On Numerical Analysis (first week)

Hamad El Kahza

Floating point arithmetic, finite precision

1 Floating point arithmetic

In computers, numbers are stored as floating point quantities in the general form:

$$\pm \cdot (d_1 d_2 d_3 \dots d_p) \cdot \beta^e,$$

where

- $p =$ precision, the number of significant bits (digits)
- $e =$ an integer exponent ranging from E_{min} to E_{max}
- $\beta =$ the number base, normally 2, 10, 16,
- d_i : ranges from 0 to $\beta - 1$
- $d_1 d_2 d_3 \dots d_p$ is called the fractional part (mantissa).

Let us examine the case $\beta = 10$ (Decimal) which is the commonly used based in our daily calculations.

$$\begin{aligned} 3216 &= 3 \cdot 10^3 + 2 \cdot 10^2 + 1 \cdot 10^1 + 6 \cdot 10^0 \\ &= 10^4 (3 \cdot 10^{-1} + 2 \cdot 10^{-2} + 1 \cdot 10^{-3} + 6 \cdot 10^{-4}) \\ &= (.3216) \cdot 10^4 \end{aligned}$$

Computers, and measurements devices, don't have a conception of real numbers. Numbers are therefore conceived as flointing point number system. Most computers primarly use a binary system or $\beta = 2$. Let's examine this case.

$$\begin{aligned} 65 &= 2^6 + 2^0 \\ &= 2^7 (2^{-1}) + 2^{-7} \\ &= (.1000001)_2 \cdot 2^7 \end{aligned}$$

$$\begin{aligned} 23 &= 2^4 + 2^3 + 2^2 \\ &= 2^5 (2^{-1} + 2^{-2} + 2^{-3}) \\ &= (.111)_2 \cdot 2^5 \end{aligned}$$

$$\begin{aligned} 85 &= 2^6 + 2^4 + 2^2 + 2^0 \\ &= 2^7(2^{-1} + 2^{-3} + 2^{-6} + 2^{-7}) \\ &= (.1010011)_2 \cdot 2^7 \end{aligned}$$

$$\begin{aligned} 5.75 &= 2^2 + 2^0 + 2^{-1} + 2^{-2} \\ &= 2^3(2^{-1} + 2^{-3} + 2^{-4} + 2^{-5}) \\ &= (.10111)_2 \cdot 2^3 \end{aligned}$$

$$0.6 = (.1001100110011001\ldots)_2$$

MATLAB and most other computing environments usefloating-point arithmetic, which involves a finite set of numbers with finite precision. This leads to phenomena like roundoff error, underflow, and overflow.

1.1 roundoff error

The four basic operations of floating point numbers are define as below:

$$\begin{aligned} x \oplus y = fl(fl(x) + fl(y)) \\ x \ominus y = fl(fl(x) - fl(y)) \\ x \otimes y = fl(fl(x) \times fl(y)) \\ x \oslash y = fl(fl(x)/fl(y)) \end{aligned}$$

where the $fl(x)$ is the rounding-off of the actual real number x.

Round off error: The error that is produced when a computer is used to perform real-number calculations is called round-off error.

1.2 Rounding

We get error from rounding a number. Since computer can only store finite digits, so we have to round-off the number to certain precision in order to save it in computer. Suppose

$$x = d_0.d_1 d_2 ... d_{k-1} d_k ... \times \beta^n$$

There are two types of rounding methods. One is round-to-nearest: we denote $fl(x)$ as the k-digit round-to-nearest of real number x:

$$fl(x) = \begin{cases} d_0.d_1 d_2 ... d_{k-1} & \text{if } 0 \le d_k < \frac{\beta}{2} \\ d_0.d_1 d_2 ... (d_{k-1} + 1) & \text{if } \frac{\beta}{2} \le d_k < \beta \end{cases}$$

The other rounding method is round-by-chopping: we denote that $chop(x)$ as the k-digit round-by-chopping of real number x:

$$chop(x) = d_0.d_1 ... d_{k-1}$$

Chopping ignores the digits after d_{k-1}.

Therefore, we have covered two ways of truncating the mantissa:

1. Chopping
2. Rounding

Ex. $13.76573 = .1376|573 \cdot 10^2$

- **4 digits chopping**: $.1376 \cdot 10^2$
- **4 digits rounding**: $.1377 \cdot 10^2$

Numbers are **rounded** when stored in the floating point format.

There are ways to measure errors. Suppose we have the original number x and the number $\hat{x}$ which is the approximation of x. We can define **Absolute Error**:

$$|x - \hat{x}|$$

and we can also define **Relative Error**::

$$\frac{|x - \hat{x}|}{|x|}$$

The floating point form is obtained by terminating the mantissa using the following two ways:

1. Chopping
2. Rounding

Absolute Error $|true\ value - approximate\ value| = |p - p^*|$

Relative Error $|\frac{true\ value\ -\ approximate\ value}{true\ value}| = |\frac{p\ -\ p^*}{p}|$

$$(true\ value)\ p_1 = 1.2,\ (approximate\ value)\ p_1^* = 1$$
$$p_2 = 1000,\ p_2^* = 1000.2$$

$$\begin{aligned} absolute\ error &= |1 - 1.2| = 0.2 \\ &= |1000 - 1000.2| = 0.2 \end{aligned}$$

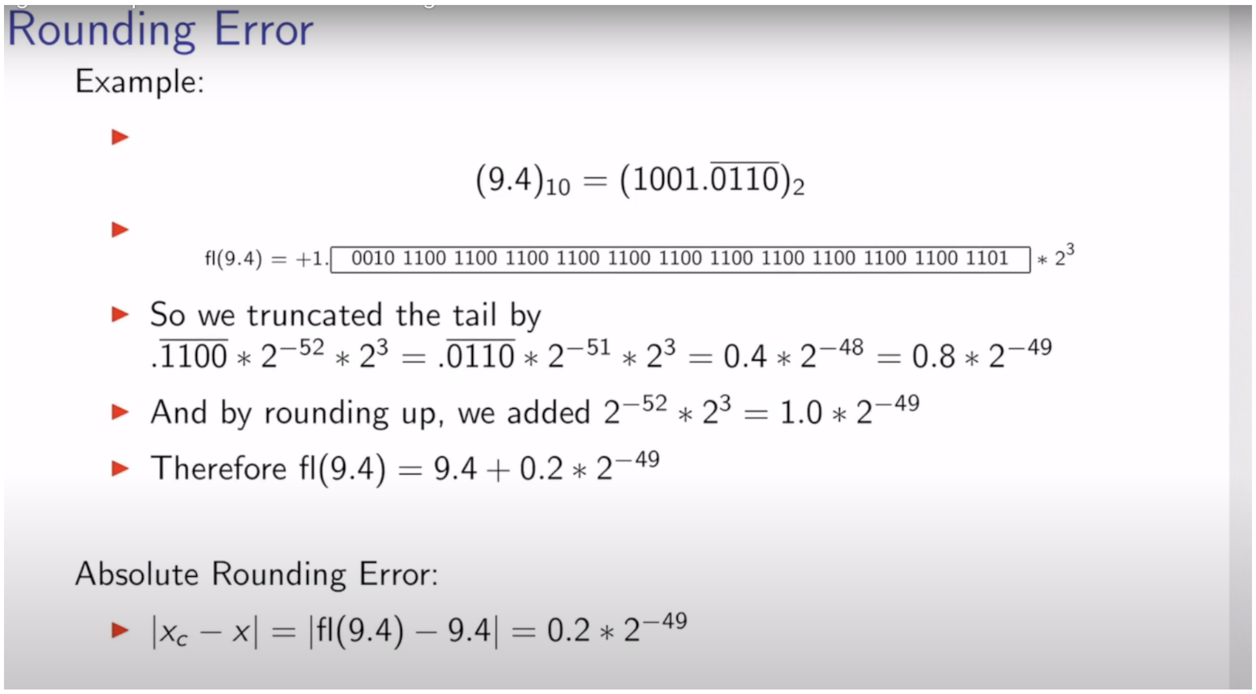
$$\begin{aligned} rel.\ error &= |\frac{1 - 1.2}{1}| = 0.2 \\ &= |\frac{1000 - 1000.2}{1000}| = 0.0002 \end{aligned}$$

$$find\ f(x) = x^3 + 61x^2 + 3.2x + 1.5\ at\ f(x = 4.71)$$

	x	x^2	x^3	$61x^2$	$3.2x$
Exact	4.7	22.1841	104.487111	1353.2301	15.072
3-digit chopping	4.7	22.1	104.0	1353	15.0
3-digit rounding	4.7	22.2	104	1353	15.1

$$Exact : -14.263899, Chopping : -13.5. Rounding : -13.4$$

$$Rel.error : Chopping : 0.0045, Rounding : 0.0025$$



Rounding Error

Example:

- $$(9.4)_{10} = (1001.\overline{0110})_2$$
- $$\mathrm{fl}(9.4) = +1.\boxed{0010\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1101} * 2^3$$
- So we truncated the tail by $.\overline{1100} * 2^{-52} * 2^3 = .\overline{0110} * 2^{-51} * 2^3 = 0.4 * 2^{-48} = 0.8 * 2^{-49}$
- And by rounding up, we added $2^{-52} * 2^3 = 1.0 * 2^{-49}$
- Therefore $\mathrm{fl}(9.4) = 9.4 + 0.2 * 2^{-49}$

Absolute Rounding Error:

- $|x_c - x| = |\mathrm{fl}(9.4) - 9.4| = 0.2 * 2^{-49}$

1.3 Machine Epsilon

We denote $\epsilon \equiv \beta^{1-k}$ and called ϵ the machine epsilon. Under IEEE double precision framework, $\beta = 2$ and $k = 53$, we have that $\epsilon = 2^{-52} \approx 2.2 * 10^{-16}$.

(a)
$$\left|\frac{x \cdot y - x \odot y}{x \cdot y}\right| \leq \epsilon$$
where the $\cdot$ denotes four basic operations: $+$, $-$, $\times$, $\div$.

(b) ϵ is the least positive number such that
$$1 + \epsilon \neq 1$$

However, in general, the relative error of other calculations under IEEE double precision framework will be at least 10^{-16}.

Example: Chip manufacturers are branching out from the traditional double-precision floating point (fp) format for high-performance computing, and are targeting the artificial intelligence and machine learning communities with processors that trade off precision for computational speed up.

Intel's Knights Mill and AMD's Radeon Vega Frontier processors can operate in a low-precision fp format consisting of 16 binary bits, which allocates 1 bit for the sign s, 5 bits for the exponent field e, and 10 bits for the fraction field f. Normalized floating point numbers are represented as $(-1)^s \cdot (1.f)_2 \cdot 2^{e-b}$, and all other representations are in analogy with the conventions of IEEE double precision arithmetic.

Bias b, and range of biased exponent $e - b$.

There are 5 bits for the biased exponent in low-precision fp format. This means that there are $2^5 = 32$ possible exponents. The maximum number that can be represented with 5 bits is $(11111)_2 = 2^5 - 1 = 31$ and the minimum number is $(00000)_2 = 0$. By IEEE convention, we reserve these two of these two exponents for special cases(overflow, underflow). This leaves us with $32 - 2 = 30$ possible exponents. We divide this number by 2 to get the bias, b.

$$\implies b = 15$$

$$\implies 1 \leq e \leq 30$$
$$\implies -14 \leq e - b \leq 15$$

Smallest normalized positive and largest fp numbers.

The smallest positive normalized fp number is of the form:

$$n = (-1)^0 \cdot (1.0000000000)_2 \cdot 2^{-14}$$
$$\implies n = 2^{-14}$$

The largest positive normalized fp number is of the form:

$$N = (-1)^0 \cdot (1.1111111111)_2 \cdot 2^{15}$$
$$\implies N = (1 + 2^{-1} + 2^{-2} + ... + 2^{-9} + 2^{-10}) \cdot 2^{15}$$

Machine epsilon (distance from 1 to next larger fp number).

Machine epsilon,ϵ, is the distance between 1 and the next larger fp number.

$$\epsilon = (-1)^0 \cdot (1.0000000000)_2 \cdot 2^0 - (-1)^0 \cdot (1.0000000001)_2 \cdot 2^0$$
$$\implies \epsilon = (0.0000000001)_2 = 2^{-10}$$

Smallest and largest positive denormalized fp numbers.

The smallest positive denormalized fp number is of the form:

$$n_d = (-1)^0 \cdot (0.0000000001)_2 \cdot 2^{-14}$$
$$\implies n_d = 2^{-10} \cdot 2^{-14} = 2^{-14}$$

The largest positive denormalized fp number is of the form:

$$N_d = (-1)^0 \cdot (0.1111111111)_2 \cdot 2^{-14}$$
$$\implies N_d = (2^{-1} + 2^{-2} + ... + 2^{-9} + 2^{-10}) \cdot 2^{-14}$$

2 Numerical examples (roots)

Consider the quadratic equation $ax^2 + bx + c$, with $a \neq 0$, which has roots

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

For the following the values $a = 10^-11$, $c = 1$, $b =, 10^-12$.

Here the implementation of a MATLAB `myroots` to implement the formula described above.

```
function myroots(a,b,c)
posAns = (-b + sqrt(b^2 - 4 * a * c)) / (2 * a);
negAns = (-b - sqrt(b^2 - 4 * a * c)) / (2 * a);
disp("Estimated Results: ");
format long;
disp(negAns);
disp(posAns);

p = [a b c];
x = roots(p);
disp("Actual Results: ");
disp(x);
end

Output:
>> roudofferror(10^-11,10^-12,1)
Estimated Results:
       -5.000000000000000e-02 - 3.162277660168339e+05i

       -5.000000000000000e-02 + 3.162277660168339e+05i

Actual Results:
    1.0e+05 *

  -0.000000500000000 + 3.162277660168340i
  -0.000000500000000 - 3.162277660168340i
```

Another different way to compute the roots that accounts for the cancellation errors.

The problem root is:

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Multiply and divide by the conjugate of the numerator:

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{(-b - \sqrt{b^2 - 4ac})}{(-b - \sqrt{b^2 - 4ac})}$$

$$\implies x_+ = \frac{-2c}{b + \sqrt{b^2 - 4ac}}$$

The following is a Matlab Implementation of this formula, we then compare it with the default implementation.

```
function myroots_acc(a, b, c)
posAns = -2 * c / (b + sqrt(b^2 - 4 * a * c));
negAns = (-b - sqrt(b^2 - 4 * a * c)) / (2 * a);
format long;
disp("Estimated Results: ");
disp(posAns);
disp(negAns);
p = [a b c];
```

```
x = roots(p);
disp(x)
end



Output of the program:
Estimated Results:
      -5.000000000000000e-02 + 3.162277660168339e+05i

      -5.000000000000000e-02 - 3.162277660168339e+05i

    1.0e+05 *

  -0.000000500000000 + 3.162277660168340i
  -0.000000500000000 - 3.162277660168340i
```

Let $f(x)$ be a differential function on the real line. The derivative is defined as

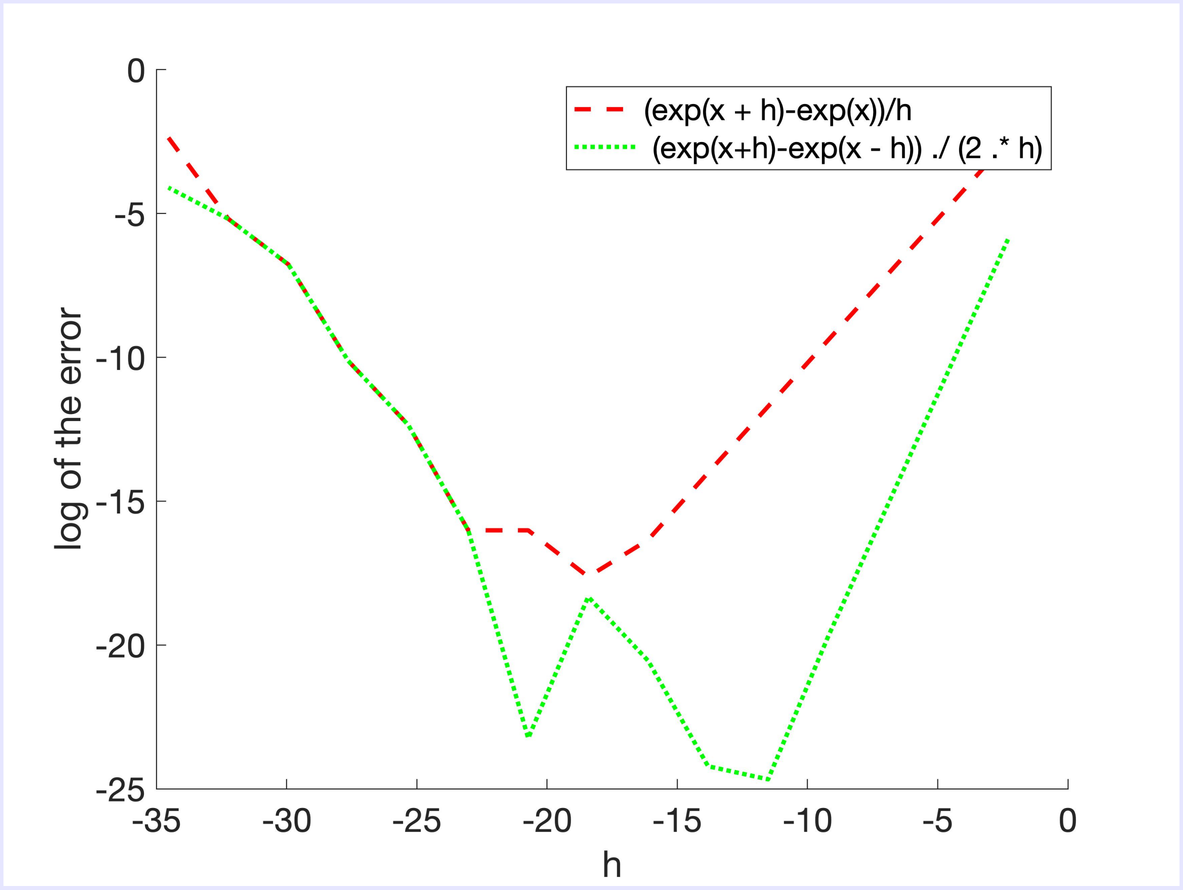
$$f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}.$$

Here are two different approximations to the derivative

$$f_1(x) = \frac{f(x+h)-f(x)}{h} \qquad f_2(x) = \frac{f(x+h)-f(x-h)}{2h}.$$

We anticipate that these approximations are accurate if $h \approx 0$. We investigate its accuracy numerically for the function $f(x) = \exp(x)$.

Let $x = 0.5$ and let $h \in \{10^{-1}, \ldots, 10^{-15}\}$. The following is a `log-log` scale plot for the relative error of the two approximations as a function of h.



In our graph $rel1(x)$ and $rel2(x)$ are the relative errors between $f'(x)$ and $f1(x)$ and $f'(x)$ and $f2(x)$, respectively. The relative errors curves are high when h is large as one would expect. Also the $rel2(x)$ is beneath $rel1(x)$, which means that the derivative approximating function $f2(x)$ is does a better job at approximating the derivative of the exponential function than $f1(x)$. The curves appear to reach a minimum and then rise again when h is very small. There must be a source of error that is becoming more noticeable as we decrease h. That source of error is dominating the error due to us approximating the derivative of the exponential function. This error could be round-off error.