CÔNG THỨC NỘI SUY TRUNG TÂM

Nhóm 18: { Lê Thị Vân Anh
Lê Nguyên Bách

CTTN Toán Tin K64 - ĐHBKHN

12/2020

Mục lục

I Giới thiệu

1 Bài toán

Cho hàm số $y = f(x)$ liên tục trên đoạn $[a, b]$, được biểu diễn qua bảng các giá trị

$$y_i = f(x_i) \ \left(x_i \in [a, b], i = \overline{0, n}\right)$$

Cụ thể hơn

x	x_0	x_1	x_2	...	x_n
$f(x)$	y_0	y_1	y_2	...	y_n

Với (x_i, y_i) là $n + 1$ bộ số cho trước

Từ bảng số liệu trên, xây dựng đa thức $P_n(x)$ có $\deg P_n(x) \leq n$ thỏa mãn $P_n(x_i) = y_i \ \left(i = \overline{0, n}\right)$. Khi đó

$$P_n(x) \approx f(x), \forall x \in [a, b]$$

Đa thức $P_n(x)$ được gọi là **đa thức nội suy**, các điểm x_i là các **mốc nội suy**

Giá trị $\overline{y} = P_n(\overline{x}) \approx f(\overline{x}) \ (\overline{x} \neq x_i)$ gọi là **giá trị nội suy** nếu $x \in (a, b)$

Hiệu số $R_n(\overline{x}) = f(\overline{x}) - P_n(\overline{x})$ gọi là **sai số** của phép tính nội suy tại điểm $\overline{x}$

2 Đa thức nội suy

Một câu hỏi được đặt ra là: "Tại sao ta chọn xấp xỉ hàm qua một đa thức mà không phải kiểu dạng hàm khác?". Đơn giản là vì đa thức có nhiều đặc tính thuận tiện cho việc tính toán như khả vi liên tục mọi cấp trên $\mathbb{R}$, lấy đạo hàm hoặc tích phân một hay nhiều lần dễ dàng hơn,...

Hiện ta đã biết được khá nhiều công thức nội suy như Lagrange, Newton,.... Vậy thì đa thức nội suy cho được bởi các công thức khác nhau có khác nhau không? Câu trả lời là không, vì đa thức nội suy là xác định và duy nhất. Điểm khác nhau ở các công thức đó là số lượng phép tính của từng công thức đối với các trường hợp khác nhau và sai số trong quá trình tính toán.

Định lý: Đa thức $P_n(x)$ bậc không vượt quá n, được sinh ra từ bảng số $\left\{(x_i, y_i)\right\}_{i=0}^{n}$ thỏa mãn $P_n(x_i) = y_i \ \left(i = \overline{0, n}\right)$ xác định và duy nhất

Chứng minh

Giả sử đa thức nội suy có dạng

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

Lần lượt thay $x = x_i$ $\left(i = \overline{0,n}\right)$, ta được một hệ phương trình ẩn a_i

$$\begin{cases} a_0 + a_1x_0 + a_2x_0^2 + ... + a_nx_0^n = P_n(x_0) \\ a_0 + a_1x_1 + a_2x_1^2 + ... + a_nx_1^n = P_n(x_1) \\ a_0 + a_1x_2 + a_2x_2^2 + ... + a_nx_2^n = P_n(x_2) \\ ... \\ a_0 + a_1x_n + a_2x_n^2 + ... + a_nx_n^n = P_n(x_n) \end{cases}$$

Viết lại hệ dưới dạng ma trận, ta được

$$\begin{pmatrix} 1 & x_0 & x_0^2 & ... & x_0^n \\ 1 & x_1 & x_1^2 & ... & x_1^n \\ 1 & x_2 & x_2^2 & ... & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & ... & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} P_n(x_0) \\ P_n(x_1) \\ P_n(x_2) \\ \vdots \\ P_n(x_n) \end{pmatrix} \Leftrightarrow \underbrace{\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}}_{A} = \begin{pmatrix} 1 & x_0 & x_0^2 & ... & x_0^n \\ 1 & x_1 & x_1^2 & ... & x_1^n \\ 1 & x_2 & x_2^2 & ... & x_2^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & ... & x_n^n \end{pmatrix}^{-1} \begin{pmatrix} P_n(x_0) \\ P_n(x_1) \\ P_n(x_2) \\ \vdots \\ P_n(x_n) \end{pmatrix}$$

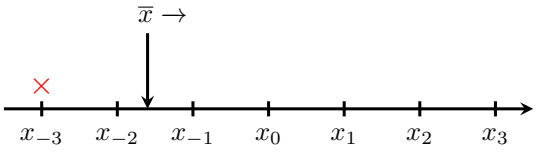
Do $P_n(x_i) = y_i$ và các bộ (x_i, y_i) xác định, do đó ma trận cột A xác định và duy nhất (đpcm)

II Công thức nội suy trung tâm

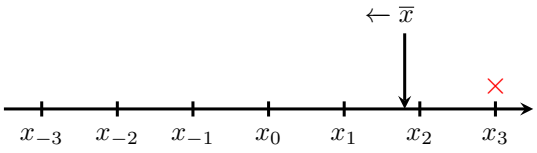
1 Đặt vấn đề

Ta đã biết tới công thức nội suy Newton

Ở trường hợp này, ta sẽ sử dụng Newton tiến, với mốc nội suy ban đầu x_{-2} và không sử dụng mốc x_{-3}



Hay như ở trường hợp này, ta sẽ sử dụng Newton lùi, với mốc nội suy ban đầu x_2 và không sử dụng mốc x_3



Công thức nội suy Newton chỉ mang đặc trưng cho một phía. Trong nhiều trường hợp, ta cần tính $f(\overline{x})$ với $\overline{x}$ ở gần giá trị chính giữa, khi đó công thức nội suy Newton sẽ bị hạn chế vì gần một nửa số mốc nội suy không được sử dụng để tính. Vì vậy trong bài báo cáo này, nhóm 18 chúng em sẽ đề cập tới công thức nội suy sử dụng cả hai phía, gọi là **công thức nội suy trung tâm**, với điều kiện các mốc nội suy **cách đều nhau**

2 Các công thức nội suy trung tâm

2.1 Công thức nội suy Gauss

Giả sử có $2n+1$ mốc nội suy $\{x_k\}_{k=-n}^{n}$ $\left(x_i < x_j, \forall i < j\right)$ **cách đều nhau** với bước nhảy h, và bảng giá trị

x	x_{-n}	x_{-n+1}	...	x_0	...	x_{n-1}	x_n
$f(x)$	y_{-n}	y_{-n+1}	...	y_0	...	y_{n-1}	y_n

Từ bảng số trên, xây dựng đa thức nội suy $P(x)$ với $\deg P(x) \leq 2n$ sao cho $P(x_i) = y_i$ $\left(\forall i = \overline{0, \pm n}\right)$

2.1.1 Công thức nội suy Gauss I

Giống như đa thức nội suy Newton nhưng thay tỉ hiệu bằng một hằng số chưa biết a_i, và thứ tự lấy mốc nội suy sẽ là "tiến" trước rồi "lùi" sau. Xuất phát từ mốc x_0, ta có

$$\begin{aligned} P(x) = a_0 &+ a_1(x-x_0) + a_2(x-x_0)(x-x_1) + a_3(x-x_{-1})(x-x_0)(x-x_1) + ... + \\ &+ a_{2n-1}\left(x-x_{-n+1}\right)(...)(x-x_{-1})(x-x_0)(x-x_1)(...)(x-x_{n-1}) + \\ &+ a_{2n}(x-x_{-n+1})(...)(x-x_{-1})(x-x_0)(x-x_1)(...)(x-x_n) \end{aligned}$$

Ta sẽ tính các hệ số a_i

Thay $x = x_0$, khi đó $P(x_0) = y_0 = a_0$

Thay $x = x_1$, khi đó

$$P(x_1) = y_1 = y_0 + a_1(x_1 - x_0)$$
$$\Rightarrow a_1 = \frac{y_1 - y_0}{x_1 - x_0} = \frac{\Delta y_0}{h}$$

Thay $x = x_{-1}$, khi đó

$$P(x_{-1}) = y_{-1} = y_0 + \frac{\Delta y_0}{h}(x_{-1} - x_0) + a_2(x_{-1} - x_0)(x_{-1} - x_1) = y_0 + \Delta y_0 + 2a_2h^2$$
$$\Rightarrow a_2 = \frac{\Delta y_{-1} - \Delta y_0}{2h^2} = \frac{\Delta^2 y_{-1}}{2h^2}$$

Thực hiện thay $x = x_k$ để tính a_{2k}, thay $x = x_{-k}$ để tính a_{2k-1}, ta được số hạng tổng quát

$$a_{2k-1} = \frac{\Delta^{2k-1} y_{-(k-1)}}{(2k-1)!h^{2k-1}} \quad ; \quad a_{2k} = \frac{\Delta^{2k} y_{-k}}{(2k)!\, h^{2k}}$$

Đặt $t = \dfrac{x - x_0}{h}$, khi đó đa thức nội suy trở thành

$$\begin{aligned} P(x) &= P(x_0 + ht) = G_1(t) \\ &= y_0 + \binom{t}{1}\Delta y_0 + \binom{t}{2}\Delta^2 y_{-1} + ... + \binom{t+n-1}{2n-1}\Delta^{2n-1} y_{-(n-1)} + \binom{t+n-1}{2n}\Delta^{2n} y_{-n} \end{aligned} \quad (1)$$

Trong đó, số hạng thứ $2k-1$ và $2k$ của đa thức $G_1(t)$ là

$$g_{2k-1}^{(1)} = \binom{t+k-1}{2k-1}\Delta^{2k-1}y_{-(k-1)} \quad ; \quad g_{2k}^{(1)} = \binom{t+k-1}{2k}\Delta^{2k}y_{-k}$$

Công thức (1) gọi là công thức nội suy Gauss I

Ở bảng sai phân trung tâm, ta thấy công thức (1) có hệ số được tính theo đường gấp khúc ↘↗

x	y	Δy	$\Delta^2 y$	$\Delta^3 y$	$\Delta^4 y$	$\Delta^5 y$	$\Delta^6 y$	...
$\vdots$	$\vdots$							
x_{-3}	y_{-3}							
		Δy_{-3}						
x_{-2}	y_{-2}		$\Delta^2 y_{-3}$					
		Δy_{-2}		$\Delta^3 y_{-3}$				
x_{-1}	y_{-1}		$\Delta^2 y_{-2}$		$\Delta^4 y_{-3}$			
		Δy_{-1}		$\Delta^3 y_{-2}$		$\Delta^5 y_{-3}$		
x_0	y_0		$\Delta^2 y_{-1}$		$\Delta^4 y_{-2}$		$\Delta^6 y_{-3}$	...
		Δy_0		$\Delta^3 y_{-1}$		$\Delta^5 y_{-2}$		
x_1	y_1		$\Delta^2 y_0$		$\Delta^4 y_{-1}$			
		Δy_1		$\Delta^3 y_0$				
x_2	y_2		$\Delta^2 y_1$					
		Δy_2						
x_3	y_3							
$\vdots$	$\vdots$							

2.1.2 Công thức nội suy Gauss II

Tương tự như công thức nội suy Gauss I nhưng thứ tự lấy mốc nội suy sẽ là "lùi" trước rồi "tiến" sau. Cũng xuất phát từ mốc x_0, ta có

$$\begin{aligned} P(x) = a_0 &+ a_1(x-x_0) + a_2(x-x_{-1})(x-x_0) + a_3(x-x_{-1})(x-x_0)(x-x_1) + ... + \\ &+ a_{2n-1}(x-x_{-n+1})(...)(x-x_{-1})(x-x_0)(x-x_1)(...)(x-x_{n-1}) + \\ &+ a_{2n}(x-x_{-n})(x-x_{-n+1})(...)(x-x_{-1})(x-x_0)(x-x_1)(...)(x-x_{n-1}) \end{aligned}$$

Thực hiện thay $x = x_k$ để tính a_{2k}, thay $x = x_{-k}$ để tính a_{2k-1}, ta được số hạng tổng quát

$$a_{2k-1} = \frac{\Delta^{2k-1}y_{-k}}{(2k-1)!h^{2k-1}} \quad ; \quad a_{2k} = \frac{\Delta^{2k}y_{-k}}{(2k)!h^{2k}}$$

Đặt $t = \dfrac{x-x_0}{h}$, khi đó đa thức nội suy trở thành

$$\begin{aligned} P(x) &= P(x_0+ht) = G_2(t) \\ &= y_0 + \binom{t}{1}\Delta y_{-1} + \binom{t+1}{2}\Delta^2 y_{-1} + ... + \binom{t+n-1}{2n-1}\Delta^{2n-1}y_{-n} + \binom{t+n}{2n}\Delta^{2n}y_{-n} \end{aligned} \tag{2}$$

Trong đó, số hạng thứ $2k$ và $2k+1$ của đa thức $G_2(t)$ là

$$g_{2k-1}^{(2)} = \binom{t+k-1}{2k-1}\Delta^{2k-1}y_{-k} \quad ; \quad g_{2k}^{(2)} = \binom{t+k}{2k}\Delta^{2k}y_{-k}$$

Công thức (2) gọi là công thức nội suy Gauss II

Ở bảng sai phân trung tâm, ta thấy công thức (2) có hệ số được tính theo đường gấp khúc ↗↘

x	y	Δy	$\Delta^2 y$	$\Delta^3 y$	$\Delta^4 y$	$\Delta^5 y$	$\Delta^6 y$	...
$\vdots$	$\vdots$							
x_{-3}	y_{-3}							
		Δy_{-3}						
x_{-2}	y_{-2}		$\Delta^2 y_{-3}$					
		Δy_{-2}		$\Delta^3 y_{-3}$				
x_{-1}	y_{-1}		$\Delta^2 y_{-2}$		$\Delta^4 y_{-3}$			
		Δy_{-1}		$\Delta^3 y_{-2}$		$\Delta^5 y_{-3}$		
x_0	y_0		$\Delta^2 y_{-1}$		$\Delta^4 y_{-2}$		$\Delta^6 y_{-3}$	...
		Δy_0		$\Delta^3 y_{-1}$		$\Delta^5 y_{-2}$		
x_1	y_1		$\Delta^2 y_0$		$\Delta^4 y_{-1}$			
		Δy_1		$\Delta^3 y_0$				
x_2	y_2		$\Delta^2 y_1$					
		Δy_2						
x_3	y_3							
$\vdots$	$\vdots$							

2.2 Công thức nội suy Stirling

2.2.1 Thiết lập công thức

Ở công thức nội suy Stirling, ta lấy trung bình cộng của hai công thức nội suy Gauss.

$$S(t) = \frac{G_1(t) + G_2(t)}{2}$$

Số hạng thứ $2k-1$ của đa thức $S(t)$ là

$$s_{2k-1} = \frac{1}{2}\left(g_{2k-1}^{(1)} + g_{2k-1}^{(2)}\right) = \binom{t+k-1}{2k-1}\frac{\Delta^{2k-1}y_{-(k-1)} + \Delta^{2k-1}y_{-k}}{2}$$

Số hạng thứ $2k$ của đa thức $S(t)$ là

$$\begin{aligned} s_{2k} &= \frac{1}{2}\left(g_{2k}^{(1)} + g_{2k}^{(2)}\right) = \frac{1}{2}\left(\binom{t+k-1}{2k}\Delta^{2k}y_{-k} + \binom{t+k}{2k}\Delta^{2k}y_{-k}\right) \\ &= \frac{\Delta^{2k}y_{-k}}{2}\left(\frac{(t+k-1)(t+k-2)(...)(t-k)}{(2k)!} + \frac{(t+k)(t+k-1)(...)(t-k+1)}{(2k)!}\right) \end{aligned}$$

$$= \frac{\Delta^{2k} y_{-k}}{2(2k)!}(t+k-1)(t+k)(...)(t-k+1).(t-k+t+k)$$

$$= \frac{\Delta^{2k} y_{-k}}{(2k)!}.t^2\left(t^2-1^2\right)\left(t^2-2^2\right)(...)\left(t^2-(k-1)^2\right) = \frac{\Delta^{2k} y_{-k}}{(2k)!}\prod_{i=0}^{k-1}\left(t^2-i^2\right)$$

Ta được đa thức nội suy Stirling

$$S(t) = y_0 + \binom{t}{1}\frac{\Delta y_{-1}+\Delta y_0}{2} + \frac{t^2}{2!}\Delta^2 y_{-1} + ... + \\ + \binom{t+n-1}{2n-1}\frac{\Delta^{2n-1}y_{-n+1}+\Delta^{2n-1}y_{-n}}{2} + \frac{\Delta^{2n}y_{-n}}{(2n)!}\prod_{i=0}^{n-1}\left(t^2-i^2\right)$$

Công thức này có khối lượng tính toán ít hơn so với các công thức Gauss I và Gauss II, chi tiết sẽ được trình bày ở phần **Thuật toán**

2.2.2 Sai số của công thức

Với $\bar{t} = \dfrac{\overline{x}-x_0}{h}$ $\left(x \neq x_j\,, j = \overline{0,\pm n}\right)$ cố định, ta có sai số tại đó là

$$R\left(\bar{t}\right) = f\left(x_0 + h\bar{t}\right) - S\left(\bar{t}\right)$$

Trong đó hàm số $f(x) = f\left(x_0 + ht\right)$ là hàm được biểu diễn bằng bảng số $\left\{(x_j, y_j)\right\}_{j=-n}^{n}$. Ta sẽ thiết lập công thức sai số tại điểm $\bar{t}$ cho đa thức nội suy Stirling. Để thuận tiện hơn, ta viết

$$\Omega_n(t) = h^{2n+1}t\left(t^2-1^2\right)\left(t^2-i^2\right)(...)\left(t^2-n^2\right) = h^{2n+1}t\prod_{i=1}^{n}\left(t^2-i^2\right)$$

Đặt

$$F(t) = R(t) - \lambda\Omega_n(t)$$

Chọn hằng số λ sao cho $F\left(\bar{t}\right) = 0$, hay $\lambda = \dfrac{R\left(\bar{t}\right)}{\Omega_n\left(\bar{t}\right)}$ $(^*)$

Dễ thấy $\Omega_n(j) = 0, \forall j = \overline{0,\pm n}$, do đó $F(j) = 0$. Vì vậy, phương trình $F(t) = 0$ có $2n+2$ nghiệm $t \in \left\{\bar{t}, 0, \pm1, \pm2, ..., \pm n\right\}$. Theo định lý Rolle, tồn tại $c \in (-n, n)$ để $F^{(2n+1)}(c) = 0$, hay

$$R^{(2n+1)}(c) - \lambda\Omega_n^{(2n+1)}(c) = 0$$

$$\Leftrightarrow f^{(2n+1)}(x_0+hc) - S^{(2n+1)}(c) - \lambda\Omega_n^{(2n+1)}(c) = 0$$

Theo định nghĩa $\Omega_n(t)$, ta có $\deg\Omega_n(t) = 2n+1$, hệ số của t^{2n+1} là h^{2n+1} nên $\Omega_n^{(2n+1)}(t) = (2n+1)!$ Hơn nữa, $\deg S(t) \leq 2n$ nên $S^{(2n+1)}(t) = 0$. Do đó

$$f^{(2n+1)}(x_0+hc) - \lambda(2n+1)! = 0$$

Hay

$$\lambda = \frac{f^{(2n+1)}(x_0+hc)}{(2n+1)!}$$

Kết hợp với $(^*)$, đặt $x_0 + hc = \xi \in (x_{-n}, x_n)$, ta được

$$R\left(\bar{t}\right) = \frac{f^{(2n+1)}(\xi)}{(2n+1)!}\Omega\left(\bar{t}\right)$$

Nếu $f(x)$ là hàm chưa biết thì ta có thể thay

$$f^{(2n+1)}(\xi) \approx \frac{1}{2h^{2n+1}} \left(\Delta^{2n+1} y_{-(n+1)} + \Delta^{2n+1} y_{-n}\right)$$

Nhưng với số lượng mốc nội suy là $2n+1$ thì ta không thể có được Δ^{2n+1}. Để khắc phục thì ta sẽ sử dụng bất đẳng thức

$$\left|\frac{\Delta^{2n+1} y_{-(n+1)} + \Delta^{2n+1} y_{-n}}{2(2n+1)!}\right| \leq \left|\frac{\Delta^{2n} y_{-n}}{(2n)!}\right|$$

Khi đó

$$\left|R\left(\bar{t}\right)\right| \leq \left|\frac{\Delta^{2n} y_{-n}}{(2n)!}\, \bar{t} \prod_{i=1}^{n} \left(\bar{t}^2 - i^2\right)\right|$$

2.3 Công thức nội suy Bessel

2.3.1 Thiết lập công thức

Khác với công thức nội suy Stirling, công thức nội suy Bessel sử dụng $2n+2$ mốc nội suy cách đều

x	x_{-n}	x_{-n+1}	...	x_0	...	x_n	x_{n+1}
$f(x)$	y_{-n}	y_{-n+1}	...	y_0	...	y_n	y_{n+1}

Sử dụng công thức Gauss II với mốc xuất phát là x_0 (Khi đó ta sẽ bỏ đi mốc x_{n+1}), ta được đa thức $G_2(t)$ với các số hạng tổng quát

$$g^{(2)}_{2k-1} = \binom{t+k-1}{2k-1} \Delta^{2k-1} y_{-k} \quad ; \quad g^{(2)}_{2k} = \binom{t+k}{2k} \Delta^{2k} y_{-k}$$

Trong đó $t = \dfrac{x - x_0}{h}$

Nhưng thay vì xuất phát tại x_0, ta sẽ cho xuất phát từ x_1 (Khi đó ta sẽ bỏ đi mốc x_{-n}). Lúc này

$$\frac{x - x_1}{h} = \frac{x - x_0}{h} + \frac{x_0 - x_1}{h} = t - 1$$

Như vậy đa thức $G_2(t)$ sẽ được chuyển thành đa thức $G'_2(t) = G_2(t-1)$, với các số hạng tổng quát

$$g'_{2k-1} = \binom{t+k-2}{2k-1} \Delta^{2k-1} y_{-(k-1)} \quad ; \quad g'_{2k} = \binom{t+k-1}{2k} \Delta^{2k} y_{-(k-1)}$$

Khi đó ta được đa thức nội suy Bessel

$$B(t) = \frac{G_1(t) + G'_2(t)}{2}$$

Trong đó $G_1(t)$ là công thức Gauss I, xuất phát tại mốc x_0

Số hạng thứ $2k-1$ của đa thức $B(t)$ là

$$b_{2k-1} = \frac{1}{2}\left(g^{(1)}_{2k-1} + g'_{2k-1}\right) = \frac{\Delta^{2k-1} y_{-(k-1)}}{2} \left(\binom{t+k-1}{2k-1} + \binom{t+k-2}{2k-1}\right)$$

$$= \frac{\Delta^{2k-1}y_{-(k-1)}}{2}\left(\frac{(t+k-1)(t+k-2)(...)(t-k+1)}{(2k-1)!} + \frac{(t+k-2)(t+k-3)(...)(t-k)}{(2k-1)!}\right)$$
$$= \frac{\Delta^{2k-1}y_{-(k-1)}}{2(2k-1)!}(t+k-2)(t+k-3)(...)(t-k+1).(t+k-1+t-k)$$
$$= \frac{\Delta^{2k-1}y_{-(k-1)}}{(2k-1)!}\left(t-\frac{1}{2}\right).(t+k-2)(t+k-3)(...)(t-k+1)$$

Số hạng thứ $2k$ của đa thức $B(t)$ là

$$b_{2k} = \frac{1}{2}\left(g_{2k}^{(1)} + g'_{2k}\right) = \binom{t+k-1}{2k}\frac{\Delta^{2k}y_{-(k-1)} + \Delta^{2k}y_{-k}}{2}$$

Ta được đa thức nội suy Bessel

$$B(t) = \frac{y_0+y_1}{2} + \left(t-\frac{1}{2}\right)\Delta y_0 + \binom{t}{2}\frac{\Delta^2 y_{-1} + \Delta^2 y_0}{2} + ... +$$
$$+ \binom{t+n-1}{2n}\frac{\Delta^{2n}y_{-(n-1)} + \Delta^{2n}y_{-n}}{2} + \frac{\Delta^{2n+1}y_{-n}}{(2n+1)!}\left(t-\frac{1}{2}\right)(t+n-1)(t+n-3)(...)(t-n)$$

2.3.2 Sai số của công thức

Tương tự như cách thiết lập công thức sai số của đa thức nội suy Stirling, ta có công thức sai số của đa thức nội suy Bessel

$$R\left(\bar{t}\right) = \frac{h^{2n+2}}{(2n+2)!}f^{(2n+2)}(\xi)\bar{t}\left(\bar{t}-(n+1)\right)\prod_{i=1}^{n}\left(\bar{t}^2-i^2\right)$$

Nếu $f(x)$ là hàm chưa biết thì ta có thể thay

$$f^{(2n+2)}(\xi) \approx \frac{1}{2h^{2n+2}}\left(\Delta^{2n+2}y_{-(n+1)} + \Delta^{2n+2}y_{-n}\right)$$

Nhưng với số lượng mốc nội suy là $2n+2$ thì ta không thể có được Δ^{2n+2}. Để khắc phục thì ta sẽ sử dụng bất đẳng thức

$$\left|\frac{\Delta^{2n+2}y_{-(n+1)} + \Delta^{2n+2}y_{-n}}{2(2n+2)!}\right| \leq \left|\frac{\Delta^{2n+1}y_{-n}}{(2n+1)!}\right|$$

Khi đó

$$\left|R\left(\bar{t}\right)\right| \leq \left|\frac{\Delta^{2n+1}y_{-n}}{(2n+1)!}\,\bar{t}\left(\bar{t}-(n+1)\right)\prod_{i=1}^{n}\left(\bar{t}^2-i^2\right)\right|$$

III Thuật toán

1 Bảng sai phân

Bảng sai phân sẽ có dạng ma trận tam giác dưới

$$\begin{pmatrix} y_2 & 0 & 0 & 0 & 0 \\ y_1 & \Delta y_1 & 0 & 0 & 0 \\ y_0 & \Delta y_0 & \Delta^2 y_0 & 0 & 0 \\ y_{-1} & \Delta y_{-1} & \Delta^2 y_{-1} & \Delta^3 y_{-1} & 0 \\ y_{-2} & \Delta y_{-2} & \Delta^2 y_{-2} & \Delta^3 y_{-2} & \Delta^4 y_{-2} \end{pmatrix}$$

Trong đó y_i là các giá trị cho trước

Để lập được bảng sai phân, ta sẽ định nghĩa hai hàm `Above_add()` và `Below_add()`

1.1 Hàm Above_add( )

Đối với hàm `Above_add()`, đầu vào sẽ gồm một bảng sai phân có dạng ma trận tam giác dưới và một giá trị bổ sung y, đầu ra sẽ là một bảng sai phân mới thu được bằng cách đặt thêm mốc mới vào đầu bảng

$$\texttt{Above\_add}\left(\left(y_0\right), y_1\right) = \begin{pmatrix} y_1 & 0 \\ y_0 & \Delta y_0 \end{pmatrix}$$

$$\texttt{Above\_add}\left(\begin{pmatrix} y_1 & 0 & 0 \\ y_0 & \Delta y_0 & 0 \\ y_{-1} & \Delta y_{-1} & \Delta^2 y_{-1} \end{pmatrix}, y_2\right) = \begin{pmatrix} y_2 & 0 & 0 & 0 \\ y_1 & \Delta y_1 & 0 & 0 \\ y_0 & \Delta y_0 & \Delta^2 y_0 & 0 \\ y_{-1} & \Delta y_{-1} & \Delta^2 y_{-1} & \Delta^3 y_{-1} \end{pmatrix}$$

Thiết lập hàm này rất đơn giản. Giả sử đầu vào gồm một bảng ma trận vuông cấp n và giá trị y. Ta thực hiện thêm một ma trận $1 \times n$ chứa toàn số 0 lên đầu bảng, sau đó thêm tiếp một ma trận $(n+1) \times 1$ vào bên phải bảng. Cuối cùng ta sửa lại các vị trí ở đường chéo chính

$$\begin{pmatrix} y_1 & 0 & 0 \\ y_0 & \Delta y_0 & 0 \\ y_{-1} & \Delta y_{-1} & \Delta^2 y_{-1} \end{pmatrix} \xrightarrow{\text{Thêm hàng}} \begin{pmatrix} 0 & 0 & 0 \\ y_1 & 0 & 0 \\ y_0 & \Delta y_0 & 0 \\ y_{-1} & \Delta y_{-1} & \Delta^2 y_{-1} \end{pmatrix}$$

$$\xrightarrow{\text{Thêm cột}} \begin{pmatrix} 0 & 0 & 0 & 0 \\ y_1 & 0 & 0 & 0 \\ y_0 & \Delta y_0 & 0 & 0 \\ y_{-1} & \Delta y_{-1} & \Delta^2 y_{-1} & 0 \end{pmatrix} \xrightarrow{\text{Sửa đường chéo}} \begin{pmatrix} y_2 & 0 & 0 & 0 \\ y_1 & \Delta y_1 & 0 & 0 \\ y_0 & \Delta y_0 & \Delta^2 y_0 & 0 \\ y_{-1} & \Delta y_{-1} & \Delta^2 y_{-1} & \Delta^3 y_{-1} \end{pmatrix}$$

```python
import numpy

def Above_add(Mat, num):
    # Thêm hàng mới (Gồm giá trị mới ở đầu hàng, còn lại là các số 0) lên trên đầu Mat
    new_row = numpy.zeros((1, len(Mat)))
    new_row[0][0] = num
    Mat = numpy.append(new_row, Mat, axis = 0)
    # Thêm cột chứa toàn số 0 vào bên phải Mat
    Mat = numpy.append(Mat, numpy.zeros((len(Mat), 1)), axis = 1)
    # Đặt các giá trị delta phù hợp vào đường chéo chính
    for i in range(1, len(Mat)):
        Mat[i][i] = Mat[i-1][i-1] - Mat[i][i-1]
    # Trả về ma trận Mat
    return Mat # Mat sẽ là ma trận tam giác dưới
```

Chạy thử

```python
Mat = [[4,  0,  0,  0],
       [5, -1,  0,  0],
       [-2, 7, -8,  0],
       [-4, 2,  5, -13]]
print(Above_add(Mat, 6))
>>> [[  6.   0.   0.   0.   0.]
     [  4.   2.   0.   0.   0.]
     [  5.  -1.   3.   0.   0.]
     [ -2.   7.  -8.  11.   0.]
     [ -4.   2.   5. -13.  24.]]
```

1.2 Hàm Below_add( )

Hơi khác một chút so với `Above_add()`, hàm `Below_add()` sẽ đặt thêm mốc mới vào đáy bảng. Đầu tiên ta sẽ thêm một ma trận $1 \times n$ chứa toàn số 0 vào bên phải bảng. Sau đó ta thêm hàng

$$\begin{pmatrix} y & \Delta y & \Delta^2 y & \cdots & \Delta^n y \end{pmatrix}$$

vào đáy bảng

$$\begin{pmatrix} y_1 & 0 & 0 \\ y_0 & \Delta y_0 & 0 \\ y_{-1} & \Delta y_{-1} & \Delta^2 y_{-1} \end{pmatrix} \xrightarrow{\text{Thêm cột}} \begin{pmatrix} y_1 & 0 & 0 & 0 \\ y_0 & \Delta y_0 & 0 & 0 \\ y_{-1} & \Delta y_{-1} & \Delta^2 y_{-1} & 0 \end{pmatrix} \xrightarrow{\text{Thêm hàng}} \begin{pmatrix} y_2 & 0 & 0 & 0 \\ y_1 & \Delta y_1 & 0 & 0 \\ y_0 & \Delta y_0 & \Delta^2 y_0 & 0 \\ y_{-1} & \Delta y_{-1} & \Delta^2 y_{-1} & \Delta^3 y_{-1} \end{pmatrix}$$

```python
import numpy

def Below_add(Mat, num):
    # Tạo hàng mới (Gồm giá trị mới ở đầu hàng, còn lại là các giá trị delta tương ứng)
    new_row = numpy.zeros((1, len(Mat)+1))
    new_row[0][0] = num
    for i in range(len(Mat[0])):
        new_row[0][i+1] = Mat[len(Mat)-1][i] - new_row[0][i]

    # Thêm cột mới (Gồm các số 0) vào bên phải Mat
    Mat = numpy.append(Mat, numpy.zeros((len(Mat), 1)), axis = 1)

    # Bây giờ mới thêm hàng mới (new_row) vào phía dưới Mat
    Mat = numpy.append(Mat, new_row, axis = 0)

    # Trả về ma trận Mat
    return Mat  # Mat sẽ là ma trận tam giác dưới
```

Chạy thử

```python
Mat = [[4,   0,   0,   0],
       [5,  -1,   0,   0],
       [-2,  7,  -8,   0],
       [-4,  2,   5, -13]]
print(Below_add(Mat, -5))
>>> [[  4.   0.   0.   0.   0.]
     [  5.  -1.   0.   0.   0.]
     [ -2.   7.  -8.   0.   0.]
     [ -4.   2.   5. -13.   0.]
     [ -5.   1.   1.   4. -17.]]
```

2 Thuật toán cho công thức nội suy Stirling

2.1 Viết lại công thức

Để thuận lợi hơn cho việc thiết lập thuật toán, ta viết lại các số hạng tổng quát của $S(t)$ như sau

$$s_{2k-1} = \underbrace{\frac{\Delta^{2k-1}y_{-(k-1)} + \Delta^{2k-1}y_{-k}}{2(2k-1)!}}_{\alpha_{2k-1}} t\prod_{i=1}^{k-1}\left(t^2 - i^2\right)$$

$$s_{2k} = \underbrace{\frac{\Delta^{2k}y_{-k}}{(2k)!}}_{\alpha_{2k}} t^2\prod_{i=1}^{k-1}\left(t^2 - i^2\right)$$

Khi đó ta có thể biểu diễn $S(t)$ dưới dạng ma trận

$$S(t) = s_0 + (s_2 + s_4 + ... + s_{2n}) + (s_1 + s_3 + ... + s_{2n-1})$$

$$= s_0 + \underbrace{\begin{pmatrix}\alpha_2 & \alpha_4 & \cdots & \alpha_{2n}\end{pmatrix}}_{A_{\text{even}}} \begin{pmatrix} t^2 \\ t^2\left(t^2-1^2\right) \\ \vdots \\ t^2\prod_{i=1}^{n-1}\left(t^2-i^2\right)\end{pmatrix} + \underbrace{\begin{pmatrix}\alpha_1 & \alpha_3 & \cdots & \alpha_{2n-1}\end{pmatrix}}_{A_{\text{odd}}} \begin{pmatrix} t \\ t\left(t^2-1^2\right) \\ \vdots \\ t\prod_{i=1}^{n-1}\left(t^2-i^2\right)\end{pmatrix}$$

Bây giờ ta sẽ thiết lập một ma trận vuông cấp n, ký hiệu là S_n, với các hàng lần lượt là các tọa độ theo cơ sở $\left\{t^2, t^4, ..., t^{2n}\right\}$ của các đa thức t^2; $t^2\left(t^2-1^2\right)$; $t^2\left(t^2-1^2\right)\left(t^2-2^2\right)$; ...; $t^2\prod_{i=1}^{n-1}\left(t^2-i^2\right)$. Khi đó

$$\begin{pmatrix} t^2 \\ t^2\left(t^2-1^2\right) \\ \vdots \\ t^2\prod_{i=1}^{n-1}\left(t^2-i^2\right)\end{pmatrix} = S_n \begin{pmatrix} t^2 \\ t^4 \\ \vdots \\ t^{2n}\end{pmatrix}$$

Hơn nữa, ta còn có

$$\begin{pmatrix} t \\ t\left(t^2-1^2\right) \\ \vdots \\ t\prod_{i=1}^{n-1}\left(t^2-i^2\right)\end{pmatrix} = \frac{1}{t}\begin{pmatrix} t^2 \\ t^2\left(t^2-1^2\right) \\ \vdots \\ t^2\prod_{i=1}^{n-1}\left(t^2-i^2\right)\end{pmatrix} = \frac{1}{t}S_n\begin{pmatrix} t^2 \\ t^4 \\ \vdots \\ t^{2n}\end{pmatrix} = S_n\begin{pmatrix} t \\ t^3 \\ \vdots \\ t^{2n-1}\end{pmatrix}$$

Thay vào $S(t)$, ta được

$$S(t) = s_0 + A_{\text{even}}S_n\begin{pmatrix} t^2 \\ t^4 \\ \vdots \\ t^{2n}\end{pmatrix} + A_{\text{odd}}S_n\begin{pmatrix} t \\ t^3 \\ \vdots \\ t^{2n-1}\end{pmatrix}$$

2.2 Khởi tạo ma trận S_n

Giờ ta phải tìm cách để thiết lập ma trận S_n. Ta viết ra một vài ma trận S_i với $i = 1, 2, ...$

$$S_1 = \begin{pmatrix}1\end{pmatrix} \quad , \quad S_2 = \begin{pmatrix}1 & 0 \\ -1 & 1\end{pmatrix} \quad , \quad S_3 = \begin{pmatrix}1 & 0 & 0 \\ -1 & 1 & 0 \\ 4 & -5 & 1\end{pmatrix} \quad , \quad S_4 = \begin{pmatrix}1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 4 & -5 & 1 & 0 \\ -36 & 49 & -14 & 1\end{pmatrix}$$

Ta sẽ định nghĩa hàm roll$(H_{1\times n}, k)$ cho phép "dời" vị trí của các phần tử trong ma trận hàng $H_{1\times n}$ sang phải k nấc. Ví dụ

$$H_{1\times 4} = \begin{pmatrix} a & b & c & d \end{pmatrix} \to \text{roll}\left(H_{1\times 4}, 1\right) = \begin{pmatrix} d & a & b & c \end{pmatrix}$$
$$H_{1\times 5} = \begin{pmatrix} a & b & c & d & e \end{pmatrix} \to \text{roll}\left(H_{1\times 5}, 3\right) = \begin{pmatrix} c & d & e & a & b \end{pmatrix}$$

Ta có thể làm điều này dễ dàng bằng cách tương ứng các giá trị $\overline{0}, \overline{1}, \overline{2}, \ldots, \overline{n-1}$ trong nhóm $(\mathbb{Z}_n, +)$, lần lượt vào các phần tử trong $H_{1\times n}$ theo thứ tự từ trái sang phải (Các giá trị được gán này giống như vị trí của phần tử trong mảng). Khi đó nếu ta lấy $\text{roll}(H_{1\times n}, k)$ thì ta chỉ cần thực hiện cộng giá trị vị trí đó với $\overline{k}$. Ví dụ dưới đây sẽ giúp phần nào hiểu được rõ hơn

$$\begin{array}{lcccc} H_{1\times 4} = \begin{pmatrix} a & b & c & d \end{pmatrix} & \xrightarrow[+\overline{1}]{\text{roll}(\cdot,1)} & \begin{pmatrix} a & b & c & d \end{pmatrix} & \Leftrightarrow & \begin{pmatrix} d & a & b & c \end{pmatrix} \\ \text{index} \quad \overline{0}\ \overline{1}\ \overline{2}\ \overline{3} & (+\overline{1}) & \overline{1}\ \overline{2}\ \overline{3}\ \overline{0} & & \overline{0}\ \overline{1}\ \overline{2}\ \overline{3} \end{array}$$

Trường hợp này $k = 1$ và $n = 4$. Giá trị vị trí của từng phần tử sẽ được cộng thêm $\overline{1}$. Tức là phần tử a sẽ ở vị trí $\overline{0} + \overline{1} = \overline{1}$, phần tử b sẽ ở vị trí $\overline{1} + \overline{1} = \overline{2}$, phần tử c sẽ ở vị trí $\overline{2} + \overline{1} = \overline{3}$, phần tử d sẽ ở vị trí $\overline{3} + \overline{1} = \overline{4} = \overline{0}$

Hàm trên có thể được viết một cách dễ dàng với C, C++, Java, Python,.... Nhưng ở Python thì ta đã có sẵn hàm `roll()` của thư viện `numpy` có thể thực hiện được giống như hàm $\text{roll}(H_{1\times n}, k)$

```
import numpy
A = [1, 4, 7, 3, 8]
B = numpy.roll(A, 2)
print(B)
>>> B = [3, 8, 1, 4, 7]
```

Giờ ta sẽ định nghĩa hàm `Stirling_coef()` với đầu vào là một số nguyên dương biểu thị cỡ củ ma trận, đầu ra là ma trận S_n

```
import numpy
def Stirling_coef(num): # num là số nguyên dương, biểu thị cấp của ma trận
    S = numpy.identity(num) # Khởi tạo ma trận đơn vị, cỡ num x num
    # Quy luật: S[i] = -(i^2)*S[i-1] + roll(S[i-1], 1)
    for i in range(1, num):
        S[i] = numpy.add(
            [-x*i*i for x in S[i-1]], # Nhân hàng S[i-1] với -i^2
            numpy.roll(S[i-1], 1)
        )
  return S
```

Ta có thể kiểm tra được dễ dàng rằng quy luật được dùng trong hàm `Stirling_coef()` là đúng Chạy thử với một số giá trị của `num`

```
print(Stirling_coef(5)) # S_5
>>> [[   1.    0.    0.    0.    0.]
     [  -1.    1.    0.    0.    0.]
     [   4.   -5.    1.    0.    0.]
     [ -36.   49.  -14.    1.    0.]
     [ 576. -820.  273.  -30.    1.]]
print(Stirling_coef(6)) # S_6
>>> [[ 1.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00]
     [-1.0000e+00  1.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00]
     [ 4.0000e+00 -5.0000e+00  1.0000e+00  0.0000e+00  0.0000e+00  0.0000e+00]
     [-3.6000e+01  4.9000e+01 -1.4000e+01  1.0000e+00  0.0000e+00  0.0000e+00]
     [ 5.7600e+02 -8.2000e+02  2.7300e+02 -3.0000e+01  1.0000e+00  0.0000e+00]
     [-1.4400e+04  2.1076e+04 -7.6450e+03  1.0230e+03 -5.5000e+01  1.0000e+00]]
```

2.3 Thuật toán - Chương trình

Ta sẽ định nghĩa hàm `Stirling_interpolation()` để tính giá trị của $f(x)$ tại điểm $\overline{x}$. **Đầu vào** sẽ là giá trị $\overline{x}$ và bảng giá trị cho trước. Bảng giá trị bao gồm hai mảng một chiều `arr_x` và `arr_y` (Trong đó mảng `arr_x` gồm $2n+1$ phần tử cách đều và được xếp theo thứ tự tăng dần, đại diện cho $2n+1$ mốc nội suy), với $f\left(\texttt{arr\_x[}i\texttt{]}\right) = \texttt{arr\_y[}i\texttt{]}, \forall i = \overline{0, \pm n}$. **Đầu ra** sẽ là giá trị $f\left(\overline{x}\right)$, in kèm theo sai số của giá trị.

```
import numpy
from math import * # Thư viện chứa các hàm toán học
def Stirling_interpolation(arr_x, arr_y, x):
```

Bước 1: Kiểm tra xem giá trị $\overline{x}$ có nằm trong khoảng nội suy hay không

► Nếu có: Tiếp tục chạy chương trình

► Nếu không: Trả về `None`

```
    # Kiểm tra xem giá trị x có nàm ngoài khoảng nội suy hay không
    if x > arr_x[len(arr_x) - 1] or x < arr_x[0]:
        print("Giá trị x nằm ngoài khoảng nội suy")
        return None
```

Chú thích: Hàm `len(<array>)` sẽ trả về độ dài của mảng

Bước 2: Xác định bước nhảy h, mốc x_0 (Là mốc gần giá trị $\overline{x}$ nhất), từ đó xác định $t = \dfrac{\overline{x} - x_0}{h}$

Do các mốc nội suy là cách đều nên ta có thể tận dụng điều này để xác định **chỉ số** `x0` của mốc x_0 trước, khi đó $x_0 = \texttt{arr\_x[x0]}$. Từ mốc x_0, ta sẽ trích ra tối đa 9 mốc nội suy thích hợp

```
    h = arr_x[1] - arr_x[0] # Bước nhảy
    x0 = round((x - arr_x[0]) / h) # Chỉ số của giá trị hoành độ gần x nhất
    max_milestone = min(2*min(len(arr_x)-1-x0, x0) + 1, 9) # Số mốc tối đa
    t = (x - arr_x[x0]) / h  # Biến phụ trong công thức nội suy Stirling
    t_square = t*t # Lưu lại giá trị t^2 để máy không phải thực hiện t*t nhiều lần
```

Bước 3: Ghép hai mảng `arr_x` và `arr_y` vào nhau để tạo ra bảng số `table` (`table` sẽ là mảng 2 chiều), sau đó sử dụng hàm `roll()` (Đã đề cập ở phía trên) để chuyển chỉ số của tất cả các mảng con trong bảng số, sao cho `table[0]` $= [x_0, y_0]$. Làm như vậy để ta có thể tận dụng được chỉ số mảng âm (Vì trong công thức nội suy ta có các giá trị $x_{-3}, y_{-2}, ...$ cần tới chỉ số âm)

```
    # Ghép hai mảng arr_x và arr_y vào nhau
    table = [list(a) for a in zip(arr_x, arr_y)]
    # Dời vị trí của  [x_0, y_0] về chỉ số 0 trong mảng
    table = numpy.roll(table, -x0, axis = 0)
```

Chú thích: Hàm `zip(<array>,<array>)` dùng để ghép hai mảng lại với nhau, nhưng chưa có định dạng `list`, do đó ta sẽ ép kiểu `list()`

Bước 4: Khởi tạo các ma trận cột

$$\texttt{T\_odd} = \begin{pmatrix} t \\ t^3 \\ \vdots \\ t^{2n-1} \end{pmatrix} \quad ; \quad \texttt{T\_even} = \begin{pmatrix} t^2 \\ t^4 \\ \vdots \\ t^{2n} \end{pmatrix} = t.\texttt{T\_odd}$$

```
    # Khởi tạo ma trận cột T_odd chứa các biến t^(2i-1)
    # T_odd = transpose(t t^3 t^5 ... t^(2n-1))
    T_odd = numpy.full((max_milestone//2, 1), t)
    for i in range(max_milestone//2 - 1):
        T_odd[i+1][0] = T_odd[i][0]*t_square
    # Khởi tạo ma trận cột T_even chứa các biến t^(2i)
    # T_even = transpose(t^2 t^4 ... t^(2n))
    T_even = [i*t for i in T_odd]
```

Chú thích: Hàm `full((<num>,<num>), <num>)` dùng để tạo ra một ma trận với kích cỡ bất kỳ, và tất cả các phần tử đều là một số tùy chọn. Hàm này thuộc thư viện `numpy`

Bước 5: Lập bảng sai phân `d_tab` bằng các hàm `Above_add()` và `Below_add()` (Các hàm này đã được trình bày ở phía trên), đồng thời khởi tạo các ma trận hàng $A_{\text{odd}}, A_{\text{even}}$

$$\texttt{A\_odd} = \begin{pmatrix} \alpha_2 & \alpha_4 & \cdots & \alpha_{2n} \end{pmatrix} \quad ; \quad \texttt{A\_even} = \begin{pmatrix} \alpha_1 & \alpha_3 & \cdots & \alpha_{2n-1} \end{pmatrix}$$

Quy trình:

▶ Xuất phát bằng các mảng $\begin{cases} \texttt{d\_tab = [[}\, y_0 \,\texttt{]]} \\ \texttt{A\_odd = [[ ]]} \\ \texttt{A\_even = [[ ]]} \end{cases}$

▶ Bắt đầu vòng lặp, mỗi một lần lặp ta thêm một mốc vào đầu bảng bằng `Above_add()`, một mốc vào cuối bảng bằng `Below_add()` (Chú ý rằng ma trận bảng sai phân luôn có cấp lẻ). Giả sử ma trận bảng sai phân đang có cấp $2m+1$. Khi đó:

- Bổ sung giá trị $\frac{1}{(2m+1)!}$.`d_tab[`$2m$`][`$2m$`]` vào bên phải `A_even`
- Bổ sung giá trị $\frac{1}{2(2m)!}$ (`d_tab[`$2m-1$`][`$2m-1$`]` + `d_tab[`$2m$`][`$2m-1$`]`) vào bên phải `A_odd`

Vòng lặp kết thúc khi đã dùng tối đa số mốc nội suy

(*Chú ý*: Nếu bảng sai phân là ma trận cấp $2m+1$ thì `d_tab[2m][2m]` là phần tử nằm ở góc dưới cùng bên phải bảng sai phân, không có `d_tab[2m+1][2m+1]`)

```
    # Lập bảng sai phân, đồng thời khởi tạo các ma trận hàng A_even và A_odd
    # A_even = (a_2 a_4 ... a_(2n))
    # A_odd = (a_1 a_3 ... a_(2n-1))
    A_even = [[ ]]
    A_odd = [[ ]]
    d_tab = [[table[0][1]]] # Bảng sai phân dưới dạng ma trận tam giác dưới
    for i in range(1, max_milestone//2 + 1):
        d_tab = Above_add(d_tab, table[i][1])
        d_tab = Below_add(d_tab, table[-i][1])
        l = len(d_tab) # Cấp của ma trận bảng sai phân
        A_odd = numpy.append(
                A_odd, [[(d_tab[l-2][l-2] + d_tab[l-1][l-2])/2/factorial(2*i-1)]], axis = 1
        )
        A_even = numpy.append(A_even, [[del_table[l-1][l-1]/factorial(2*i)]], axis = 1)
```

Chú thích: Hàm `factorial()` dùng để tính giai thừa của một số nguyên dương. Hàm này thuộc thư viện `math`

Bước 6: Tính $f(\overline{x})$ bằng công thức

$$f(\overline{x}) = S(t) = y_0 + \texttt{A\_even.S.T\_even + A\_odd.S.T\_odd}$$

Trong đó `S = Stirling_coef(n)` (Hàm `Stirling_coef()` đã được trình bày ở trên)

```
    # Công thức nội suy Stirling: S(t) = A.S.T
    S = Stirling_coef(max_milestone//2)
    value = table[0][1] + numpy.add(A_even @ S @ T_even, A_odd @ S @ T_odd)[0][0]
```

Chú thích: Toán tử `@` (a còng) dùng để nhân ma trận trong Python. Hàm `add()` dùng để cộng hai ma trận cùng cỡ, hàm này thuộc thư viện `numpy`

Bước 7: Trả về giá trị $f(\overline{x})$ và in kèm theo sai số của giá trị. Sai số được tính theo công thức

$$|R(t)| \leq \left| \frac{\Delta^{2n} y_{-n}}{(2n)!}\, t \prod_{i=1}^{n} \left(t^2 - i^2\right) \right|$$

Giá trị $\dfrac{\Delta^{2n} y_{-n}}{(2n)!}$ nằm ở tận cùng bên phải của ma trận hàng `A_even`, nên ta có thể gọi giá trị đó ra bằng `A_even[0][len(A_even)]`

```
    # Tính toán sai số
    error = A_even[0][len(A_even[0])-1]
    for i in range(1, (max_milestone + 1)//2):
        error *= (t_square - i*i)
    print("Sai số |R(x)| <", abs(error), "\n")
    print("Have a good day!! :D\n")
    # Trả về giá trị
    return value
```

2.4 Chạy thử chương trình

Ta sẽ lập ra một bảng giá trị của hàm số

$$y = \ln\left(x^2 + 1\right) + \sin 3x$$

với các mốc nội suy nằm trong đoạn $\left[0, \dfrac{5}{2}\right]$, bước nhảy $h = \dfrac{1}{4}$

```
arr_x = [0.000, 0.250, 0.500, 0.750, 1.000, 1.250, 1.500, 1.750, 2.000, 2.250, 2.500]
arr_y = [0.000, 0.074, 0.249, 0.486, 0.745, 1.006, 1.257, 1.493, 1.713, 1.920, 2.112]
```

Ta sẽ thử với $\overline{x} = 3.22$. Nhận thấy rằng $\overline{x} = 3.22 \notin \left[0, \dfrac{5}{2}\right]$ nên đầu ra mong đợi sẽ là `None`

```
Stirling_interpolation(arr_x, arr_y, 3.22)
>>> Giá trị x nằm ngoài khoảng nội suy
```

Ta thử với giá trị khác, $\overline{x} = 1.274$

```
Stirling_interpolation(arr_x, arr_y, 1.274)
>>> Ma trận chuyển vị của T_odd
     [[9.60000000e-02 8.84736000e-04 8.15372698e-06 7.51447478e-08]]
>>> Ma trận chuyển vị của T_even
     [[9.21600000e-03 8.49346560e-05 7.82757790e-07 7.21389579e-09]]
>>> Ma trận A_odd
     [[2.56000000e-01 -1.41666667e-03 -1.66666667e-05 -7.93650794e-07]]
>>> Ma trận A_even
     [[-5.00000000e-03  2.91666667e-04 -2.77777778e-06  4.46428571e-07]]
>>> Bảng sai phân
     [[ 1.92000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
        0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 1.71300000e+00  2.07000000e-01  0.00000000e+00  0.00000000e+00
        0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 1.49300000e+00  2.20000000e-01 -1.30000000e-02  0.00000000e+00
        0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 1.25700000e+00  2.36000000e-01 -1.60000000e-02  3.00000000e-03
        0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 1.00600000e+00  2.51000000e-01 -1.50000000e-02 -1.00000000e-03
        4.00000000e-03  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 7.45000000e-01  2.61000000e-01 -1.00000000e-02 -5.00000000e-03
        4.00000000e-03  1.66533454e-15  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 4.86000000e-01  2.59000000e-01  2.00000000e-03 -1.20000000e-02
        7.00000000e-03 -3.00000000e-03  3.00000000e-03  0.00000000e+00  0.00000000e+00]
      [ 2.49000000e-01  2.37000000e-01  2.20000000e-02 -2.00000000e-02
        8.00000000e-03 -1.00000000e-03 -2.00000000e-03  5.00000000e-03  0.00000000e+00]
      [ 7.40000000e-02  1.75000000e-01  6.20000000e-02 -4.00000000e-02
        2.00000000e-02 -1.20000000e-02  1.10000000e-02 -1.30000000e-02  1.80000000e-02]]
>>> Ma trận S
     [[  1.   0.   0.   0.]
      [ -1.   1.   0.   0.]
      [  4.  -5.   1.   0.]
      [-36.  49. -14.   1.]]
>>> Giá trị của f(x) tại x = 1.274 là  1.0306581380462374
>>> Sai số |R(x)| < 0.0002537794837883185
```

Ở lần này code đã được chỉnh sửa để in ra các giá trị trung gian (Bảng sai phân, `A_odd`, `A_even`, `T_odd`, `T_even`, `Stirling_coef()`) để theo dõi kỹ hơn.

Với máy tính cầm tay, ta tính được giá trị $y(1.274) = 1.031004699$, độ chênh lệnh của giá trị này với giá trị tính được trong máy là 0.00034656. Vậy tức là sai số in ra của máy bị sai? Thật ra sai số của máy in ra còn phụ thuộc vào bộ dữ liệu đầu vào. Bản thân bộ dữ liệu này cũng đã bị "xấp xỉ" các giá trị y_i rồi, nên sai số có thể cũng sẽ bị chênh lệch

3 Thuật toán cho công thức nội suy Bessel

3.1 Viết lại công thức

Từ công thức nội suy Bessel, đặt $u = t - \frac{1}{2}$, khi đó ta được đa thức mới $B\left(u + \frac{1}{2}\right)$, với các số hạng tổng quát

$$\begin{aligned} b_{2k} &= \frac{\Delta^{2k} y_{-k} + \Delta^{2k} y_{-(k-1)}}{2} \left(u + k - \frac{1}{2}\right)\left(u + k - \frac{3}{2}\right)(...)\left(u - k + \frac{1}{2}\right) \\ &= \underbrace{\frac{\Delta^{2k} y_{-k} + \Delta^{2k} y_{-(k-1)}}{2}}_{\alpha_{2k}} \prod_{i=0}^{k-1} \left(u^2 - \left(i + \frac{1}{2}\right)^2\right) \end{aligned}$$

$$\begin{aligned} b_{2k+1} &= \frac{\Delta^{2k+1} y_{-k}}{(2k+1)!}\, u.\left(u + k - \frac{1}{2}\right)\left(u + k - \frac{3}{2}\right)(...)\left(u - k + \frac{1}{2}\right) \\ &= \underbrace{\frac{\Delta^{2k-1} y_{-(k-1)}}{(2k-1)!}}_{\alpha_{2k-1}}\, u \prod_{i=0}^{k-1} \left(u^2 - \left(i + \frac{1}{2}\right)^2\right) \end{aligned}$$

Khi đó ta có thể biểu diễn $B\left(u + \frac{1}{2}\right)$ dưới dạng ma trận

$$B\left(u + \frac{1}{2}\right) = (b_0 + b_2 + b_4 + ... + b_{2n}) + (b_1 + b_3 + ... + b_{2n+1})$$

$$= \underbrace{\begin{pmatrix} \alpha_0 & \alpha_2 & \cdots & \alpha_{2n} \end{pmatrix}}_{A_{\text{even}}} \begin{pmatrix} 1 \\ u^2 - \left(\frac{1}{2}\right)^2 \\ \vdots \\ \prod_{i=0}^{n-1} \left(u^2 - \left(i + \frac{1}{2}\right)^2\right) \end{pmatrix} + \underbrace{\begin{pmatrix} \alpha_1 & \alpha_3 & \cdots & \alpha_{2n+1} \end{pmatrix}}_{A_{\text{odd}}} \begin{pmatrix} u \\ u\left(u^2 - \left(\frac{1}{2}\right)^2\right) \\ \vdots \\ u\prod_{i=0}^{n-1} \left(u^2 - \left(i + \frac{1}{2}\right)^2\right) \end{pmatrix}$$

Bay giờ ta sẽ thiết lập một ma trận vuông cấp $n+1$, ký hiệu là B_{n+1}, với các hàng lần lượt là các tọa độ theo cơ sở $\left\{1, u^2, ..., u^{2n}\right\}$ của các đa thức $1\,; u^2 - \left(\frac{1}{2}\right)^2\,; ...\,; \prod_{i=0}^{n-1}\left(u^2 - \left(i + \frac{1}{2}\right)^2\right)$

Khi đó

$$\begin{pmatrix} 1 \\ u^2 - \left(\frac{1}{2}\right)^2 \\ \vdots \\ \prod_{i=0}^{n-1} \left(u^2 - \left(i + \frac{1}{2}\right)^2\right) \end{pmatrix} = B_{n+1} \begin{pmatrix} 1 \\ u^2 \\ \vdots \\ u^{2n} \end{pmatrix}$$

Hơn nữa, ta còn có

$$\begin{pmatrix} u \\ u\left(u^2-\left(\frac{1}{2}\right)^2\right) \\ \vdots \\ u\prod_{i=0}^{n-1}\left(u^2-\left(i+\frac{1}{2}\right)^2\right) \end{pmatrix} = u\begin{pmatrix} 1 \\ \left(u^2-\left(\frac{1}{2}\right)^2\right) \\ \vdots \\ \prod_{i=0}^{n-1}\left(u^2-\left(i+\frac{1}{2}\right)^2\right) \end{pmatrix} = uB_{n+1}\begin{pmatrix} 1 \\ u^2 \\ \vdots \\ u^{2n} \end{pmatrix} = B_{n+1}\begin{pmatrix} u \\ u^3 \\ \vdots \\ u^{2n+1} \end{pmatrix}$$

Thay vào $B\left(u+\frac{1}{2}\right)$, ta được

$$B'(u) = B\left(u+\frac{1}{2}\right) = A_{\text{even}}B_{n+1}\begin{pmatrix} 1 \\ u^2 \\ \vdots \\ u^{2n} \end{pmatrix} + A_{\text{odd}}B_{n+1}\begin{pmatrix} u \\ u^3 \\ \vdots \\ u^{2n+1} \end{pmatrix}$$

Phép đặt ẩn u này cũng khiến công thức sai số bị thay đổi. Ta sẽ viết lại công thức sai số

$$\left|R\left(\bar{t}\right)\right| = \left|R'\left(\overline{u}\right)\right| \leq \left|\frac{\Delta^{2n+1}y_{-n}}{(2n+1)!}\prod_{i=0}^{n}\left(\overline{u}^2-\left(i+\frac{1}{2}\right)^2\right)\right|$$

3.2 Khởi tạo ma trận B_n

Tương tự như cách ta khởi tạo ma trận S_n, ta định nghĩa hàm `Bessel_coef()` với đầu vào là một số nguyên dương, biểu thị cỡ của ma trận, đầu ra là ma trận B_n

```
import numpy
def Bessel_coef(num):
    B = numpy.identity(num) # Khởi tạo ma trận đơn vị, cỡ num x num
    for i in range(1, num):
        B[i] = numpy.add(
            [-x* (i-0.5)**2 for x in B[i-1]],
            numpy.roll(B[i-1], 1)
        )
    return B
```

Chạy thử với một số giá trị của `num`

```
print(Bessel_coef(3)) # B_3
>>> [[ 1.      0.      0.    ]
     [-0.25    1.      0.    ]
     [ 0.5625 -2.5     1.    ]]
```

```
print(Bessel_coef(5)) # B_5
>>> [[   1.           0.           0.           0.           0.       ]
     [  -0.25         1.           0.           0.           0.       ]
     [   0.5625      -2.5          1.           0.           0.       ]
     [  -3.515625    16.1875      -8.75         1.           0.       ]
     [  43.06640625 -201.8125     123.375      -21.          1.       ]]
```

3.3 Thuật toán - Chương trình

Cũng tương tự như hàm `Stirling_interpolation()`, ta sẽ định nghĩa hàm `Bessel_interpolation()` để tính giá trị của $f(x)$ tại điểm $\overline{x}$. Đầu vào và đầu ra của hai hàm này khá tương đồng, chỉ khác một chỗ là các mảng `arr_x` và `arr_y` có độ dài $2n+2$

```
import numpy
from math import * # Thư viện chứa các hàm toán học
def Bessel_interpolation(arr_x, arr_y, x):
```

Bước 1: Kiểm tra xem $\overline{x}$ có nằm trong khoảng nội suy hay không

▶ Nếu có: Tiếp tục chạy chương trình

▶ Nếu không: Trả về `None`

```
    # Kiểm tra xem giá trị x có nàm ngoài khoảng nội suy hay không
    if x > arr_x[len(arr_x) - 1] or x < arr_x[0]:
        print("Giá trị x nằm ngoài khoảng nội suy")
        return None
```

Bước 2: Xác định bước nhảy h, mốc x_0 (Là mốc bên trái $\overline{x}$ và gần $\overline{x}$ nhất). Từ đó xác định $u = \dfrac{\overline{x} - x_0}{2} - \dfrac{1}{2}$. Do các mốc nội suy là cách đều nên ta có thể tận dụng điều này để xác định **chỉ số** `x0` của mốc x_0 trước, khi đó x_0 = `arr_x[x0]`. Từ mốc x_0, ta sẽ trích ra tối đa 8 mốc nội suy thích hợp

```
    h = arr_x[1] - arr_x[0] # Bước nhảy
    x0 = int(floor((x - arr_x[0]) / h)) # Chỉ số của giá trị hoành độ nằm bên trái x
    max_milestone = int(min(2*min(x0+1, len(arr_x)-1-x0), 8)) # Số mốc tối đa
    u = (x - arr_x[x0]) / h - 0.5  # Biến phụ trong công thức nội suy Bessel
    u_square = u*u # Lưu lại giá trị u^2 để máy không phải thực hiện u*u nhiều lần
```

Bước 3: Ghép hai mảng `arr_x` và `arr_y` vào nhau để tạo ra bảng số `table` (`table` sẽ là mảng 2 chiều), sau đó sử dụng hàm `roll()` (Đã đề cập ở phía trên) để chuyển chỉ số của tất cả các mảng

con trong bảng số, sao cho `table[0]` $= [x_0, y_0]$. Làm như vậy để ta có thể tận dụng được chỉ số mảng âm (Vì trong công thức nội suy ta có các giá trị $x_{-3}, y_{-2}, ...$ cần tới chỉ số âm)

```
    # Ghép hai mảng arr_x và arr_y vào nhau
    table = [list(a) for a in zip(arr_x, arr_y)]
    # Dời vị trí của [x_0, y_0] về chỉ số 0 trong mảng
    table = numpy.roll(table, -x0, axis = 0)
```

Bước 4: Khởi tạo các ma trận cột

$$\texttt{U\_odd} = \begin{pmatrix} u \\ u^3 \\ \vdots \\ u^{2n+1} \end{pmatrix} \quad ; \quad \texttt{U\_even} = \begin{pmatrix} 1 \\ u^2 \\ \vdots \\ u^{2n} \end{pmatrix}$$

```
    # Khởi tạo ma trận cột U_even chứa các biến u^(2i)
    # U_even = transpose(1 u^2 u^4 ... u^(2n))
    U_even = numpy.full((max_milestone//2, 1), 1.0)
    for i in range(max_milestone//2 - 1):
        U_even[i+1][0] = U_even[i][0]*u_square
    # Khởi tạo ma trận cột U_even chứa các biến u^(2i+1)
    # U_even = transpose(u u^3 u^5 ... u^(2n+1))
    U_odd = [i*u for i in U_even]
```

Bước 5: Lập bảng sai phân `d_tab` bằng các hàm `Above_add()` và `Below_add()` (Các hàm này đã được trình bày ở phía trên), đồng thời khởi tạo các ma trận hàng $A_{\text{odd}}, A_{\text{even}}$

$$\texttt{A\_even} = \begin{pmatrix} \alpha_0 & \alpha_2 & \cdots & \alpha_{2n} \end{pmatrix} \quad ; \quad \texttt{A\_odd} = \begin{pmatrix} \alpha_1 & \alpha_3 & \cdots & \alpha_{2n-1} \end{pmatrix}$$

Quy trình:

► Xuất phát bằng bảng sai phân `d_tab = [[`y_0`]]`, thực hiện thêm một mốc y_1 lên đầu bảng bằng `Above_add()`

$$\begin{pmatrix} y_0 \end{pmatrix} \longrightarrow \begin{pmatrix} y_1 & 0 \\ y_0 & \Delta y_0 \end{pmatrix}$$

► Khởi tạo các mảng $\begin{cases} \texttt{A\_even = [[} \dfrac{y_0 + y_1}{2} \texttt{]]} \\ \texttt{A\_odd = [[}\Delta y_0\texttt{]]} \end{cases}$

► Bắt đầu vòng lặp, mỗi một lần lặp ta thêm một mốc vào đầu bảng bằng `Above_add()`, một mốc vào cuối bảng bằng `Below_add()` (Chú ý rằng ma trận bảng sai phân luôn có cấp chẵn). Giả sử ma trận bảng sai phân lúc này đang có cấp $2m$. Khi đó:

- Bổ sung giá trị $\frac{1}{2(2m)!}$ (`d_tab[2m-2][2m-2]` + `d_tab[2m-1][2m-2]`) vào bên phải `A_even`
- Bổ sung giá trị $\frac{1}{(2m-1)!}$`d_tab[2m-1][2m-1]` vào bên phải `A_odd`

Vòng lặp kết thúc khi đã sử dụng tối đa số mốc nội suy

```
    # Khởi tạo ma trận hàng chứa các hệ số a_i
    # A = (a_0 a_1 a_2 a_3 ... a_n)
    A_even = [[(table[0][1] + table[1][1])/2]]
    d_tab = [[table[0][1]]]  # Bảng sai phân dưới dạng ma trận tam giác dưới
    d_tab = Above_add(d_tab, table[1][1])
    A_odd = [[d_tab[1][1]]]
    for i in range(1, max_milestone//2):
        d_tab = Above_add(d_tab, table[i+1][1])
        d_tab = Below_add(d_tab, table[-i][1])
        l = len(d_tab)
        A_even = numpy.append(
                 A_even, [[(d_tab[l-2][l-2] + d_tab[l-1][l-2])/2/factorial(2*i)]], axis = 1
        )
        A_odd = numpy.append(A_odd, [[d_tab[l-1][l-1] / factorial(2*i+1)]], axis = 1)
```

Bước 6: Tính $f(\overline{x})$ bằng công thức

$$f(\overline{x}) = B\left(u + \frac{1}{2}\right) = \texttt{A\_even.B.U\_even+A\_odd.B.T\_odd}$$

Trong đó `B=Bessel_coef(`$n+1$`)` (Hàm `Bessel_coef()` đã được trình bày ở trên)

```
    # Công thức nội suy Bessel: B(u+0.5) = A.B.U
    B = Bessel_coef(max_milestone//2)
    value = numpy.add(A_even @ B @ U_even, A_odd @ B @ U_odd)[0][0]
```

Bước 7: Trả về giá trị $f(\overline{x})$ và in kèm theo sai số của giá trị. Sai số được tính theo công thức

$$\left|R(\overline{u})\right| \leq \left|\frac{\Delta^{2n+1}y_{-n}}{(2n+1)!}\prod_{i=0}^{n}\left(\overline{u}^2 - \left(i+\frac{1}{2}\right)^2\right)\right|$$

Giá trị $\frac{\Delta^{2n+1}y_{-n}}{(2n+1)!}$ nằm ở tận cùng bên phải của ma trận hàng `A_odd`, nên ta có thể gọi giá trị đó ra bằng `A_odd[0][len(A_odd)]`

```
    # Tính toán sai số
    error = A_odd[0][len(A_odd[0])-1]
    for i in range(max_milestone//2):
        error *= (u_square - (i+0.5)**2)
    print("Sai số |R(x)| < ", abs(error), "\n")
```

```
    # Trả về giá trị
    return value
```

3.4 Chạy thử chương trình

Ta vẫn sẽ sử dụng bảng dữ liệu đầu vào như đã làm với hàm `\Stirling_interpolation()`, sau đó so sánh hai kết quả với nhau

```
Bessel_interpolation(arr_x, arr_y, 1.274)
>>> Ma trận chuyển vị của U_odd
     [[-0.404      -0.06593926 -0.01076234 -0.00175659]]
>>> Ma trận chuyển vị của U_even
     [[1.         0.163216   0.02663946 0.00434799]]
>>> Bảng sai phân
     [[ 1.92000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
        0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 1.71300000e+00  2.07000000e-01  0.00000000e+00  0.00000000e+00
        0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 1.49300000e+00  2.20000000e-01 -1.30000000e-02  0.00000000e+00
        0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 1.25700000e+00  2.36000000e-01 -1.60000000e-02  3.00000000e-03
        0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 1.00600000e+00  2.51000000e-01 -1.50000000e-02 -1.00000000e-03
        4.00000000e-03  0.00000000e+00  0.00000000e+00  0.00000000e+00]
      [ 7.45000000e-01  2.61000000e-01 -1.00000000e-02 -5.00000000e-03
        4.00000000e-03  1.66533454e-15  0.00000000e+00  0.00000000e+00]
      [ 4.86000000e-01  2.59000000e-01  2.00000000e-03 -1.20000000e-02
        7.00000000e-03 -3.00000000e-03  3.00000000e-03  0.00000000e+00]
      [ 2.49000000e-01  2.37000000e-01  2.20000000e-02 -2.00000000e-02
        8.00000000e-03 -1.00000000e-03 -2.00000000e-03  5.00000000e-03]]
>>> Ma trận A_odd
     [[ 2.51000000e-01 -8.33333333e-04 -2.50000000e-05  9.92063492e-07]]
>>> Ma trận A_even
     [[ 1.13150000e+00 -6.25000000e-03  2.29166667e-04  6.94444444e-07]]
>>> Ma trận B
     [[ 1.         0.         0.         0.       ]
      [-0.25       1.         0.         0.       ]
      [ 0.5625    -2.5        1.         0.       ]
      [-3.515625 16.1875    -8.75       1.       ]]
>>> Giá trị của f(x) tại x = 1.274 là  1.0306521900895862
>>> Sai số |R(x)| <  1.3217681447316492e-05
```

Nhìn chung, cả hai công thức nội suy Stirling và Bessel đều cho kết quả tương đối chính xác.

IV Kết luận

1 Ưu điểm

► Việc lấy một mốc nội suy gần nhất làm mốc đầu tiên, sau đó quét qua các mốc nội suy có tính đối xứng với mốc nội suy trung tâm nên bất cứ giá trị nào cũng có thể được quét qua, tăng độ tin tưởng cho kết quả đầu ra

► Việc trích ra số lượng mốc nội suy vừa phải (Đối với Stirling tối đa là 9, còn với Bessel tối đa là 8) đã giảm thiểu được phần nào khối lượng tính toán, mà vẫn đem lại kết quả có độ chính xác cao

► Công thức nội suy trung tâm thường được ưu tiên sử dụng đối với những bài toán xấp xỉ hàm số có mốc cách đều, không như những phương pháp khác, chỉ ưu tiên về một phía

► Phương pháp đưa về tính toán ma trận đã được trình bày ở trên giúp cho đa thức nội suy dễ kiểm soát hơn, không như cách tính bằng đệ quy thông thường. Chẳng hạn nếu ta muốn tính gần đúng đạo hàm hoặc tích phân của một hàm số, ta có thể xấp xỉ hàm số đó bằng một đa thức, rồi đạo hàm (tích phân) đa thức đó một cách dễ dàng.

(Tính đạo hàm)

$$\left.\frac{df(x)}{dx}\right|_{x=\overline{x}} \approx \left.\frac{dS\left(x_0+ht\right)}{dt}.\frac{dt}{dx}\right|_{t=\overline{t}} = \left.\frac{1}{h}.\frac{dS\left(x_0+ht\right)}{dt}\right|_{t=\overline{t}}$$

(Tính tích phân)

$$\int_a^b f(x)dx = \left(\int_a^x f(x)dx\right)\Bigg|_{x=b} \approx h\left(\int_{\frac{a-x_0}{h}}^{t} S(x_0+ht)dt\right)\Bigg|_{t=\frac{b-x_0}{h}}$$

2 Nhược điểm

► Công thức nội suy không xử lý được đối với trường hợp giá trị $\overline{x}$ nằm ở (gần) đầu bảng

► Phương pháp đưa về ma trận tuy lợi về tính toán nhưng thiệt nhiều về bộ nhớ

V Tài liệu tham khảo

[1] Lê Trọng Vinh. *Giáo trình Giải tích số*. Nhà xuất bản Khoa học và Kỹ thuật. 2007

[2] Ake Björck. *Numerical Method in Matrix Computation*. Springer. 2016