LECTURE 3: Polynomial interpolation and numerical differentiation

September 19, 2011

1 Introduction

An interpolation task usually involves a given set of data points: where the values y_i can,

x_i	x_0	x_1	...	x_n
$f(x_i)$	y_0	y_1	...	y_n

for example, be the result of some physical measurement or they can come from a long numerical calculation. Thus we know the value of the underlying function $f(x)$ at the set of points $\{x_i\}$, and we want to find an analytic expression for f.

In interpolation, the task is to estimate $f(x)$ for arbitrary x that lies between the smallest and the largest x_i. If x is outside the range of the x_i's, then the task is called extrapolation, which is considerably more hazardous.

By far the most common functional forms used in interpolation are the polynomials. Other choices include, for example, trigonometric functions and spline functions (discussed later during this course).

Examples of different types of interpolation tasks include:

1. Having the set of $n+1$ data points $\{x_i, y_i\}$, we want to know the value of y in the whole interval $x = [x_0, x_n]$; i.e. we want to find a simple formula which reproduces the given points exactly.

2. If the set of data points involve errors (e.g. if they are measured values), then we ask for a formula that represents the data, and if possible, filters out the errors.

3. A function f may be given in the form of a computer procedure which is expensive to evaluate. In this case, we want to find a function g which gives a good approximation of f and is simpler to evaluate.

2 Polynomial interpolation

2.1 Interpolating polynomial

Given a set of $n+1$ data points $\{x_i, y_i\}$, we want to find a polynomial curve that passes through all the points. Thus, we look for a *continuous* curve which takes on the values y_i for each of the $n+1$ distinct x_i's.

A polynomial p for which $p(x_i) = y_i$ when $0 \leq i \leq n$ is said to *interpolate* the given set of data points. The points x_i are called *nodes*.

The trivial case is $n = 0$. Here a *constant* function $p(x) = y_0$ solves the problem.

The simplest case is $n = 1$. In this case, the polynomial p is a straight line defined by

$$\begin{aligned} p(x) &= \left(\frac{x - x_1}{x_0 - x_1}\right) y_0 + \left(\frac{x - x_0}{x_1 - x_0}\right) y_1 \\ &= y_0 + \left(\frac{y_1 - y_0}{x_1 - x_0}\right)(x - x_0) \end{aligned}$$

Here p is used for *linear interpolation*.

As we will see, the interpolating polynomial can be written in a variety of forms, among these are the Newton form and the Lagrange form. These forms are equivalent in the sense that the polynomial in question is the one and the same (in fact, the solution to the interpolation task is given by a unique polynomial)

We begin by discussing the conceptually simpler Lagrange form. However, a straightforward implementation of this form is somewhat awkward to program and unsuitable for numerical evaluation (e.g. the resulting algorithm gives no error estimate). The Newton form is much more convenient and efficient, and therefore, it is used as the basis for numerical algorithms.

2.2 Lagrange form

First define a system of $n+1$ special polynomials l_i known as *cardinal functions*. These have the following property:

$$l_i(x_j) = \delta_{ij} = \begin{cases} 0 & \text{if} \quad i \neq j \\ 1 & \text{if} \quad i = j \end{cases}$$

The cardinal functions l_i can be written as the product of n linear functions (the variable x occurs only in the numerator of each term; the denominators are just numbers):

$$\begin{aligned} l_i(x) &= \prod_{\substack{j \neq i \\ j=0}}^{n} \left(\frac{x - x_j}{x_i - x_j}\right) \quad (0 \leq i \leq n) \\ &= \left(\frac{x - x_0}{x_i - x_0}\right)\left(\frac{x - x_1}{x_i - x_1}\right) \cdots \left(\frac{x - x_{i-1}}{x_i - x_{i-1}}\right)\left(\frac{x - x_{i+1}}{x_i - x_{i+1}}\right) \cdots \left(\frac{x - x_n}{x_i - x_n}\right) \end{aligned}$$

Thus l_i is a polynomial of degree n.

Note that when $l_i(x)$ is evaluated at $x = x_i$, each factor in the preceding equation becomes 1. But when $l_i(x)$ is evaluated at some other *node* x_j, one of the factors becomes 0, and thus $l_i(x_j) = 0$ for $i \neq j$.

Once the cardinal functions are defined, we can interpolate any function f by the following **Lagrange form of the interpolation polynomial**:

$$p_n(x) = \sum_{i=0}^{n} l_i(x) f(x_i)$$

Writing out the summation, we get

$$\begin{aligned} p_n(x) = &\left(\frac{x-x_1}{x_0-x_1}\right)\ldots\left(\frac{x-x_i}{x_0-x_i}\right)\ldots\left(\frac{x-x_n}{x_0-x_n}\right) f(x_0) \quad +\ldots+ \\ &+\left(\frac{x-x_0}{x_i-x_0}\right)\left(\frac{x-x_1}{x_i-x_1}\right)\ldots\left(\frac{x-x_{i-1}}{x_i-x_{i-1}}\right)\left(\frac{x-x_{i+1}}{x_i-x_{i+1}}\right)\ldots\left(\frac{x-x_n}{x_i-x_n}\right) f(x_i) \quad +\ldots+ \\ &+\left(\frac{x-x_0}{x_n-x_0}\right)\ldots\left(\frac{x-x_i}{x_n-x_i}\right)\ldots\left(\frac{x-x_{n-1}}{x_n-x_{n-1}}\right) f(x_n) \end{aligned}$$

It is easy to check that this function passes through all the nodes, i.e. $p_n(x_j) = f(x_j) = y_j$. Since the function p_n is a linear combination of the polynomials l_i, it is itself a polynomial of degree n or less.

Example.

Find the Lagrange form of the interpolating polynomial for the following table of values:

x	$1/3$	$1/4$	1
y	2	-1	7

The cardinal functions are:

$$\begin{aligned} l_0(x) &= \frac{(x-1/4)(x-1)}{(1/3-1/4)(1/3-1)} = -18(x-1/4)(x-1) \\ l_1(x) &= \frac{(x-1/3)(x-1)}{(1/4-1/3)(1/4-1)} = 16(x-1/3)(x-1) \\ l_2(x) &= \frac{(x-1/3)(x-1/4)}{(1-1/3)(1-1/4)} = 2(x-1/3)(x-1/4) \end{aligned}$$

Therefore, the interpolating polynomial in Lagrange's form is

$$\begin{aligned} p_2(x) = &-36(x-1/4)(x-1) \\ &-16(x-1/3)(x-1) \\ &+14(x-1/3)(x-1/4) \end{aligned}$$

2.3 Existence of interpolating polynomial

Theorem. If points $x_0, x_1, \ldots, x_n$ are distinct, then for arbitrary real values $y_0, y_1, \ldots, y_n$, there is a unique polynomial p of degree $\leq n$ such that $p(x_i) = y_i$ for $0 \leq i \leq n$.

The theorem can be proven by inductive reasoning. Suppose that we have succeeded in finding a polynomial p that reproduces a part of the given set of data points; e.g. $p(x_i) = y_i$ for $0 \leq i \leq k$.

We then attempt to add another term to p such that the resulting curve will pass through yet another data point x_{k+1}. We consider

$$q(x) = p(x) + c(x - x_0)(x - x_1)\ldots(x - x_k)$$

where c is a constant to be determined.

This is a polynomial and it reproduces the first k data points because p does so and the added term is 0 at each of the points $x_0, x_1, \ldots, x_k$.

We now adjust the value of c in such a way that the new polynomial q takes the value y_{k+1} at x_{k+1}. We obtain

$$q(x_{k+1}) = p(x_{k+1}) + c(x_{k+1} - x_0)(x_{k+1} - x_1)\ldots(x_{k+1} - x_k) = y_{k+1}$$

The proper value of c *can* be obtained from this equation since all the x_i's are distinct.

2.4 Newton form

The inductive proof of the existence of interpolating polynomial theorem provides a method for constructing an interpolating polynomial. The method is known as the *Newton algorithm*.

The method is based on constructing successive polynomials $p_0, p_1, p_2, \ldots$ until the desired degree n is reached (determined by the number of data points = $n+1$).

The first polynomial is given by

$$p_0(x) = y_0$$

Adding the second term gives

$$p_1(x) = p_0(x) + c_1(x - x_0) = y_0 + c_1(x - x_0)$$

Interpolation condition is $p_1(x_1) = y_1$. Thus we obtain

$$y_0 + c_1(x_1 - x_0) = y_1$$

which leads to

$$c_1 = \frac{y_1 - y_0}{x_1 - x_0}$$

The value of c is evaluated and placed in the formula for p_1.

We then continue to the third term which is given by

$$p_2(x) = p_1(x) + c_2(x - x_0)(x - x_1) = y_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1)$$

We again impose the interpolation condition $p_2(x_2) = y_2$ and use it to calculate the value of c.

The iteration for the kth polynomial is

$$p_k(x) = p_{k-1}(x) + c_k(x - x_0)(x - x_1)\ldots(x - x_{k-1})$$

In the end we obtain the **Newton form of the interpolating polynomial**:

$$p_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1)\ldots + a_n(x - x_0)\ldots(x - x_{n-1})$$

where the coefficients $y_0, c_1, c_2, \ldots$ have been denoted by a_i.

Example.

Find the Newton form of the interpolating polynomial for the following table of values:

x	$1/3$	$1/4$	1
y	2	-1	7

We will construct three successive polynomials p_0, p_1 and p_2. The first one is

$$p_0(x) = 2$$

The next one is given by

$$p_1(x) = p_0(x) + c_1(x - x_0) = 2 + c_1(x - 1/3)$$

Using the interpolation condition $p_1(1/4) = -1$, we obtain $c_1 = 36$, and

$$p_1(x) = 2 + 36(x - 1/3)$$

Finally,

$$\begin{aligned} p_2(x) &= p_1(x) + c_2(x - x_0)(x - x_1) \\ &= 2 + 36(x - 1/3) + c_2(x - 1/3)(x - 1/4) \end{aligned}$$

The interpolation condition gives $p_2(1) = 7$, and thus $c_2 = -38$.
The Newton form of the interpolating polynomial is

$$p_2(x) = 2 + 36(x - 1/3) - 38(x - 1/3)(x - 1/4)$$

Comparison of Lagrange and Newton forms

Lagrange form

$$\begin{aligned} p_2(x) = &- 36(x - 1/4)(x - 1) \\ &- 16(x - 1/3)(x - 1) \\ &+ 14(x - 1/3)(x - 1/4) \end{aligned}$$

Newton form

$$p_2(x) = 2 + 36(x - 1/3) - 38(x - 1/3)(x - 1/4)$$

These two polynomials are equivalent, i.e:

$$p_2(x) = -38x^2 + (349/6)x - 79/6$$

2.5 Nested multiplication

For efficient evaluation, the Newton form can be rewritten in a so-called nested form. This form is obtained by systematic factorisation of the original polynomial.

Consider a general polynomial in the Newton form

$$p_n(x) = a_0 + a_1(x-x_0) + a_2(x-x_0)(x-x_1)\ldots + a_n(x-x_0)\ldots(x-x_{n-1})$$

This can be written as

$$p_n(x) = a_0 + \sum_{i=1}^{n} a_i \left[\prod_{j=0}^{i-1}(x-x_j)\right]$$

Let us now rewrite the Newton form of the interpolating polynomial in the nested form by using the common terms $(x-x_i)$ for factoring. We obtain

$$p_n(x) = a_0 + (x-x_0)(a_1 + (x-x_1)(a_2 + (x-x_2)(a_3 + \ldots + (x-x_{n-1})a_n)\ldots))$$

This is called the *nested form* and its evaluation is done by *nested multiplication*.

When evaluating $p(x)$ for a given numerical value of $x = t$, we naturally begin with the innermost parenthesis:

$$\begin{aligned} v_0 &= a_n \\ v_1 &= v_0(t-x_{n-1}) + a_{n-1} \\ v_2 &= v_1(t-x_{n-2}) + a_{n-2} \\ &\vdots \\ v_n &= v_{n-1}(t-x_0) + a_0 \end{aligned}$$

The quantity v_n is $p(t)$.

In the actual implementation, we only need *one* variable v. The following pseudocode shows how the evaluation of $p(x=t)$ is done efficiently once the coefficients a_i of the Newton form are known. The array $(x_i)_{0:n}$ contains the $n+1$ nodes x_i.

```
real array (a_i)_{0:n}, (x_i)_{0:n}
integer i, n
real t, v
v ← a_n
for i=n-1 to 0 step -1 do
    v ← v(t − x_i) + a_i
end for
```

2.6 Divided differences

The next step is to discuss a method for determining the coefficients a_i efficiently. We start with a table of values of a function f: We have seen that there exists a unique polynomial

x	x_0	x_1	$\ldots$	x_n
$f(x)$	$f(x_0)$	$f(x_1)$	$\ldots$	$f(x_n)$

p of degree $\leq n$. In the compact notation of the Newton form p is written as

$$p_n(x) = \sum_{i=0}^{n} a_i \left[\prod_{j=0}^{i-1} (x - x_j) \right]$$

in which $\prod_{j=0}^{-1}(x-x_j)$ is interpreted as 1.

Notice that the coefficients a_i do not depend on n. In other words, p_n is obtained from p_{n-1} by adding one more term, without altering the coefficients already present in p_{n-1}.

A way of systematically determining the coefficients a_i is to set x equal to each of the points x_i $(0 \leq i \leq n)$ at a time. The resulting equations are:

$$\begin{cases} f(x_0) & = a_0 \\ f(x_1) & = a_0 + a_1(x_1 - x_0) \\ f(x_2) & = a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) \\ & \vdots \end{cases}$$

In compact form

$$f(x_k) = \sum_{i=0}^{k} a_i \prod_{j=0}^{i-1} (x_k - x_j) \quad (0 \leq k \leq n)$$

The coefficients a_k are *uniquely* determined by these equations. The coefficients can now be solved starting with a_0 which depends on $f(x_0)$, followed by a_1 which depends on $f(x_0)$ and $f(x_1)$, and so on.

In general, a_k depends on the values of f at the nodes $x_0, \ldots, x_k$. This dependence is formally written in the following notation

$$a_k = f[x_0, x_1, \ldots, x_k]$$

The quantity $f[x_0, x_1, \ldots, x_k]$ is called the **divided difference of order k** for f.
(Suomeksi: jaettu erotus.)

Example. Determine the quantities $f[x_0]$, $f[x_0,x_1]$, and $f[x_0,x_1,x_2]$ for the following table

x	1	-4	0
$f(x)$	3	13	-23

The system of equations for the coefficients a_k:

$$\begin{cases} 3 & = a_0 \\ 13 & = a_0 + a_1(-4-1) \\ -23 & = a_0 + a_1(0-1) + a_2(0-1)(0+4) \end{cases}$$

We get

$$\begin{cases} a_0 & = 3 \\ a_1 & = [13 - a_0]/(-4-1) = -2 \\ a_2 & = [-23 - a_0 - a_1(0-1)]/(0+4) = 7 \end{cases}$$

Thus for this function, $f[1] = 3$, $f[1,-4] = -2$, and $f[1,-4,0] = 7$.

With the new notation for divided differences, the **Newton form of the interpolating polynomial** takes the form

$$p_n(x) = \sum_{i=0}^{n} \left\{ f[x_0,x_1,\ldots,x_i] \prod_{j=0}^{i-1} (x - x_j) \right\}$$

The required divided differences $f[x_0,x_1,\ldots,x_k]$ (the coefficients) can be computed *recursively*. We can write

$$\begin{aligned} f(x_k) &= \sum_{i=0}^{k} a_i \prod_{j=0}^{i-1} (x_k - x_j) \\ &= f[x_0,x_1,\ldots,x_k] \prod_{j=0}^{k-1} (x_k - x_j) + \sum_{i=0}^{k-1} f[x_0,x_1,\ldots,x_i] \prod_{j=0}^{i-1} (x_k - x_j) \end{aligned}$$

and

$$f[x_0,x_1,\ldots,x_k] = \frac{f(x_k) - \sum_{i=0}^{k-1} f[x_0,x_1,\ldots,x_i] \prod_{j=0}^{i-1} (x_k - x_j)}{\prod_{j=0}^{k-1} (x_k - x_j)}$$

Thus one possible **algorithm** for computing the divided differences is

i. Set $f[x_0] = f(x_0)$.
ii. For $k = 1,2,\ldots,n$, compute $f[x_0,x_1,\ldots,x_k]$ by the equation above.

This can be easily programmed; the algorithm is capable of computing the divided differences at the cost of $\frac{1}{2}n(3n+1)$ additions/subtractions, $(n-1)(n-2)$ multiplications, and n divisions.

There is, however, a more refined method for calculating the divided differences.

2.7 Recursive property of divided differences

The efficient method for calculating the divided differences is based on two properties of the divided differences.

First, the divided differences obey the formula (*the recursive property of divided differences*)

$$f[x_0, x_1, \ldots, x_k] = \frac{f[x_1, x_2, \ldots, x_k] - f[x_0, x_1, \ldots, x_{k-1}]}{x_k - x_0}$$

The proof is left to the student (see e.g., Cheney and Kincaid, pp. 134, Theorem 2).

Second, because of the uniqueness of the interpolating polynomials, the divided differences are not changed if the nodes are permuted. This is expressed more formally by the following theorem.

Invariance theorem.
The divided difference $f[x_0, x_1, \ldots, x_k]$ is invariant under all permutations of the arguments $x_0, x_1, \ldots, x_k$.

Thus, the **recursive formula** can also be written as

$$f[x_i, x_{i+1}, \ldots, x_{j-1}, x_j] = \frac{f[x_{i+1}, x_{i+2}, \ldots, x_j] - f[x_i, x_{i+1}, \ldots, x_{j-1}]}{x_j - x_i}$$

We can now determine the divided differences as follows:

$$f[x_i] = f(x_i) \quad f[x_{i+1}] = f(x_{i+1}) \quad f[x_{i+2}] = f(x_{i+2})$$

$$f[x_i, x_{i+1}] = \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i} \qquad f[x_{i+1}, x_{i+2}] = \frac{f[x_{i+2}] - f[x_{i+1}]}{x_{i+2} - x_{i+1}}$$

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}$$

Using the recursive formula, we can now construct a divided-differences table for a function f

x	$f[\,]$	$f[\,,\,]$	$f[\,,\,,\,]$	$f[\,,\,,\,,\,]$
x_0	$f[x_0]$			
		$f[x_0,x_1]$		
x_1	$f[x_1]$		$f[x_0,x_1,x_2]$	
		$f[x_1,x_2]$		$f[x_0,x_1,x_2,x_3]$
x_2	$f[x_2]$		$f[x_1,x_2,x_3]$	
		$f[x_2,x_3]$		
x_3	$f[x_3]$			

Example.

Construct a divided differences table and write out the Newton form of the interpolating polynomial for the following table of values:

x_i	1	3/2	0	3
$f(x_i)$	3	13/4	3	5/3

Step 1.

The second column of entries is given by $f[x_i] = f(x_i)$. Thus

x	$f[\,]$	$f[\,,\,]$	$f[\,,\,,\,]$	$f[\,,\,,\,,\,]$
1	3			
		$f[x_0,x_1]$		
3/2	13/4		$f[x_0,x_1,x_2]$	
		$f[x_1,x_2]$		$f[x_0,x_1,x_2,x_3]$
0	3		$f[x_1,x_2,x_3]$	
		$f[x_2,x_3]$		
2	5/3			

Step 2.

The third column of entries is obtained using the values in the second column:

$$f[x_i,x_{i+1}] = \frac{f[x_{i+1}]-f[x_i]}{x_{i+1}-x_i}$$

For example,

$$f[x_0,x_1] = \frac{f[x_1]-f[x_0]}{x_1-x_0} = \frac{13/4-3}{3/2-1} = \frac{1}{2}$$

The table with a completed second column is

x	$f[\,]$	$f[\,,\,]$	$f[\,,\,,\,]$	$f[\,,\,,\,,\,]$
1	3			
		1/2		
3/2	13/4		$f[x_0,x_1,x_2]$	
		1/6		$f[x_0,x_1,x_2,x_3]$
0	3		$f[x_1,x_2,x_3]$	
		-2/3		
2	5/3			

Step 3.

The first entry in the fourth column is obtained using the values in the third column:

$$f[x_0,x_1,x_2] \;=\; \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0} = \frac{1/6-1/2}{0-1} = \frac{1}{3}$$

The table with a completed third column is

x	$f[\,]$	$f[\,,\,]$	$f[\,,\,,\,]$	$f[\,,\,,\,,\,]$
1	3			
		1/2		
3/2	13/4		1/3	
		1/6		$f[x_0,x_1,x_2,x_3]$
0	3		-5/3	
		-2/3		
2	5/3			

Step 4.

The final entry in the table is

$$f[x_0,x_1,x_2,x_3] \;=\; \frac{f[x_1,x_2,x_3]-f[x_0,x_1,x_2]}{x_3-x_0} = \frac{-5/3-1/3}{2-1} = -2$$

The completed table is

x	$f[\,]$	$f[\,,\,]$	$f[\,,\,,\,]$	$f[\,,\,,\,,\,]$
1	**3**			
		1/2		
3/2	13/4		**1/3**	
		1/6		**-2**
0	3		-5/3	
		-2/3		
2	5/3			

Using the values shown in bold text in the divided differences table, we can now write out the Newton form of the interpolating polynomial. We get

$$\begin{aligned} p_n(x) &= \sum_{i=0}^{n} \left\{ f[x_0, x_1, \ldots, x_i] \prod_{j=0}^{i-1} (x - x_j) \right\} \\ &= f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) \\ &+ f[x_0, x_1, x_2, x_3](x - x_0)(x - x_1)(x - x_2) \\ \\ &= 3 + \frac{1}{2}(x-1) + \frac{1}{3}(x-1)(x-\frac{3}{2}) - 2(x-1)(x-\frac{3}{2})x \end{aligned}$$

2.8 Algorithms

We now use the same procedure to construct an algorithm which computes all the divided differences $a_{ij} \equiv f[x_i, x_{i+1}, \ldots, x_j]$.

Notice that there is no need to store all the values in the table since only $f[x_0], f[x_0, x_1], \ldots, f[x_0, x_1, \ldots, x_n]$ are needed to construct the Newton form of the interpolating polynomial.

Denote $a_i \equiv f[x_0, x_1, \ldots, x_i]$. Using this notation, the Newton form is given by

$$p_n(x) = \sum_{i=0}^{n} a_i \prod_{j=0}^{i-1} (x - x_j)$$

We can use a one-dimensional array $(a_i)_{0:n}$ for storing the divided differences and overwrite the entries each time from the last storage location backward. At each step, the new value of each new divided difference is obtained using the recursion formula:

$$a_i \leftarrow (a_i - a_{i-1})/(x_i - x_{i-j})$$

where j is the index of the step in question (j=1,...,n).

The following procedure *Coefficients* calculates the coefficients a_i required in the Newton interpolating polynomial. The input is the number n (degree of the polynomial) and the arrays for x_i, y_i and a_i (all contain $n+1$ elements). The arrays x_i and y_i contain the data points. a_i is the array where the coefficients are to be computed.

```
void Coefficients(int n, double *x, double *y, double *a)
{
  int i, j;

  for(i=0; i<=n; i++)
    a[i] = y[i];

  for(j=1; j<=n; j++)
    for(i=n; i>=j; i--)
      a[i] = (a[i]-a[i-1])/(x[i]-x[i-j]);
}
```

In the procedure *Coefficients*, the arrays are one-dimensional and the divided differences are calculated from the last storage location backward so that in the end only the desired coefficients remain.

The same result could be accomplished in a simpler way which, however, would require more storage:

```
  double x[n+1], y[n+1], a[n+1][n+1];

  for(i=0; i<=n; i++)
    a[i] = y[i];

  for(j=1; j<=n; j++)
    for(i=0; i<=n-j; i++)
      a[i][j] = (a[i+1][j+1]-a[i][j-1])/(x[i+j]-x[i]);
```

The procedure *Evaluate* returns the value of the interpolating polynomial at point t. The input is the array x_i and the array a_i which is obtained using the procedure *Coefficients*. Also the value of the point t is given.

```
double Evaluate(int n, double *x, double *a, double t)
{
  int i;
  double pt;

  pt = a[n];
  for(i=n-1; i>=0; i--)
    pt = pt*(t-x[i])+a[i];

  return(pt);
}
```

Notice that only the value of t should change in subsequent calls of *Evaluate*.

These procedures *Coefficients* and *Evaluate* can be used to construct a program that determines the Newton form of the interpolating polynomial for a given function $f(x)$. In such a program, the procedure *Coefficients* is called once to determine the coefficients and then the procedure *Evaluate* is called as many time as needed to determine the value of the interpolating polynomial at some given set points of interest.

Example.

In the following example, the task is to find the interpolating polynomial for the function $f(x) = \sin(x)$ at ten equidistant points in the interval $[0, 1.6875]$ (these ten points are the nodes). The value of the resulting polynomial $p(x)$ is then evaluated at 37 equally spaced points in the same interval and also the absolute error $|f(x) - p(x)|$ is calculated.

Output

The following coefficients for the Newton form of the interpolating polynomial are obtained:

```
  i       ai
  0   0.00000000
  1   0.99415092
  2  -0.09292892
  3  -0.15941590
  4   0.01517217
  5   0.00738018
  6  -0.00073421
  7  -0.00015560
  8   0.00001671
  9   0.00000181
```

Figure 1 shows a plot of the data points $f(x_i) = \sin(x_i)$, $0 \leq i \leq 9$ and the interpolating polynomial (evaluated at 37 points).

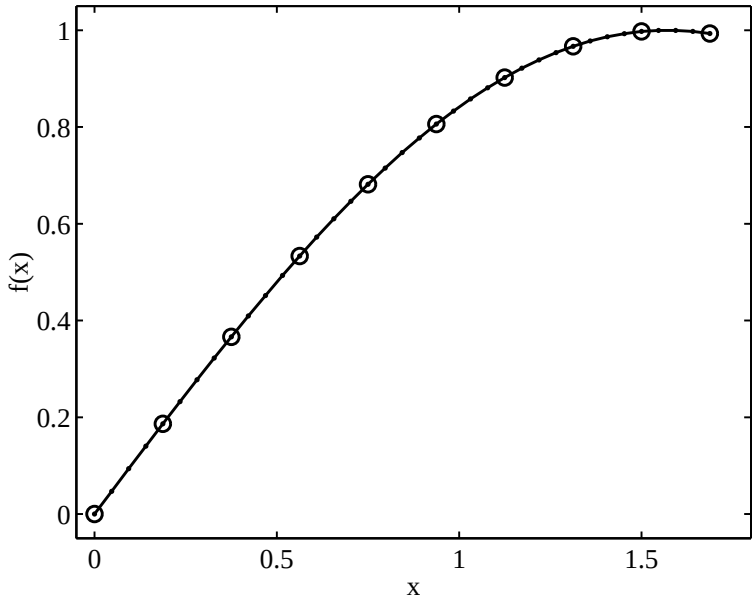

Figure 1: A plot of the original data points (nodes) and the interpolating polynomial.

From Fig. 2, we see that the deviation between $p(x)$ and $f(x)$ is *zero* at each interpolation node (within the accuracy of the computer). Between the nodes, we obtain deviations that are less than 5×10^{-10} in magnitude.

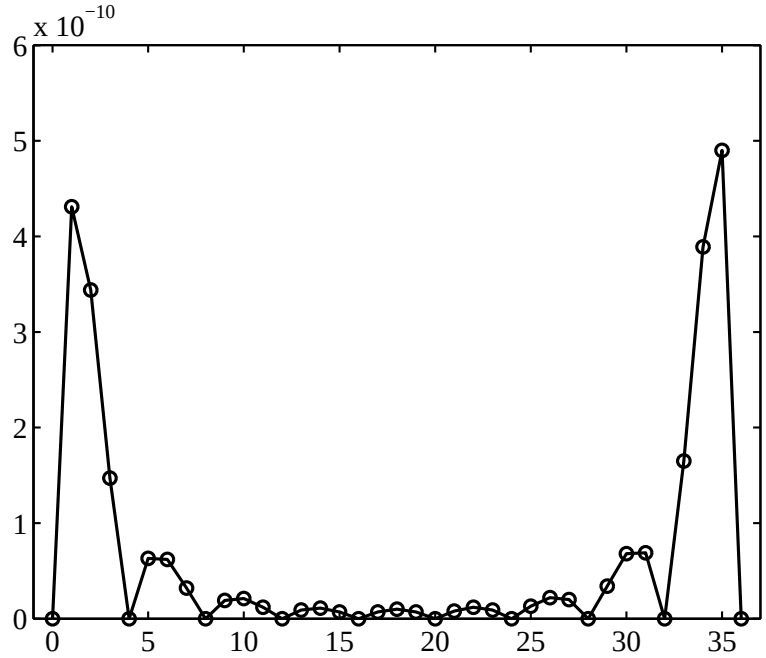

Figure 2: The absolute error $|f(x) - p(x)|$ between the interpolating polynomial p and the underlying function f at the 37 interpolation points.

2.9 Inverse interpolation

A process called inverse interpolation can be used to approximate an inverse function. Suppose that the values $y_i = f(x_i)$ have been computed at $x_0, x_1, \ldots, x_n$.

We can form the interpolation polynomial as

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

The original relationship $y = f(x)$ has an inverse under certain conditions. This can be approximated by $x = p(y)$.

The procedures *Coefficients* and *Evaluate* can be used to carry out the inverse interpolation by reversing the arguments x and y.

Inverse interpolation can be used to locate the *root* of a given function f; i.e. we want to solve $f(x) = 0$. First create a set of function values $\{f(x_i), x_i\}$ and find the interpolating polynomial $p(y_i) = x_i$.

The approximate root is now given by $r \approx p(y = 0)$.

Example.

Having the following set of values

```
      y            x
 -0.57892000   1.00000000
 -0.36263700   2.00000000
 -0.18491600   3.00000000
 -0.03406420   4.00000000
  0.09698580   5.00000000
```

We can use the interpolation program with
$x \leftarrow y$ and $y \leftarrow x$.

This gives us the coefficients for the polynomial $p(y)$.

Evaluating p at point $y = 0$, gives $p(0) = 4.24747001$.

2.10 Neville's algorithm

Another widely used method for obtaining the interpolating polynomial for a given table of values is given by *Neville's algorithm*. It builds up the polynomial in steps, just as the Newton algorithm does.

Let $P_{a,b,\ldots,s}$ be the polynomial interpolating the given data at a sequence of nodes $x_a, x_b, \ldots, x_s$. We start with constant polynomials $P_i(x) = f(x_i)$. Selecting two nodes x_i and x_j with $i > j$, we define recursively

$$P_{u,\ldots,v}(x) = \left(\frac{x - x_j}{x_i - x_j}\right) P_{u,\ldots,j-1,j+1,\ldots,v}(x) + \left(\frac{x_i - x}{x_i - x_j}\right) P_{u,\ldots,i-1,i+1,\ldots,v}(x)$$

For example, the first three constant polynomials are

$$P_0(x) = f(x_0) \quad P_1(x) = f(x_1) \quad P_2(x) = f(x_2)$$

Using P_0 and P_1, we can construct $P_{0,1}$:

$$P_{0,1}(x) = \left(\frac{x - x_0}{x_1 - x_0}\right) P_1 + \left(\frac{x_1 - x}{x_1 - x_0}\right) P_0$$

and similarly

$$P_{1,2}(x) = \left(\frac{x - x_1}{x_2 - x_1}\right) P_2 + \left(\frac{x_2 - x}{x_2 - x_1}\right) P_1$$

Using these two polynomials of degree one, we can construct another polynomial of degree two:

$$P_{0,1,2}(x) = \left(\frac{x - x_0}{x_2 - x_0}\right) P_{1,2} + \left(\frac{x_2 - x}{x_2 - x_0}\right) P_{0,1}$$

Similarly,

$$P_{0,1,2,3}(x) = \left(\frac{x - x_0}{x_3 - x_0}\right) P_{1,2,3} + \left(\frac{x_3 - x}{x_3 - x_0}\right) P_{0,1,2}$$

Using this formula repeatedly, we can create an array of polynomials:

x_0	$P_0(x)$				
x_1	$P_1(x)$	$P_{0,1}(x)$			
x_2	$P_2(x)$	$P_{1,2}(x)$	$P_{0,1,2}(x)$		
x_3	$P_3(x)$	$P_{2,3}(x)$	$P_{1,2,3}(x)$	$P_{0,1,2,3}(x)$	
x_4	$P_4(x)$	$P_{3,4}(x)$	$P_{2,3,4}(x)$	$P_{1,2,3,4}(x)$	$P_{0,1,2,3,4}(x)$

Here each successive polynomial can be determined from two adjacent polynomials in the previous column.

We can simplify the notation by

$$S_{ij}(x) = P_{i-j,i-j+1,\ldots,i-1,i}(x)$$

where $S_{ij}(x)$ for $i \geq j$ denotes the interpolating polynomial of degree j on the $j+1$ nodes $x_{i-j}, x_{i-j+1}, \ldots, x_{i-1}, x_i$. For example, S_{44} is the interpolating polynomial of degree 4 on the five nodes x_0, x_1, x_2, x_3 and x_4.

Using this notation, we can rewrite the recurrence relation as

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

The array becomes

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	$S_{40}(x)$	$S_{41}(x)$	$S_{42}(x)$	$S_{43}(x)$	$S_{44}(x)$

Algorithm

A pseudocode to evaluate $S_{nn}(x)$ at point $x = t$ can be written as follows:

```
real array (x_i)_{0:n}, (y_i)_{0:n}, (S_ij)_{0:n×0:n}
integer i, j, n
for i=0 to n
   S_i0 ← y_i
end for
for j=1 to n
  for i=j to n
     S_ij ← [(t − x_{i−j})S_{i,j−1} + (x_i − t)S_{i−1,j−1}]/(x_i − x_{i−j})
  end for
end for
return S_nn
```

Here $(x_i)_{0:n}$ and $(y_i)_{0:n}$ are the arrays containing the initial data points, and the procedure returns the value of the interpolating polynomial at point $x = t$.

3 Errors in polynomial interpolation

When a function f is approximated on an interval $[a,b]$ by an interpolating polynomial p, the discrepancy between f and p will be zero at each node, i.e., $p(x_i) = f(x_i)$ (at least theoretically). Based on this, it is easy to expect that when the number of nodes n increases, the function f will be better approximated at all intermediate points as well.

This is wrong. In many cases, high-degree polynomials can be very unsatisfactory representations of functions even if the function is continuous and well-behaved.

Example. Consider the Runge function:

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

on the interval [-5,5]. Let p_n be the polynomial that interpolates this function at $n+1$ equally spaced points on the interval [-5,5]. Then

$$\lim_{n\to\infty} \max_{-5\le x\le 5} |f(x) - p(x)| = \infty$$

Thus increasing the number of node points increases the the error at nonnodal points beyond all bounds! Figure 3 below shows f and p for $n = 9$.

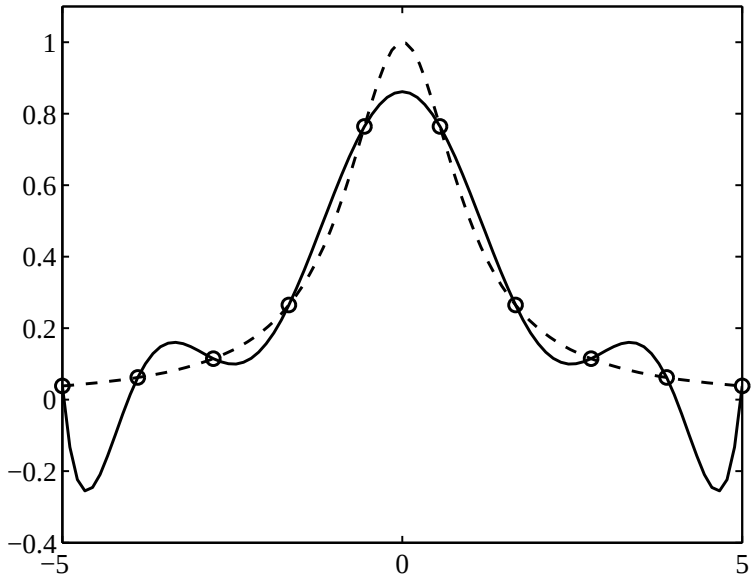

Figure 3: The Runge function f (shown in dashed line) and the polynomial interpolant p obtained using 9 equidistant nodes.

3.1 Choice of nodes

Contrary to intuition, equally distributed nodes are usually not the best choice in interpolation. A much better choice for $n+1$ nodes in [-1,1] is the set of **Chebyshev nodes**:

$$x_i = \cos\left[\left(\frac{i}{n}\right)\pi\right] \quad (0 \leq i \leq n)$$

The corresponding set of nodes on an arbitrary interval [a,b] can be derived from a linear mapping to obtain

$$x_i = \frac{1}{2}(a+b) + \frac{1}{2}(b-a)\cos\left[\frac{i}{n}\pi\right] \quad (0 \leq i \leq n)$$

Notice that the Chebyshev nodes are numbered from right to left (i.e.the largest value has the lowest index).

Figure 4 shows the polynomial interpolant of the Runge function obtained with 13 Chebyshev nodes.

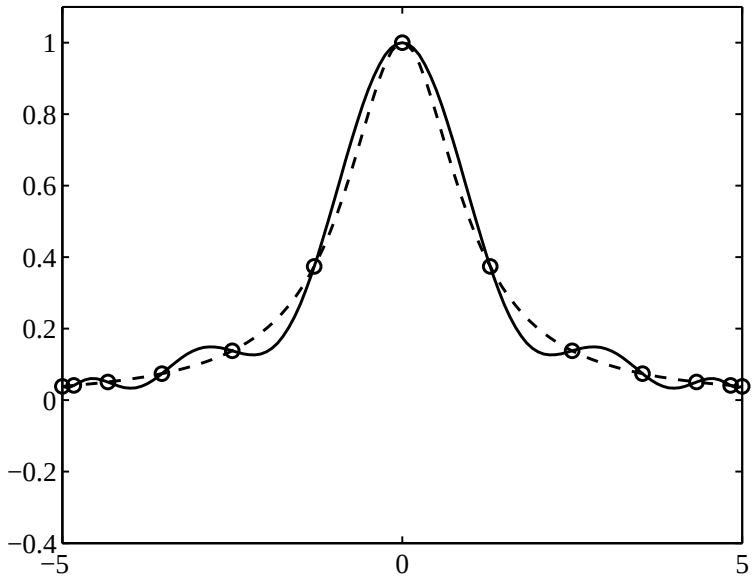

Figure 4: The Runge function f (shown in dashed line) and the polynomial interpolant p obtained using 13 Chebyshev nodes.

3.2 Theorems on interpolation errors

Theorem 1.
If p is the polynomial of degree at most n that interpolates f at the $n+1$ distinct nodes x_i $(0 \leq i \leq n)$ belonging to an interval $[a,b]$ and if $f^{(n+1)}$ (the nth derivative of f) is continuous, then for each x in $[a,b]$, there is a ξ in (a,b) for which

$$f(x) - p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^{n} (x - x_i)$$

Theorem 2.
Let f be a function such that $f^{(n+1)}$ is continuous on $[a,b]$ and satisfies $|f^{(n+1)}(x)| \leq M$. Let p be the polynomial of degree $\leq n$ that interpolates f at $n+1$ equally spaced nodes in $[a,b]$, including the endpoints. Then on $[a,b]$,

$$|f(x) - p(x)| = \frac{1}{4(n+1)} M \left(\frac{b-a}{n} \right)^{n+1}$$

Theorem 3.
If p is the polynomial of degree n that interpolates the function f at nodes x_i $(0 \leq i \leq n)$, then for any x not a node

$$f(x) - p(x) = f[x_0, x_1, \ldots, x_n, x] \prod_{i=0}^{n} (x - x_i)$$

Example.

In a previous example, we constructed the interpolating polynomial of $f(x) = sin(x)$ at ten equidistant points in $[0, 1.6875]$.

Figure 5 shows a plot of the computed error $|f(x) - p(x)|$. At the largest, the error is approximately 5×10^{-10}.

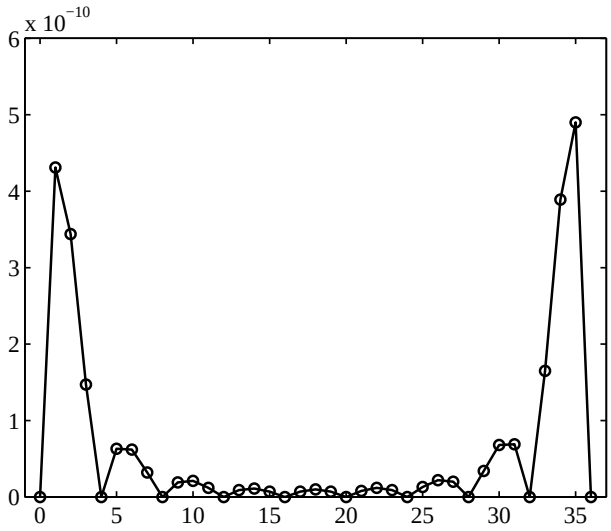

Figure 5: The absolute error $|f(x) - p(x)|$ in polynomial interpolation of $f(x) = sin(x)$ using ten equidistant nodes.

Now use Theorem 2 with $f(x) = sin(x)$, $n = 9$, $a = 0$ and $b = 1.6875$. The 10th derivative of f is $f^{(10)}(x) = -sin(x)$, and thus $|f^{(10)}(x)| \leq 1$. Thus we can let $M = 1$.

The resulting bound for the error is

$$|f(x) - p(x)| \leq \frac{1}{4(n+1)} M \left(\frac{b-a}{n} \right)^{n+1} = 1.34 \times 10^{-9}$$

Divided differences & derivatives theorem.
If $f^{(n)}$ is continuous on $[a,b]$ and if $x_0, x_1, \ldots, x_n$ are any $n+1$ distinct points in $[a,b]$, then for some ξ in (a,b),

$$f[x_0, x_1, \ldots, x_n] = \frac{1}{n!} f^{(n)}(\xi)$$

Divided differences corollary.
If f is a polynomial of degree n, then all of the divided differences $f[x_0, x_1, \ldots, x_i]$ are zero for $i \geq n+1$.

Example.

Consider the following table of values:

x	1	-2	0	3	-1	7
y	-2	-56	-2	4	-16	376

Constructing a divided differences table shows that the fourth-order divided differences are all zero.

x	$f[\,]$	$f[\,,\,]$	$f[\,,\,,\,]$	$f[\,,\,,\,,\,]$	$f[\,,\,,\,,\,,\,]$
1	-2				
		18			
-2	-56		-9		
		27		2	
0	-2		-5		0
		2		2	
3	4		-3		0
		5		2	
-1	-16		11		
		49			
7	376				

The Newton form of the interpolation polynomial is

$$p_3(x) = -2 + 18(x-1) - 9(x-1)(x+2) + 2(x-1)(x+2)x$$

Thus the data can be represented by a cubic polynomial because all the fourth-order divided differences are zero.

4 Numerical differentiation

We now move on to a slightly different subject, namely numerical differentiation. This subject is often not discussed in great detail, but we include it here because the central idea of a process called Richardson extrapolation is also used in numerical integration.

Determining the derivative of a function f at point x is not a trivial numerical problem. Specifically, if $f(x)$ can be computed with n digits of precision, it is difficult to calculate $f'(x)$ numerically with n digits of precision. This is caused by subtraction of nearly equal quantities (loss of precision).

4.1 First-derivative formulas via Taylor series

Consider first the obvious method based on the definition of $f'(x)$:

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

What is the error involved in this formula? By Taylor's theorem:

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

Rearranging this gives:

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

We see that the first approximation has a *truncation error* $-\frac{1}{2}hf''(\xi)$ where ξ is in the interval $[x, x+h]$. In general, as $h \to 0$, the error approaches zero at the same rate as h does - that is $O(h)$.

It is advantageous to have the convergence of numerical processes occur with high powers of some quantity approaching zero. In the present case, we want an approximation to $f'(x)$ in which the error behaves like $O(h^2)$.

One such method is obtained using the following two Taylor series:

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

By subtraction, we obtain

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

From this we obtain the following formula for f':

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

Thus we obtain an approximation formula for f' with an error whose leading term is $-\frac{1}{6}h^2 f'''(x)$, which makes it $O(h^2)$:

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

The approximation formula with the error term can be written as

$$f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)] - \frac{1}{6}h^2 f'''(\xi)$$

where ξ is some point on the interval $[x-h, x+h]$. This is based only on the assumption that f''' is continuous on $[x-h, x+h]$.

4.2 Richardson extrapolation

Richardson extrapolation is a powerful technique for obtaining accurate results from a numerical process such as the approximation formula for f' obtained via Taylor series:

$$f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)] + a_2h^2 + a_4h^4 + a_6h^6 + \dots$$

where the constants a_i depend on f and x.

Holding f and x fixed, we define a function of h by the formula

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

where $\varphi(h)$ is an approximation to $f'(x)$ with an error of order $O(h^2)$.

Our objective is to compute the limit $\lim_{h\to 0}\varphi(h)$ because this is the quantity $f'(x)$. Since we cannot actually compute the value of $\varphi(0)$ from the equation above, Richardson extrapolation seeks to estimate the limiting value at 0 from some computed values of φ near 0.

Suppose now that we compute $\varphi(h)$ and $\varphi(h/2)$ for some h:

$$\begin{cases} \varphi(h) = f'(x) - a_2h^2 - a_4h^4 - a_6h^6 - \dots \\ \varphi(\frac{h}{2}) = f'(x) - a_2\left(\frac{h}{2}\right)^2 - a_4\left(\frac{h}{2}\right)^4 - a_6\left(\frac{h}{2}\right)^6 - \dots \end{cases}$$

We can now use simple algebra to eliminate the dominant term in the error series. We multiply the bottom series by 4 and subtract it from the top equation. The result is

$$\varphi(h) - 4\varphi(\frac{h}{2}) = -3f'(x) - \frac{3}{4}a_4h^4 - \frac{15}{16}a_6h^6 - \dots$$

Divide this by -3 and rearrange to get

$$\varphi(\frac{h}{2}) + \frac{1}{3}\left[\varphi(\frac{h}{2}) - \varphi(h)\right] = f'(x) + \frac{1}{4}a_4h^4 + \frac{5}{16}a_6h^6 + \dots$$

Thus we have improved the precision to $O(h^4)$!! The same procedure can be repeated over and over again to "kill" higher order terms from the error series. This is called *Richardson extrapolation*.

4.3 Richardson extrapolation theorem

The same procedure will be needed later in the derivation of the Romberg's algorithm for numerical integration. We therefore want to have a general discussion of the procedure.

Let φ be a function such that

$$\varphi(h) = L - \sum_{k=1}^{\infty} a_{2k} h^{2k}$$

where the coefficients a_{2k} are not known. It is assumed that $\varphi(h)$ can be computed accurately for any $h > 0$. Our objective is to approximate L accurately using φ.

We select a convenient h and compute the numbers

$$D(n,0) = \varphi(\frac{h}{2^n}) \quad (n \geq 0)$$

We have

$$D(n,0) = L + \sum_{k=1}^{\infty} A(k,0) \left(\frac{h}{2^n}\right)^{2k}$$

where $A(k,0) = -a_{2k}$. These numbers $D(n,0)$ give a crude estimate of the unknown number $L = \lim_{x \to 0} \varphi(x)$.

More accurate estimates are obtained via Richardson extrapolation:

$$D(n,m) = \frac{4^m}{4^m - 1} D(n,m-1) - \frac{1}{4^m - 1} D(n-1,m-1)$$

This is called the *Richardson extrapolation formula.*

Theorem.
The quantities $D(n,m)$ obey the equation

$$D(n,m) = L + \sum_{k=m+1}^{\infty} A(k,m) \left(\frac{h}{2^n}\right)^{2k} \quad (0 \leq m \leq n)$$

4.4 Algorithm for Richardson extrapolation

1. Write a procedure function for φ.
2. Decide suitable values for N and h.
3. For $i = 0, 1, \ldots, N$, compute $D(i,0) = \varphi(h/2^i)$.
4. For $0 \leq i \leq j \leq N$, compute
   $D(i,j) = D(i,j-1) + (4^j - 1)^{-1}[D(i,j-1) - D(i-1,j-1)]$.

In this algorithm the computation of $D(i,j)$ has been rearranged slightly to improve its numerical properties.

Example

The Richardson extrapolation algorithm is implemented in the following procedure *Derivative*:

```
void Derivative(double (*f)(double x), double x,
                int n, double h, double D[][NCOLS])
{
  int i, j;
  double hh;

  hh=h;
  for(i=0; i<=n; i++) {
    D[i][0] = (f(x+hh)-f(x-hh))/(2.0*hh);
    for(j=0; j<=i-1; j++)
      D[i][j+1] = D[i][j]
                + (D[i][j]-D[i-1][j])/(pow(4.0,j+1.0)-1.0);
    hh = hh/2.0;
  }
}
```

Here f is the function for which we seek to evaluate the derivative f' at a specified point x (a pointer to the function f and the value of x are given as arguments). The other arguments to the procedure are the number of iterations n, the initial value of h and the array for the estimates D.

Choosing $x = 1.2309594154$ for $f(x) = \sin(x)$ with $h = 1$, we get the following output(for n=4):

```
 n    D(n,0)      D(n,1)      D(n,2)      D(n,3)      D(n,4)
 0  0.28049033
 1  0.31961703  0.33265926
 2  0.32987195  0.33329025  0.33333232
 3  0.33246596  0.33333063  0.33333332  0.33333333
 4  0.33311636  0.33333317  0.33333333  0.33333334  0.33333334
```

The correct answer is $f'(x_0) = \cos(x_0) = 1/3$. We notice that the values $D(i,0)$, which are obtained without any extrapolation, are not very accurate. Acceptable accuracy is reached with Richardson extrapolation with $n = 4$.

4.5 First-derivative formulas via interpolation polynomials

An important stratagem can be used to approximate derivatives (as well as integrals and other quantities). The function f is first approximated by a polynomial p. Then we simply approximate the derivative of f by the derivative of p: $f'(x) \approx p'(x)$. Of course, we must be careful with this strategy because the behavior of p may be oscillatory.

In practice, the approximating polynomial is often determined by interpolation at a few points. For two nodes, we have

$$p_1(x) = f(x_0) + f[x_0, x_1](x - x_0)$$

Consequently,

$$f'(x) \approx p'(x) = f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

If $x_0 = x$ and $x_1 = x + h$, this formula gives

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

If $x_0 = x - h$ and $x_1 = x + h$, we get

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

Now consider interpolation at three nodes

$$p_2(x) = f(x_0) + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1)$$

The derivative is

$$p_2'(x) = f[x_0, x_1] + f[x_0, x_1, x_2](2x - x_0 - x_1)$$

Here the first term is the same as in the previous case and the second term is a refinement or a correction term.

The accuracy of the leading term depends on the choice of the nodes x_0 and x_1. If we have $x = (x_0 + x_1)/2$, then the correction term is zero. Thus the first term must be more accurate than in other cases. This is why choosing a *central difference* $x_0 = x - h$ and $x_1 = x + h$ gives more accurate results than $x_0 = x$ and $x_1 = x + h$.

In general, an analysis of errors goes as follows: Suppose that p_n is the polynomial of least degree that interpolates f at the nodes $x_0, x_1, \ldots, x_n$. Then according to first theorem on interpolating errors (Sec.3.2):

$$f(x) - p_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) w(x)$$

with ξ dependent on x and $w(x) = (x - x_0) \ldots (x - x_n)$. Differentiating gives

$$f'(x) - p_n'(x) = \frac{1}{(n+1)!} w(x) \frac{d}{dx} f^{(n+1)}(\xi) + \frac{1}{(n+1)!} f^{(n+1)}(\xi) w'(x)$$

The first observation is that $w(x)$ vanishes at each node. The evaluation is simpler if it is done at a node x_i:

$$f'(x_i) = p_n'(x_i) + \frac{1}{(n+1)!} f^{(n+1)}(\xi) w'(x_i)$$

The second observation is that it becomes simpler if x is chosen such that $w'(x) = 0$. Then

$$f'(x_i) = p_n'(x_i) + \frac{1}{(n+1)!} w(x) \frac{d}{dx} f^{(n+1)}(\xi)$$

Example.

Derive a first-derivative formula for $f'(x)$ using an interpolating polynomial $p_3(x)$ obtained with four nodes x_0, x_1, x_2 and x_3. Use central difference in choosing the nodes.

Solution. The interpolating polynomial written in the Newton form is

$$\begin{aligned} p_3(x) = {} & f(x_0) + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) \\ & + f[x_0, x_1, x_2, x_3](x - x_0)(x - x_1)(x - x_2) \end{aligned}$$

Its derivative is

$$\begin{aligned} p_3'(x) = {} & f[x_0, x_1] + f[x_0, x_1, x_2](2x - x_0 - x_1) \\ & + f[x_0, x_1, x_2, x_3]\{(x - x_1)(x - x_2) + (x - x_0)(x - x_2) \\ & + (x - x_0)(x - x_1)\} \end{aligned}$$

Formulas for the divided differences can be obtained recursively:

$$\begin{aligned} f[x_i] &= f(x_i) \\ f[x_i, x_{i+1}] &= \frac{f[x_{i+1}] - f[x_i]}{x_{i+1} - x_i} \\ f[x_i, x_{i+1}, x_{i+2}] &= \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i} \end{aligned}$$

We now choose the nodes using central difference:
$x_0 = x - h, \quad x_1 = x + h, \quad x_2 = x - 2h, \quad x_3 = x + 2h.$

This allows us to write out the formulas for the divided differences:

$$f[x_0,x_1] = \frac{f(x_1)-f(x_0)}{x_1-x_0} = \frac{f(x+h)-f(x-h)}{2h}$$

and

$$\begin{aligned} f[x_0,x_1,x_2] &= \frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_1} \\ &= \frac{1}{-h}\left[\frac{f(x-2h)-f(x+h)}{-3h} - \frac{f(x+h)-f(x-h)}{2h}\right] \\ &= \frac{1}{6h^2}\left[2f(x-2h)-3f(x-h)+f(x+h)\right] \end{aligned}$$

Similarly for $f[x_1,x_2,x_3]$.

Finally, we obtain

$$\begin{aligned} &f[x_0,x_1,x_2,x_3] = \\ &\frac{1}{12h^3}\left[f(x+2h)-2f(x+h)+2f(x-h)-f(x-2h)\right] \end{aligned}$$

The two remaining factors in the equation for p_3 are

$$(2x-x_0-x_1) = 2x-x+h-x-h = 0$$

and

$$\begin{aligned} (x-x_1)(x-x_2)+(x-x_0)(x-x_2)+(x-x_0)(x-x_1) = \\ -2h^2+2h^2-h^2 = -h^2 \end{aligned}$$

The resulting formula is

$$\begin{aligned} f'(x) \approx &\frac{1}{2h}\left[f(x+h)-f(x-h)\right] \\ &-\frac{1}{12h}\left[f(x+2h)-2f(x+h)+2f(x-h)-f(x-2h)\right] \end{aligned}$$

Numerical evaluation

The obtained formula

$$f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)] \\ - \frac{1}{12h}[f(x+2h) - 2f(x+h) + 2f(x-h) - f(x-2h)]$$

can be used in the Richardson extrapolation algorithm to speed up the convergence. As an example, the algorithm is applied to evaluate f' at $x = 1.2309594154$ for $f(x) = \sin(x)$ and $h = 1$.

We obtain the following output:

```
D(n,0)      D(n,1)      D(n,2)      D(n,3)      D(n,4)
0.32347058
0.33265926  0.33572215
0.33329025  0.33350059  0.33335248
0.33333063  0.33334409  0.33333365  0.33333335
0.33333317  0.33333401  0.33333334  0.33333334  0.33333334
```

Using the original formula (first term of the equation above)

```
D(n,0)      D(n,1)      D(n,2)      D(n,3)      D(n,4)
0.28049033
0.31961703  0.33265926
0.32987195  0.33329025  0.33333232
0.33246596  0.33333063  0.33333332  0.33333333
0.33311636  0.33333317  0.33333333  0.33333334  0.33333334
```

Notice that the first column of the first output has the same values as the second column of the second output! This is a good example of how the Richardson algorithm works in refining the approximation.