

Divided Difference Algorithm

a pseudocode for computing divided difference is

```
real array  $(a_{ij})_{0:n \times 0:n}$ ,  $(x_i)_{0:n}$ 
integer  $i, j, n$ 
for  $i = 0$  to  $n$  do
     $a_{i0} \leftarrow f(x_i)$ 
end for
for  $j = 1$  to  $n$  do
    for  $i = 0$  to  $n - j$  do
         $a_{ij} \leftarrow (a_{i+1,j-1} - a_{i,j-1}) / (x_{i+j} - x_i)$ 
    end for
end for
```

this algorithm computes and stores all components of the divided difference. The coefficients of the Newton interpolating polynomial are stored in the first row of the array $(a_{ij})_{0:n \times 0:n}$, i.e., in $a(0 : n, 0)$

Computing the Coefficients only

if we compute divided difference only for constructing Newton interpolating polynomial, there is no need to store the unnecessary divided difference terms. (But they will be computed, used, and discarded.)

```
real array  $(a_i)_{0:n}, (x_i)_{0:n}$   
integer  $i, j, n$   
for  $i = 0$  to  $n$  do  
     $a_i \leftarrow f(x_i)$   
end for  
for  $j = 1$  to  $n$  do  
    for  $i = n$  to  $j$  step  $-1$  do  
         $a_i \leftarrow (a_i - a_{i-1}) / (x_i - x_{i-j})$   
    end for  
end for
```

the two algorithms assume the same computational cost

Memory Allocations

here we show how the memory is occupied and updated in computing coefficients of Newton interpolating polynomial

1st	$f[x_0]$	$f[x_1]$	$f[x_2]$	$f[x_3]$	$f[x_4]$
2nd					
3rd					
4th					
5th					

computation must be done backward to avoid erasing needed memory locations

Inverse Interpolation

it is also possible to use a polynomial to approximate the inverse of a function $y = f(x)$. Given a table

y	y_0	y_1	\dots	y_n
x	x_0	x_1	\dots	x_n

an interpolation polynomial

$$p(y) = \sum_{i=0}^n c_i \prod_{j=0}^{i-1} (y - y_j)$$

can be constructed such that $p(y_i) = x_i$. This interpolating polynomial is useful to find the approximate location of a root of a function $f(x)$. E.g., there is a root in $[4.0, 5.0]$ for this table

y	-0.579	-0.363	-0.185	-0.034	0.097
x	1.0	2.0	3.0	4.0	5.0

Neville's Algorithm

Neville proposed a different scheme to construct interpolation polynomial step by step. Start with zero degree polynomials $P_i(x) = f(x_i)$, we construct higher degree interpolation polynomials by the recurrence relation

$$S_{ij}(x) = \left(\frac{x - x_{i-j}}{x_i - x_{i-j}} \right) S_{i,j-1}(x) + \left(\frac{x_i - x}{x_i - x_{i-j}} \right) S_{i-1,j-1}(x)$$

with $S_{i0}(x) = P_i(x) = f(x_i)$. The relation table can be written as

x_0	$S_{00}(x)$				
x_1	$S_{10}(x)$	$S_{11}(x)$			
x_2	$S_{20}(x)$	$S_{21}(x)$	$S_{22}(x)$		
x_3	$S_{30}(x)$	$S_{31}(x)$	$S_{32}(x)$	$S_{33}(x)$	
x_4	$S_{40}(x)$	$S_{41}(x)$	$S_{42}(x)$	$S_{43}(x)$	$S_{44}(x)$

Interpolation Property

redefine constant polynomials as $P_i^{(0)}(x) = y_i$ for $0 \leq i \leq n$, we can define higher order polynomial as

$$P_i^{(j)}(x) = \left(\frac{x - x_{i-j}}{x_i - x_{i-j}} \right) P_i^{(j-1)}(x) + \left(\frac{x_i - x}{x_i - x_{i-j}} \right) P_{i-1}^{(j-1)}(x)$$

the range of j is $1 \leq j \leq n$ and that of i is $j \leq i \leq n$

the interpolation properties of these polynomials are:

The polynomial $P_i^{(j)}$ defined above interpolate as follows (see p. 153 for a proof)

$$P_i^{(j)}(x_k) = y_k \quad (0 \leq i - j \leq k \leq i \leq n)$$

Higher Dimensional Interpolation

it is possible to define interpolation polynomials of several variables. The **tensor-product interpolation** is used on a rectangular domain $[a, b] \times [\alpha, \beta]$. Select n nodes in $[a, b]$ and define the Lagrange polynomials as

$$l_i(x) = \prod_{j \neq i, j=1}^n \frac{x - x_j}{x_i - x_j} \quad (1 \leq i \leq n).$$

Select m nodes in $[\alpha, \beta]$ and define

$$h_i(y) = \prod_{j \neq i, j=1}^m \frac{y - y_j}{y_i - y_j} \quad (1 \leq i \leq m).$$

then function

$$P(x, y) = \sum_{i=1}^n \sum_{j=1}^m f(x_i, y_j) l_i(x) h_j(y)$$

a two dimensional table with data

$$(x_i, y_j, f(x_i, y_j))$$

Computing First Derivative

the first derivative can be approximated as

$$f'(x) \approx \frac{1}{h}[f(x+h) - f(x)] \quad (1)$$

for accurate approximation, h should be small. Thus $f(x+h)$ and $f(x)$ are close to each other. This may cause loss of significant digits in finite precision computation

using Taylor's theorem, we have

$$f(x+h) = f(x) + h f'(x) + \frac{1}{2} h^2 f''(\xi)$$

for ξ between x and $x+h$. It follows that

$$f'(x) = \frac{1}{h}[f(x+h) - f(x)] - \frac{1}{2} h f''(\xi)$$

the approximation error of (1) is $-\frac{1}{2} h f''(\xi)$, or of order $O(h)$. This is a first order (or sided) approximation of first derivative. The error goes to 0 as fast as $h \rightarrow 0$

Higher Order Approximation

it is desirable to have some higher order (faster) approximation schemes

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2!}h^2f''(x) + \frac{1}{3!}h^3f'''(x) + \frac{1}{4!}h^4f^{(4)}(x) + \dots$$

$$f(x-h) = f(x) - hf'(x) + \frac{1}{2!}h^2f''(x) - \frac{1}{3!}h^3f'''(x) + \frac{1}{4!}h^4f^{(4)}(x) - \dots$$

subtracting these two equations, we have

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{2}{3!}h^3f'''(x) + \frac{2}{5!}h^5f^{(5)}(x) + \dots$$

2nd Order Approximation

it follows that

$$f'(x) = \frac{1}{2h}[f(x+h) - f(x-h)] \\ - \frac{h^2}{3!}f'''(x) - \frac{h^4}{5!}f^{(5)}(x) - \dots$$

after dropping the higher order terms, we have a second order approximation formula as

$$f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)]$$

the leading truncated terms of this approximation scheme is $-\frac{h^2}{6}f'''(x)$. Hence the approximation is of $O(h^2)$. The approximation error goes to 0 as fast as $h^2 \rightarrow 0$. The exact truncation error is

$$-\frac{1}{6}h^2 \left[\frac{f'''(\xi_1) + f'''(\xi_2)}{2} \right] = -\frac{1}{6}h^2 f'''(\xi)$$