POLYTECHNIC UNIVERSITY
Department of Computer and Information Science

Floating-Point Numbers in Digital Computers

K. Ming Leung

Abstract: We explain how floating-point numbers are represented and stored in modern digital computers.

Directory

Last Revision Date: February 3, 2005

Table of Contents

1. Introduction to Floating-Point Number Systems

Most quantities encountered in scientific and engineering computing are continuous and are therefore described by real or complex numbers. The number of real numbers even within the smallest interval is uncountably infinite, and therefore real numbers cannot be represented exactly on a digital computer. Instead they are approximated by what are known as floating-point numbers.

A floating-point number is similar to the way real numbers are expressed in scientific notation in the decimal system. For example the speed of light in vacuum, c, is defined to have the value $c = 2.99792458 \times 10^8$ in meters per second. The part containing 2.99792458 is called the mantissa. The dot is the decimal point. The base (or radix) is 10 and the exponent is 8.

This value can be rewritten in many alternate ways, such as $c = 0.299792458 \times 10^9$, $c = 0.0299792458 \times 10^{10}$, and $c = 2997.92458 \times 10^5$. The decimal point moves or floats as we desire, that is why these numbers are called floating-point numbers. When the decimal point floats one unit to the left, the exponent increases by 1, and when the

decimal point floats one unit to the right, the exponent decreases by 1.

One can make the floating-point representation of any number, except 0, unique by requiring the leading digit in the mantissa to be nonzero and the decimal point to appear always right after the leading digit. Zero is treated separately. The resulting floating-point system is then referred to as normalized.

There are good reasons for normalization:

1. representation of each integer is unique
2. no digits wasted on leading zeros
3. additional advantage in binary system: leading digit is always a 1 and therefore need not be stored

The scientific notation to represent a number can be generalized to bases other than 10. Some useful ones are shown in the following table.

base	name	application
$2 = 2^1$	binary	practically all digital computers
$8 = 2^3$	octal	old Digital computers
10	decimal	HP calculators
$16 = 2^4$	hexadecimal	PCs and workstations

In the above example for the number representing the speed of light, the mantissa has 9 digits and the exponent has 1 digit.

1.1. Specification of a Floating-Point System

In general a floating-point system is characterized by 4 positive integers

base or radix	β
precision	p
smallest exponent allowed	L
largest exponent allowed	U

A floating-point number, x, is represented as

$$x = \pm \left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots + \frac{d_{p-1}}{\beta^{p-1}} \right) \beta^E,$$

where for $i = 0, 1, \ldots, p-1$, d_i is an integer between 0 and $\beta - 1$, and E is an integer between L and U. The plus or minus signs indicate the sign of the floating-point number. Some terminologies:

mantissa	$d_0 d_1 \ldots d_{p-1}$
exponent	E
fractional part	$d_1 d_2 \ldots d_{p-1}$

The following table lists some typical floating-point systems.

System	β	p	L	U
IEEE Single-precision	2	24	-126	127
IEEE Double-precision	2	53	-1022	1023
Cray	2	48	-16383	16384
HP Calculator	10	12	-499	499
IBM Mainframes	16	6	-64	64

PCs and workstations typically use the IEEE floating-point systems.

Clearly a floating-point system can only contain a finite set of discrete numbers. The total number of normalized floating-point numbers is:

$$2(\beta - 1)\beta^{p-1}(U - L + 1) + 1.$$

2 for the 2 possible signs

$\beta - 1$ since d_0 can be any integer between 1 and $\beta - 1$

β^{p-1} each of the $p-1$ fractional digit $d_1 d_2 \ldots d_{p-1}$ can be any integer between 0 and $\beta - 1$

$U - L + 1$ is the total number of different exponents

We add 1 to the product of the above factors since the number 0 is treated separately.

1.2. Characteristics of a Floating-Point System

Out of all those numbers, two numbers have important meanings. The smallest positive normalized floating-point number is called the un-

derflow level, UFL. UFL must have the smallest normalized mantissa of $1.0\ldots0$, and the smallest exponent of L, and so

$$UFL = \beta^L.$$

It has a value of

$$UFL = 2^{-1022} \approx 2.23 \times 10^{-308},$$

in IEEE double-precision. In MATLAB this number is named `realmin`.

The largest positive floating-point number is called the overflow level, OFL. It has a mantissa having the largest value. Therefore the leading digit as well as remaining ones must have the largest value of $\beta - 1$, *i.e.* $d_0 = d_1 = \ldots = d_{p-1} = \beta - 1$. The mantissa then has a value of

$$\beta - 1 + \frac{\beta - 1}{b} + \frac{\beta - 1}{b^2} + \ldots + \frac{\beta - 1}{b^{p-1}}.$$

To see how we can sum up this finite series, we consider a particular example where $\beta = 10$ and $p = 5$. We see that the series gives

$$9 + \frac{9}{10} + \frac{9}{10^2} + \frac{9}{10^3} + \frac{9}{10^4} = 9.9999.$$

Notice that by adding 0.0001 to the result, we get 10, which is β. Written in terms of β and p, we see that $0.0001 = 1/\beta^{p-1}$. The sum of the series must therefore be given by

$$\beta \frac{1}{\beta^{p-1}} = \left(1 - \frac{1}{\beta^p}\right) \beta.$$

The OFL must have the largest exponent of U. Therefore we have

$$OFL = \left(1 - \frac{1}{\beta^{p-1}}\right) \beta\beta^U = \left(1 - \frac{1}{\beta^p}\right) \beta^{U+1}.$$

In IEEE double-precision,

$$OFL = \left(1 - \frac{1}{2^{53}}\right) 2^{1024},$$

which is slightly less than 2^{1024}. An approximate value for the OFL is 1.8×10^{308}.

Ideally a floating-point system should have a large p for high precision, a large and negative L, so that numbers with very small magnitudes can be represented, and a large and positive U so that numbers with large magnitudes can be represented.

1.3. A Simply Example of A Floating-Point System

To obtain more insight into the floating-point system, we will now consider in detail a floating-point system, where $\beta = 2$, $p = 3$, $L = -1$, and $U = 1$. This is a "toy" system, so simple that we can write down all of its numbers.

First 0 is always present in any floating-point number system. For every positive floating-point number is a corresponding negative number. So we can concentrate on getting all the positive floating-point numbers first and then reverse their signs to get all the negative floating point numbers.

Immediately to the right of 0 is the UFL, which is given by

$$UFL = \beta^L = 2^{-1} = \frac{1}{2} = (0.5)_{10},$$

or equivalently by $(1.00)_2 \times 2^{-1}$. This floating-point number is shown by the red mark in the following diagram. The next floating-point number (also in red) is $(1.01)_2 \times 2^{-1}$. This number has a value $(1.00)_2 \times 2^{-1} + (0.01)_2 \times 2^{-1} = UFL + \frac{1}{8} = (0.625)_{10}$. The next floating-point number (also in red) is $(1.10)_2 \times 2^{-1} = UFL + 2 \times \frac{1}{8} = (0.75)_{10}$.

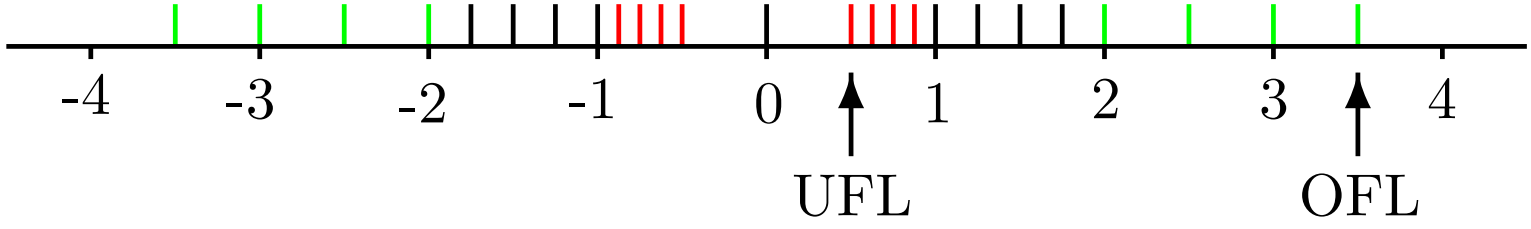
And the next one is $(1.11)_2 \times 2^{-1} = UFL + 3 \times \frac{1}{8} = (0.875)_{10}$, also shown in red. The mantissa reaches its highest possible value.

For the next number, the exponent increases by 1 and the mantissa returns to its smallest normalized value of $(1.00)_2$. The number (in black) is $(1.00)_2 \times 2^0 = (1)_{10}$. The next number (also in black) is $(1.01)_2 \times 2^0 = (1)_{10} + (0.01)_2 \times 2^0 = (1)_{10} + \frac{1}{4} = 1.25_{10}$. The next 2 numbers (also in black) are $(1.10)_2 \times 2^0 = (1)_{10} + 2 \times \frac{1}{4} = (1.5)_{10}$ and $(1.11)_2 \times 2^0 = (1)_{10} + 3 \times \frac{1}{4} = (1.75)_{10}$.

Again the mantissa reaches its highest possible value. For the next number, once again the exponent has to increase by 1 and the mantissa has to return to its smallest normalized value of $(1.00)_2$. The number (in green) is $(1.00)_2 \times 2^1 = (2)_{10}$. The next 3 numbers (also in green) are $(1.01)_2 \times 2^1 = (2)_{10} + (0.01)_2 \times 2^1 = (2)_{10} + \frac{1}{2} = (2.5)_{10}$, $(1.01)_2 \times 2^1 = (2)_{10} + 2 \times \frac{1}{2} = (3)_{10}$, and $(1.11)_2 \times 2^1 = (2)_{10} + 3 \times \frac{1}{2} = (3.5)_{10} = OFL$. There is no ordinary floating-point number larger than OFL.

Therefore this system has a total of 12 positive floating-point numbers. Putting a negative sign in front of these numbers gives 12 negative floating-point numbers. Together with 0, this system has a total

of 25 floating-point numbers. Notice that except for 0, and depends on how they are grouped (disjoint groups or as overlapping groups), all the floating-point numbers appear in groups of 4 ($= \beta^{p-1}$) or 5 ($= \beta^{p-1}+1$). Floating-point numbers are not distributed uniformly, except for numbers within the same group.



In summary, a normalized floating-point system has the following characteristics.

1. can only represent a total of $2(\beta-1)\beta^{p-1}(U-L+1)+1$ floating-point number exactly.
2. these number are discrete (not like the real numbers which are continuous)
3. these numbers are unevenly spaced except within the same group
4. 0 is always represented (also whole numbers with magnitudes

$= \beta^p$)

5. the smallest positive normalized floating-point number is UFL $= \beta^L$
6. the largest positive floating-point number is OFL $= \left(1 - \frac{1}{\beta^p}\right) \beta^{U+1}$
7. most other real numbers are not represented
8. those real numbers that can be represented exactly are called machine numbers
9. Basically all normalized floating-point systems exhibit the same characteristics. System appears less grainy and less uneven as p becomes large, L becomes more negative, and U becomes larger

2. Rounding Rules

Most real numbers are non-machine numbers. How can they be treated? Even if we start with machine numbers, most of the time operations involving them will result in non-machine numbers. For example, in the above toy floating-point system, the numbers 2 and

2.5 are machine numbers. However 2 divided by 2.5 gives a true value of 0.8, but 0.8 is not a machine number. It falls between two machine numbers, 0.75 and 0.875.

If x is a real non-machine number, then it is approximated by a "nearby" machine number, denoted by $fl(x)$. This process is called rounding. The error introduced by rounding is called rounding or roundoff error.

Two commonly used rounding rules are:

chop truncate base-β expansion of x after the $(p-1)$st digit. This rule is called "round toward zero".

round-to-nearest $fl(x)$ is given by the floating-point number nearest to x. In case of a tie, use the one whose last stored digit, the so-called least significant digit (LSD) is even. This rule is called "round-to-even".

Of these two rules, round-to-nearest is more accurate (in the sense described in the next section) and is the default rounding rule in IEEE systems. We will consider the round-to-nearest rule here.

Again using the toy floating-point system as an example, we see

that in the calculation $0.75 + 1 = 1.75$, all the numbers involved are machine numbers. The calculation is therefore performed exactly. We are extremely lucky. On the other hand, the true sum of the machine numbers 0.875 and 1.5 is 2.375, which lies between the two consecutive machine numbers, 2 and 2.5. Since 2.375 lies closer to 2.5 than to 2, it is approximated by 2.5. Therefore $0.875 + 1.5$ gives 2.5 in this floating-point system. As another example, the true sum of machine numbers 0.625 and 1 is 1.625. This number lies exactly between the two consecutive machine numbers, 1.5 and 1.75, which have representations $(1.10)_2 \times 2^0$ and $(1.11)_2 \times 2^0$, respectively. The round-to-nearest rule therefore picks 1.5 as the answer, since its LSD is even.

2.1. Machine Precision

Accuracy of a floating-point system is characterized by ϵ_{mach}, referred to as the machine epsilon, machine precision, or unit roundoff. It is defined to be the smallest number ϵ such that

$$fl(1 + \epsilon) > 1.$$

Since 1 is a machine number represented by $(1.0\ldots0)\times\beta^0$, and the next machine number larger than 1 is $(1.0\ldots01)\times\beta^0 = 1+\frac{1}{\beta^{p-1}}$. This number is larger than 1 by β^{1-p}. Therefore in the round-to-nearest rule

$$\epsilon_{\text{mach}} = \beta^{1-p}/2.$$

The machine epsilon is sometimes also defined to be the maximum possible relative error in representing a nonzero real number x in a floating-point system. That means that the maximum relative error in representing any number is bounded by the machine epsilon:

$$\left|\frac{fl(x)-x}{x}\right| < \epsilon_{\text{mach}}.$$

For our toy floating-point system, $\epsilon_{\text{mach}} = 2^{1-3}/2 = 0.125$. For the IEEE floating-point system, in single precision $\epsilon_{\text{mach}} = 2^{1-24}/2 = 2^{-24} \approx 6.0\times10^{-8}$, *i.e.* about 7 digits of precision, and in double precision $\epsilon_{\text{mach}} = 2^{1-53}/2 = 2^{-53} \approx 1.1\times10^{-16}$, *i.e.* about 16 digits of precision. Note that ϵ_{mach} is much larger than the UFL in all practical floating-point number system.

MATLAB defines `eps` as the distance between 1 and the next larger floating-point number. That distance is β^{1-p}. Thus we see that `eps` is $2\epsilon_{\text{mach}}$. Thus we see that a real number x that is slightly larger than $1+\epsilon_{\text{mach}}$ is represented as $fl(x) = 1+eps = 1+2*\epsilon_{\text{mach}}$, the next machine number larger than 1. However if x is either equal to or slightly less than $1 + \epsilon_{\text{mach}}$ then it is represented as $fl(x) = 1$.

On the other hand, the next machine number less than 1 is $1 - eps/2 = 1 - \epsilon_{\text{mach}}$. Thus a real number x that is slightly less than $1 - \epsilon_{\text{mach}}/2$ is then represented as $fl(x) = 1 - eps/2 = 1 - \epsilon_{\text{mach}}$, the next machine number smaller than 1. However if x is slightly larger than or equal to $1 - \epsilon_{\text{mach}}/2$, then it is represented as $fl(x) = 1$.

2.2. Subnormals and Gradual Underflow

Normalization causes the floating-point system to have no numbers other than 0 in the interval $[-UFL,\ UFL]$. Thus any number with a magnitude less than $UFL/2 = \beta^L/2$ at any step in a calculation is set to 0. Most of the floating-point systems, including the single- and double-precision IEEE systems, attempt to fill in the above interval

by allowing subnormal or denormalized floating-point numbers. These numbers are introduced by relaxing the normalization requirement by allowing the leading digits in the mantissa to be zero if the exponent is at its lowest value L.

These subnormal floating-point numbers added to the system all have the form

$$(0.d_1 d_2 \ldots d_{p-1}) \times \beta^L,$$

where $d_1, d_2 \ldots d_{p-1}$ are integers between 0 and $\beta - 1$. (The number for which all the fractional digits are zero is clearly the floating-point number 0, which has been considered before and is not a subnormal number.) Since there are $p-1$ fractional digits and each digit can take on β possible values, thus by allowing for subnormals we add a total of $2(\beta^{p-1} - 1)$ floating-points to the system. The factor of 2 comes form the 2 possible signs, and we subtract 1 because 0 was considered already.

The smallest subnormal floating-point number is

$$(0.0 \ldots 01)_\beta \times \beta^L = \frac{1}{\beta^{p-1}} \beta^L = \beta^{L-p+1}.$$

Now any floating-point number whose magnitude is less than or equal to $\beta^{L-p+1}/2$ (rather than $UFL/2 = \beta^L/2$) is then set to 0. This augmented system is said to exhibit gradual underflow.

Note that ϵ_{mach} still has the same value as before. Subnormal numbers extend the range of magnitudes representable but they have less precision than the normalized ones since 1 or more of their leading digits are zero.

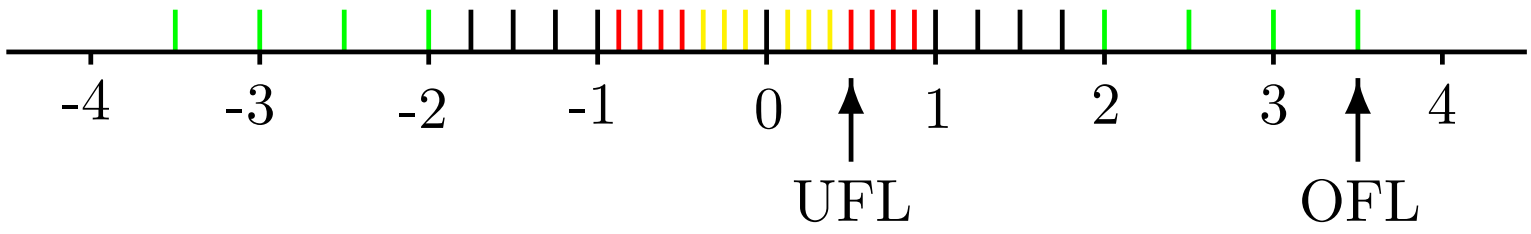
For our toy floating-point system, since $\beta = 2$ and $p = 3$, the total number of subnormals is $2(\beta^{p-1} - 1) = 6$. Thus subnormals add three numbers on each side of the origin. The three positive ones are

$$(0.01)_2 \times 2^{-1} = \frac{1}{8}$$

$$(0.10)_2 \times 2^{-1} = \frac{2}{8} = \frac{1}{4}$$

$$(0.11)_2 \times 2^{-1} = \frac{3}{8}$$

The floating-point numbers of the entire floating-point system are shown in the following diagram. The subnormals are shown in yellow.



We will now consider examples with subnormals in IEEE double-precision systems. We define two normalized real numbers $a = 1 \times 10^{15}$ and $b = 1 \times 10^{-307}$. Clearly both numbers lie between the UFL and the OFL, and therefore have full precision given by ϵ_{mach}. Notice that b is very small, only slightly larger than the UFL ($\approx 2.23 \times 10^{-308}$). We now compute the ratio $r = b/a$ using for example MATLAB. The result to 16 digit precision is $r = 9.881312916824931 \times 10^{-323}$, which is smaller than the UFL and is clearly a subnormal. This number cannot have full precision. In fact the correct answer for r is $(1 \times 10^{15}) \times (1 \times 10^{-307}) = 1 \times 10^{-322}$. The relative error for r is 0.012, and that is much much larger than ϵ_{mach}.

The smallest subnormal is $s = UFL/(2^{52})$, which has a value of $4.940656458412465 \times 10^{-324}$. Any number whose magnitude is less than of equal to half of s is rounded to zero. One can check to see

that $s/1.99999$ gives s again, and $s/2$ gives 0.

3. Exceptional Values

IEEE floating-point standard provides special values to indicate two exceptional situations:

- Inf, which stands for infinity, results from dividing a finite number by zero, such as $1/0$
- NaN, which stands for not-a-number, results from undefined or indeterminate operations such as $0/0$, $0 * \infty$, or ∞/∞

Inf and NaN are implemented in IEEE arithmetic through special reserved values of the exponent field

Some languages like MATLAB can sensibly handle and propagate these exceptional values through a calculation. For example, Inf - 1 = Inf, 5* NaN +7 = NaN. Data such as

x	1	2	3	4	5	6
y	0.25	0.37	NaN	0.46	0.32	0.21

can be plotted in MAT-

LAB. The third data point is omitted in the plot, with a warning about having a NaN in the data.

3.1. Floating-Point Arithmetic

Addition or subtraction: The exponents of two floating-point numbers must be made to match by shifting the mantissas before they can be added or subtracted. But shifting the mantissa may cause loss of some and possibly even all the digits of the smaller number.

Multiplication: Product of two p-digit mantissas contains up to 2p digits, so result may not be representable

Division: Quotient of two p-digit mantissas may contain more than p digits, such as nonterminating binary expansion of 1/10

Result of floating-point arithmetic operation may differ from the result of the corresponding real arithmetic operation on the same operands.

3.2. Example: Floating-Point Arithmetic

Assume we are using a floating-point system where $\beta = 10$, $p = 6$. Let $x = 1.92403 \times 10^2$, $y = 6.35782 \times 10^{-1}$, floating-point addition gives the true result $1.92403 \times 10^2 + 0.00635782 \times 10^2 = 1.93038782 \times 10^2$. But the floating-point system has only a precision of $p = 6$, therefore the result is 1.93039×10^2, assuming rounding to nearest. The calculation was done by shifting the decimal point in the mantissa of the smaller number so that its exponent matches with that of the larger number.

We can also do the calculation by shifting the decimal point of the larger number so that its exponent matches with that of the smaller number. This gives the true result $1924.03 \times 10^{-1} + 6.35782 \times 10^{-1} = 1930.38782 \times 10^{-1}$. Rounding to nearest then gives the same result as the previous calculation 1.93039×10^2.

Notice that the last two digits of y do not affect the final result, and with even smaller exponent, y could have had no effect on the result at all.

Multiplication gives the true result $x * y = 1.22326556549 \times 10^2$. Rounding to nearest then gives the floating-point multiplication value

of 1.22326×10^2. Notice that half of the digits of the true product are discarded.

Division is similar to multiplication except that the result of a division by two machine numbers may not be a machine number. For example, 1 is a machine number represented by $(1.00\ldots0)_\beta\beta^0$, and 10 is a machine number represented by $(1.010\ldots0)_\beta\beta^3$. But 1 divided by 10 is 0.1 and it is not a machine number. In fact it has a non-terminating (repeating) representation $(1.10011001100\ldots)_\beta\beta^{-4}$, which is the counterpart of repeating decimals in a decimal system.

Real result may also fail to be representable because its exponent is beyond available range.

Over flow is usually more serious than under flow because there is no good approximation to arbitrarily large magnitudes in a floating-point system, whereas zero is often reasonable approximation for arbitrarily small magnitudes.

On many computer systems over flow is fatal, but an under flow may be silently set to zero.

Ideally, x flop y = (x op y), i.e., floating point arithmetic operations produce correctly rounded results. Computers satisfying IEEE

floating-point standard achieve this ideal as long as x op y is within range of floating-point system. But some familiar laws of real arithmetic not necessarily valid in floating-point system. Floating-point addition and multiplication commutative but not associative.

Example: if ϵ is positive floating-point number slightly smaller than ϵ_{mach}, $(1+\epsilon)+\epsilon = 1$; but $1+(\epsilon+\epsilon) > 1$.

4. Cancellation

Subtraction between two p-digit numbers having the same sign and magnitudes differing by less than a factor of 2, the leading digit(s) will cancel. The result will have fewer number of significant digits, although it is usually exactly representable.

Cancellation loses the most significant (leading) bit(s) and is therefore much worst than rounding, this loses the least significant (trailing) bit(s).

Again assume we are using a floating-point system where $\beta = 10$, $p = 6$. Let $x = 1.92403 \times 10^2$, $z = 1.92275 \times 10^2$, floating-point difference of these numbers gives the true result 0.00128×10^2,

which has only 3 significant digits! It can therefore can be represented exactly as 1.28000×10^{-1}.

Despite exactness of result, cancellation often implies serious loss of information Operands often uncertain due to rounding or other previous errors, so relative uncertainty in difference may be large Example: if ϵ is positive floating-point number slightly smaller than ϵ_{mach},

$$(1 + \epsilon) - (1 - \epsilon) = 1 - 1 = 0,$$

in floating-point arithmetic. The true result of the overall computation, 2ϵ, has been completely lost.

Subtraction itself not at fault: it merely signals loss of information that had already occurred

Because of cancellation, it is generally a bad idea to compute any small quantity as the difference of large quantities, since rounding error is likely to dominate the result.

The total energy, E of helium atom is the sum of kinetic (K.E.) and potential energies (P.E.), which are computed separately and have opposite signs, so suffer cancellation.

The following table gives a sequence of values obtained over 18 years. During this span the computed values for the K.E. range from 12.22 to 13.0, a 6.3% variation, and the P.E. range from −14.0 to −14.84, a 6.0% variation. However the computed values for E range from −1.0 to −2.44, a 144% variation!

Year	K.E.	P.E.	E = K.E. + P.E.
1971	13.0	-14.0	-1.0
1977	12.76	-14.02	-1.26
1980	12.22	-14.35	-2.13
1985	12.28	-14.65	-2.37
1988	12.40	-14.84	-2.44

4.1. Example: Quadratic Formula

The two roots of the quadratic equation

$$ax^2 + bx + c = 0,$$

are given by

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Naive use of the above quadratic formula can suffer over flow, or under flow, or severe cancellation.

Rescaling coefficients can help avoid over flow and harmful under flow.

Cancellation inside the square root cannot be easily avoided without using higher precision arithmetic.

Cancellation between $-b$ and the square root can be avoided by computing one root using the following alternative formula:

$$\begin{aligned} x_{1,2} &= \left(\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}\right)\left(\frac{-b \mp \sqrt{b^2 - 4ac}}{-b \mp \sqrt{b^2 - 4ac}}\right) \\ &= \frac{b^2 - (b^2 - 4ac)}{2a(-b \mp \sqrt{b^2 - 4ac})} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}. \end{aligned}$$

To see how that works, let us be specific and assume that b is negative. If $4ac$ is positive and very small compared to b^2, then the

square-root is real and is only slightly smaller than $-b$. Large cancellation is expected in computing x_2 using the original formula, but not for x_1. On the other hand, large cancellation is expected in computing x_1 using the alternate formula, but not for x_2.

As an example, let us consider a floating-point system where $\beta = 10$ and $p = 4$. We want to compute the roots for a quadratic equation where $a = 0.05010$, $b = -98.78$, and $c = 5.015$. First, $fl(b^2) = fl(9757.488\ldots) = 9757$, $4a = 0.2004$ without rounding, $4ac = fl(0.2004 \times 5.015) = 1.005$, $fl(b^2 - 4ac) = 9756$, and so $fl(\sqrt{b^2 - 4ac}) = 98.77$. Also $fl(2a) = fl(2 \times 0.0510) = 0.1002$, and $fl(2c) = fl(2 \times 5.015) = 10.03$ all without rounding.

Using the original formula to compute x_1, the numerator is $fl(98.78 + 98.77) = fl(197.55) = 197.6$ (not 197.5), and so $x_1 = fl(197.6/0.1002) = 1972$. This result for x_1 is correct to all 4 digits. The numerator for x_2 is $fl(98.78 - 98.77) = fl(0.0100) = 0.0100$, and so $x_2 = fl(0.0100/0.1002) = 0.09980$. But even the leading digit of this result for x_2 is incorrect.

If we use the alternate formula to compute x_1, we have $x_1 = fl(10.03/0.010) = 1003$. This result is far from being correct. How-

ever for x_2, we have $x_2 = fl(10.03/197.6) = fl(0.0507591) = 0.05076$. The result correct to 4 significant digits is 0.05077.

The above example was made up to bring out the problem associated with cancellation. With increasing precision p, b^2 has to be even much larger than $4ac$ before the problem shows up. In fact in IEEE double precision, using the above values for a, b and c, the roots are correctly computed up to about 14 significant digits even if the "wrong" formula is used.

4.2. Standard Deviation

The mean of a sequence $x_i, i = 1, 2, \ldots n$, is given by

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i,$$

and the standard deviation by

$$\sigma = \left[\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right]^{\frac{1}{2}}.$$

Some people use the mathematically equivalent formula

$$\sigma = \left[\frac{1}{n-1} \left(\sum_{i=1}^{n} x_i^2 - n\bar{x}^2 \right) \right]^{\frac{1}{2}}.$$

to avoid making two passes through the data. Unfortunately, the two terms in the one-pass formula are usually large and nearly equal and so the single cancellation error at the end is more damaging numerically than all of cancellation errors in the two-pass formula combined.