

Computer arithmetic

Victor Eijkhout

Fall 2019

First we dig into bits
Integers
Floating point numbers
Floating point math
Examples
More

Justification

This short session will explain the basics of floating point arithmetic, mostly focusing on round-off and its influence on computations.

Numbers in scientific computing

- Integers: $\dots, -2, -1, 0, 1, 2, \dots$
- Rational numbers: $1/3, 22/7$: not often encountered
- Real numbers $0, 1, -1.5, 2/3, \sqrt{2}, \log 10, \dots$
- Complex numbers $1 + 2i, \sqrt{3} - \sqrt{5}i, \dots$

Computers use a finite number of bits to represent numbers, so only a finite number of numbers can be represented, and no irrational numbers (even some rational numbers).

Bit operations

	boolean	bitwise
and	& &	&
or		
not	!	
xor		^

Bit string operations:

left shift	<<
right shift	>>

First we dig into bits
Integers
Floating point numbers
Floating point math
Examples
More

Exercise 1: Bit operations

Use bit operations to test whether a number is odd or even.

Integers

Scientific computation mostly uses real numbers. Integers are mostly used for array indexing.

16/32/64 bit: `short, int, long, long long` in C, size not standardized, use `sizeof(long)` et cetera. (Also `unsigned int` et cetera)

`INTEGER*2/4/8` Fortran, also `KIND`

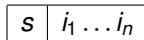
Exercise 2: Powers of two

Print 2^n for $n = 0, \dots, 31$. There are at least two ways of generating these powers.

Also print the bit pattern. What is unexpected?

Negative integers

Use of sign bit: typically first bit



Simplest solution: $n > 0$, $\text{rep}(n) = 0, i_1, \dots, i_{31}$, then
 $\text{rep}(-n) = 1, i_1, \dots, i_{31}$

Problem: $+0$ and -0 ; also impractical in other ways.

Sign bit

bitstring	00...0	...	01...1	10...0	...	11...1
as unsigned int	0	...	$2^{31} - 1$	2^{31}	...	$2^{32} - 1$
as naive signed	0	...	$2^{31} - 1$	-0	...	$-2^{31} + 1$

Shifting

Interpret unsigned number n as $n - B$

bitstring	00...0	...	01...1	10...0	...	11...1
as unsigned int	0	...	$2^{31} - 1$	2^{31}	...	$2^{32} - 1$
as shifted int	-2^{31}	...	-1	0	...	$2^{31} - 1$

2's Complement

Let m be a signed integer, then the 2's complement 'bit pattern' $\tau(m)$ is a non-negative integer defined as follows:

- If $0 \leq m \leq 2^{31} - 1$, the normal bit pattern for m is used, that is

$$0 \leq m \leq 2^{31} - 1 \Rightarrow \tau(m) = m.$$

- For $-2^{31} \leq n \leq -1$, n is represented by the bit pattern for $2^{32} - |n|$:

$$-2^{31} \leq n \leq -1 \Rightarrow \tau(m) = 2^{32} - |n|.$$

Bit pattern to integer: $\eta = \tau^{-1}$.

Addition in 2's complement

Add $m + n$, where m, n are representable:

$$0 \leq |m|, |n| < 2^{31}.$$

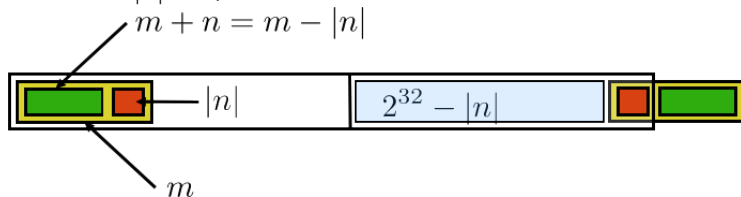
The easy case is $0 < m, n$, as long as there is no overflow.

Addition in 2's complement (cont'd)

Case $m > 0$, $n < 0$, and $m + n > 0$. Then $\tau(m) = m$ and $\tau(n) = 2^{32} - |n|$, so the unsigned addition becomes

$$\tau(m) + \tau(n) = m + (2^{32} - |n|) = 2^{32} + m - |n|.$$

Since $m - |n| > 0$, this result is $> 2^{32}$.



However, this is basically $m + n$ with the overflow bit set.

Subtraction in 2's complement

Subtraction $m - n$:

- Case: $m < n$. Observe that $-n$ has the bit pattern of $2^{32} - n$. Also, $m + (2^{32} - n) = 2^{32} - (n - m)$ where $0 < n - m < 2^{31} - 1$, so $2^{32} - (n - m)$ is the 2's complement bit pattern of $m - n$.
- Case: $m > n$. The bit pattern for $-n$ is $2^{32} - n$, so $m + (-n)$ as unsigned is $m + 2^{32} - n = 2^{32} + (m - n)$. Here $m - n > 0$. The 2^{32} is an overflow bit; ignore.

Overflow

There is a limited number of bits, so numbers that are too large in absolute value can not be represented.

Overflow.

This is not a fatal error: your program continues with the wrong result.

Exercise 3: Integer overflow

Investigate what happens when you perform an integer calculation that leads to overflow. What does your compiler say if you try to write down a nonrepresentable number explicitly, for instance in a declaration or assignment statement?

Floating point numbers

Analogous to scientific notation $x = 6.022 \cdot 10^{23}$:

$$x = \pm \sum_{i=0}^{t-1} d_i \beta^{-i} \beta^e$$

- sign bit
- β is the base of the number system
- $0 \leq d_i \leq \beta - 1$ the digits of the *mantissa*:
one digit before the *radix point*, so mantissa $< \beta$
- $e \in [L, U]$ exponent, stored with bias: unsigned int where $\text{fl}(L) = 0$

Examples of floating point systems

	β	t	L	U
IEEE single (32 bit)	2	24	-126	127
IEEE double (64 bit)	2	53	-1022	1023
Old Cray 64bit	2	48	-16383	16384
IBM mainframe 32 bit	16	6	-64	63
packed decimal	10	50	-999	999

BCD is tricky: 3 decimal digits in 10 bits

(we will often use $\beta = 10$ in the examples, because it's easier to read for humans, but all practical computers use $\beta = 2$)

Internal processing in 80 bit

Limitations

Overflow: more than $\beta(1 - \beta^{-t+1})\beta^U$ or less than $\beta(1 - \beta^{-t+1})\beta^L$

Underflow: numbers less than $\beta^{-t+1} \cdot \beta^L$

Exercise 4: Floating point overflow

For real numbers x, y , the quantity $g = \sqrt{(x^2 + y^2)/2}$ satisfies

$$g \leq \max\{|x|, |y|\}$$

so it is representable if x and y are. What can go wrong if you compute g using the above formula? Can you think of a better way?

Normalized numbers

Require first digit in the mantissa to be nonzero.

Equivalent: mantissa part $1 \leq x_m < \beta$

Unique representation for each number,

(do you see a problem?)

also: in binary this makes the first digit 1, so we don't need to store that.

With normalized numbers, underflow threshold is $1 \cdot \beta^L$;

'gradual underflow' possible, but usually not efficient.

IEEE 754

sign	exponent	mantissa
s	$e_1 \cdots e_8$	$s_1 \dots s_{23}$
31	30 \cdots 23	22 \cdots 0

$(e_1 \cdots e_8)$	numerical value
$(0 \cdots 0) = 0$	$\pm 0.s_1 \cdots s_{23} \times 2^{-126}$
$(0 \cdots 01) = 1$	$\pm 1.s_1 \cdots s_{23} \times 2^{-126}$
$(0 \cdots 010) = 2$	$\pm 1.s_1 \cdots s_{23} \times 2^{-125}$
\dots	
$(01111111) = 127$	$\pm 1.s_1 \cdots s_{23} \times 2^0$
$(10000000) = 128$	$\pm 1.s_1 \cdots s_{23} \times 2^1$
\dots	
$(11111110) = 254$	$\pm 1.s_1 \cdots s_{23} \times 2^{127}$
$(11111111) = 255$	$\pm \infty$ if $s_1 \cdots s_{23} = 0$, otherwise NaN

Representation error

Error between number x and representation \tilde{x} :

absolute $x - \tilde{x}$ or $|x - \tilde{x}|$

relative $\frac{x - \tilde{x}}{x}$ or $|\frac{x - \tilde{x}}{x}|$

Equivalent: $\tilde{x} = x \pm \epsilon \Leftrightarrow |x - \tilde{x}| \leq \epsilon \Leftrightarrow \tilde{x} \in [x - \epsilon, x + \epsilon]$.

Also: $\tilde{x} = x(1 + \epsilon)$ often shorthand for $|\frac{\tilde{x} - x}{x}| \leq \epsilon$

Example

Decimal, $t = 3$ digit mantissa: let $x = 1.256$, $\tilde{x}_{\text{round}} = 1.26$,
 $\tilde{x}_{\text{truncate}} = 1.25$

Error in the 4th digit: $|\epsilon| < \beta^{t-1}$ (this example had no exponent, how about if it does?)

Exercise 5: Round-off

The number $e \approx 2.72$, the base for the natural logarithm, has various definitions. One of them is

$$e = \lim_{n \rightarrow \infty} (1 + 1/n)^n.$$

Write a single precision program that tries to compute e in this manner. Evaluate the expression for an upper bound $n = 10^k$ with $k = 1, \dots, 10$. Explain the output for large n . Comment on the behaviour of the error.

Machine precision

Any real number can be represented to a certain precision:

$\tilde{x} = x(1 + \varepsilon)$ where

truncation: $\varepsilon = \beta^{-t+1}$

rounding: $\varepsilon = \frac{1}{2}\beta^{-t+1}$

This is called *machine precision*: maximum relative error.

32-bit single precision: $mp \approx 10^{-7}$

64-bit double precision: $mp \approx 10^{-16}$

Maximum attainable accuracy.

Another definition of machine precision: smallest number ε such that
 $1 + \varepsilon > 1$.

Exercise 6: Machine epsilon

Write a small program that computes the machine epsilon for both single and double precision. Does it make any difference if you set the *compiler optimization levels* low or high?

(For C++ programmers: can you write a templated program that works for single and double precision?)

Addition

1. align exponents
2. add mantissas
3. adjust exponent to normalize

Example: $1.00 + 2.00 \times 10^{-2} = 1.00 + .02 = 1.02$. This is exact, but what happens with $1.00 + 2.55 \times 10^{-2}$?

Example: $5.00 \times 10^1 + 5.04 = (5.00 + 0.504) \times 10^1 \rightarrow 5.50 \times 10^1$

Any error comes from limiting the mantissa: if x is the true sum and \tilde{x} the computed sum, then $\tilde{x} = x(1 + \epsilon)$ with $|\epsilon| < 10^{-2}$

The ‘correctly rounded arithmetic’ model

Assumption (enforced by IEEE 754):

The numerical result of an operation is the rounding of the exactly computed result.

$$\text{fl}(x_1 \odot x_2) = (x_1 \odot x_2)(1 + \epsilon)$$

where $\odot = +, -, *, /$

Note: this holds only for a single operation!

Guard digits

Correctly rounding is not trivial, especially for subtraction.

Example: $t = 2, \beta = 10$: $1.0 - 9.5 \times 10^{-1}$, exact result
 $0.05 = 5.0 \times 10^{-2}$.

- Simple approach:

$$1.0 - 9.5 \times 10^{-1} = 1.0 - 0.9 = 0.1 = 1.0 \times 10^{-1}$$

- Using 'guard digit':

$$1.0 - 9.5 \times 10^{-1} = 1.0 - 0.95 = 0.05 = 5.0 \times 10^{-2}, \text{ exact.}$$

In general 3 extra bits needed.

Fused Mul-Add instructions

$$a \leftarrow a * b + c \quad \text{or} \quad c \leftarrow a * b + c$$

- Addition plus multiplication, but not independent
- Processors can have dedicated hardware for FMA (also IEEE 754-2008)
- Internally evaluated in higher precision: 80-bit.
- Very useful for certain linear algebra (which?) Not for other operations (examples?)

Associativity

Compute $4 + 6 + 7$ in one significant digit.

Evaluation left-to-right gives:

$$\begin{aligned}(4 \cdot 10^0 + 6 \cdot 10^0) + 7 \cdot 10^0 &\Rightarrow 10 \cdot 10^0 + 7 \cdot 10^0 && \text{addition} \\ &\Rightarrow 1 \cdot 10^1 + 7 \cdot 10^0 && \text{rounding} \\ &\Rightarrow 1.0 \cdot 10^1 + 0.7 \cdot 10^1 && \text{using guard digit} \\ &\Rightarrow 1.7 \cdot 10^1 \\ &\Rightarrow 2 \cdot 10^1 && \text{rounding}\end{aligned}$$

On the other hand, evaluation right-to-left gives:

$$\begin{aligned}4 \cdot 10^0 + (6 \cdot 10^0 + 7 \cdot 10^0) &\Rightarrow 4 \cdot 10^0 + 13 \cdot 10^0 && \text{addition} \\ &\Rightarrow 4 \cdot 10^0 + 1 \cdot 10^1 && \text{rounding} \\ &\Rightarrow 0.4 \cdot 10^1 + 1.0 \cdot 10^1 && \text{using guard digit} \\ &\Rightarrow 1.4 \cdot 10^1 \\ &\Rightarrow 1 \cdot 10^1 && \text{rounding}\end{aligned}$$

Error propagation under addition

Let $s = x_1 + x_2$, and $x = \tilde{s} = \tilde{x}_1 + \tilde{x}_2$ with $\tilde{x}_i = x_i(1 + \varepsilon_i)$

$$\begin{aligned}\tilde{x} &= \tilde{s}(1 + \varepsilon_3) \\ &= x_1(1 + \varepsilon_1)(1 + \varepsilon_3) + x_2(1 + \varepsilon_2)(1 + \varepsilon_3) \\ &= x_1 + x_2 + x_1(\varepsilon_1 + \varepsilon_3) + x_2(\varepsilon_2 + \varepsilon_3) \\ \Rightarrow \tilde{x} &= s(1 + 2\varepsilon)\end{aligned}$$

\Rightarrow errors are added

Assumptions: all ε_i approximately equal size and small;
 $x_i > 0$

Multiplication

1. add exponents
2. multiply mantissas
3. adjust exponent

Example:

$$.123 \times .567 \times 10^1 = .069741 \times 10^1 \rightarrow .69741 \times 10^0 \rightarrow .697 \times 10^0.$$

What happens with relative errors?

Subtraction

Correct rounding only applies to a single operation.

Example: $1.24 - 1.23 = 0.01 \rightarrow 1. \times 10^{-2}$:
result is exact, but only one significant digit.

What if $1.24 = \text{fl}(1.244)$ and $1.23 = \text{fl}(1.225)$? Correct
result 1.9×10^{-2} ; almost 100% error.

- *Cancellation* leads to loss of precision
- subsequent operations with this result are inaccurate
- this can not be fixed with guard digits and such
- \Rightarrow avoid subtracting numbers that are likely close.

ABC-formula

Example: $ax^2 + bx + c = 0 \rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

suppose $b > 0$ and $b^2 \gg 4ac$ then the '+' solution will be inaccurate

Better: compute $x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$ and use $x_+ \cdot x_- = -c/a$.

Serious example

Evaluate $\sum_{n=1}^{10000} \frac{1}{n^2} = 1.644834$

in 6 digits: machine precision is 10^{-6} in single precision

First term is 1, so partial sums are ≥ 1 , so $1/n^2 < 10^{-6}$ gets ignored,
 \Rightarrow last 7000 terms (or more) are ignored, \Rightarrow sum is 1.644725: 4
 correct digits

Solution: sum in reverse order; exact result in single precision

Why? Consider ratio of two terms:

$$\frac{n^2}{(n-1)^2} = \frac{n^2}{n^2 - 2n + 1} = \frac{1}{1 - 2/n + 1/n^2} \approx 1 + \frac{2}{n}$$

with aligned exponents:

$$\begin{array}{r|l} n-1: & .00 \cdots 0 \quad 10 \cdots 00 \\ n: & .00 \cdots 0 \quad 10 \cdots 01 \quad 0 \cdots 0 \\ & k = \log(n/2) \text{ positions} \end{array}$$

The last digit in the smaller number is not lost if $n < 2/\epsilon$

Another serious example

Previous example was due to finite representation; this example is more due to algorithm itself.

Consider $y_n = \int_0^1 \frac{x^n}{x-5} dx = \frac{1}{n} - 5y_{n-1}$ (monotonically decreasing)
 $y_0 = \ln 6 - \ln 5$.

In 3 decimal digits:

computation

correct result

$$y_0 = \ln 6 - \ln 5 = .182|322 \times 10^1 \dots$$

1.82

$$y_1 = .900 \times 10^{-1}$$

.884

$$y_2 = .500 \times 10^{-1}$$

.0580

$$y_3 = .830 \times 10^{-1}$$

going up? .0431

$$y_4 = -.165$$

negative? .0343

Reason? Define error as $\tilde{y}_n = y_n + \epsilon_n$, then

$$\tilde{y}_n = 1/n - 5\tilde{y}_{n-1} = 1/n + 5\epsilon_{n-1} = y_n + 5\epsilon_{n-1}$$

so $\epsilon_n \geq 5\epsilon_{n-1}$: exponential growth.

Stability of linear system solving

Problem: solve $Ax = b$, where b inexact.

$$A(x + \Delta x) = b + \Delta b.$$

Since $Ax = b$, we get $A\Delta x = \Delta b$. From this,

$$\left\{ \begin{array}{l} Ax = b \\ \Delta x = A^{-1} \Delta b \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} \|A\| \|x\| \geq \|b\| \\ \|\Delta x\| \leq \|A^{-1}\| \|\Delta b\| \end{array} \right.$$
$$\Rightarrow \frac{\|\Delta x\|}{\|x\|} \leq \|A\| \|A^{-1}\| \frac{\|\Delta b\|}{\|b\|}$$

‘Condition number’. Attainable accuracy depends on matrix properties

Consequences of roundoff

Multiplication and addition are not associative:
problems for parallel computations.

Operations with “same” outcomes are not equally stable:
matrix inversion is unstable, elimination is stable

Exercise 7: Fixed-point iteration

Consider the iteration

$$x_{n+1} = f(x_n) = \begin{cases} 2x_n & \text{if } 2x_n < 1 \\ 2x_n - 1 & \text{if } 2x_n \geq 1 \end{cases}$$

Does this function have a fixed point, $x_0 \equiv f(x_0)$, or is there a cycle $x_1 = f(x_0)$, $x_0 \equiv x_2 = f(x_1)$ et cetera?

Now code this function and see what happens with various starting points x_0 . Can you explain this?

Complex numbers

Two real numbers: real and imaginary part.

Storage:

- Store real/imaginary adjacent: easy to pass address of one number
- Store array of real, then array of imaginary. Better for stride 1 access if only real parts are needed. Other considerations.

First we dig into bits
Integers
Floating point numbers
Floating point math
Examples
More

Other arithmetic systems

Some compilers support higher precisions.

Arbitrary precision: GMPlib

Interval arithmetic