, $\approx 2.2250738585072014 \times e^{-308}$ to $\approx 1.7976931348623157^{308}$. Since the sign of floating-point numbers is given in a separate bit, the range for negative numbers is given by the negation of the above values.

Note that -126 is the smallest exponent for a normalized number because 1 is the smallest possible nonzero value for E for normalized numbers. Similarly, +127 is the largest exponent for a normalized number because 254 is the largest possible value for E for normalized numbers. When E is out of this allowable range of values but is still representible in 8 bits the result is encoded to the following values:

1. E = 0 and m = 0 encodes to ± 0
2. E = 0 and $-1 < m < 1$ encodes to $\pm 0.m \times 2^{-126}$
3. E = 255 and $m \neq 0$ encodes to NaN
4. E = 255 and $m = 0$ encodes to $\pm\infty$

NaN is an abbreviation for the phrase *not a number* implying that the encoding represents the result of an operation that cannot be represented in floating-point.

This means that there are two zeroes, $+0$ (s is set to 0) and -0 (s is set to 1) and two infinities $+\infty$ (s is set to 0) and $-\infty$ (s is set to 1). *NaN*s may have a sign and a significand, but these have no meaning other than for diagnostics; the first bit of the significand is often used to distinguish *signaling NaN*s from *quiet NaN*s. Note that NaNs and infinities have all 1's in the E field.

An interesting case is the second situation described above which allows numbers smaller than the smallest normalized number representible in the floating-point format to be stored as a unnormalized floating-point number. As noted above, the smallest normalized number representible in the floating-point format is $(\pm 1.0, -126)_2$. Numbers with smaller exponents than -126 can be stored in the floating point format by denormalizing the number and shifting the radix pont a number of places to the left so that the exponent is always adjusted to -127. For example, the number $(+1.0, -127)_2$ can be stored in denormalized form because it is equal to $(+0.1, -126)_2$, and will be in single precision 32-bit floating-point as:

0	00000000	100000 ··· 0

Thus, these numbers are representable in floating-point format with E=0 and $m = 0.f$, and are called *subnormal* or *denormalized* numbers. The smallest non-zero positive and largest non-zero negative denormalized numbers (represented by all 0's in the E field and the binary value 1 in the f field) are 2^{-149} (or $\approx 1.4012985 \times 10^{-45}$ in decimal).

Operations on special numbers are well-defined by the IEEE standard. In the simplest case, any operation with a *NaN* yields a *NaN* result. These and other operations are summarized in the following table:

Operation	Result
number $\div \infty$	0
$\pm\infty \times \pm\infty$	$\pm\infty$
$\pm$nonzero $\div 0$	$\pm\infty$
$\infty + \infty$	∞
$\pm 0 \div \pm 0$	NaN
$\pm\infty - \pm\infty$	NaN
$\pm\infty \div \pm\infty$	NaN
$\pm\infty \times 0$	NaN

Floating-point Arithmetic

Because of the finite precision of floating-point numbers, floating-point arithmetic can only approximate real arithmetic. In particular, floating-point arithmetic is commutatative but not associative, implying that arithmetic expressions written in different order of operations may result in different results.

Since many real numbers do not have floating-point equivalents, floating-point operations such as addition and multiplication, may result in approximations to the exact answer obtained using exact arithmetic. IEEE standard includes *rounding modes* which are methods to select a floating-point number to approximate the true result. The symbols $\oplus, \ominus, \otimes$ and $\oslash$ are used to represent floating-point operations equivalent to real operations of $+, -, \times$, and $\div$, respectively. The IEEE standard requires that these operations, plus square root,

remainder, and conversion between integer and floating-point be *correctly rounded*. That is, the result must be computed exactly and then rounded to the nearest floating-point number. The IEEE standard has four different rounding modes, the default mode being *round to even* or *unbiased rounding*, which rounds to the nearest value. If the number falls midway it is rounded to the nearest value with an even (or zero) least significant bit. Other rounding modes allowed are towards zero, towards positive infinity, and towards negative infinity.

While different machines may implement floating-point arithmetic somewhat differently, the basic steps are common and the operations are carried out in floating-point registers capable of holding additional bits than the source operands require. This means that floating-point arithmetic operations may utilize more bits than are used to represent the individual operands. However, some of the operations discussed below may result in bits falling off the least significant end even when additional bits are used. Usually, the FP register has a *guard digit* to hold at least one bit of overflow of least significant bits so that it could be used for rounding later. As an example, consider the addition (or subtraction) of two numbers. The steps needed can be summarized as follows:

1. The two operands are first brought into FP registers.
2. The exponents E of the two operands are compared and the mantissa of the number with the smaller exponent is shifted to the right a number of digits equal to the difference between the two exponents. This operation is called `alignment` and may result in a carry to the guard digit.
3. The larger of the two exponents is made the exponent of the result.
4. Add (or subtract) the two mantissa. This uses the guard digit and may alter its value.
5. The correct sign of the result is determined (using an algorithm).
6. The resultant mantissa is `normalized` by shifting digits to the left and the exponent is decreased by the number of digits needed to be so shifted.

Some examples that illustrate this process follow:

1. Two decimal floating-point numbers are added using decimal floating-point arithmetic using infinite precision:

$$8.451 \times 10^3 + 3.76 \times 10^{-1} \equiv (+8.451, 3)_{10} + (+3.76, -1)_{10}$$

$$\begin{aligned} &8.451000 \times 10^3 \\ &+0.000376 \times 10^3 \\ &= 8.451376 \times 10^3 \end{aligned}$$

   The result is already normalized; so the answer is 8.451376×10^4.

2. The same two decimal floating-point numbers are added using 5-digit floating point arithmetic with a guard digit:

$$\begin{aligned} &8.4510[0] \times 10^3 \\ &+0.0003[7] \times 10^3 \\ &= 8.4513[7] \times 10^3 \end{aligned}$$

Rounding to the nearest gives the result 8.4514×10^4

3. adding $(+3.0, 0)_{10} = (+1.100, 1)_2$ to $(+2.0, 0)_{10} = (+1.000, 1)_2$ in 32-bit floating point arithmetic with a guard digit:

$$\begin{aligned} &1.10000000000000000000000[0] \times 2^1 \\ &+1.00000000000000000000000[0] \times 2^1 \\ &= 10.10000000000000000000000[0] \times 2^1 \end{aligned}$$

   The answer is then normalized to $(+1.010, 2)$ which is equal to $(+5.0, 0)_{10}$

4. Add $(3.0, 0)_{10} = (1.100, 1)_2$ to $(0.75, 0)_{10} = (1.100, -2)_2$:

$$\begin{aligned} &1.10000000000000000000000[0] \times 2^1 \\ &+0.01100000000000000000000[0] \times 2^1 \\ &= 1.11100000000000000000000[0] \times 2^1. \end{aligned}$$

   Note that no rounding is necessary as the guard digit is set to zero. The result is already normalized and is thus $(+1.1110, 1)_2$ or $(+3.75, 0)_{10}$

5. Now consider adding 3 to 3×2^{-23} or $\equiv (+1.100, 1)_2 + (+1.100, -22)_2$. Then, :

$$\begin{aligned} &1.10000000000000000000000[0] \times 2^1 \\ &+0.00000000000000000000001[1] \times 2^1 \\ &= 1.10000000000000000000001[1] \times 2^1 \end{aligned}$$

   The result is first rounded to the nearest by rounding to the even number by adding 1 to the last digit *because the guard digit is set to 1.* The answer is already normalized and is thus equal to $(1.10000000000000000000010, 1)_2$

In the light of the above exposition, it is possible to demonstrate using 5-digit decimal floating-point arithmetic that floating-point arithmetic is not associative. Consider the the three numbers $a = 3 = (+3.0000, 0)_{10}$,$b = 40004 = (+4.0004, 4)_{10}$,$c = 60000 = (+6.0000, 4)_{10}$. It is easy to check that the expressions $(a + b) + c$ and $a + (b + c)$ result in $(+1.0001, 4)_{10}$ and $(+1.0000, 4)$, respectively, assuming a guard digit and rounding to the nearest takes place in every intermediate computation.

In much of the following discussion, hypothetical computers implementing decimal floating-point representation and floating-point arithmetic with finite precision and rounding is used to illustrate various implications of floating-point arithmetic. In general, this helps to fix ideas and understand the concept and avoids the complication of dealing with binary arithmetic and large numbers of binary digits.

Implications of floating-point representation and floating-point arithmetic

a) As observed earlier not all real numbers can be represented exactly as a floating-point number conforming to the IEEE standard. For example, it may be observed that the numbers between 6.3247 and 6.3247 are not representable using 5-digit decimal floating-point format. Also note that the same number of real numbers in each of the ranges $[10^0, 10^1]$, $[10^1, 10^2]$, ... can be representable in the 5-digit decimal floating-point format.

   In the IEEE floating-point format the corresponding intervals are $[2^0, 2^1]$, $[2^1, 2^2]$, ... Thus the same number of real numbers are representable in intervals that are getting wider. The denseness of representable numbers gets lesser as the range increases. This implyies that floating-point number representable are not evenly distributed over the set of real numbers. The expectation is that IEEE floating-point format allows the representation of most real numbers to an acceptable degree of accuracy. For example, the decimal number 3.1415926535 is stored in single precision floating-point as

0	10000000	10010010000111111011011

   resulting in a value of 3.1415927 but in double precision floating-point representation as

0	10000000000	1001001000011111101101010100010000010001011101000100

   that results in the decimal equivalent of 3.1415926535000000, i.e. exactly.

b) The error in representing a real number x as the nearest floating-point number $fl(x)$ is measured as the relative error $\epsilon = |fl(x) - x|/|x|$ which can be expressed in the form $fl(x) = x(1 \pm \epsilon)$. For example, the relative error in representing 3.1415926535 in 5-digit decimal floating-point is $|3.1416 - 3.1415926535|/|3.1415926535| = .000002339$ implying that 3.1456 is actually 3.1415926535 times the factor 1.000002339. It can be shown that $|\epsilon| \leq u$ where u is called the *machine unit* and is the upper bound on the relative error in representing a number x in floating-point. For the IEEE binary floating-point standard with rounding to the nearest $u = \frac{1}{2}2^{-23}$

c) In addition to the creation of special values like NaN's and infinities as a result of arithmetic operations, arithmetic exceptions called *underflow* and *overflow* may occur if a computed result is outside the range representible. Negative numbers less than $-(2 - 2^{-23} \times 2^{127}$ cause *negative overflow*, and positive numbers greater than $(2 - 2^{-23} \times 2^{127}$ *positive overflow*. In the denormalized range, negative numbers greater than -2^{-149} result in negative underflow and positive numbers less than 2^{-149}, positive underflow. The IEEE standard requires exceptions to be flagged or an overflow/underflow *trap handler* to be called.

d) `Round-off error` occurs when floating-point arithmetic is carried out. For example, consider the addition of the two numbers 100.0 and 1.01 in floating-point in 3-digit decimal arithmetic (assuming rounding to the nearest and a single guard digit). In effect, the operation is performed as $(+1.00, 2)_{10} + (+0.010, 2)_{10}$ The exact sum is 101.01 which and the floating-point computation gives $(+1.01, 2)$ giving result of this operation as 101.0. Thus

$$\begin{aligned} \text{Absolute Error: } |fl(op) - op| &= .01 \\ \text{Relative Error: } \frac{|fl(op) - op|}{|op|} &= \frac{.01}{101.01} \sim .0001 \end{aligned}$$

Number of correct digits can be calcualted using $-\log_{10}(\text{Relative Error})$ which in this example is ≈ 4.
Accumulated Round-off (or Error Propagation) is a major problem in computation, For e.g., consider the computation of the inner product $\mathbf{x}^T\mathbf{y} = \sum x_i y_i$.

f) `Catastrophic Cancellation` is the extreme loss of significant digits when small numbers additively computed from large numbers. For example, consider the computation of the sample variance of 101, 102, 103, 104, 105 on a 4-digit decimal arithmetic using the formula $(\Sigma x^2 - n\,\bar{x}^2)/(n-1)$. The intermediate results are:
$101^2 \longrightarrow (+1.020, 4)$
$102^2 \longrightarrow (+1.040, 4)$
$103^2 \longrightarrow (+1.061, 4)$
$104^2 \longrightarrow (+1.082, 4)$
$105^2 \longrightarrow (+1.102, 4)$
$101^2 + 102^2 + \cdots + 105^2 \longrightarrow (+5.305, 4)$
$(101 + 102 + \cdots + 105)/5 \longrightarrow 103$
$103^2 \longrightarrow (+1.061, 4)$
$5 \times 103^2 \longrightarrow (+5.305, 4)$
Thus the final answer is computed as $\Sigma x^2 - n\,\bar{x}^2 = (+5.305, 4) - (+5.305, 4) = 0!$

This occurs because only 4 digits of the result of each computation are carried and these digits cancel each other out. The true answer depends entirely on those digits that were not carried.

Stability of Algorithms

In the previous example, data is not the problem but the algorithm is. The formula used does not provide a good algorithm for computing the sample variance. Algorithms that produce the best answer in FP for a wide array of input data are called *stable*; in particular *small* perturbations in the data must not produce *large* changes in the value computed using the algorithm. For the computation of sample variance it can easily be checked that on the same machine, the algorithm based on the formula $\Sigma(x_i - \bar{x})^2/(n-1)$ gives the exact answer, 2.5, for the sample variance of the data considered in the previous example, using the same precision arithmetic. In a later section, results of a study comparing the stability of these two algorithms are presented.

Other well-known examples are the computation of e^{-a} for $a > 0$ using a series expansion and evaluation of polynomials. Consider computing exp(−5.5) using the series:

$$exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots .$$

The required computations are summarized in the table given below.

Term Number	Term for $x = -5.5$	Cumul. Sum	Term for $x = 5.5$	Cumul. Sum
1	1.0000E+00	1.0000E+00	1.0000E+00	1.0000E+00
2	-5.5000E+00	-4.5000E+00	5.5000E+00	6.5000E+00
3	1.5125E+01	1.0625E+01	1.5125E+01	2.1625E+01
4	-2.7728E+01	-1.7103E+01	2.7728E+01	4.9350E+01
5	3.8127E+01	2.1024E+01	3.8127E+01	8.7477E+01
6	-4.1940E+01	-2.0916E+01	4.1940E+01	1.2941E+01
7	3.8444E+01	1.7528E+01	3.8444E+01	1.6785E+01
8	-3.0206E+01	-1.2678E+01	3.0206E+01	1.9805E+02
9	2.0767E+01	8.0890E+00	2.0767E+01	2.1881E+02
10	-1.2690E+01	-4.6010E+00	1.2690E+01	2.3150E+02
11	6.9799E+00	2.3789E+00	6.9799E+00	2.3847E+02
12	3.4900E+00	-1.1111E+00	3.4900E+00	2.4196E+02
13	1.5996E+00	4.8850E−01	1.5996E+00	2.4355E+02
14	-6.7674E−01	-1.8824E−01	6.7674E−01	2.4422E+02
15	2.6586E−01	7.7620E−02	2.6586E−01	2.4448E+02
16	-9.7483E−02	-1.9863E−02	9.7483E−02	2.4457E+02
17	3.3511E−02	1.3648E−02	3.3511E−02	2.4460E+02
18	-1.0841E−02	2.8070E−03	1.0841E−02	2.4461E+02
19	3.3127E−03	6.1197E−03	3.3127E−03	2.4461E+02
20	-9.5897E−04	5.1608E−03	9.5897E−04	
21	2.6370E−04	5.4245E−03	2.6370E−04	
22	-6.9068E−05	5.3555E−03	6.9068E−05	
23	1.7266E−05	5.3727E−03	1.7266E−05	
24	-4.1288E−06	5.3686E−03	4.1288E−06	
25	9.4623E−07	5.3695E−03	9.4623E−07	

From this table, the result is exp(−5.5) = 0.0053695 using 25 terms of the expansion directly. An alternative is to use the reciprocal of exp(5.5) to compute exp(−5.5). Using this method,

$\exp(-5.5) = 1/\exp(5.5) = 1/244.61 = 0.0040881$ is obtained using only 19 terms. The true value is 0.0040867. Thus using the truncated series directly to compute $\exp(-5.5)$ does not result in a single correct significant digit whereas using the reciprocal of $\exp(5.5)$ computed using the same series gives 3 correct digits are obtained.
When calculating polynomials of the form

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n \tag{3}$$

re-expressing $f(x)$ in other forms, some of which is discussed below, often leads to increased accuracy. The reason for this is that these representations result in more numerically stable algorithms than direct use of the `power form` given above. One such representation is called the shifted form where $f(x)$ is expressed in the form

$$f(x) = b_0 + b_1(x-c) + b_2(x-c)^2 + \cdots + b_n(x-c)^n \tag{4}$$

where the center c is a predetermined constant. It is easy to see that, given c, by comparing coefficients in (1) and (2) one can obtain values of $b_0, b_1, \ldots, b_n$ from those of $a_0, a_1, \ldots, a_n$. Note that this shifting can be accomplished by making a change of variable $t = x - c$.

Consider a simple parabola centered at $x = 5555.5$:

$$f(x) = 1 + (x - 5555.5)^2\,,$$

whose power form representation is

$$f(x) = 30863581.25 - 11111x + x^2$$

If we now compute f(5555) and f(5554.5) in 6-digit decimal arithmetic directly from the power form, we get

$$\begin{aligned}
f(5555) &= 3.08636 \times 10^7 - 1.11110 \times 10^4(5.5550 \times 10^3) + (5.5550 \times 10^3)^2 \\
&= 3.08636 \times 10^7 - 6.17216 \times 10^7 + 3.08580 \times 10^7 \\
&= 0 \\
f(5554.5) &= 3.08636 \times 10^7 - 6.17160 \times 10^7 + 3.08525 \times 10^7 \\
&= 0.00001 \times 10^7 = 100
\end{aligned}$$

the same arithmetic on the shifted form show that its evaluation produces accurate results.

Given a polynomial $f(x)$ in the power form, the center c is usually chosen somewhere in the 'center' of the x values for which $f(x)$ is expected to be evaluated. Once a value for c is determined, one needs to convert $f(x)$ from the power form to the shifted form by evaluating coefficients $b_0, b_1, \ldots, b_n$.

The coding of the shifted form may be accomplished by first rewriting f(x) expressed in shifted form as in (2), as follows:

$$f(x) = \{\ldots\{[b_n(x-c) + b_{n-1}](x-c) + b_{n-2}\}(x-c)\ldots\}(x-c) + b_0 \tag{5}$$

Given the coefficients $b_0, b_1, \ldots, b_n$ and the center c, this computation may be coded using the following algorithm:

Set $d_n = b_n$ and $z = x - c$.
For $i = n-1, n-2, \ldots, 0$ do
 set $d_i = b_i + z d_{i+1}$
 end do
Set $f(x) = d_0$.

In the simplest implementation of this algorithm, the center c is taken to be zero in expression (3) and the coefficients b_i are then equal to a_i for all $i = 1, \ldots, n$. $f(x)$ of (1) is then re-expressed in the form

$$f(x) = \{\ldots\{[a_n x + a_{n-1}]x + a_{n-2}\}x \ldots\}x + a_0 \,.$$

The resulting algorithm is often called `Horner's rule`. Consider the evaluation of

$$f(x) = 7 + 3x + 4x^2 - 9x^3 + 5x^4 + 2x^5 \,. \tag{6}$$

A C expression for evaluating (4) using Horner's rule is

$$((((2.0 * x + 5.0) * x - 9.0) * x + 4.0) * x + 3.0) * x + 7.0$$

which requires only 5 multiplications, which is 10 multiplications less than evaluating it using the expression

$$7.0 + 3.0 * x + 4.0 * x * x - 9.0 * x * x * x + \ldots$$

Condition Number

The term `condition` is used to denote a characteristic of input data relative to a computational problem. It measures the sensitivity of computed results to small relative changes in the input data.

`Ill-conditioned` problems produce a large relative change in the computed result from a small relative change in the data. Thus the `condition number` measures the relative change in the computed result due to a unit relative change in the data. Suppose that a numerical procedure is described as producing `output` from some `input`, i.e., output=f(input). Then we define a constant κ such that

$$\frac{|f(\text{input} + \delta) - \text{output}|}{\text{output}} \le \kappa \frac{|\delta|}{\text{input}}$$

κ is called the `condition number` and small condition numbers are obviously desirable. If this number is large for a particular computation performed on some data then that data set is said to be `ill-conditioned` with respect to the computation.

For some problems it is possible to define a `condition number` in closed-form. Suppose, for example, that the computational problem is to evaluate a function $f()$ at a specified point x. If x is perturbed slightly to $x+h$, the relative change in $f(x)$ can be approximated by

$$\frac{f(x+h) - f(x)}{f(x)} \approx \frac{h\, f'(x)}{f(x)} = \left(\frac{x\, f'(x)}{f(x)}\right) \frac{h}{x}$$

Thus

$$\kappa = \left| \frac{x\, f'(x)}{f(x)} \right|$$

represent the condition number for this problem.

`Example:`
Evaluate the condition number for computing $f(x) = \sin^{-1} x$ near $x = 1$. Obviously,

$$\frac{x\, f'(x)}{f(x)} = \frac{x}{\sqrt{1-x^2}\,\sin^{-1} x}$$

For x near 1, $\sin^{-1} x \approx \pi/2$ and the condition number becomes infinitely large. Thus small relative errors in x leads to large relative errors in $\sin^{-1} x$ near $x = 1$.

`Example:`
Chan and Lewis(1978) has shown that the condition number for the sample variance computation problem using the one-pass machine algorithm (see later in this note) is

$$\kappa = \sqrt{1 + \frac{\bar{x}^2}{s^2}} = \sqrt{1 + (CV)^{-2}}$$

Thus the coefficient of variation (CV) is a good measure of the condition of a data set relative the computation of s^2.

It is important to note that the same problem may be `well-conditioned` with respect to another set of data.

`Example:`
Consider the computation of

$$f(x) = \frac{1.01 + x}{1.01 - x}$$

for x near 2.0 and for x near 1.0.

For x near 2.0: $f(2) = -3.0404 \qquad f(2.01) = -3.02$

.5% change in data causes only .67% change in the computed result. Thus the problem is well-conditioned for x near 2.

For x near 1.0: $f(1) = 201 \qquad f(1.005) = 403$

.5% change in x produces a 100% change in $f(x)$. Thus the problem is seen to be ill-conditioned for x near 1. We may verify this by computing the condition number κ for this $f(x)$.

$$f'(x) = \frac{-2.02x}{(1.01 - x)^2}$$

Thus

$$\begin{aligned} \kappa &= \left| \frac{x\, f'(x)}{f(x)} \right| \\ &= \frac{2.02x^2}{1.01^2 - x^2} = \frac{2.02}{\frac{1.01^2}{x^2} - 1} \end{aligned}$$

Thus we see that κ is large near $x = 1$ leading to the conclusion that this computation is ill-conditioned near $x = 1$. Note, however, that the algorithm used (i.e., direct computation) is `stable` in FP. It gives “right” answers in both cases.

Thus the aim should be to develop algorithms that will be stable for both well-conditioned and ill-conditioned data with respect to a specified computational problem. In many cases simply rexpressing computational formulas may work, However, for difficult problems more complex approaches may be neccessary. As we shall see later, the Youngs-Cramer algorithm is an example of a stable algorithm for the computation of the sample variance s^2

Order of Convergence

Order Relations

These are central to the development of asymptotics. Consider functions $f(x)$ and $g(x)$ defined in the interval I (∞ or $-\infty$ can be a boundary of I). Let x_0 be either an interval point or a boundary point of I with $g(x) \neq 0$ for x near x_0 (but $g(x_0)$ itself may be zero). By definition:

1. If $\exists$ a constant M such that
$$|f(x)| \leq M\, |g(x)|$$
as $x \to x_0$ then $f(x) = O(g(x))$.

2. If
$$\lim_{x\to x_0} \frac{f(x)}{g(x)} = 0$$
then $f(x) = o(g(x))$. Note: $f(x) = o(g(x))$ implies $f(x) = O(g(x))$ thus $O()$ is the weeker relation.

3. If
$$\lim_{x\to x_0} \frac{f(x)}{g(x)} = 1$$
then $f(x) \asymp g(x)$ i.e.,$f(x)$ and $g(x)$ are asymptotically equivalent.

In practice, $f(x)$ and $g(x)$ are usually defined in $\{1,2,\ldots\}$ instead of in an interval and $x_0 = \infty$.

Examples:

1. $I = (1, \infty)$ $\quad e^x = O(\sinh x)$ as $x \to \infty$

   Proof:
$$\frac{e^x}{(e^x - e^{-x})/2} = \frac{2}{1 - e^{-2x}} \leq \frac{2}{1 - e^{-2}}$$

2. $I = (0, \infty)$ $\quad \sin^2 x = o(x)$ as $x \to 0$

   Proof:
$$\lim_{x\to 0} \frac{\sin^2 x}{x} = \lim_{x\to 0} \sin x \lim_{x\to 0} \frac{\sin x}{x} = 0 \times 1.$$

3. $I = (0, \infty)$ $\quad \frac{x^2+1}{x+1} \asymp x$ as $x \to \infty$

   Proof:
$$\begin{aligned}
\frac{x^2+1}{x+1} &= \frac{x^2+1}{x(1+1/x)} \\
&= (x + 1/x) \sum_{k=0}^{\infty} (\frac{-1}{x})^k \\
&= x - 1 - 2 \sum_{k=1}^{\infty} (\frac{-1}{x})^k
\end{aligned}$$

   Thus we can say
$$\frac{x^2+1}{x+1} = x - 1 + O(1/x)$$

Applications in Computing

If an infinite series (e.g., log) is approximated by a finite sum, the omitted terms represent `approximation error`. Here, we need to estimate this error (or a bound for it) theoretically. Usually this error depends on some parameters: e.g.,

n = number of terms in the finite series

h = "step size" in numerical differentiation/integration

In an iterative procedure we need to be able to test whether the process has `converged`.

In addition, we need to be able to measure the `rate of convergence`; for e.g. say that a method converges like $1/n$ or h^2 or that the approximation error is of the order $1/n^2$. We use "big O" to measure `order of convergence`. If we have sequences $\{a_n\}$ and $\{b_n\}$ we say that

$$a_n = O(b_n) \text{ if } \frac{|a_n|}{|b_n|} < \infty \text{ as } n \to \infty$$

For e.g., $\frac{5}{n^2}$, $\frac{10}{n^2} + \frac{1}{n^3}$, $\frac{-6.2}{n^2} + \frac{e^{-n}}{n}$ are all of $O(1/n^2)$ as $n \to \infty$ and $4h$, $3h + \frac{h^2}{\log h}$, $-h + h^2 - h^3$ are all of $O(h)$ as $h \to 0$

Definition of Convergence Rate:

If s = solution and $\epsilon_i = |x_i - s|$ where $x_1, x_2, \ldots$ are successive iterates then the iteration is said to exhibit *convergence rate* β if the limit

$$\lim_{i\to\infty} \frac{\epsilon_{i+1}}{\epsilon_i^\beta} = c$$

exists. c is called the *rate constant*. We talk of linear convergence, super-linear convergence, quadratic convergence etc. depending on the value of β and c. For example linear convergence implies that the error of the $(i+1)^{th}$ iterate is linearly related to error of ith iterate, thus the error is decreasing at linear rate. That is, for $\beta = 1$ if $0 < c < 1$, the sequence is said to converge *linearly*. For any $1 < \beta < 2$, and c is finite, the convergence is said to be *superlinear*. If c is finite for $\beta = 2$, the iteration converges *quadratically*.

For example, consider the Newton-Raphson algorithm for solving $f(x) = 0$. The iteration is given by:

$$x_{(n+1)} = x_{(n)} - \frac{f(x_{(n)})}{f'(x_{(n)})} \quad n = 0, 1, \ldots$$

If X is a true root of $f(x) = 0$ define

$$\epsilon_{(n)} = X - x_{(n)} \qquad \text{and} \qquad \epsilon_{(n+1)} = X - x_{(n+1)}$$

and

$$\epsilon_{(n+1)} = \epsilon_{(n)} + \frac{f(x_{(n)})}{f'(x_{(n)})}$$

.

Expanding $f(X)$ around $x_{(n)}$, we have

$$0 \equiv f(X) = f(x_{(n)}) + \epsilon_{(n)} f'(x_{(n)}) + 1/2\epsilon_{(n)}^2 f''(x^*)$$

where $x^* \epsilon\, [x_{(n)}, x_{(n)} + \epsilon_{(n)}]$, which implies that

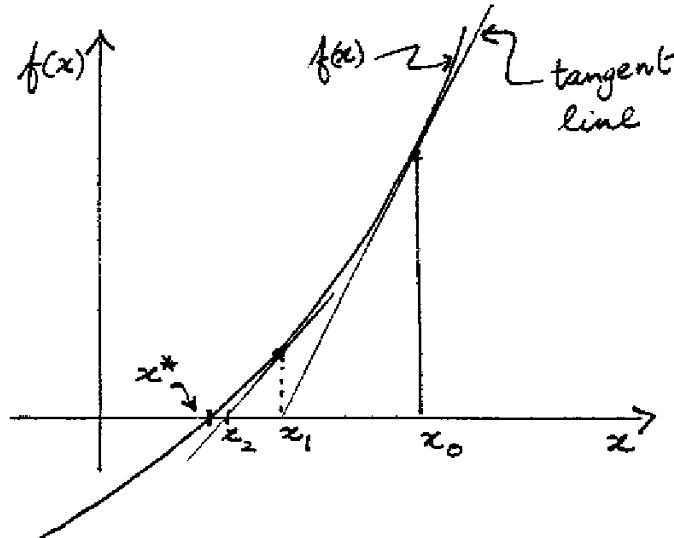
$$\epsilon_{(n+1)} = \epsilon_{(n)}^2 \frac{f''(x^*)}{2f'(x_{(n)})}$$

This shows that Newton-Raphson iteration is a 2nd order method or that it is quadratically convergent.

Numerical Root Finding

Consider the problem of solving $f(x) = 0$ where $f(x)$ is a (univariate) nonlinear function in x. There are several approaches to obtaining iterative algorithms for this problem. We shall look at a few methods briefly.

Newton's Method



From the diagram, $f'(x_0) = \frac{f(x_0)}{x_0 - x_1}$ which implies that $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. This is generalized to obtain the iterative formula

$$x_{(i+1)} = x_{(i)} - \frac{f(x_{(i)})}{f'(x_{(i)})}$$

where $f'(x_i)$ is the first derivative of $f(x)$ evaluated at x_i and x_0 is an initial solution. To begin the iteration a suitable value for x_0 is needed. As an illustration of the use of this algorithm it is applied to find a solution of the cubic equation

$$f(x) = x^3 - 18.1x - 34.8 = 0$$

To estimate a value for x_0, first evaluate $f(x)$ for a few values of x:

x	f(x)
4	–43.2
5	–.3
6	72.6
7	101.5

This type of a search is known as a grid search, the terminology coming from the use of the technique in the multidimensional case. It is clear that a zero for $f(x)$ exists between 5 and 6. Hence, a good starting value is $x_0 = 5.0$. To use the above algorithm the first derivative of $f(x)$ is also needed:

$$f'(x) = 3x^2 - 18.1$$

Thus we obtain

$$x_{i+1} = x_i - \left(x_i^3 - 18.1x_i - 34.8\right) / \left(3x_i^2 - 18.1\right) .$$

Hence

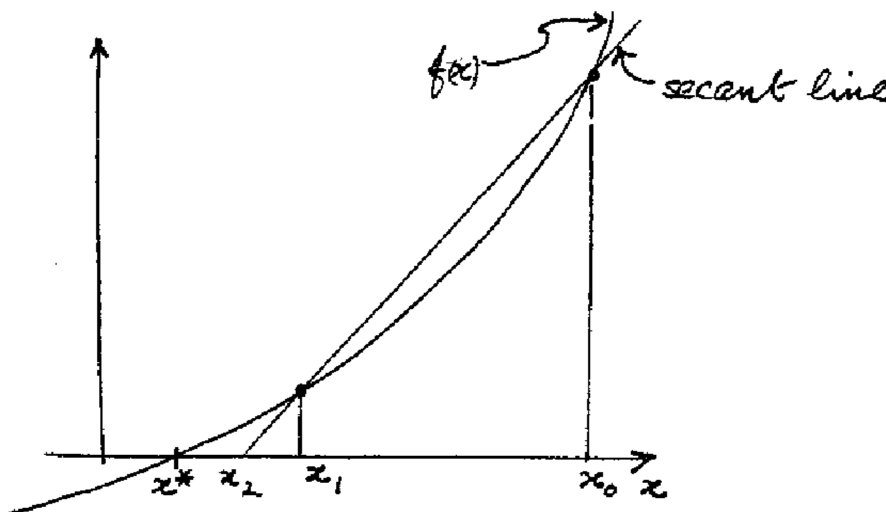
$$\begin{aligned} x_1 &= x_0 - (x_0^3 - 18.1x_0 - 34.8)/(3x_0^2 - 18.1) \\ &= 5.005272408 . \end{aligned}$$

Similarly,

$$\begin{aligned} x_2 &= x_1 - (x_1^3 - 18.1x_1 - 34.8)/(3x_1^2 - 18.1) \\ &= 5.005265097 . \end{aligned}$$

It is known that a true root of the above equation is 5.005265097. Newton-Raphson iteration is quadratically convergent when $x_{(0)}$ is selected close to an actual root.

Secant Method



If the approximation $f'(x_{(i)} \approx \frac{f(x_{(i)})-f(x_{(i-1)}}{x_{(i)}-x_{(i-1)}}$ is plugged in Newton's formula we obtain the iteration:

$$x_{(i+1)} = \frac{x_{(i-1)}f(x_{(i)}) - x_{(i)}f(x_{(i-1)})}{f(x_{(i)}) - f(x_{(i-1)})}$$

This is thus a variation of Newton's method that doe not require the availability of the first derivative of $f(x)$. The iterative formula can also be directly obtained by noting that the triangles formed by x_2, $x_1\, f(x_1)$ and x_2, $x_0\, f(x_0)$ in the above diagram are similar. An equivalent iterative formula is:

$$x_{(i+1)} = x_{(i)} - \frac{f(x_{(i)})(x_{(i)} - x_{(i-1)})}{f(x_{(i)}) - f(x_{(i-1)})}$$

Usually, the Secant method takes a few more iterations than Newton-Raphson.

Fixed Point Iteration
Re-expressing $f(x) = 0$ in the form $x = g(x)$ one can obtain an iterative formula $x_{(i+1)} = g(x_{(i)})$ for finding roots of nonlinear equations. . One way to do this is to set

$$x_{(i+1)} = x_{(i)} + f(x_{(i)})$$

Thisted examines solving

$$f(x) = \frac{-3062(1-\xi)e^{-x}}{\xi + (1-\xi)e^{-x}} - 1013 + \frac{1628}{x},$$

where $\xi = .61489$, by expressing $g(x)$ in several different forms:

$$\begin{aligned} g(x) &= x + f(x) \\ g_4(x) &= x + \frac{f(x)}{1000} \\ g_2(x) &= \frac{1628[\xi + (1-\xi)e^{-x}]}{3062(1-\xi)e^{-x} + 1013[\xi + (1-\xi)e^{-x}]} \end{aligned}$$

The iteration based on $g(x)$ fails to converge. $g_4(x)$ is a scaled version of $g(x)$.

Table 1: Convergence of two methods for fixed-point iteration. The first pair of columns gives the results using $g_2(x)$. The second pair gives the results for $g_4(x)$. Both of these versions are extremely stable, although convergence is not rapid when x_0 is far from the solution s.

	$g_2(x)$		$g_4(x)$	
i	x_i	$f(x_i)$	x_i	$f(x_i)$
0	2.00000000	-438.259603329	2.00000000	-438.259603329
1	1.30004992	-207.166772208	1.56174040	-326.146017453
2	1.11550677	-75.067503354	1.23559438	-166.986673341
3	1.06093613	-24.037994642	1.06860771	-31.623205997
4	1.04457280	-7.374992947	1.03698450	0.583456921
5	1.03965315	-2.232441655	1.03756796	-0.033788786
6	1.03817307	-0.673001091	1.03753417	0.001932360
7	1.03772771	-0.202634180	1.03753610	-0.000110591
8	1.03759369	-0.060988412	1.03753599	-0.000006329
9	1.03755336	-0.018354099	1.03753600	-0.000000362
10	1.03754122	-0.005523370	1.03753600	0.000000021
11	1.03753757	-0.001662152	1.03753600	-0.000000001
12	1.03753647	-0.000500191	1.03753600	0.000000000
13	1.03753614	-0.000150522	1.03753600	-0.000000000
14	1.03753604	-0.000045297	1.03753600	0.000000000

The iterative process

$$x_{(i+1)} = g(x_{(i)}) \quad i = 0, 1, 2, \ldots$$

is said to have a fixed point if for every $x \in [a, b]$, $g(x) \in [a, b]$ for a specified interval $[a, b]$. Any $x_{(0)} \in [a, b]$ will produce a sequence of iterates $x_{(1)}, x_{(2)}, \ldots$ that lie in the specified interval; however, there is no guarantee that the sequence converges to a root of $f(x) = x - g(x) = 0$. For this to occur, there must be at most one root of $x - g(x) = 0$ in $[a, b]$. We can check whether the function $g(x)$ and the interval $[a, b]$ satisfy one of two conditions to verify whether $x - g(x)$ has a unique root in $[a, b]$.

Condition 1: $g(x)$ is a continuous function defined in $[a, b]$ and $|g'(x)| \leq L$ with $L < 1$ for all $x \in [a, b]$.
Condition 2: (Lipshitz condition) $|g(x_1) - g(x_2)| \leq L\ |x_1 - x_2|$ with $0 < L < 1$ for $x_1, x_2 \in [a, b]$.

If one of the above is satisfied and if the iterative process has a fixed point in $[a, b]$, then it converges to the unique root of $f(x) = 0$ in $[a, b]$ for any $x_{(0)}$ in $[a, b]$.

Example: Find the root of $f(x) = x^2 - 4x + 2.3 = 0$ contained in the interval $[0, 1]$ by an iterative method. Consider the possible iterative processes that may be obtained by writing $f(x) = 0$ in the forms:
(i) $x = (x^2 + 2.3)/4$, (ii) $x = (4x - 2.3)^{1/2}$, and (iii) $x = -2.3/(x - 4)$.

(i) This iterative process has a fixed point and $g'(x) = \frac{x}{2}$ implies $|g'(x)| \leq 1/2$ in $[0, 1]$. Thus condition 1 is satisfied and the iteration is convergent. Applying the iteration with $x_{(0)} = 0.6$ we get the sequence:

$$x_{(1)} = 0.665$$
$$x_{(2)} = 0.68556$$
$$x_{(3)} = 0.69250$$
$$x_{(4)} = 0.69489$$
$$x_{(5)} = 0.69572.$$

`Note:`To verify Condition 2, note that by the Mean Value Theorem

$$|g(x_1) - g(x_2)| = 1/4|x_1^2 - x_2^2| = 2|\xi|/4|x_1 - x_2|,$$

where $\xi \in (x_1,\ x_2)$. Thus $|g(x_1) - g(x_2)| \leq 1/2|x_1 - x_2|$ for $x_1,\ x_2 \in [0,\ 1]$.

(ii) Here the iteration does not have a fixed point since at $x = 0$ $g(x) = (-2.3)^{1/2}$ which is not contained in $[0, 1]$ and $|g'(x)| > 1$ for some $x \geq .575$.

(iii) This also has a fixed point and $g'(x) = +2.3/(x - 4)^2 \leq \frac{2.3}{9}$ in $[0, 1]$. Condition 1 is satisfied. Applying this iteration with $x_{(0)} = 0.6$ we get the sequence:

$$x_{(0)} = 0.6$$
$$x_{(1)} = 0.67647$$
$$x_{(2)} = .692035$$
$$x_{(3)} = 0.69529$$
$$x_{(4)} = 0.69598$$
$$x_{(5)} = 0.69612$$

Aitken Acceleration

First consider $n+1$ iterates of a linearly convergent iteration:

$$x_{(0)}, x_{(1)}, \ldots, x_{(n)}$$

Suppose the method converges to the unique solution X at rate α. That is,

$$\lim_{i\to\infty} \frac{|\epsilon_{(i+1)}|}{|\epsilon_{(i)}|} = \alpha, \quad 0 < \alpha < 1$$

by our previous definition, for a linearly convergent sequence, where $\epsilon_{(i)} = X - x_{(i)}$. Thus, as $i \to \infty$, we should have

$$\frac{|\epsilon_{(i+1)}|}{|\epsilon_{(i)}|} \simeq \frac{|\epsilon_{(i+2)}|}{|\epsilon_{(i+1)}|}$$

resulting in the relationship

$$\frac{X - x_{(i+1)}}{X - x_{(i)}} \simeq \frac{X - x_{(i+2)}}{X - x_{(i+1)}}$$

From this we have

$$X \simeq \frac{x_{(i)}x_{(i+2)} - x^2_{(i+1)}}{x_{(i+2)} - 2x_{(i+1)} + x_{(i)}} = x_{(i)} - \frac{(x_{(i+1)} - x_{(i)})^2}{x_{(i+2)} - 2x_{(i+1)} + x_{(i)}}$$

Defining the first and second order forward differences $\Delta x_{(i)} = x_{(i+1)} - x_{(i)}$ and

$$\Delta^2 x_{(i)} = \Delta(\Delta x_{(i)}) = \Delta(x_{(i+1)} - x_{(i)}) = x_{(i+2)} - 2x_{(i+1)} + x_{(i)}$$

we have

$$X \simeq x_{(i)} - \frac{(\Delta x_{(i)})^2}{\Delta^2 x_{(i)}}.$$

This is called the Aitken's Δ^2 method. If a sequence $x^*_{(0)}, x^*_{(1)}, \ldots$ is defined such that

$$x^*_{(i)} = x_{(i)} - \frac{(\Delta x_{(i)})^2}{\Delta^2 x_{(i)}} \quad \text{for } i = 0, 1, \ldots$$

then it can be shown that the new sequence will converge faster to X than the convergent sequence $x_{(0)}, x_{(1)}, \ldots$ in the sense that

$$\lim_{i\to\infty} \frac{x^*_{(i)} - X}{x_{(i)} - X} = 0.$$

We can observe that the Aitken's method attempts to eliminate the first order term in the error by the correction $(\Delta x_{(i)})^2/\Delta^2 x_{(i)}$. However, since the assumption above that the error rate α is constant is only an approximation, one would expect that the convergence to be not exactly quadratic although considerably better than linear.

Aitken's Δ^2 method can be applied to accelerate any linearly convergent sequence, such as iterates resulting from the fixed-point method. Suppose $x_{(0)}, x_{(1)}, \ldots, x_{(n+1)}$ are the successive iterates from a convergent iteration $x_{(i+1)} = g(x_{(i)})$. The sequence $x^*_{(0)}, x^*_{(1)}, \ldots$ defined by

$$x^*_{(i)} = \frac{x_{(i)}x_{(i+2)} - x^2_{(i+1)}}{x_{(i+2)} - 2x_{(i+1)} + x_{(i)}}, \qquad i = 0, 1, \ldots,$$

are the Aitken accelerated iterates.

`Steffensen's method` is a slight modification of this straight Aitken acceleration of the original sequence. Instead of of using all of the original iterates to obtain the new sequence, say, a new $x^*_{(0)}$ is computed using

$$x^*_{(0)} = \frac{x_{(0)}x_{(2)} - x^2_{(1)}}{x_{(2)} - 2x_{(1)} + x_{(0)}}$$

and it is then is used to generate 2 new iterates applying the iterating function:

$$x^*_{(1)} = g(x^*_{(0)}), \qquad x^*_{(2)} = g(x^*_{(1)})$$

successively. From these three iterates, a new $x^*_{(0)}$ is computed, from which two iterates $x^*_{(1)}$ and $x^*_{(2)}$ are generated, and so on. This results in an algorithm that is closer to achieving quadratic convergence than the direct application of Aitken above.

`Steffensen's Algorithm:`

Select $x = g(x)$, starting value x_0 an ϵ, and set $i = 0$.
While $i < N$ do

1. Set $x_1 = g(x_0)$ and $x_2 = g(x_1)$
2. Set $x^* = \frac{x_0 x_2 - x_1^2}{x_2 - 2x_1 + x_0}$
3. If $|x_0 - x^*| < \epsilon$ then
   Return x^*
   Stop
4. Else,
   Set $x_0 = x^*$
   Set $i = i + 1$

`Example:`
For solving $x^3 - 3x^2 + 2x - 0.375 = 0$ using fixed point iteration write

$$x = \left((x^3 + 2x - .375)/3\right)^{1/2}.$$

It can be shown that this has a fixed point and results in a convergent iteration. We compare below the output from using the 5 iteration schemes; fixed point, Aitken, Newton-Raphson, Steffensen, Secant.

First Order	Aitken	Newton-Raphson	Steffensen	Secant
0.7000000	0.7000000	0.7000000	0.7000000	0.7000000
0.6752777	0.5268429	0.5602740	0.5268429	0.5590481
0.6540851	0.5246187	0.5119343	0.5019479	0.5246936
0.6358736	0.5221342	0.5007367	0.5000138	0.5055804
0.6201756	0.5196318	0.5000032	0.5000000	0.5006874
0.6065976	0.5172448		0.5000000	0.5000221
0.5948106	0.5150407			0.5000001
0.5845412	0.5130476			
0.5755615	0.5112714			
0.5676824	0.5097044			
0.5607459	0.5083327			
0.5546202	0.5071387			
0.5491944	0.5061042			
0.5443754	0.5052110			
0.5400844	0.5044423			
0.5362545	0.5037822			
0.5328287	0.5032166			
0.5297581	0.5027328			
0.5270007	0.5023197			
0.5245205	0.5019675			
0.5222860	0.5016675			
0.5202700	0.5014122			
0.5184487	0.5011953			
0.5168013	0.5010112			
0.5153095	0.5008549			
0.5139573	0.5007225			
0.5127304	0.5006102			
0.5116163	0.5005152			
0.5106039	0.5004349			
0.5096831	0.5003669			
0.5088451	0.5003095			
0.5080820	0.5002609			
0.5073868	0.5002200			
0.5067530	0.5001854			
0.5061751	0.5001562			
0.5056477	0.5001316			
0.5051664	0.5001108			
0.5047269	0.5000933			
0.5043255	0.5000785			
0.5039588	0.5000661			
0.5036236	0.5000556			
0.5033172	0.5000468			
0.5030371				
0.5027809				
0.5025465				
0.5023321				
0.5021359				
0.5019564				
0.5017921				
0.5016416				
0.5015039				
0.5013778				
0.5012624				
0.5011566				
0.5010598				
0.5009711				
0.5008898				
0.5008154				
0.5007472				
0.5006848				
0.5006276				
0.5005751				
0.5005271				
0.5004831				
0.5004427				
0.5004058				
0.5003719				
0.5003409				
0.5003124				
0.5002864				
0.5002625				
0.5002406				
0.5002205				
0.5002021				
0.5001853				
0.5001698				
0.5001556				
0.5001427				
0.5001308				
0.5001199				
0.5001099				

Bracketing Methods

Methods such as Newton-Raphson may be categorized as *analytical* iterative methods for finding roots of $f(x) = 0$ since they depend on the behaviour of the first derivative $f'(x)$ even if it may not be explicitly calculated. The class of methods where only the value of the function at different points are used are known as *search* methods. The best known iterative search methods are called *bracketing* methods since they attempt to reduce the width of the interval where the root is located while keeping the successive iterates bracketed within that interval. These methods are *safe*, in the sense that they avoid the possibility that an iterate may be thrown far away from the root, such as that may happen with the Newton's or Secant methods when a *flat* part of the function is encountered. Thus these methods are guaranteed to converge to the root.

The simplest bracketing method is the *bisection algorithm* where the interval of uncertainty is halved at each iteration, and from the two resulting intervals, the one that *straddles* the root is chosen as the next interval. Consider the function $f(x)$ to be continuous in the interval $[x_0, x_1]$ where $x_0 < x_1$ and that it is known that a root of $f(x) = 0$ exists in this interval.

`Bisection Algorithm:`

Select starting values $x_0 < x_1$ such that $f(x_0) * f(x_1) < 0$, ϵ_x, and ϵ_f,

Repeat

1. set $x_2 = (x_0 + x_1)/2$
2. if $f(x_2) * f(x_0) < 0$

   set $x_1 = x_2$
3. else

   set $x_0 = x_2$

until $|x_1 - x_0| < \epsilon_x$ or $|f(x_2)| < \epsilon_f$

Return x_2

The *Golden-section search* is a bracketing algorithm that reduces the length of the interval of uncertainty by a fixed factor α in each iteration (instead of by .5 as in bisection method), where α satisfies

$$\frac{1}{\alpha_1} = \frac{\alpha_1}{1 - \alpha_1}.$$

This gives $\alpha = (\sqrt{5} - 1)/2 = .618033989$, which is known as the *golden ratio.* The Fibonacci search, which is not discussed here, is the optimal method of this type of search methods.

The main problem with bisection is that convergence is extremely slow. A search algorithm that retains the *safety* feature of the bisection algorithm but achieves a superlinear convergence rate is the *Illinois method*. As in bisection, this algorithm begins with two points $[x_0, x_1]$ that straddle the root and uses the Secant algorithm to obtain the next iterate x_2.

Then instead of automatically discarding x_0 as in Secant, the next two iterates are either $[x_0, x_2]$ or $[x_2, x_1]$ depending on which interval straddles the root. This is determined by examining the sign of $f(x_2)$ and comparing with the sign of $f(x_1)$. Thus one of the end-points of the previous interval is retained in forming the next interval. If the same end-point is retained in the next iteration also that iteration will not be a true Secant iteration (since the oldest iterate will always be discarded in Secant). If this occurs, the value of $f()$ at that end-point is halved to force a true Secant step to be performed. The halving of an end-point is continued until a true Secant iteration takes place.

`Illinois Algorithm:`

Select starting values $x_0 < x_1$ such that $f(x_0) * f(x_1) < 0$, ϵ_x, and ϵ_f,

> Set $f_0 = f(x_0)$
> Set $f_1 = f(x_1)$
> Repeat
>
> 1. set $x_2 = (x_0 f_1 - x_1 f_0)/(f_1 - f_0)$
> 2. if $f(x_2) * f_0 < 0$ then
>
>   set $f_{prev} = f_1$
>   set $x_1 = x_2$
>   set $f_1 = f(x_1)$
>   if $f_{prev} * f_1 > 0$ set $f_0 = f_0/2$
>
> 3. else
>
>   set $f_{prev} = f_0$
>   set $x_0 = x_2$
>   set $f_0 = f(x_0)$
>   if $f_0 * f_{prev} > 0$ set $f_1 = f_1/2$
>
> until $|x_1 - x_0| < \epsilon_x$ or $f(x_2) < \epsilon_f$
>
> Return x_2

The Illinois could be thought of as an improvement of the Secant algorithm instead of an improvement of the Bisection method, by changing the Secant algorithm to a *safe* method. The secant method has superlinear convergence rate with $|\epsilon_{n+1}| = O(|\epsilon_n|^{1.618}$. Illinois is nearly as good with a rate of 1.442, far superior to other bracketing methods. Note that Illinois requires that the two starting values straddle the root as in Bisection, but that Secant does not.

Iterative Solution of Linear Systems

We will introduce *direct methods* for solution of linear systems of equations such as those based on Gauss-Jordan row operations or on the Cholesky factorization during the discussion of matrix methods later. Compared to algorithms based on direct methods which produce exact solutions with a fixed number of computations, the methods introduced here are iterative in nature — they produce a sequence of approximations to the solution that converge to the exact solution. These methods are important since variations of them are used in many computer-intensive nonlinear statistical methods. Consider the solution of the linear system of equations:

$$A\mathbf{x} = \mathbf{b}$$

where A is an $n \times n$ matrix of coefficients of full rank, $\mathbf{b}$ is a given $n \times 1$ vector of constants. By rewriting the first row, we have

$$x_1 = \{b_1 - (a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n)\}/a_{11}$$

provided $a_{11} \neq 0$. Similarly, we can write

$$x_i = (b_i - \sum_{j \neq i} a_{ij}x_j)/a_{ii}$$

for $i = 1, \ldots, n$, provided $a_{ii} \neq 0$ for all i. This relation suggests an iterative algorithm of the fixed point type for obtaining an updated value $x_i^{(k)}$ given $x_i^{(k-1)}$ for $i = 1, \ldots, n$. In particular, starting from an initial guess $\mathbf{x}^{(0)}$, we can compute the sequence $\mathbf{x}^{(k)}$, $k = 1, 2, \ldots$, where

$$x_i^{(k)} = (b_i - \sum_{j \neq i} a_{ij}x_j^{(k-1)})/a_{ii},\ \ i = 1, \ldots, n\ . \tag{1}$$

This procedure is called the *Jacobi iteration*. This is also called a method of *simultaneous displacement* since no element of $\mathbf{x}^{(k)}$ is used until *every* element of $\mathbf{x}^{(k)}$ has been computed, and then $\mathbf{x}^{(k)}$ replaces $\mathbf{x}^{(k-1)}$ entirely for the next cycle.

Let us consider A to be a real symmetric positive definite matrix and let $D = \text{diag}(A)$ and $N = A - D$. Then the above iteration (1) can be expressed in vector form as

$$\mathbf{x}^{(k)} = D^{-1}(\mathbf{b} - N\mathbf{x}^{(k-1)})\ . \tag{2}$$

Rewriting (2) setting $H = -D^{-1}N$,

$$\mathbf{x}^{(k)} = H\mathbf{x}^{(k-1)} + D^{-1}\mathbf{b}\ . \tag{3}$$

The error $\mathbf{e}^{(k)}$ at the kth step of the Jacobi iteration is easily derived to be:

$$\begin{aligned} \mathbf{e}^{(k)} &= \mathbf{x}^{(k)} - \mathbf{x} \\ &= H\mathbf{e}^{(k-1)}\ . \end{aligned} \tag{4}$$

Notice that H is a matrix that is fixed throughout the iteration which is a characteristic of *stationary* methods. This also gives

$$\mathbf{e}^{(k)} = H^k\mathbf{e}^{(0)} \tag{5}$$

which shows that for this iteration to converge it is clearly necessary that the largest eigenvalues of H, called the *spectral radius*, to be less than unity. However, it can be shown that even in a convergent iteration, individual elements of $\mathbf{e}^{(k)}$ may exceed the corresponding elements of $\mathbf{e}^{(0)}$ in absolute value. These elements will not decrease exponentially as k increases unless the largest eigenvalue is larger than the others. That is, unless absolute values of off-diagonals of A are negligible compared to the diagonals, the convergence is slow.

In equation (1), if the elements of $\mathbf{x}^{(k)}$ replaces those of $\mathbf{x}^{(k-1)}$ as soon as they have been computed, this becomes a method of *successive displacement*. The resulting algorithm is called the *Gauss-Seidel iteration* and is given by

$$x_i^{(k)} = \{b_i - (\sum_{j<i} a_{ij}x_j^{(k)} + \sum_{j>i} a_{ij}x_j^{(k-1)})\}/a_{ii} \tag{6}$$

for $k = 1, 2, \ldots$. Again by setting $D = \text{diag}(A)$ and $N = A - D = L + U$, where L and U are strictly lower triangular and strictly upper triangular, respectively, the iteration can be expressed in the form

$$\begin{aligned} \mathbf{x}^{(k)} &= (D + L)^{-1}(\mathbf{b} - U\mathbf{x}^{(k-1)}) \\ &= H\mathbf{x}^{(k-1)} + B^{-1}\mathbf{b} \end{aligned} \tag{7}$$

where $B = D + L$ (i.e., B is the lower triangle of A) and $H = -B^{-1}U$. The error $\mathbf{e}^{(k)}$ at the kth step of the Gauss-Seidel iteration is given by

$$\mathbf{e}^{(k)} = H\mathbf{e}^{(k-1)} = H^k\mathbf{e}^{(0)} ,$$

as before. In particular, convergence of Gauss-Seidel will occur for any starting value $\mathbf{x}^{(0)}$, if and only if $H = B^{-1}U$ has all eigenvalues less than unity. Convergence is rapid when diagonals of A dominate the off-diagonals, but is quite slow if the largest eigenvalue of H is close to one. When this is the case Gauss-Seidel can be accelerated using the method of *successive over-relaxation* (SOR). Suppose that the kth Gauss-Seidel iterate is:

$$\mathbf{x}^{(k)} = B^{-1}\mathbf{b} + H\mathbf{x}^{(k-1)} .$$

Then the kth SOR iterate is given by

$$\mathbf{x}_w^{(k)} = w\mathbf{x}^{(k)} + (1 - w)\mathbf{x}^{(k-1)} \tag{8}$$

where $w \neq 0$. When $w = 1$, the SOR iteration reduces to the Gauss-Seidel. Typically, in SOR, $w > 1$, which means we are "overcorrecting" or taking a bigger step away from the kth estimate than the Gauss-Seidel iteration would (but in the same direction) because

$$\mathbf{x}_w^{(k)} = \mathbf{x}^{(k-1)} + w(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}) .$$

In general, the *relaxation factor* w must be estimated. Various choices are discussed in Chapter 6 of Young(1971). As a practical consideration of implementing these algorithms, the original system $A\mathbf{x} = \mathbf{b}$ is first replaced by

$$(D^{-1/2}AD^{-1/2})(D^{1/2}\mathbf{x}) = D^{-1/2}\mathbf{b},$$

where $D^{1/2} = diag(\sqrt{(a_{ii})})$. Thus all divisions involved are performed beforehand. Let us assume that this new system is denoted by the same notation i.e.,$A\mathbf{x} = \mathbf{b}$. $D = diag(A)$ is now an identity matrix and L anf U matrices strictly lower triangular and strictly upper triangular, respectively. The resulting Jacobi iteration is then given by

$$\mathbf{x}^{(k)} = (L + U)\mathbf{x}^{(k-1)} + \mathbf{b}$$

and the Gauss-Seidel iteration by

$$\mathbf{x}^{(k)} = L\mathbf{x}^{(k)} + U\mathbf{x}^{(k-1)} + \mathbf{b}.$$

To exploit the symmetry that exists in certain statistical problems, Gauss-Seidel method has been applied using a "double sweep", i.e., after an iteration updating $x^{(k)}$ through $x_1^{(k)}$ are updated in that order. This double sweep ensures exponential decrease in the size of $\mathbf{e}$.

Example

Formulate the Jacobi and point Gauss-Seidel methods for the system of linear equations

$$\begin{aligned} 20x_1 + 2x_2 - x_3 &= 25 \\ 2x_1 + 13x_2 - 2x_3 &= 30 \\ x_1 + x_2 + x_3 &= 2 \end{aligned}$$

which has the solution $x_1 = 1, x_2 = 2, x_3 = -1$.

Using the definition of the Jacobi method, Eq. (1), we obtain

$$\begin{aligned} x_{1(1)} &= \frac{1}{20}\Big(- 2x_{2(0)} + x_{3(0)} + 25\Big) \\ x_{2(1)} &= \frac{1}{13}\Big(- 2x_{1(0)} + x_{3(0)} + 30\Big) \\ x_{3(1)} &= \Big(- x_{1(0)} - x_{2(0)} + 2\Big) \end{aligned}$$

At the $(1 + n)$th iteration this becomes

$$\begin{aligned} x_{1(n+1)} &= \frac{1}{20}\Big(- 2x_{2(n)} + x_{3(n)} + 25\Big) \\ x_{2(n+1)} &= \frac{1}{13}\Big(- 2x_{1(n)} + 2x_{3(n)} + 30\Big) \\ x_{3(n+1)} &= \Big(- x_{1(n)} - x_{2(n)} + 2\Big) \end{aligned}$$

where $x_{1(0)}, x_{2(0)}$, and $x_{3(0)}$ are initial guesses.

In contrast, the Gauss-Seidel method, Eq. (6), may be written for the first iteration

$$\begin{aligned}
x_{1(1)} &= \frac{1}{20}\Big(- 2x_{2(0)} + x_{3(0)} + 25\Big) \\
x_{2(1)} &= \frac{1}{13}\Big(- 2x_{1(1)} + 2x_{3(0)} + 30\Big) \\
x_{3(1)} &= \Big(- x_{1(1)} - x_{2(1)} + 2\Big)
\end{aligned}$$

and at the $(1 + n)$th iteration

$$\begin{aligned}
x_{1(n+1)} &= \frac{1}{20}\Big(- 2x_{2(n)} + x_{3(n)} + 25\Big) \\
x_{2(n+1)} &= \frac{1}{13}\Big(- 2x_{1(n+1)} + 2x_{3(n)} + 30\Big) \\
x_{3(n+1)} &= \Big(- x_{1(n+1)} - x_{2(n+1)} + 2\Big)
\end{aligned}$$

where $x_{2(0)}$ and $x_{3(0)}$ are initial guesses.

Starting with the initial guesses $x_{1(0)} = x_{2(0)} = x_{3(0)} = 0$, the two algorithms produce the results given in Tables 1 and 2. The faster convergence of the Gauss-Seidel method is evident from the tables. This is often the case although examples can be constructed where the Jacobi is faster.

Table 1 Jacobi results

Iteration $(n+1) =$	x_1	x_2	x_3
1	0	0	0
2	1.25	2.307 692	2
3	1.119 231	2.423 077	-1.557 692
4	0.929 807 7	1.895 858	-1.542 308
5	0.983 298 8	1.927 367	-0.825 665 7
6	1.015 98	2.029 39	-0.910 665 7
7	1.001 528	2.011 285	-1.045 37
8	0.996 603	1.992 785	-1.012 813
9	1.000 081	1.998 551	-0.989 387 9
10	1.000 675	2.001 62	-0.998 632 2
11	0.999 906 4	2.000 106	-1.002 296
12	0.999 874 6	1.999 661	-1.000 013
13	1.000 033	2.000 017	-0.999 535 8
14	1.000 021	2.000 066	-1.000 051

Table 2 Gauss-Seidel results

Iteration $(n+1) =$	x_1	x_2	x_3
1	1.25	2.115 385	-1.365 385
2	0.970 192 3	1.948 373	-0.918 565 1
3	1.009 234	2.011 108	-1.020 342
4	0.997 872 1	1.997 198	-0.995 069 9
5	1.000 527	2.000 677	-1.001 204
6	0.999 872 1	1.999 834	-0.999 706 5
7	1.000 031	2.000 04	-1.000 072
8	0.999 992 4	1.999 99	-0.999 982 5
9	1.000 002	2.000 002	-1.000 004
10	0.999 999 6	1.999 999	-0.999 998 9
11	1	2	-1

Convergence Criteria

As we have seen, many numerical algorithms are iterative. When iterative procedures are implemented in a program it is required to check whether the iteration has converged to a solution or is near to a solution to a desired degree of accuracy. Such a test is called a *stopping rule* and is derived using a *convergence criterion*. In the root finding problem, we may use the convergence criterion that the change in successive iterates $x_{(n)}$ and $x_{(n+1)}$ is sufficiently small or, alternatively, whether $f(x_{(n)})$ is close enough to zero. In the univariate root finding problem, the stopping rules thus may be one or more of the tests

$$\begin{aligned} |x_{(n+1)} - x_{(n)}| &\leq \epsilon_x \\ |x_{(n+1)} - x_{(n)}| &\leq \delta * |x_{(n)} - x_{(n-1)}| \end{aligned}$$

$$\begin{aligned} |f(x_{(n))}| &\leq \epsilon_f \\ |f(x_{(n+1)}) &- f(x_{(n)}| \leq \delta_f \end{aligned}$$

A variety of reasons exist for these criteria to fail. Some of these are discussed with examples in Rice(1983) where it is recommended that composite tests be used. In the case that x is a vector then some norm of the difference of iterates, denoted by $\|x_{(n+1)} - x_{(n)}\|$ is used in the stopping rule. Usually the norm used is the square root of the Euclidean distance between the two vectors, called the *2-norm*.

Algorithms for the Computation of Sample Variance

Standard algorithms

The sample variance of a given set of n data values $x_1, x_2, \ldots, x_n$ is denoted by s^2 and is defined by the formula

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

where

$$\bar{x} = \left(\sum_{i=1}^{n} x_i\right)/n\,.$$

The numerator of the formula for s^2, $\sum_{i=1}^{n}(x_i - \bar{x})^2$ which will be denoted by S, is the sum of squared deviations of the n data values from their sample mean $\bar{x}$. The computation of S using the above formula requires two passes through the data since it requires going through the data twice: once to compute $\bar{x}$ and then again to compute S. Hence this method of computation is called a `two-pass algorithm`. If the sample size is large and data is not stored in memory a two-pass algorithm may turn out to be not very efficient.

The following is generic code for implementing the two-pass algorithm for computing the sample variance, s^2. In this and other programs given in this note, the data vector $x_1, x_2, \ldots, x_n$ is assumed to be available in a standard data structure, such as a properly dimensioned data array `x`.

```
xsum=x(1)
for i = 2,...,n
{
   xsum=xsum+x(i)
}
xmean=xsum/n
xvar=(x(1)-xmean)^2
for i = 2,...,n
{
  xdev=x(i)-xmean
  xvar=xvar+xdev*xdev
}
xvar=xvar/(n-1)
```

The `one-pass machine` (or textbook) `formula` for computing s^2 is obtained by expressing S in the form

$$S = \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2/n\,.$$

It is easy to show that $\sum_{i=1}^{n}(x_i - \bar{x})^2$ and the above formula are mathematically equivalent. The formula requires that the quantities $\sum_{i=1}^{n} x_i^2$ and $\sum_{i=1}^{n} x_i$ be computed and this can be achieved, as easily seen, by a single pass through the data. However, the quantities $\sum x_i^2$ and $(\sum x_i)^2/n$ may be quite large in practice and will generally be computed with some `round-off error`. In this case, as seen by observing the above expression, the value

of S is computed by the subtraction of two (positive-valued) quantities which may have been computed inaccurately. Thus, the possibility exists for S to be computed incorrectly due to the effect of `cancellation` of leading digits of the two numbers being subtracted. (Cancellation is a topic to be discussed in detail elsewhere in the notes). The following `generic` code may be used to program the one-pass machine formula for computing s^2 (and $\bar{x}$):

```
xsum=x(1)
xssq=x(1)^2
for i = 2,...,n
{
   xsum=xsum+x(i)
   xssq=xssq+x(i)^2
}
xvar=(xssq-(xsum^2)/n)/(n-1)
```

Several alternative one-pass algorithms which are more stable than the above algorithm have been suggested. In the next section one of these algorithms is presented and the `Fortran` and `C` implementation of it is discussed.

Updating Algorithms

During the 60's and the 70's several researchers proposed algorithms to overcome the numerical instability associated with the machine formula for computing s^2. These algorithms involve the computation of the numerator S for subsets of data increasing in size, at each stage adjusting the previously computed value to account for the effect of adding successive data values. These are called `updating algorithms`. Youngs and Cramer in 1971 introduced the following algorithm which has turned out to be the most frequently implemented one-pass algorithm for computing the sample variance.

1. Set $V_1 = 0$ and $T_1 = x_1$.
2. Then for $i = 2, 3, \ldots, n$, compute

$$T_i = T_{i-1} + x_i$$

   and

$$V_i = V_{i-1} + \frac{(ix_i - T_i)^2}{i(i-1)}$$

3. Set $\bar{x} = T_n/n$ and $s^2 = V_n/(n-1)$.

For single precision computations this algorithm is more stable than the one-pass machine algorithm because it avoids the computation of $\sum x_i^2$ and $(\sum x_i)^2$ as intermediate values. Moreover, in this algorithm S is computed as a sum of nonnegative quantities which thus avoids any problems arising due to cancellation of significant digits.

The following is a proof of the Youngs-Cramer algorithm. First, note that in the algorithm, T_i denotes the sum of the first i observations and V_i denotes the sum of squared deviations of the first i observations about their mean, $M_i = T_i/i$. Thus,

$$\begin{aligned} V_i &= \sum_{r=1}^{i}(x_r - M_i)^2 \\ &= \sum_{r=1}^{i-1}(x_r - M_i)^2 + (x_i - M_i)^2 \\ &= \sum_{r=1}^{i-1}\{(x_r - M_{i-1}) + (M_{i-1} - M_i)\}^2 + (x_i - M_i)^2 \\ &= \sum_{r=1}^{i-1}(x_r - M_{i-1})^2 + \sum_{r=1}^{i-1}(M_{i-1} - M_i)^2 \end{aligned}$$

because the cross-product term

$$\sum_{r=1}^{i-1}(x_r - M_{i-1})(M_{i-1} - M_i)$$

reduces to zero. Hence,

$$V_i = V_{i-1} + (i-1)(M_i - M_{i-1})^2 + (x_i - M_i)^2 \,.$$

However, since $T_i = T_{i-1} + x_i$, it follows that $iM_i = (i-1)M_{i-1} + x_i$ and therefore

$$(i-1)(M_i - M_{i-1}) = x_i - M_i$$

. Substituting this in the above expression for V_i, we obtain

$$\begin{aligned} V_i &= V_{i-1} + \frac{(i-1)(x_i - M_i)^2}{(i-1)^2} + (x_i - M_i)^2 \\ &= V_{i-1} + \frac{i(x_i - \frac{T_i}{i})^2}{(i-1)} \,. \end{aligned}$$

Thus,

$$V_i = V_{i-1} + \frac{(ix_i - T_i)^2}{i(i-1)} \,.$$

Possible Fortran or C implementations of the Youngs and Cramer algorithm may be accomplished as follows:

```
   xmean=x(1)
   xvar=0.0
   do 10 i=2,n
    xmean=xmean+x(i)
    zi =x(i)*i-xmean
    xvar =xvar+zi*zi/(i*(i-1))
10 continue
   xmean=xmean/n
   xvar =xvar/(n-1)
```

```
*xmean = x[0];
*xvar = 0.0;
z = 1.0;
if (n < 2)
     return;
for (i = 1; i < n; i++)
  {
    z += 1.0;
    *xmean +=  x[i];
    zx = z *  x[i] - *xmean;
    *xvar += zx * zx / (z *(z - 1.0));
   }
 *xmean /= z;
 *xvar /= (z - 1.0);
```

The above is one of the most stable algorithms available for the computation of sample variance in single precision arithmetic. On the other hand the algorithm based on the single-pass machine formula can be relied upon to give accurate results for s^2 when double precision arithmetic is employed in all intermediate computations, when the coefficient of variation for the data is not very large. (See our discussion on condition number).

The updating algorithm given above can be generalized for the computation of sample covariance and the sample variance-covariance matrix required in multiple regression computations.

Accuracy Comparisons

The following table compares the accuracy of s^2 when computed using the two-pass method, the one-pass textbook formula and the updating algorithm, respectively, using single precision. Data for a sample size of 1024 were randomly generated from the $N(1, \sigma^2)$ distribution for various values of σ^2. The results shown in the table are the number of correct leading digits averaged over 20 samples. The number of correct digits is defined as — $\log_{10}(E)$ where E is the relative error involved in the computation. The relative error of a computed quantity is defined elsewhere in the notes. The 'true' value needed for computing E was obtained by computing s^2 using quadruple precision. All results presented were computed on an IBM 370. Below κ refers to the condition number of data sets generated for each fixed value of σ^2 and is computed as $\sqrt{1 + CV^{-2}}$.

Table 1: Number of Corrected Digits for n = 1024

σ^2	CV	κ	Two-pass	Textbook	Updating
1.0	1.0	1.41	4.0	4.1	4.0
0.1	0.316	3.32	4.2	3.0	4.2
0.01	0.1	10.05	4.4	2.0	4.5
0.001	0.0316	31.64	4.5	1.0	4.1
0.0001	0.01	100.05	4.4	0.0	3.6