Elements of Floating-point Arithmetic

Sanzheng Qiao

Department of Computing and Software
McMaster University

July, 2012

Outline

Representing floating-point numbers

On paper we write a floating-point number in the format:

$$\pm d_1.d_2 \cdots d_t \times \beta^e$$

$0 < d_1 < \beta$, $0 \leq d_i < \beta$ $(i > 1)$

- t: precision
- β: base (or radix), almost universally 2, other commonly used bases are 10 and 16
- e: exponent, integer

Examples

1.00×10^{-1}

$t = 3$ (trailing zeros count), $\beta = 10$, $e = -1$

1.234×10^{2}

$t = 4$, $\beta = 10$, $e = 2$

1.10011×2^{-4}

$t = 6$, $\beta = 2$ (binary), $e = -4$

Characteristics

A floating-point number system is characterized by four (integer) parameters:

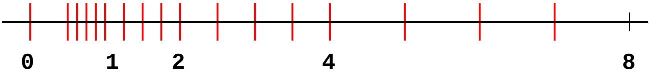
- base β (also called radix)
- precision t
- exponent range $e_{\min} \leq e \leq e_{\max}$

Some properties

A floating-point number system is

- discrete (not continuous)
- not equally spaced throughout
- finite

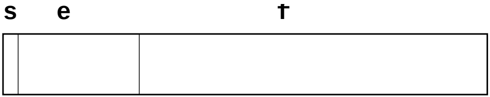
Example. The 33 points in a small system: $\beta = 2$, $t = 3$, $e_{\min} = -1$, and $e_{\max} = 2$. (Negative part not shown.)



In general, how many numbers in a system: β, t, $e_{\min}$, $e_{\max}$?

Storage

In memory, a floating-point number is stored in three consecutive fields:



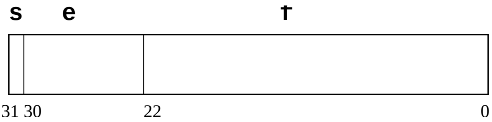
sign (1 bit)
exponent (depends on the range)
fraction (depends on the precision)

Standards

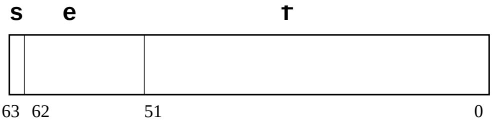
In order for a memory representation to be useful, there must be a standard.

IEEE floating-point standards:

single precision



double precision



Machine precision

A real number representing the accuracy.

Machine precision

Denoted by ϵ_M, defined as the distance between 1.0 and the next larger floating-point number, which is 0.0...01 $\times \beta^0$.

Thus, $\epsilon_M = \beta^{1-t}$.

Equivalently, the distance between two consecutive floating-point numbers between 1.0 and β. (The floating-point numbers between 1.0($= \beta^0$) and β are equally spaced: 1.0...000, 1.0...001, 1.0...010, ..., 1.1...111.)

Machine precision (cont.)

How would you compute the underlying machine precision?

The smallest ϵ such that $1.0 + \epsilon > 1.0$.

For $\beta = 2$:

```
eps = 1.0;
while (1.0 + eps > 1.0)
    eps = eps/2;
end
2*eps,
```

Examples. ($\beta = 2$)

When $t = 24$, $\epsilon_M = 2^{-23} \approx 1.2 \times 10^{-7}$
When $t = 53$, $\epsilon_M = 2^{-52} \approx 2.2 \times 10^{-16}$

Approximations of real numbers

Since floating-point numbers are discrete, a real number, for example, $\sqrt{2}$, may not be representable in floating-point. Thus real numbers are approximated by floating-point numbers.

We denote

$$\mathrm{fl}(x) \approx x.$$

as a floating-point approximation of a real number x.

Approximations of real numbers (cont.)

Example

The floating-point number $1.10011001100110011001101 \times 2^{-4}$ can be used to approximate 1.0×10^{-1}. The best single precision approximation of decimal 0.1.

1.0×10^{-1} is not representable in binary. (Try to convert decimal 0.1 into binary.)

When approximating, some kind of rounding is involved.

Error measurements: ulp and u

If the nearest rounding is applied and $\mathrm{fl}(x) = d_1.d_2...d_t \times \beta^e$, then the absolute error is bounded by

$$|\mathrm{fl}(x) - x| \leq \frac{1}{2}\beta^{1-t}\beta^e,$$

half of the unit in the last place (ulp);

the relative error is bounded by

$$\frac{|\mathrm{fl}(x) - x|}{|\mathrm{fl}(x)|} \leq \frac{1}{2}\beta^{1-t}, \text{ since } |\mathrm{fl}(x)| \geq 1.0 \times \beta^e,$$

called the *unit of roundoff* denoted by u.

Unit of roundoff u

When $\beta = 2$, $u = 2^{-t}$.
How would you compute u?

The largest number such that $1.0 + u = 1.0$.
Also, when $\beta = 2$, the distance between two consecutive floating-point numbers between $1/2 (= \beta^{-1})$ and $1.0 (= \beta^{0})$
$(1.0...0 \times 2^{-1}, ..., 1.1...1 \times 2^{-1}, 1.0.)$
$1.0 + 2^{-t} = 1.0$ (Why?)

```
u = 1.0;
while (1.0 + u > 1.0)
    u = u/2;
end
u,
```

Four parameters

Base $\beta = 2$.

	single	double
precision t	24	53
e_{min}	-126	-1022
e_{max}	127	1023

Formats:

	single	double
Exponent width	8 bits	11 bits
Format width in bits	32 bits	64 bits

$x \neq y \Rightarrow 1/x \neq 1/y$?

How many single precision floating-point numbers in $[1, 2)$?

$$1.00...00 \rightarrow 1.11...11$$

2^{23}, evenly spaced.

How many single precision floating-point numbers in $(1/2, 1]$?

$$1.00...01 \times 2^{-1} \rightarrow 1.00...00$$

2^{23}, evenly spaced.

$x \neq y \Rightarrow 1/x \neq 1/y$? (cont.)

How many single precision floating-point numbers in $[3/2, 2)$?

$(1/2) \times 2^{23}$

How many single precision floating-point numbers in $(1/2, 2/3]$?

$(1/3) \times 2^{23}$.

Since $(1/2) \times 2^{23} > (1/3) \times 2^{23}$, there exist $x \neq y \in [3/2, 2)$ such that $1/x = 1/y \in (1/2, 2/3]$.

Hidden bit and biased representation

Since the base is 2 (binary), the integer bit is always 1. This bit is not stored and called *hidden bit*.

The exponent is stored using the biased representation. In single precision, the bias is 127. In double precision, the bias is 1023.

Example

Single precision $1.10011001100110011001101 \times 2^{-4}$ is stored as

```
0 01111011 10011001100110011001101
```

Special quantities

The special quantities are encoded with exponents of either $e_{\max} + 1$ or $e_{\min} - 1$. In single precision, `11111111` in the exponent field encodes $e_{\max} + 1$ and `00000000` in the exponent field encodes $e_{\min} - 1$.

Signed zeros

Signed zeros: ± 0

Binary representation:

`X 00000000 00000000000000000000000`

- When testing for equal, $+0 = -0$, so the simple test `if (x == 0)` is predictable whether x is $+0$ or -0.
- The relation $1/(1/x) = x$ holds when $x = \pm\infty$.
- $\log(+0) = -\infty$ and $\log(-0) = NaN$; $\mathrm{sign}(+0) = 1$ and $\mathrm{sign}(-0) = -1$.

Signed zeros

If $z = -1$, $\sqrt{1/z} = i$, but $1/\sqrt{z} = -i$.
$\sqrt{1/z} \neq 1/\sqrt{z}$!
Why? Square root is multivalued, can't make it continuous in the entire complex plane. However, it is continuous for $z = \cos\theta + i\sin\theta$, $-\pi \leq \theta \leq \pi$, if a branch cut consisting of all negative real numbers is excluded from the consideration.
With signed zeros, for the numbers with negative real part, $-x + i(+0)$, $x > 0$, has a square root of $i\sqrt{x}$; $-x + i(-0)$ has a square root of $-i\sqrt{x}$.

$z = -1 = -1 + i(+0)$, $1/z = -1 + i(-0)$, then
$\sqrt{1/z} = -i = 1/\sqrt{z}$

However, $+0 = -0$, and $1/(+0) \neq 1/(-0)$. (Shortcoming)

Infinities

Infinities: $\pm\infty$

Binary Representation:

`X 11111111 00000000000000000000000`

- Provide a way to continue when exponent gets too large, $x^2 = \infty$, when x^2 overflows.
- When $c \neq 0$, $c/0 = \pm\infty$.
- Avoid special case checking, $1/(x + 1/x)$, a better formula for $x/(x^2 + 1)$, with infinities, there is no need for checking the special case $x = 0$.

NaN

NaNs (not a number)

Binary representation:

`X 11111111 nonzero fraction`

Provide a way to continue in situations like

Operation	NaN Produced By
+	$\infty + (-\infty)$
$*$	$0 * \infty$
/	0/0, ∞/∞
REM	x REM 0, ∞ REM y
sqrt	sqrt(x) when $x < 0$

Example for NaN

The function `zero(f)` returns a zero of a given quadratic polynomial `f`.

If

$$f = x^2 + x + 1,$$

$d = 1 - 4 < 0$, thus $\sqrt{d} = NaN$ and

$$\frac{-b \pm \sqrt{d}}{2a} = NaN,$$

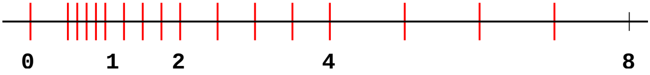
no zeros.

Denormalized numbers

Denormalized Numbers

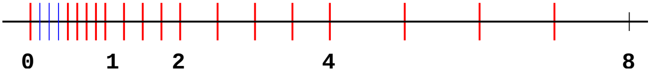
The small system: $\beta = 2$, $t = 3$, $e_{\min} = -1$, $e_{\max} = 2$

Without denormalized numbers (negative part not shown)



With (six) denormalized numbers (negative part not shown)

$$0.01 \times 2^{-1},\ 0.10 \times 2^{-1},\ 0.11 \times 2^{-1}$$



Denormalized numbers

Binary representation:

`X 00000000 nonzero fraction`

When $e = e_{\min} - 1$ and the bits in the fraction are $b_2, b_3, ..., b_t$, the number being represented is $0.b_2 b_3 ... b_t \times 2^{e+1}$ (no hidden bit)

- Guarantee the relation: $x = y \Longleftrightarrow x - y = 0$
- Allow gradual underflow. Without denormals, the spacing abruptly changes from $\beta^{-t+1}\beta^{e_{\min}}$ to $\beta^{e_{\min}}$, which is a factor of β^{t-1}.

Example for denormalized numbers

Complex division

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}.$$

Underflows when a, b, c, and d are small.

Example for denormalized numbers

Smith's formula

$$\frac{a+b(d/c)}{c+d(d/c)} + i\,\frac{b-a(d/c)}{c+d(d/c)} \qquad \textit{if } |d| < |c|$$

$$\frac{b+a(c/d)}{d+c(c/d)} + i\,\frac{-a+b(c/d)}{d+c(c/d)} \qquad \textit{if } |d| \geq |c|$$

For $a = 2\beta^{e_{\min}}$, $b = \beta^{e_{\min}}$, $c = 4\beta^{e_{\min}}$, and $d = 2\beta^{e_{\min}}$, the result is 0.5 with denormals ($a + b(d/c) = 2.5\beta^{e_{\min}}$) or 0.4 without denormals ($a + b(d/c) = 2\beta^{e_{\min}}$).

It is typical for denormalized numbers to guarantee error bounds for arguments all the way down to $\beta^{e_{\min}}$.

IEEE floating-point representations

Exponent	Fraction	Represents
$e = e_{\text{min}} - 1$	$f = 0$	± 0
$e = e_{\text{min}} - 1$	$f \neq 0$	$0.f \times 2^{e_{\text{min}}}$
$e_{\text{min}} \leq e \leq e_{\text{max}}$		$1.f \times 2^{e}$
$e = e_{\text{max}} + 1$	$f = 0$	$\pm\infty$
$e = e_{\text{max}} + 1$	$f \neq 0$	NaN

Examples (IEEE single precision)

- 1 10000001 11100000000000000000000
  represents: $-1.111_2 \times 2^{129-127} = -7.5_{10}$
- 0 00000000 11000000000000000000000
  represents: $0.11_2 \times 2^{-126}$
- 0 11111111 00100000000000000000000
  represents: NaN
- 1 11111111 00000000000000000000000
  represents: $-\infty$.

Underflow

An arithmetic operation produces a number with an exponent that is too small to be represented in the system.

Example.
In single precision,

$$a = 3.0 \times 10^{-30},$$

$a * a$ underflows.
By default, it is set to zero.

Overflow

An arithmetic operation produces a number with an exponent that is too large to be represented in the system.

Example.
In single precision,

$$a = 3.0 \times 10^{30},$$

$a * a$ overflows.
In IEEE standard, the default result is ∞.

Avoiding unnecessary underflow and overflow

Sometimes, underflow and overflow can be avoided by using a technique called scaling.

Given $x = (a, b)^T$, $a = 1.0 \times 10^{30}$, $b = 1.0$, compute $c = \|x\|_2 = \sqrt{a^2 + b^2}$.

scaling: $s = \max\{|a|, |b|\} = 1.0 \times 10^{30}$

$a \leftarrow a/s$ (1.0),
$b \leftarrow b/s$ (1.0×10^{-30})
$t = \sqrt{a * a + b * b}$ (1.0)
$c \leftarrow t * s$ (1.0×10^{30})

Example: Computing 2-norm of a vector

Compute

$$\sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$$

Efficient and robust:

Avoid multiple loops:
searching for the largest; Scaling; Summing.

Result: One single loop

Technique: Dynamic scaling

Example: Computing 2-norm of a vector

```
scale = 0.0;
ssq = 1.0;
for i=1 to n
   if (x(i) != 0.0)
      if (scale<abs(x(i))
         tmp = scale/x(i);
         ssq = 1.0 + ssq*tmp*tmp;
         scale = abs(x(i));
      else
         tmp = x(i)/scale;
         ssq = ssq + tmp*tmp;
      end
   end
end
nrm2 = scale*sqrt(ssq);
```

Correctly rounded operations

Correctly rounded means that the result of the floating-point operation must be the same as if it were computed exactly and then rounded, usually to the nearest floating-point number.

For example, if $\oplus$ denotes the floating-point addition, then given two
floating-point numbers a and b,

$$a \oplus b = \mathrm{fl}(a + b).$$

Correctly rounded operations

Examples

$\beta = 10$, $t = 4$

$a = 1.234 \times 10^0$ and $b = 5.678 \times 10^{-3}$
Exact: $a + b = 1.239678$ Floating-point: $\mathrm{fl}(a + b) = 1.240 \times 10^0$

$a = 4.563 \times 10^{-3}$ and $b = 5.678 \times 10^{-3}$
Exact: $a + b = 10.241 \times 10^{-3}$ Floating-point: $\mathrm{fl}(a + b) = 1.024 \times 10^{-2}$

$a = 1.234 \times 10^0$ and $b = -1.221 \times 10^0$
Exact: $a + b = 0.012$ Floating-point: $\mathrm{fl}(a + b) = 1.200 \times 10^{-2}$ (exact)

Correctly rounded operations

IEEE standards require the following operations are correctly rounded:

- arithmetic operations $+$, $-$, $*$, and $/$
- square root and remainder
- conversions of formats (binary, decimal)

Rounding error

Due to finite precision arithmetic, a computed result must be rounded to fit storage format.

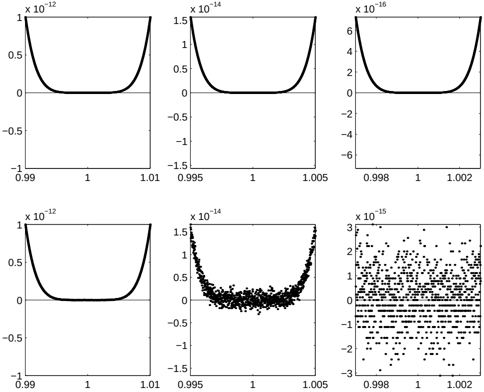
Example
$\beta = 10$, $t = 4$ ($u = 0.5 \times 10^{-3}$)
$a = 1.234 \times 10^0$, $b = 5.678 \times 10^{-3}$
$x = a + b = 1.239678 \times 10^0$ (exact)
$\hat{x} = \mathrm{fl}(a + b) = 1.240 \times 10^0$
the result was rounded to the nearest computer number.

Rounding error: $\mathrm{fl}(a + b) = (a + b)(1 + \epsilon)$, $|\epsilon| \leq u$.

$1.240 = 1.239678(1 + 2.59... \times 10^{-4})$, $|2.59... \times 10^{-4}| < u$

Effect of rounding errors

Top: $y = (x-1)^6$
Bottom: $y = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1$



Two ways of evaluating the polynomial $(x-1)^6$

Real to floating-point

```
double x = 0.1;
```

What is the value of x stored?

$$1.0 \times 10^{-1} = 1.100110011001100110011... \times 2^{-4}$$

Decimal 0.1 cannot be exactly represented in binary. It must be rounded to

$$\begin{aligned} & 1.10011001100...1100110\textcolor{red}{10} \times 2^{-4} \\ > \quad & 1.10011001100...110011001100 11... \end{aligned}$$

slightly larger than 0.1.

Real to floating-point

```
double x, y, h;
x = 0.0;
h = 0.1;

for i=1 to 10
   x = x + h;
end
y = 1.0 - x;
```

$$y > 0 \quad \text{or} \quad y < 0 \quad \text{or} \quad y = 0?$$

Answer: $y \approx 1.1 \times 10^{-16} > 0$

Real to floating-point (cont.)

Why?

$$i = 1$$
$$x = h = 1.100...110011010 \times 2^{-4} > 1.0 \times 10^{-1}$$
$$i = 2$$
$$x = 1.100...110011010 \times 2^{-3} > 2.0 \times 10^{-1}$$
$$i = 3$$
$$x = 1.001...100110011\textcolor{red}{10} \times 2^{-2}$$
$$\rightarrow 1.001...10100 \times 2^{-2} > 3.0 \times 10^{-1}$$
$$i = 4$$
$$x = 1.100...110011010\textcolor{red}{10} \times 2^{-2}$$
$$\rightarrow 1.100...110011010 \times 2^{-2} > 4.0 \times 10^{-1}$$

Real to floating-point (cont.)

$$
\begin{aligned}
&i = 5\\
&x = 1.000...0000{\color{red}010} \times 2^{-1}\\
&\rightarrow 1.000...0000 \times 2^{-1} = 5.0 \times 10^{-1}\\
&i = 6\\
&x = 1.001...100110011{\color{red}010} \times 2^{-1}\\
&\rightarrow 1.001...100110011 \times 2^{-1} < 6.0 \times 10^{-1}\\
&\vdots
\end{aligned}
$$

Rounding errors in floating-point addition.

Integer to floating-point

Fallacy

Java converts an integer into its mathematically equivalent floating-point number.

```
long k = 18014398509481985;
long d = k - (long)((double) k);
```

Note $18014398509481985 = 2^{54} + 1$

$d = 0$?

No, d = 1!

Integer to floating-point

Why?

$$
\begin{aligned}
k &= 1.00\ldots00\textcolor{red}{01} \times 2^{54} \\
(\texttt{double})\, k &= 1.00\ldots00 \times 2^{54}
\end{aligned}
$$

Truncation error

When an infinite series is approximated by a finite sum, truncation error is introduced.
Example. If we use

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

to approximate

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

then the truncation error is

$$\frac{x^{n+1}}{(n+1)!} + \frac{x^{n+2}}{(n+2)!} + \cdots .$$

Discretization error

When a continuous problem is approximated by a discrete one, discretization error is introduced.
Example. From the expansion

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(\xi),$$

for some $\xi \in [x, x+h]$, we can use the following approximation:

$$y_h(x) = \frac{f(x+h) - f(x)}{h} \approx f'(x).$$

The discretization error is $E_{\text{dis}} = |f''(\xi)|h/2$.

Example

Let $f(x) = e^x$, compute $y_h(1)$.
The discretization error is

$$E_{\text{dis}} = \frac{h}{2}|f''(\xi)| \leq \frac{h}{2}e^{1+h} \approx \frac{h}{2}e \quad \text{for small} \quad h.$$

The computed $y_h(1)$:

$$\widehat{y}_h(1) = \frac{(e^{(1+h)(1+\epsilon_1)}(1+\epsilon_2) - e(1+\epsilon_3))(1+\epsilon_4)}{h}(1+\epsilon_5),$$

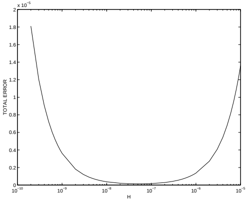
$|\epsilon_i| \leq u$.
The rounding error is

$$E_{\text{round}} = \widehat{y}_h(1) - y_h(1) \approx \frac{7u}{h}e.$$

Example (cont.)

The total error:

$$E_{\text{total}} = E_{\text{dis}} + E_{\text{round}} \approx \left(\frac{h}{2} + \frac{7u}{h} \right) e.$$



Total error in the computed $y_h(1)$.

The optimal h: $h_{\text{opt}} = \sqrt{14u} \approx \sqrt{u}$.

Backward errors

Recall that

$$a \oplus b = \mathrm{fl}(a + b) = (a + b)(1 + \eta), \quad |\eta| \leq u$$

In other words,

$$a \oplus b = \tilde{a} + \tilde{b}$$

where $\tilde{a} = a(1 + \eta)$ and $\tilde{b} = b(1 + \eta)$, for $|\eta| \leq u$, are slightly different from a and b respectively.

The computed sum (result) is the exact sum of slightly different a and b (inputs).

Example

$\beta = 10$, $p = 4$ ($u = 0.5 \times 10^{-3}$)
$a = 1.234 \times 10^0$, $b = 5.678 \times 10^{-3}$
$a \oplus b = 1.240 \times 10^0$, $a + b = 1.239678$

$1.240 = 1.239678(1 + 2.59... \times 10^{-4})$, $|2.59... \times 10^{-4}| < u$

$1.240 = a(1 + 2.59... \times 10^{-4}) + b(1 + 2.59... \times 10^{-4})$

The computed sum (result) is the exact sum of slightly different *a* and *b* (inputs).

Backward errors (cont.)

A general example

$$s_n = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

The computed result ($x_1 \oplus \cdots \oplus x_n$) is the exact result of the problem with slightly perturbed data. ($x_1(1+\eta_1), ..., x_n(1+\eta_n)$).
Backward errors:
$|\eta_1| \leq 1.06(n-1)u$
$|\eta_i| \leq 1.06(n-i+1)u,\ i = 2, 3, ..., n$
If the backward errors are small, then we say that the algorithm is backward stable.

Example

Example: $a + b$

$a = 1.23$, $b = 0.45$, $s = a + b = 1.68$

Slightly perturbed
$\widehat{a} = a(1 + 0.01)$, $\widehat{b} = b(1 + 0.001)$, $\widehat{s} = \widehat{a} + \widehat{b} = 1.69275$

Relative perturbations in data (a and b) are at most 0.01.
Causing a relative change in the result $|\widehat{s} - s|/|s| \approx 0.0076$, which is about the same as the perturbation 0.01

The result is insensitive to the perturbation in data.

Example (cont.)

$a = 1.23$, $b = -1.21$, $s = a + b = 0.02$

Slightly perturbed
$\widehat{a} = a(1 + 0.01)$, $\widehat{b} = b(1 + 0.001)$, $\widehat{s} = \widehat{a} + \widehat{b} = 0.03109$

Relative perturbations in data (a and b) are at most 0.01.
Causing a relative change in the result $|\widehat{s} - s|/|s| \approx 0.5545$, which is more than 55 times as the perturbation 0.01

The result is sensitive to the perturbation in the data.

Perturbation analysis

Example: $a + b$

$$\begin{aligned} & \frac{|a(1+\delta_a) + b(1+\delta_b) - (a+b)|}{|a+b|} \\ \leq \quad & \frac{|a| + |b|}{|a+b|}\,\delta, \quad \delta = \max(\delta_a, \delta_b). \end{aligned}$$

Condition number: $(|a| + |b|)/|a+b|$, magnification of the relative error.

$$\frac{\text{relative error in result}}{\text{relative error in data}} \leq \text{cond}$$

Condition number is a measurement (an upper bound) of the sensitivity of the problem to changes in data.

Example

Two methods for calculating $z(x+y)$:

$$z \otimes x \oplus z \otimes y \quad \text{and} \quad z \otimes (x \oplus y)$$

$\beta = 10$, $t = 4$
$x = 1.002$, $y = -0.9958$, $z = 3.456$
Exact $z(x+y) = 2.14272 \times 10^{-2}$
$z \otimes (x \oplus y) = \mathrm{fl}(3.456 * 6.200 \times 10^{-3})$
$= 2.143 \times 10^{-2}$
error: 2.8×10^{-6}
$(z \otimes x) \oplus (z \otimes y)) = \mathrm{fl}(3.463 - 3.441)$
$= 2.200 \times 10^{-2}$
error: 5.7×10^{-4}
More than 200 times!

Example (cont.)

Backward error analyses

$$
\begin{aligned}
& z \otimes x \oplus z \otimes y \\
= \; & (zx(1+\epsilon_1) + zy(1+\epsilon_2))(1+\epsilon_3) \\
= \; & z(1+\epsilon_3)(x(1+\epsilon_1) + y(1+\epsilon_2)), \; |\epsilon_i| \leq u
\end{aligned}
$$

$$
\begin{aligned}
& z \otimes (x \oplus y) \\
= \; & z((x+y)(1+\epsilon_1))(1+\epsilon_3) \\
= \; & z(1+\epsilon_3)(x(1+\epsilon_1) + y(1+\epsilon_1)), \; |\epsilon_i| \leq u
\end{aligned}
$$

Both methods are backward stable.

Example (cont.)

Perturbation analysis

$$
\begin{aligned}
& z(1+\delta_z)(x(1+\delta_x)+y(1+\delta_y)) \\
\approx\ & zx(1+\delta_z+\delta_x)+zy(1+\delta_z+\delta_y) \\
=\ & z(x+y)+zx(\delta_z+\delta_x)+zy(\delta_z+\delta_y) \\
=\ & z(x+y)(1+(\delta_z+\delta_x)+(\delta_y-\delta_x)/(x/y+1))
\end{aligned}
$$

$$
\frac{|z(1+\delta_z)(x(1+\delta_x)+y(1+\delta_y))-z(x+y)|}{|z(x+y)|}
\leq \left(2+\frac{2}{|\frac{x}{y}+1|}\right)\delta, \quad \delta=\max(|\delta_x|,|\delta_y|,|\delta_z|)
$$

The condition number can be large if $y \approx -x$ and $\delta_x \neq \delta_y$.

Example (cont.)

Forward error analysis

$$
\begin{aligned}
& z \otimes x \oplus z \otimes y \\
= \quad & z(1+\epsilon_3)(x(1+\epsilon_1)+y(1+\epsilon_2)) \\
\approx \quad & z(x+y)(1+(\epsilon_3+\epsilon_1)+(\epsilon_2-\epsilon_1)/(x/y+1)), \quad |\epsilon_i| \leq u
\end{aligned}
$$

$$
\frac{|(z \otimes x \oplus z \otimes y)-z(x+y)|}{|z(x+y)|} \leq \left(2+\frac{2}{|\frac{x}{y}+1|}\right) u
$$

Example (cont.)

Forward error analysis (cont.)

$$z \otimes (x \oplus y) \approx z(x + y)(1 + \epsilon_1 + \epsilon_3), \quad |\epsilon_i| \leq u$$

$$\frac{|z \otimes (x \oplus y) - z(x + y)|}{|z(x + y)|} \leq 2u$$

Remarks

$$\text{forward error} \leq \text{cond} \cdot \text{backward error}$$

- If we can prove the algorithm is stable, in other words, the backward errors are small, say, no larger than the measurement errors in data, then we know that large forward errors are due to the ill-conditioning of the problem.
- If we know the problem is well-conditioned, then large forward errors must be caused by unstable algorithm.
- Condition number is an upper bound. It is possible that a well-designed stable algorithm can produce good results even the problem is ill-conditioned.

Example of stability/sensitivity

Compute the integrals

$$E_n = \int_0^1 x^n e^{x-1} dx, \qquad n = 1, 2, \ldots.$$

Using integration by parts,

$$\int_0^1 x^n e^{x-1} dx = x^n e^{x-1}|_0^1 - \int_0^1 n x^{n-1} e^{x-1} dx,$$

or

$$E_n = 1 - nE_{n-1}, \qquad n = 2, \ldots,$$

where $E_1 = 1/e$.

Example of stability/sensitivity

$$E_n = 1 - nE_{n-1}$$

Double precision

$$\begin{array}{lll} E_1 \approx 0.3679 & E_7 \approx 0.1124 & E_{13} \approx 0.0669 \\ E_2 \approx 0.2642 & E_8 \approx 0.1009 & E_{14} \approx 0.0627 \\ E_3 \approx 0.2073 & E_9 \approx 0.0916 & E_{15} \approx 0.0590 \\ E_4 \approx 0.1709 & E_{10} \approx 0.0839 & E_{16} \approx 0.0555 \\ E_5 \approx 0.1455 & E_{11} \approx 0.0774 & E_{17} \approx 0.0572 \\ E_6 \approx 0.1268 & E_{12} \approx 0.0718 & E_{18} \approx -0.0295 \end{array}$$

Apparently, $E_{18} > 0$.

Unstable algorithm or ill-conditioned problem?

Example of stability/sensitivity

$E_n = 1 - nE_{n-1}$

Perturbation analysis. Suppose that we perturb E_1:

$$\tilde{E}_1 = E_1 + \epsilon,$$

then $\tilde{E}_2 = 1 - 2\tilde{E}_1 = E_2 - 2\epsilon$.
In general,

$$\tilde{E}_n = E_n - (-1)^{n+1} n!\,\epsilon.$$

Thus this problem is ill-conditioned.

We can show that this algorithm is backward stable.

Example of stability/sensitivity

$E_{n-1} = (1 - E_n)/n$
Note that E_n goes to zero as n goes to ∞.
Start with $E_{40} = 0.0$

$E_{35} \approx 0.0270$	$E_{29} \approx 0.0323$	$E_{23} \approx 0.0401$
$E_{34} \approx 0.0278$	$E_{28} \approx 0.0334$	$E_{22} \approx 0.0417$
$E_{33} \approx 0.0286$	$E_{27} \approx 0.0345$	$E_{21} \approx 0.0436$
$E_{32} \approx 0.0294$	$E_{26} \approx 0.0358$	$E_{20} \approx 0.0455$
$E_{31} \approx 0.0303$	$E_{25} \approx 0.0371$	$E_{19} \approx 0.0477$
$E_{30} \approx 0.0313$	$E_{24} \approx 0.0385$	$E_{18} \approx 0.0501$

Example of stability/sensitivity

$E_{n-1} = (1 - E_n)/n$

Perturbation analysis. Suppose that we perturb E_n:

$$\tilde{E}_n = E_n + \epsilon,$$

then $\tilde{E}_{n-1} = E_{n-1} - \epsilon/n$.

In general,

$$\tilde{E}_k = E_k + \epsilon_k, \quad |\epsilon_k| = \frac{\epsilon}{n(n-1)...(k+1)}.$$

Thus this problem is well-conditioned.

Note that we view these two methods as two different problems, since they have different inputs and outputs.

Example revisited

$\beta = 10$, $t = 4$

$x = 1.002$, $y = -0.9958$, $z = 3.456$

Exact $z(x+y) = 2.14272 \times 10^{-2}$

$z \otimes (x \oplus y) = \mathrm{fl}(3.456 * 6.200 \times 10^{-3})$

$= 2.143 \times 10^{-2}$

error: 2.8×10^{-6}

$(z \otimes x) \oplus (z \otimes y)) = \mathrm{fl}(3.463 - 3.441)$

$= 2.200 \times 10^{-2}$

error: 5.7×10^{-4}

More than 200 times!

Why?

Example revisited

$(z \otimes x) \oplus (z \otimes y)) = \mathrm{fl}(3.463 - 3.441)$
$= 2.200 \times 10^{-2}$
error: 5.7×10^{-4}

Cancellation in subtracting two computed (contaminated) numbers. (Catastrophic)

$z \otimes (x \oplus y) = \mathrm{fl}(3.456 * 6.200 \times 10^{-3})$
$= 2.143 \times 10^{-2}$
error: 2.8×10^{-6}

Cancellation in subtracting two original (not contaminated) numbers. (Benign)

Catastrophic cancellation v.s. benign cancellation.

Example myexp

Using the Taylor series

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

we write a function myexp:

```
oldy = 0.0;
y = 1.0;
term = 1.0;
k = 1;
while (oldy ~= y)
    term = term*(x/k);
    oldy = y;
    y = y + term;
    k = k + 1;
end
```

Example `myexp`

About the stopping criterion

- When x is negative, the terms have alternating signs, then it is guaranteed that the truncation error is smaller then the last term in the program.
- When x is positive, all the terms are positive, then it is not guaranteed that the truncation error is smaller than the last term in the program. For example, when $x = 678.9$, the last two digits of the computed result are inaccurate.

Example `myexp`

When $x = -7.8$

k	sum	term
1	1.00000000000000E+0	−7.80000000000000E+0
$\vdots$		
10	−1.322784174621715E+2	2.29711635560962E+2
11	9.743321809879086E+1	−1.62886432488682E+2
12	−6.545321438989151E+1	1.05876181117643E+2
$\vdots$		

Example `myexp`

k	sum	term
26	1.092489579672046E$-$4	3.88007967049995E$-$04
$\vdots$		
49	4.097349789682480E$-$4	$-$8.48263272621995E$-$20
50	4.097349789682479E$-$4	1.32329070529031E$-$20

The MATLAB result:

$$4.09734978979786 8\mathrm{E}-4$$

Example myexp

An explanation

When $k = 10$, the absolute value of the intermediate sum reaches the maximum, about 10^{+2}, that is, the ulp is about 10^{-13}. After that, cancellation occurs, so the final result is about 10^{-4}. We expect the error in the final result is 10^{-13}, in other words, ten digit accuracy.

Cancellation magnifies the relative error.

An accurate method.

Break x into the integer part m and the fraction part f. Compute e^m using multiplications, then compute e^f when $-1 < f < 0$ or $1/e^{-f}$ when $0 < f < 1$.

Using this method, the computed $e^{-7.8}$ is

$$4.0973497897978\text{E}-4$$

A classic example of avoiding cancellation

Solving quadratic equation

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

Text book formula:

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

Computational method:

$$x_1 = \frac{2c}{-b - \mathrm{sign}(b)\sqrt{b^2 - 4ac}}, \quad x_2 = \frac{c}{ax_1}$$

Question

Suppose $\beta = 10$ and $t = 8$ (single precision), solve

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

where

$$a = 1, \quad b = -10^5, \quad \text{and} \quad c = 1,$$

using the both methods.

Fallacies

- Cancellation in the subtraction of two nearly equal numbers is always bad.
- The final computed answer from an algorithm cannot be more accurate than any of the intermediate quantities, that is, errors cannot cancel.
- Arithmetic much more precise than the data it operates upon is needless and wasteful.
- Classical formulas taught in school and found in handbooks and software must have passed the Test of Time, not merely withstood it.

Summary

- A computer number system is determined by four parameters: Base, precision, e_{min}, and e_{max}
- IEEE floating-point standards, single precision and double precision. Special quantities: Denormals, $\pm\infty$, NaN, ± 0, and their binary representations.
- Error measurements: Absolute and relative errors, unit of roundoff, unit in the last place (ulp)
- Sources of errors: Rounding error (computational error), truncation error (mathematical error), discretization error (mathematical error). Total error (combination of rounding error and mathematical errors)
- Issues in floating-point computation: Overflow, underflow, cancellations (benign and catastrophic)
- Error analysis: Forward and backward errors, sensitivity of a problem and stability of an algorithm

References

[1] George E. Forsyth and Michael A. Malcolm and Cleve B. Moler. Computer Methods for Mathematical Computations. Prentice-Hall, Inc., 1977.
Ch 2.

[2] David Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys, vol. 23, no. 1, 1991, 5–48.

[3] Nicholas J. Higham. Accuracy and Stability of Numerical Algorithms. Second Edition. SIAM. Philadelphia, PA, 2002.
Ch 1, Ch 2, Ch 27.