Floating-Point Numbers

Floating-point number system characterized by four integers:

β	base or radix
p	precision
$[L, U]$	exponent range

Number x represented as

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

where

$0 \leq d_i \leq \beta - 1, \ i = 0, \ldots, p-1,$ and $L \leq E \leq U$

$d_0 d_1 \cdots d_{p-1}$ called *mantissa*

E called *exponent*

$d_1 d_2 \cdots d_{p-1}$ called *fraction*

Typical Floating-Point Systems

Most computers use binary ($\beta = 2$) arithmetic

Parameters for typical floating-point systems shown below

system	β	p	L	U
IEEE SP	2	24	-126	127
IEEE DP	2	53	-1022	1023
Cray	2	48	-16383	16384
HP calculator	10	12	-499	499
IBM mainframe	16	6	-64	63

IEEE standard floating-point systems almost universally adopted for personal computers and workstations

Normalization

Floating-point system *normalized* if leading digit d_0 always nonzero unless number represented is zero

In normalized system, mantissa m of nonzero floating-point number always satisfies

$$1 \leq m < \beta$$

Reasons for normalization:

- representation of each number unique
- no digits wasted on leading zeros
- leading bit need not be stored (in binary system)

Properties of Floating-Point Systems

Floating-point number system finite and discrete

Number of normalized floating-point numbers:

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

Smallest positive normalized number:

$$\text{underflow level} = \text{UFL} = \beta^{L}$$

Largest floating-point number:

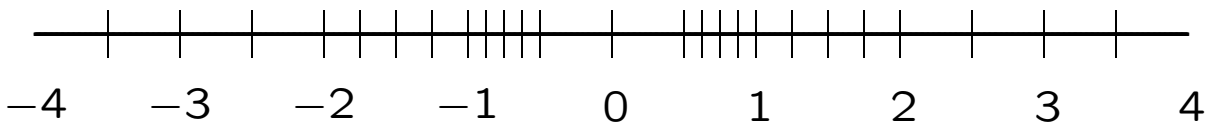
$$\text{overflow level} = \text{OFL} = \beta^{U+1}(1 - \beta^{-p})$$

Floating-point numbers equally spaced only between powers of β

Not all real numbers exactly representable; those that are are called *machine numbers*

Example: Floating-Point System

Tick marks indicate all 25 numbers in floating-point system having $\beta = 2$, $p = 3$, $L = -1$, and $U = 1$



$\text{OFL} = (1.11)_2 \times 2^1 = (3.5)_{10}$

$\text{UFL} = (1.00)_2 \times 2^{-1} = (0.5)_{10}$

At sufficiently high magnification, all normalized floating-point systems look grainy and unequally spaced like this

Rounding Rules

If real number x not exactly representable, then approximated by "nearby" floating-point number fl(x)

Process called *rounding*, and error introduced called *rounding error*

Two commonly used rounding rules:

- *chop*: truncate base-β expansion of x after $(p-1)$st digit; also called *round toward zero*
- *round to nearest*: fl(x) nearest floating-point number to x, using floating-point number whose last stored digit is even in case of tie; also called *round to even*

Round to nearest most accurate, and is default rounding rule in IEEE systems

Machine Precision

Accuracy of floating-point system characterized by *unit roundoff*, *machine precision*, or *machine epsilon*, denoted by ϵ_{mach}

With rounding by chopping, $\epsilon_{\text{mach}} = \beta^{1-p}$

With rounding to nearest, $\epsilon_{\text{mach}} = \frac{1}{2}\beta^{1-p}$

Alternative definition is smallest number ϵ such that $\text{fl}(1 + \epsilon) > 1$

Maximum *relative error* in representing real number x in floating-point system given by

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

Machine Precision, continued

For toy system illustrated earlier,

$$\epsilon_{\text{mach}} = 0.25 \text{ with rounding by chopping}$$

$$\epsilon_{\text{mach}} = 0.125 \text{ with rounding to nearest}$$

For IEEE floating-point systems,

$$\epsilon_{\text{mach}} = 2^{-24} \approx 10^{-7} \text{ in single precision}$$

$$\epsilon_{\text{mach}} = 2^{-53} \approx 10^{-16} \text{ in double precision}$$

IEEE single and double precision systems have about 7 and 16 decimal digits of precision

Though both are "small," unit roundoff error ϵ_{mach} should not be confused with underflow level UFL

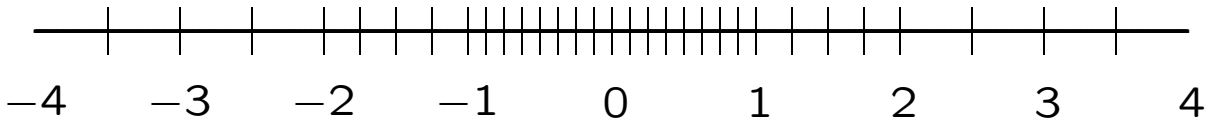
In all practical floating-point systems,

$$0 < \text{UFL} < \epsilon_{\text{mach}} < \text{OFL}$$

Subnormals and Gradual Underflow

Normalization causes gap around zero in floating-point system

If leading digits allowed to be zero, but only when exponent at its minimum value, then gap "filled in" by additional *subnormal* or *denormalized* floating-point numbers



Subnormals extend range of magnitudes representable, but have less precision than normalized numbers, and unit roundoff is no smaller

Augmented system exhibits *gradual underflow*

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 $0/0$, $0 * \texttt{Inf}$, or $\texttt{Inf}/\texttt{Inf}$

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

Floating-Point Arithmetic

Addition or subtraction: Shifting of mantissa to make exponents match may cause loss of some digits of smaller number, possibly all of them

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 non-terminating binary expansion of $1/10$

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

Example: Floating-Point Arithmetic

Assume $\beta = 10$, $p = 6$

Let $x = 1.92403 \times 10^2$, $y = 6.35782 \times 10^{-1}$

Floating-point addition gives

$$x + y = 1.93039 \times 10^2,$$

assuming rounding to nearest

Last two digits of y do not affect result, and with even smaller exponent, y could have had no effect on result

Floating-point multiplication gives

$$x * y = 1.22326 \times 10^2,$$

which discards half of digits of true product

Floating-Point Arithmetic, continued

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

Overflow usually more serious than underflow because there is *no* good approximation to arbitrarily large magnitudes in floating-point system, whereas zero is often reasonable approximation for arbitrarily small magnitudes

On many computer systems overflow is fatal, but an underflow may be silently set to zero

Example: Summing a Series

Infinite series

$$\sum_{n=1}^{\infty} \frac{1}{n}$$

has finite sum in floating-point arithmetic even though real series is divergent

Possible explanations:

- Partial sum eventually overflows
- $1/n$ eventually underflows
- Partial sum ceases to change once $1/n$ becomes negligible relative to partial sum:

$$1/n < \epsilon_{\text{mach}} \sum_{k=1}^{n-1} (1/k)$$

Floating-Point Arithmetic, continued

Ideally, x `flop` y = fl(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},

$$(1 + \epsilon) + \epsilon = 1, \text{ but } 1 + (\epsilon + \epsilon) > 1$$

Cancellation

Subtraction between two p-digit numbers having same sign and similar magnitudes yields result with *fewer* than p digits, so it is usually exactly representable

Reason is that leading digits of two numbers *cancel* (i.e., their difference is zero)

Example:

$$1.92403 \times 10^2 - 1.92275 \times 10^2 = 1.28000 \times 10^{-1},$$

which is correct, and exactly representable, but has only three significant digits

Cancellation, continued

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, which is correct for actual operands of final subtraction, but true result of overall computation, 2ϵ, has been completely lost

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

Cancellation, continued

Digits lost to cancellation are most significant, leading digits, whereas digits lost in rounding are least significant, trailing digits

Because of this effect, it is generally bad idea to compute any small quantity as difference of large quantities, since rounding error is likely to dominate result

For example, summing alternating series, such as

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

for $x < 0$, may give disastrous results due to catastrophic cancellation

Example: Cancellation

Total energy of helium atom is sum of kinetic and potential energies, which are computed separately and have opposite signs, so suffer cancellation

Year	Kinetic	Potential	Total
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

Although computed values for kinetic and potential energies changed by only 6% or less, resulting estimate for total energy changed by 144%

Example: Quadratic Formula

Two solutions of quadratic equation

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

given by

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Naive use of formula can suffer overflow, or underflow, or severe cancellation

Rescaling coefficients can help avoid overflow and harmful underflow

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

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}$$

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

Example: Standard Deviation

Mean of sequence x_i, $i = 1, \ldots, n$, is given by

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

and standard deviation by

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

Mathematically equivalent formula

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

avoids making two passes through data

Unfortunately, single cancellation error at end of one-pass formula is more damaging numerically than all of cancellation errors in two-pass formula combined

Mathematical Software

High-quality mathematical software is available for solving most commonly occurring problems in scientific computing

Use of sophisticated, professionally written software has many advantages

We will seek to understand basic ideas of methods on which such software is based, so that we can use software intelligently

We will gain hands-on experience in using such software to solve wide variety of computational problems

Desirable Qualities of Math Software

- Reliability
- Robustness
- Accuracy
- Efficiency
- Maintainability
- Portability
- Usability
- Applicability

Sources of Math Software

`FMM`: From book by Forsythe/Malcolm/Moler

`HSL`: Harwell Subroutine Library

`IMSL`: Internat. Math. & Stat. Libraries

`KMN`: From book by Kahaner/Moler/Nash

`NAG`: Numerical Algorithms Group

`Netlib`: Free software available via Internet

`NR`: From book *Numerical Recipes*

`NUMAL`: From Math. Centrum, Amsterdam

`SLATEC`: From U.S. Government labs

`SOL`: Systems Optimization Lab, Stanford U.

`TOMS`: ACM Trans. on Math. Software

Scientific Computing Environments

Interactive environments for scientific computing provide

- powerful mathematical capabilities
- sophisticated graphics
- high-level programming language for rapid prototyping

`MATLAB` is popular example, available for most personal computers and workstations

Similar, "free" alternatives include `octave`, `RLaB`, and `Scilab`

Symbolic computing environments, such as Maple and Mathematica, also useful