Numerical methods

Author: Daria Shutina

23-02-02

Org stuff

Grade:

- 100% exam
- bonus 10% via homeworks

Now the lecture starts

Approaches to solving a problem:

- iteration: $x_0 = c;\ x_n = f(x_{n-1})$
- interpolation -- choose a function which is the closest to the initial function
- integration -- use integrals

Taylor series

Given a function $f\ :\ R \to R$, which is infinitely differentiable at $c \in R$. The Taylor series of f at c is:

$$f(x) = \sum_{n=0}^{+\infty} \frac{f^{(n)}(x_0)}{n!}(x-c)^n$$

If $c = 0$, then it is called ***the Maclaurin series***.

Note: A power series have an interval/radius of convergence. $f^{(n)} \in \textbf{radius of conv.}$

Given a function $f = \sum a_n x^n$. Then a radius of convergence of f is $R = \frac{1}{\lim\limits_{n\to\infty} \sup \sqrt[n]{|a_n|}}$

Note: The smaller the difference between x and c, the faster the Taylor series converge.

Taylor theorem

version 1

$f \in C^{n+1}([a, b])$ (n+1 times continuously differentiable in [a, b])

Then for $\forall c \in [a, b]$ we have that $f = \sum\limits_{k=0}^{n} \frac{f^{(k)}(c)}{k!}(x-c)^k + \underbrace{\frac{f^{(n+1)}(\xi_x)}{(n+1)!}(x-c)^{n+1}}_{E_n(x)\ -\ remainder}$, where ξ_x is between x and c and depends on x.

version 2

$f \in C^{n+1}([a,b])$

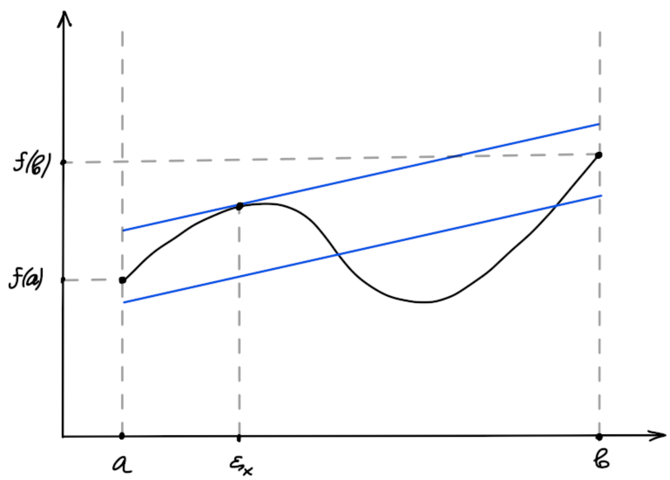
For $x, x+h \in [a,b]$ $\ f(x+h) = \sum\limits_{k=0}^{n} \frac{f^{(k)}(x)}{k!} h^k + \frac{f^{(n+1)}(\xi_x)}{(n+1)!} h^{n+1}$, where $E_n(x) = O(h^{n+1})$

23-02-03

Mean value theorem

For $n = 0$ $f(x) = f(c) + f'(\xi_x)(x-c)$

$x := b,\ c := a \ \Rightarrow\ f(b) = f(a) + f'(\xi_x)(b-a) \ \Rightarrow\ f'(\xi_x) = \frac{f(b)-f(a)}{b-a}$



Definition: The Taylor series *represents* f at $(.)x$ iff the Taylor series converge at $(.)x$.

$f = e^x$

$c = 0,\ e^x = \sum\limits_{0}^{n} \frac{x^k}{k!} + \frac{e^{\xi_x}}{(n+1)!} x^{n+1} (*)$

For $\forall x \in R\ \exists s \in R_0^+\ :\ |x| \leqslant s\ \wedge\ |\xi_x| \leqslant s$

e^x is monotone increasing $\Rightarrow e^{\xi_x} \leqslant e^s \;\Rightarrow\; \lim\limits_{n\to\infty} |\frac{e^{\xi_x}}{(n+1)!} x^{n+1}| \leqslant e^s \cdot \lim\limits_{n\to\infty} |\frac{s^{n+1}}{(n+1)!}| = 0 \;\Rightarrow\; (*)$ represents e^x at x.

$f = \ln(1+x)$ \todo

$c = 0$

$f^{(k)}(x) = (-1)^{k-1}(k-1)!\frac{1}{(1+x)^k}$

$g(x) = \sum\limits_0^n \frac{(-1)^{k-1}}{k} x^k + \frac{(-1)^n}{n+1} \cdot \frac{1}{(1+\xi_x)^{n+1}} \cdot x^{n+1}$

$\lim\limits_{n\to\infty} E_n(x) = \underbrace{\lim\limits_{n\to\infty} \frac{(-1)^n}{n+1}}_{\to 0} \cdot \lim\limits_{n\to\infty} (\frac{x}{\xi_x+1})^{n+1} = 0 \;\Rightarrow\; 0 < \frac{x}{\xi_x+1} < 1$

$\Rightarrow\; x \leqslant 1,\; if\; \xi_x \in [0, x]$ and $x > -1,\; if\; \xi_x \in [x, 0] \;\Rightarrow\; g$ represents f at $x \in (-1, 1)$.

Counting $\cos(0,1)$

$f = \cos x$

$g(x) = \sum\limits_0^n (-1)^k \frac{x^{2k}}{(2k)!} + (-1)^{n+1} \cos\xi_x \frac{x^{2(n+1)}}{(2(n+1))!},\; c = 0$

$|(-1)^{n+1} \underbrace{\cos\xi_x}_{\leqslant 1} \cdot \frac{x^{2(n+1)}}{(2(n+1))!}| \leqslant |\frac{x^{2(n+1)}}{(2(n+1))!}|$

$|\frac{0{,}1^{2(n+1)}}{(2(n+1))!}| \underset{n\to\infty}{\to} 0 \;\Rightarrow\; g$ represents f at $(.)0, 1$.

23-02-09

Base representation

Every number $x \in N$ can be written in the folowing form as a unique expansion with the resect to the base b, where $b \in N/\{0\}$, using digits a_i:

$x = \sum\limits_{i=0}^n a_i b^i.$

For a real number $x \in R$ we can write: $x = \sum_{i=1}^{+\infty} a_{-i} b^{-i}$.

General remarks:

- A number with simple representation in one base may be complicated to represent in another base:

  $0.1_{10} = (0.0001100110011\ldots)_2$
- $b = 2$ is binary, $b = 8$ is octal, $b = 16$ is hexadecimal
- To convert from base b to base 10, we perform the dolowwing computation:

  $y_b = \overline{a_n \ldots a_0}_b = \sum_{i=0}^{n} a_n b^n = x_{10}$
- Conversion $2 \to 8$: three digits with base 2 represent one digit with base 8
- Conversion $2 \to 16$: four digits with base 2 represent one digit with base 16

Example

$b = 2;\ 1011_2 = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 = 11_{10}$

Euclid's algorithm

Euclid's algorithm converts x_{10} to y_b.

1. Input x_{10}
2. Determine $\min n\ :\ x < b^{n+1}$
3. for $i := n$ to 0 do:

   $a_i = x\ \mathtt{div}\ b^i$

   $x = x\ \mathtt{mod}\ b^i$
4. Output result $\overline{a_n a_{n-1} \ldots a_0} = y_b$.

Problems:

1. Step 2 is inefficient
2. Division by large numbers can be problematic

Example

1. $13_{10} \longrightarrow y_2$
2. $\min n = 3 \ : \ 13 < 2^4$
3. $i = 3; \ \ a_3 = 1, \ x = 5$

   $i = 2; \ \ a_2 = 1, \ x = 1$

   $i = 1; \ \ a_1 = 0, \ x = 1$

   $i = 0; \ \ a_0 = 1, \ x =$
4. $\overline{a_3 a_2 a_1 a_0} = 1101_2$

Horner’s scheme

- no division by large numbers
- no need in finding the amount of digits for y_b (*aka* n in Euclid's algorithm)
- applicable to real numbers, but needs a condition when we need to stop if representation with new base is infinite

1. Input x_{10}. $i := 0$.
2. ```
   while (x > 0) {
       a[i++] = x % b;
       x /= b;
   }
   ```
3. $\overline{a_n a_{n-1} \ldots a_0} = y_b$

23-02-10

Number normalization

$x = 0.a_1 a_2 \ldots a_k \cdot b^n$ with $a_i \in \{0, \ldots, b-1\}$.

$a_1, \ldots, a_k$ are digits.

b is the base.
   ```

k is called *precision*. This is the actual amount of digits.

$\overline{a_1 \dots a_k}$ is called *mantissa*. $a_1 \neq 0$.

n is called *exponent*. This is the distance from the current position of the floating point. $n > 0 \Rightarrow$ move right, otherwise move left.

This form is called *normalization*. It makes representation of a number unique.

Examples

$0.099 \;\Leftrightarrow\; 0.99 \cdot 10^{-1}$. We do not save leading zeros in the mantissa.

$32.213 \;\Leftrightarrow\; 0.32213 \circ 10^2$.

$1.101 \;\Leftrightarrow\; 0.1101 \circ 2^1$.

Single precision

There are 4 bytes = 32 bits.

1 bit is for the sign of the number.

1 bit for the sign of the exponent.

7 bits for for the exponent. 7-bit largest number is $\underbrace{1 \dots 1_2}_{7} = 127$. $2^{127} \approx 10^{38}$, so we have numbers from -10^{38} to 10^{38}.

23 bits for mantissa. Actually, we are "able" to store 24 bits of a number, because, as we know, $a_1 \neq 0 \Rightarrow$ it is always equal to 1 at base $b = 2 \Rightarrow$ it can be omitted.

We can have a better representation using *double precision* (8 bytes = 64 bits).

Problems with double numbers

Example 1: sum of numbers

Adding numbers is commutative, but not always associative: $z + (y + w) \neq (z + y) + w$.

For example, let us take $x = y = 0.00000033 = 0.33 \cdot 10^{-6}$, $z = 0.00000034 = 0.34 \cdot 10^{-6}$, $w = 1.000000 = 0.1 \cdot 10^{1}$.

$((x + y) + z) + w = 0.1 \cdot 10^{-5} + 0.1 \cdot 10^{1} = 1.000001$

$x + (y + (z + w)) = 1.000000$, since $z + w$ can have only 7 significant digits in the mantissa.

Thus, it would be better to add numbers in increasing order of their size.

Example 2: subtraction of numbers

Let us compute $x - \sin x$ with x close to 0. For example, $x = \frac{1}{15}$.

$x = 0.6666666667 \cdot 10^{-1}$

$\sin x = 0.6661729492 \cdot 10^{-1}$

$x - \sin x = 0.0004937175 \cdot 10^{-1} = 0.4937175000 \cdot 10^{-4}$.

Three zeros at the end of the mantissa mean a precision loss.

Thus, it would be better to avoid subtracting numbers of similar size.

The better idea is to use the Taylor series:

$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$

Th. (about the lost of precision during subtraction)

Let x, y be normalized floating point numbers with $x > y > 0$ and base $b = 2$.

If $\exists p, q \in N_0 \ : \ 2^{-p} \leqslant 1 - \frac{y}{x} \leqslant 2^{-q}$, then at most p and at least q significant bits (digits at base 2) are lost during subtraction.