7

INTERPOLATION

Interpolation is one of the methods of approximating functions. In the following paragraphs we take into consideration a function defined through a table, so-called *tabular function*. Knowing the value for the function in certain points (x_i, $f(x_i)$) in which the function is defined, we pose the problem of trying to find out the values of the function in some other, different points, where the function is currently unknown. This problem can appear in the case of the graphical representation of the tabular function or in another case that necessitates knowing the value of the function in a point for which the value isn't known - that is, the ordinate value for some unknown abscissa.

When one knows *the analytical expression* for some function: $y = f(x)$, the unknown function's values (the ordinates) are calculated by replacing the proper argument in that function. When we do not know the function's analytical expression, we approximate the function with another one, which we think it best approximates the first function in the points where the unknown function's value is required. Approximating a given function either by a linear variation or by an nth degree polynomial can lead to ordinates that are very close to the original ones.

7.1. POLYNOMIAL INTERPOLATION. LAGRANGE'S INTERPOLATION POLYNOMIAL

Let's consider the tabular function defined in the Table 7.1:

Table 7.1.

x	x_0	x_1	.	x_k	.	x_{n-1}
y	y_0	y_1	.	y_k	.	y_{n-1}

We know the values of the function in n points. Through n points we can draw a uniquely defined polynomial of $(n-1)^{th}$ degree.

Given the polynomial:

$$P_{n-1}(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1 x + a_0 \tag{7.1}$$

According to Table 7.1 we have the system:

$$\begin{cases} a_{n-1}x_0^{\;n-1} + a_{n-2}x_0^{\;n-2} + \cdots\cdots + a_1x_0 + a_0 = y_0 \\ a_{n-1}x_1^{\;n-1} + a_{n-2}x_1^{\;n-2} + \cdots\cdots + a_1x_1 + a_0 = y_1 \\ ------------------------ \\ a_{n-1}x_{n-1}^{n-1} + a_{n-2}x_{n-1}^{n-2} + \cdots\cdots + a_1x_{n-1} + a_0 = y_{n-1} \end{cases} \tag{7.2}$$

This system has *n* equations with *n* unknowns, $a_0, a_1, a_2, ..., a_{n-2}, a_{n-1}.$, so it is a linear system of equations.

We consider the *homogenous* system:

$$\begin{cases} a_{n-1}x_0^{\;n-1} + a_{n-2}x_0^{\;n-2} + \cdots\cdots + a_1x_0 + a_0 = 0 \\ a_{n-1}x_1^{\;n-1} + a_{n-2}x_1^{\;n-2} + \cdots\cdots + a_1x_1 + a_0 = 0 \\ ------------------------ \\ a_{n-1}x_{n-1}^{n-1} + a_{n-2}x_{n-1}^{n-2} + \cdots\cdots + a_1x_{n-1} + a_0 = 0 \end{cases} \tag{7.3}$$

The homogenous system's determinant is different than 0. If the determinant would be null, the *(n-1)*th degree polynomial would have *n* solutions: $x_0, x_1, x_2, ..., x_{n-1}$ which is impossible. This homogenous system only accepts the trivial solution. Therefore the determinant of the system is not null. This determinant is also the determinant of the system (7.2). This system is a CRAMER system having unique solutions. Consequently the *(n-1)*th degree polynomial is *unique*. For the sake of computations' simplification we will write the polynomial using the following form:

$$P_{n-1}(x) = x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0 \tag{7.4}$$

It is obtained from the polynomial (7.1) through division by a_{n-1} while leaving the after division coefficients notation unchanged.

Let us consider the following polynomials, that verify (7.2) or (7.3):

$$\begin{aligned} &\Pi_0(x) = (x - x_1)(x - x_2)...(x - x_{n-1}) \\ &\Pi_1(x) = (x - x_0)(x - x_2)...(x - x_{n-1}) \\ &------------------- \\ &\Pi_{n-1}(x) = (x - x_0)(x - x_1)...(x - x_{n-2}) \end{aligned} \tag{7.5}$$

We will write the polynomial $P_{n-1}(x)$ under the form:

$$P_{n-1}(x) = b_0\Pi_0(x) + b_1\Pi_1(x) + ... + b_k\Pi_k + ... + b_{n-1}\Pi_{n-1}(x) \tag{7.6}$$

We must find out the coefficients $b_0, b_1, ..., b_{n-1}$.

$$b_0 = \frac{P_{n-1}(x_0)}{\Pi_0(x_0)},\ b_1 = \frac{P_{n-1}(x_1)}{\Pi_2(x_1)}, ..., b_k = \frac{P_{n-1}(x_k)}{\Pi_k(x_k)}, ..., b_{n-1} = \frac{P_{n-1}(x_{n-1})}{\Pi_{n-1}(x_{n-1})} \tag{7.7}$$

Following as such, by replacing (7.7) into (7.6) we can write:

$$P_{n-1}(x)=\sum_{i=0}^{n-1}P_{n-1}(x_i)\frac{\Pi_i(x)}{\Pi_i(x_i)}=\sum_{i=0}^{n-1}y_i\frac{\prod_{j=0,j\neq i}^{n-1}\left(x-x_j\right)}{\prod_{j=0,j\neq i}^{n-1}\left(x_i-x_j\right)}=\sum_{i=0}^{n-1}y_i\prod_{j=0,j\neq i}^{n-1}\frac{x-x_j}{x_i-x_j}$$

The *(n-1)th* degree polynomial that passes through *n* given points, also called the *Lagrange interpolation polynomial*, has the formula:

$$P_{n-1}(x)=\sum_{i=0}^{n-1}\left(y_i\cdot\prod_{j=0,j\neq i}^{n-1}\left(\frac{x-x_j}{x_i-x_j}\right)\right) \tag{7.8}$$

The expression of the Lagrange polynomial depends upon the known data point coordinates and the variable x. With the aid of polynomial formula (7.8), which approximates a function, we can compute the function's value in any unknown point placed in-between x_0 and x_{n-1}, so we can say that we compute the ordinate's pair for an unknown abscissa.

7.1.1. TRUNCATION ERROR IN LAGRANGE'S INTERPOLATION

The truncation error is given by the difference between the function we want to interpolate $f(x)$ and the interpolation polynomial in the LAGRANGE form:

$$e_T=f(x)-\sum_{i=0}^{n-1}y_i\prod_{j=0,j\neq i}^{n-1}\frac{x-x_j}{x_i-x_j}=F(x) \tag{7.9}$$

We build the function:

$$G(x)=(x-x_1)(x-x_2)...(x-x_n) \tag{7.10}$$

With both $F(x)$ from (7.9) and $G(x)$ from (7.10) we conclude the following function:

$$H(t)=F(x)G(t)-F(t)G(x) \tag{7.11}$$

that has the following properties:

1) Since any $H(x_j)=0$ for j = 1,...,n-1, while replacing $x=x_j$ in formula (7.9) the resulting $F(x_j)=0$; j = 1, 2,...,(n-1) and $G(x_j)=0$; j=1, 2, . . . ,(n-1);
2) $H(x)=0$

On the basis of the *mean value theorem* it results that there are *n* points $\xi_0,\xi_1,\xi_2,...,\xi_{n-1}$ for which the derivative $H'(t)=0$,

$$H'(\xi_0),H'(\xi_1),H'(\xi_2),...,H'(\xi_n),H'(\xi_{n+1}),\quad \xi_i\in[x_0,x_n]\ i=0,...,n-1$$

Continuing to apply the mean value theorem, we end up to the equality:

$$H^{(n)}(\xi)=0\ \text{, where } x_1<\xi<x_n$$

From (7.11) we have:

$$H^{(n)}(t) = F(x)G^{(n)}(t) - F^{(n)}(t)G(x) = n!F(x) - f^{(n)}(t)G(x)$$
$$H^{(n)}(\xi) = 0 = n!F(x) - f^{(n)}(\xi)G(x) \qquad (7.12)$$

or, equivalently:

$$e_T = F(x) = -\frac{f^{(n)}(\xi)}{n!}G(x), \text{ with: } x_1 < \xi < x_n \qquad (7.13)$$

This formula represents the *truncation error* for the Lagrange's interpolation method.

7.1.2 ROUNDING ERROR IN LAGRANGE'S INTERPOLATION

We consider the LAGRANGE polynomial given by equation (7.8) which we will write under the form:

$$P(x) = \sum_{i=0}^{n-1} y_i z_i \text{, where } z_i = \prod_{j=0, j\neq i}^{n-1} \frac{x - x_j}{x_i - x_j} \text{ and } p_i = \sum_{k=0}^{i} y_k z_k \text{ , } P(x) = p_n \qquad (7.14)$$

We build the procedural graph (see *Chapter 1 - Errors*) for computing the rounding error for expression (7.14) which represents the ith order product from the Lagrange polynomial expression. In the product expression there are operations of *subtraction*, *division* and *multiplication*. For each node in the graph where there is an arithmetic operation involved we give the symmetric type rounding error. For the total rounding error calculation of the Lagrange polynomial expression we create a procedure graph and for the sum expression in (7.14).

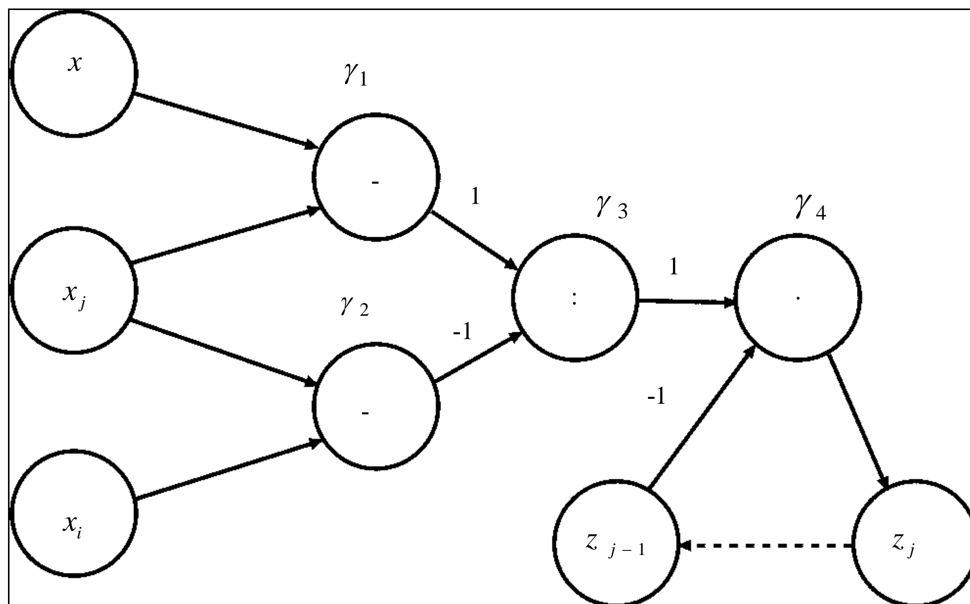

Fig.7.1 - The procedural graph for expression z.

Let's consider all the values x_i, $i=1,2,\dots,n$, without errors and we note with $\pi_k^i, k=0\dots n-1; i=0,1,\dots n-1$, the errors in the members of the product. In these conditions, for z_0 we obtain the rounding error: $\pi_0=0$, because z_0 is initialized with 1 and therefore is assumed as an exact representation, that is its error is null.

$$
\begin{aligned}
&\pi_0 = 0 \\
&\pi_0^i = \gamma_{11}^i - \gamma_{21}^i + \gamma_{31}^i + \gamma_{41}^i \\
&\pi_1^i = \gamma_{12}^i - \gamma_{22}^i + \gamma_{32}^i + \gamma_{42}^i + \pi_0^i \\
&-------------------- \\
&\pi_{n-1}^i = \gamma_{1,n-1}^i - \gamma_{2,n-1}^i + \gamma_{3,n-1}^i + \gamma_{4,n-1}^i + \pi_{n-2}^i
\end{aligned}
\tag{7.15}
$$

In the expression (7.15) the term π_i^i is missing. From (7.15) results:

$$\pi_{n-1}^i = \sum_{\substack{j=0 \\ i\neq j}}^{n-1} \left(\gamma_{1j} - \gamma_{2j} + \gamma_{3j} + \gamma_{4j}\right) \tag{7.16}$$

If we consider $\gamma_{kj} \leq 10^{-t} = \gamma$ where t is the computing machine and note with π_i the product z_i's error, we have

$$\pi_i \leq 4(n-1)\gamma \tag{7.17}$$

or, for z_i, $i=0,1,2,\dots n-1$, we have

$$\pi_i \leq 4(n-1)\gamma \qquad i=0,1,2,\dots,n-1 \tag{7.18}$$

We build the procedure graph for p_i.

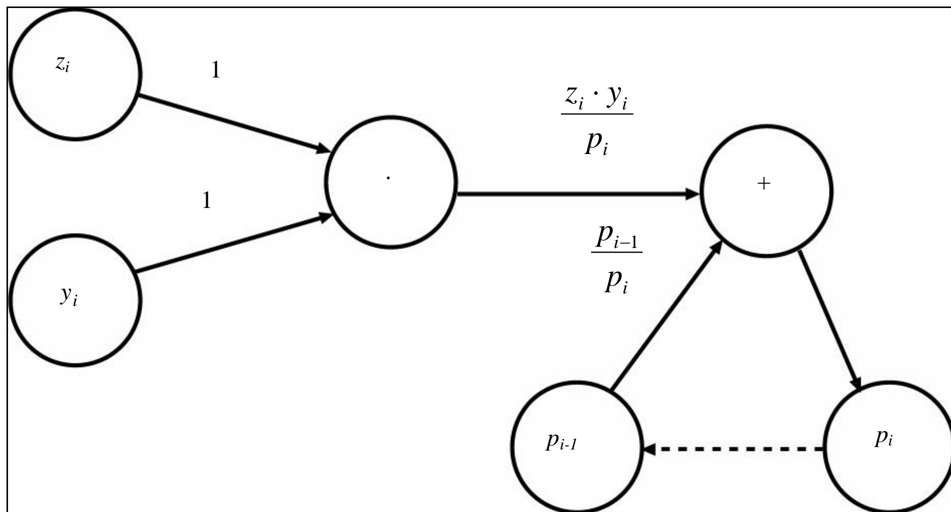

Fig.7.2 - The procedural graph for the expression $p_i = \sum_{k=0}^{n-1} y_k z_k$

We consider the initial error in point p_i to be null, the rounding errors in nodes 5, 6 being noted as $\gamma_{5i}, \gamma_{6i}, i = 0,1,..,n-1$ and the errors in the sum's terms $\varepsilon_i, i = 0,1,...n-1$.

$$\varepsilon_0 = (\pi_0 + e_0 + \gamma_{50})\frac{z_0 y_0}{p_0} + \gamma_{60}$$

$$\varepsilon_1 = (\pi_1 + e_1 + \gamma_{31})\frac{z_1 y_1}{p_1} + \varepsilon_1 \frac{p_1}{p_2} + \gamma_{61}$$

--- (7.19)

$$\varepsilon_{n-1} = (\pi_{n-1} + e_{n-1} + \gamma_{5,n-1})\frac{z_n y_{n-1}}{p_{n-1}} + \varepsilon_{n-1} \frac{p_{n-2}}{p_{n-1}} + \gamma_{6,n-1}$$

We know that $\pi_i \le 4(n-1)\gamma$ for *i=0,1,..,n-1* and consider that $|e_i| < e$ for *i*=0.1….,n-1. In these conditions we will obtain: $|\gamma_{kj}| < \gamma$

$$\varepsilon_{n-1} \mid p_{n-1} \mid \le [(4n-3)\gamma + e]\sum_{i=0}^{n-1}|z_i y_i| + \gamma\sum_{i=0}^{n-1}|p_i| \le [(4n-3)\gamma + e]\sum_{i=0}^{n-1} p_i + \gamma\sum_{i=0}^{n-1}|p_i|$$

We can perform the following increase $\sum_{i=0}^{n-1} | p_i | \le np_{n-1}$ and result in:

$$\varepsilon_{n-1} \le 2\gamma n(2n-1) + en \qquad (7.20)$$

The expression that represents the rounding error is actually the rounding error for the Lagrange's interpolation.

7.1.3. SINGULARITIES FOR LAGRANGE'S POLYNOMIAL

If we consider *two points,* the interpolation becomes linear:

$$y = y_1 \frac{x - x_2}{x_1 - x_2} + y_2 \frac{x - x_1}{x_2 - x_1} \qquad (7.21)$$

The *truncation error* for linear interpolation becomes:

$$e_T = \frac{f''(\xi)}{2!}(x - x_1)(x - x_2) \quad x_1 < \xi < x_2 \qquad (7.22)$$

and the *rounding error*:

$$\varepsilon_2 \le 12\gamma + 2e \qquad (7.23)$$

where γ and e are stated in paragraph 7.1.2.

For *three points*, the interpolation becomes a 2nd degree polynomial interpolation, that is, a *quadratic interpolation polynomial* - in fact a *parabola*).

7.1.4. Algorithm 7.1. The LAGRANGE'S interpolation polynomial

```
Arguments
(
   n: number of known points, integer;
   x: horizontal axis coordinate for known points, array;
   y: vertical axis coordinate for known points, array;
   x: interpolation point, real;
)
{
      i, j: counters, integer;
      sum: variable to retain sum, real;
      prod: variable to retain product, real;

      sum=0;
      for i=0..n;
       { prod=1;
          for j=0..n if j≠0 then
                     prod = prod * (x̄ - x_j)/(x_i - x_j);
          compute
          sum = sum + y_i * prod;
       }
      The interpolated value is sum;
}
```

7.1.5. Implementation for Algorithm 7.1

```
/* Function that implements the Lagrange's method
 * of interpolation.
 * Function returns the interpolated value.
 **/
double Lagrange(    int n,
                    double x[],
                    double y[],
                    double point
                    )
{
   int i,j;
   double sum=0,prod;
   for(i=0; i<=n; i++)
   {
       prod=1;
       for(j=0;j<=n;j++)
          if(j!=i)prod*=(point-x[j])/(x[i]-x[j]);
       sum+=y[i]*prod;
   }
   return sum;
}
```

7.2. NEWTON'S FIRST KIND INTERPOLATION POLYNOMIAL

This interpolation polynomial is expressed as a function of *finite differences*. Let us assume some function $f:[a,b]\rightarrow\mathbf{R}$ and the net $x_1, x_2, x_3,\ldots,x_n$ with a constant step k.

Definition 7.1. We denote by ***first order finite difference*** the expression:

$$\Delta f(x) = f(x+h) - f(x) \tag{7.24}$$

where h is the constant step among abscissas. The nth order *finite difference* is then computed using:

$$\Delta^n f(x) = \Delta\left(\Delta^{n-1} f(x)\right) \tag{7.25}$$

The finite differences have the following properties:

1. The finite difference operator is *linear*:

$$\Delta(c_1 f_1 + c_2 f_2) = c_1 \Delta f_1 + c_2 \Delta f_2 \tag{7.26}$$

2. The nth order finite difference is computed with the formula:

$$\Delta^n f(x) = \sum_{k=0}^{n} (-1)^k C_n^k f(x + (n-k)h) \tag{7.27}$$

3. The finite differences can also be obtained using Table 7.2 below.

Definition 7.2. We define the *generalized* n^{th} *order power* of x the expression:

$$x^{[n]} = x(x-h)(x-2h)\ldots(x-(n-1)h) \tag{7.28}$$

For $h=0$ the generalized power is the same as the ordinary power.

1. The finite difference over the generalized power is computed as:

$$\Delta x^{[n]} = nhx^{[n-1]} \tag{7.29}$$

2. The kth order finite difference of generalized power is:

$$\Delta^k x^{[n]} = n(n-1)(n-2)\ldots(n-k+1)x^{[n-k]} \tag{7.30}$$

Table 7.2

x_i	y_i	Δy_i	$\Delta^2 y_i$	$\Delta^3 y_i$	$\Delta^4 y_i$	$\Delta^5 y_i$	$\Delta^6 y_i$
x_0	y_0	Δy_0					
x_1	y_1	Δy_1	$\Delta^2 y_0$	$\Delta^3 y_0$			
x_2	y_2	Δy_2	$\Delta^2 y_1$	$\Delta^3 y_1$	$\Delta^4 y_0$	$\Delta^5 y_0$	
x_3	y_3	Δy_3	$\Delta^2 y_2$	$\Delta^3 y_2$	$\Delta^4 y_1$	$\Delta^5 y_1$	$\Delta^6 y_0$
x_4	y_4	Δy_4	$\Delta^2 y_3$	$\Delta^3 y_3$	$\Delta^4 y_2$		
x_5	y_5	Δy_5	$\Delta^2 y_4$				
x_6	y_6						

Let us consider the tabular function from table (7.1), where the net $x_0, x_1, x_2, x_3, ..., x_n$ has a constant step *h*.

An nth order polynomial passes through *n+1* points and we can look for it using the form:

$$P_n(x) = C_0 + C_1(x-x_0)^{[1]} + C_2(x-x_0)^{[2]} + ... + C_n(x-x_0)^{[n]} \quad (7.31)$$

where $(x-x_0)^{[i]} = (x-x_0)(x-x_1)...(x-x_{i-1})$, $i = 1, 2, ..., n$

and the coefficients $C_0, C_1, ..., C_n$ represent unknowns that we will compute. We can observe that

$$P_n(x_0) = y_0 = C_0 \quad (7.32)$$

We calculate the 1st order finite difference:

$$\Delta P_n(x) = C_1 h + 2C_2 h(x-x_0)^{[1]} + ... + nC_n h(x-x_0)^{[n-1]} \quad (7.33)$$

After performing the substitution $x = x_0$ we get $\Delta P_n(x_0) = C_1 1! h$.

We can also compute: $C_1 = \frac{\Delta P_n(x_0)}{1!h}$ (7.34)

By continuing the finite difference calculus in point x_0 we observe that:

$$C_k = \frac{\Delta^{(k)} P_n(x_0)}{k! h^k}, \; \Delta^k P_n(x_0) = \Delta^k y_0, \text{ with } k=0, 1, 2, ... n. \quad (7.35)$$

Taking into account the coefficient computation formulas, the 1st kind Newton interpolation polynomial can be written as:

$$P_n(x_0) = y_0 + \frac{\Delta y_0}{1!h}(x-x_0)^{[1]} + \frac{\Delta^2 y_0}{2!h^2}(x-x_0)^{[2]} + \cdots + \frac{\Delta^n y_0}{n!h^n}(x-x_0)^{[n]} \quad (7.36)$$

Because in the computation required for the coefficients we have used *forward finite differences* (see Table 7.2), the polynomial is called NEWTON's interpolation polynomial of the ***first kind***.

If the approximation point is placed near x_0 (or in the *neighborhood* of x_0), we recommend the utilization of this method because it leads to smaller errors.

7.2.1. Algorithm 7.2. NEWTON'S first kind interpolation method

```
Arguments
(
    n: degree of the interpolation polynomial, integer;
    h: step between abscissas of the known points, real;
    y: the ordinates for known points, array;
    xp: abscissa of the interpolation point;
    x0: abscissa for the first known point, real;
)
{
  sum: partial sum container, real;
  prod: product store variable, real;
  i,j: counters, integer;
  sum = y0;
```

```
    prod = 1;
    for i=1...n
    {
        for j=0...n-1 compute yj = yj+1 - yj
        prod = prod*(xp - (x0 - (i-1)*h))*1/h*1/i;
        sum = sum + y0*prod;
    }
    The interpolated value is sum;
}
```

7.2.2. Algorithm 7.2. Implementation for algorithm 7.2

```
/* The function that implements Newton's first kind
 * interpolation method.
 * Function returns the interpolated value.
 **/
double Newton1(     int n,
                    double vi,
                    double pas,
                    double y[],
                    double point
                )
{
double sum=y[0], prod=1;
int i, j;
for(i=1; i<=n; i++)
{
       for(j=0; j<=n-i; j++) y[j] = y[j+1]-y[j];
       prod *= (point-(vi+(i-1)*pas))/(i*pas);
       sum += y[0]*prod;
}
return sum;
}
```

7.3. NEWTON'S SECOND KIND INTERPOLATION POLYNOMIAL

For the function given in Table 7.1 we look for an nth degree polynomial that passes through all *n+1* points. The interpolation polynomial is of the following form:

$$P_n(x) = C_0 + C_1(x - x_n) + C_2(x - x_n)(x - x_{n-1}) + ... + C_n(x - x_n)(x - x_{n-1})...(x - x_1) \quad (7.37)$$

The above polynomial can also be written as a function of *generalized power* as:

$$P_n(x) = C_0 + C_1(x - x_n)^{[1]} + C_2(x - x_{n-1})^{[2]} + C_3(x - x_{n-2})^{[3]} + ... + C_n(x - x_1)^{[n]} \quad (7.38)$$

It can be observed that: $P_n(x_n) = y_n = C_0$ (7.39)

We compute the 1st order *finite difference* over the polynomial in (7.38):

$$\Delta P_n(x) = C_1 1!h + 2C_2 h(x - x_{n-1})^{[1]} + 3C_3 h(x - x_{n-2})^{[2]} + ... + nC_n h(x - x_1)^{[n-1]} \quad (7.40)$$

For $x = x_{n-1}$, from (7.40) we have $\Delta y_{n-1} = C_1 1!h$ and therefore: $C_1 = \frac{\Delta y_{n-1}}{1!h}$.

By continuing the finite difference calculus in the points $x_{n-2}, x_{n-3},\dots,x_{n-k}$, for the rank k we obtain the general formula used to compute the C_k coefficient:

$$C_k = \frac{\Delta^{(k)} y_{n-k}}{k!h^k} \tag{7.41}$$

By substituting $k = 0,1,2,..,n$ in (7.41) we obtain all the polynomials' coefficients. Taking into account formula (7.41) then the nth degree polynomial (7.38) can be written as:

$$P_n(x) = y_n + \frac{\Delta y_{n-1}}{1!h}(x-x_n)^{[1]} + \frac{\Delta^2 y_{n-2}}{2!h^2}(x-x_{n-1})^{[2]} + \cdots + \frac{\Delta^n y_0}{n!h^n}(x-x_1)^{[n]} \tag{7.42}$$

This polynomial is called NEWTON's interpolation polynomial of ***second kind***, because the *backward finite differences* were used (see Table 7.2).

If the approximation point is placed near x_n (or in the *neighborhood* of x_n), we recommend the utilization of this method because it leads to smaller errors.

7.3.1. Algorithm 7.3. NEWTON's second kind interpolation polynomial

```
Arguments
(
   n: the polynomial's degree, integer;
   h: the step between the known abscissas, real;
   y: the known ordinates, array;
   xp: the abscissa for the interpolation point;
   xn: maximum value among the known abscissas, real;
)
{
   sum: variable to retain partial sums, real;
   prod: variable to retain products, real;
   i, j: counters, integer;
   {
      sum = yn;
      prod = 1;
      for i = 1 to n
      {
         for j = n downto i compute yj = yj - yj-1
         prod = prod*(xp-(xn-(i-1)*h))*1/h*1/i;
         sum=sum + yn*prod;
      }
   The interpolated value is sum;
}
```

7.3.1.1. Implementation for Algorithm 7.3

```
/* Function that implements Newton's second kind
 * interpolation formula
 * The function returns the interpolated value.
 */
double Newton2(     int n,
```

```
                        double vi,
                        double pas,
                        double y[],
                        double point
                      )
{
    double sum, prod, vf;
    int i, j;

    vf = vi + n*pas;
    sum = y[n]; prod=1;
    for(i=1; i<=n; i++)
    {
        for(j=n; j>=i; j--) y[j] = y[j] - y[j-1];
        prod *= (point-(vf-(i-1)*pas))/(i*pas);
        sum += y[n]*prod;
    }
    return sum;
}
```

7.3. NEWTON'S INTERPOLATION POLYNOMIAL USING DIVIDED DIFFERENCES *(aka Newton's third kind interpolation polynomial)*

Let us consider the function $f(x)$ given under the form described in Table 7.1 where the network $x_0, x_1, x_2, ..., x_n$ from the definition domain of the function hasn't got a constant step.

Definition 7.3. We call *divided difference of (k+i)th order* of some function *f(x)* the expression:

$$f(x_{i-1}, x_i, x_{i+1}, ..., x_{i+k}) = \frac{f(x_i, x_{i+1}, ..., x_{i+k}) - f(x_{i-1}, x_i, ..., x_{i+k-1})}{x_{i+k} - x_{i-1}} \qquad (7.43)$$

We consider the nth degree polynomial, under the following format:

$$P_n(x) = C_0 + C_1(x - x_0) + C_2(x - x_0)(x - x_1) + ... + C_n(x - x_0)(x - x_1)...(x - x_{n-1}) \qquad (7.44)$$

knowing $n+1$ points (x_i, y_i), i=0, ...,n , that verify the polynomial.

It can be observed that: $P_n(x_0) = y_0 = C_0$ (since our polynomial must pass through the pairs of point that we have in the input table).

We compute the divided difference of first order for the polynomial $P_n(x)$ and substitute $x = x_1$ in (7.44). We get: $P_n(x_0, x_1) = C_1$.

Continuing to compute, we determine the kth order divided difference and taking $x = x_k$ we obtain the value for coefficient C_k :

$$C_k = P_n(x_0, x_1, x_2, ..., x_k),\ k=0,\ 1, ...,\ n \qquad (7.45)$$

Taking into account formula (7.45), the polynomial in (7.44) can be written as:

$$P_n(x) = y_0 + P_n(x_0, x_1)(x - x_0) + P_n(x_0, x_1, x_2)(x - x_0)(x - x_1) + ... \\ + P_n(x_0, ..., x_n)(x - x_0)(x - x_1)...(x - x_{n-1}) \tag{7.46}$$

where each divided difference is computed using (7.43). The resulting polynomial (7.46) is also called *NEWTON's interpolation polynomial using divided difference.*

7.4.1. Algorithm 7.4. NEWTON'S interpolation polynomial using divided differences

```
Arguments
(
   n: number of given points in function, integer;
   x: horizontal coordinates for given points, array;
   y: the ordinates for given points, array;
   x̄ : the abscissa for which we find the interpolated
      value, real;
)
{
   i, j: counters, integer;
   sum: variable to retain partial sums, real;
   prod: variable to retain partial products, real;

   sum = y0;
   prod = 1;
   for i = 1 to n
   {
      for j=0 to n-i      compute   y_j = (y_{j+1} - y_j)/(x_{j+i} - x_j);
      prod = prod*(x̄-x_{i-1});
      sum = sum + y0*prod;
   }
   print "The interpolated value is: sum";
}
```

7.4.2. Implementation for Algorithm 7.4

```
/* Function to implement Newton's interpolation
 * polynomial using divided differences.
 * This function returns the interpolated value.
 */
double NewtonDD
(
    int n,
    double x[],
    double y[],
    double point
)
{
   int i, j;
   double sum, prod;
   sum = y[0]; prod=1;
   for(i=1; i<=n; i++)
   {
```

```
            for(j=0; j<=n-i; j++)
                  y[j]=(y[j+1]-y[j])/(x[j+i]-x[j]);
            prod *= (point-x[i-1]);
            sum += y[0]*prod;
        }
        return sum;
}
```

7.4. AITKEN'S INTERPOLATION METHOD

AITKEN's interpolation method gives the same result as LAGRANGE's method, except that using this method the result is not a polynomial, but we realize multiple *linear interpolations*. With each interpolation, the number of remaining points is shrunk by 1, and the function that passes through two points increases with a degree of 1, so in the final iteration, if $n+1$ points are given, we obtain a nth degree function.

Let us consider the tabular function in Table 7.3.

Table 7.3

x	**y**					
x_0	y_0					
x_1	y_1	y_{01}				
x_2	y_2	y_{02}	y_{012}			
x_3	y_3	y_{03}	y_{013}	y_{0123}		
x_4	y_4	y_{04}	y_{014}	y_{0124}	----	
-----	----	----	----	----	----	
x_n	y_n	y_{0n}	y_{01n}	y_{012n}	----	$y_{0123...n}$

Then, writing the straight-lines' analytical equations, given the first pair (x_0, y_0) combined successively with all the other pairs (x_i, y_i) with i != 0 we get, for the first step, all the equations y_{0j} of the following straight-lines:

$$y_{0j} = \frac{x_j - \bar{x}}{x_j - x_0} y_0 + \frac{\bar{x} - x_0}{x_j - x_0} y_j \qquad j=1, 2, 3, ..., n$$

Similarly, at the second step of linear interpolation, the first pair is (x_1, y_{01}) and the other pairs are (x_j, y_{0j}), with $j=1, ..., n$. The new 'straight-lines' are now denoted by y_{01j}:

$$y_{01j} = \frac{x_j - \bar{x}}{x_j - x_1} y_{01} + \frac{\bar{x} - x_1}{x_j - x_1} y_{0j} \qquad j=2, 3, ..., n \qquad (7.47)$$

We continue likewise, by using the analytical equation for a straight line. In fact, what we are going to do is not indeed a straight line. We only use that equation, but at every step we use the equations computed before.

Finally, the only two pairs remain: (x_{n-1}, $y_{012...n-2}$) and (x_n, $y_{012...n-1}$).

Conclusively, the AITKEN's final interpolation polynomial, for some abscissa $\bar{x}$, is given by the following formula:

$$y_{012\dots n} = \frac{x_n - \bar{x}}{x_n - x_{n-1}} y_{012\dots(n-2)(n-1)} + \frac{\bar{x} - x_{n-1}}{x_n - x_{n-1}} y_{012\dots(n-2)n} \qquad (7.48)$$

7.4.1. Algorithm 7.5. AITKEN'S interpolation method

```
Arguments
(
   n: number of given points in function, integer;
   x: horizontal coordinates for given points, array;
   y: vertical coordinates for given points, array;
   xp: interpolation point, real;
)
{
   i, j: counters, integer ;

   for i=1...n;
      for j=i...n;
```

$$y_j = y_{j-1}\frac{x_p - x_j}{x_{i-1} - x_j} + y_j\frac{x_p - x_{i-1}}{x_j - x_{i-1}};$$

```
         compute
}
 Interpolated value is yn;
}
```

7.4.2. Implementation of algorithm 7.5

```
/* Function to implement Aitken's interpolation method.
 * It returns the interpolated value.
 */
double Aitken
(
    int n,
    double x[],
    double y[],
    double point
)
{
  int i, j;

  for(i=0; i<=n; i++)
    for(j=i; j<=n; j++)
      y[j]= y[j-1]*(point-x[j])/(x[i-1]-x[j])+
             y[j]*(point-x[i-1])/(x[j]-x[i-1]);
  return y[n];
}
```

7.5. *SPLINE* INTERPOLATION

The word "*spline*" means an elastic ruler under which, if weights get attached of it, can be made to pass through different points, in-between the endpoints of the ruler. Figure 7.1 depicts such a construction.

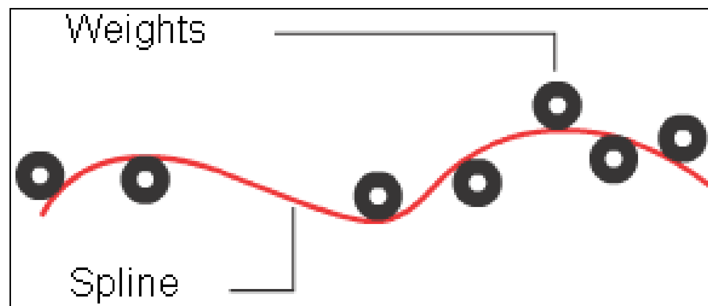

Figure 7.1 - A Spline curve, constructed from a mechanical point of view[*)].

Consider the function represented in table 7.4.

Table 7.4

x	x_1	x_2	x_3	----	x_n
y	y_1	y_2	y_3	----	y_n

We also consider $x_1 = a$ and $x_n = b$, $[a,b]$ being function f's interval of definition, and the horizontal coordinates $x_1, x_2, \ldots, x_n$ a division Δ pf the definition interval. The function values are

$$y_i = f(x_i),\ i = 1,2,\ldots,n \tag{7.49}$$

Definition 7.4. We call n^{th} *order spline function relative to division* Δ of some interval $[a,b]$ a function $S:[a,b] \to \mathbf{R}$ of class $C^{m-1}[a,b]$ whose restrictions $S_i(x)$ on each interval $[x_i, x_{i+1}]$ of division are the mth order polynomials, meaning:

$$S_i(x) = P_m^i(x) \qquad \text{if} \qquad x \in [x_i, x_{i+1}],\ \ i = 1,2,\ldots,(n-1) \tag{7.50}$$

The function $S(x)$ is a piecewise smooth function because its first (m-1) derivatives are continuous on $[a,b]$, and the mth order derivative is discontinuous in x_i, $i = 1,2,\ldots,n$. The smoothness degree of the function is m.

The function's restrictions are the polynomials:

$$S_i(x) = A_i x^m + B_i x^{m-1} + C_i x^{m-2} + E_i x^{m-3} + \ldots + R_i \tag{7.51}$$

if, $x \in [x_i, x_{i+1}]$, $i = 1,2,\ldots,(n-1)$

---

[*] According to:
http://www.autodesk.com/techpubs/aliasstudio/2010/index.html?url=WS1a9193826455f5ff4e421d7d11bf108001d-68c6.htm,topicNumber=d0e43698

These functions are differentiable up to (*n-1*) and are continuous along with their derivatives. The (*n-1*)th order derivative of $S_i(x)$ on the interval $[x_i, x_{i+1}]$ is a linear function that passes through the points (x_i, D_i) and (x_{i+1}, D_{i+1}). We denote by D_i the following: $D_i = S_i^{(m-1)}(x)$ $i=1,2,...,n$.

The linear equation results:

$$S_i^{(m-1)}(x) = \frac{D_{i+1}(x-x_i) + D_i(x_{i+1}-x)}{h_i} \tag{7.52}$$

where $h_i = x_{i+1} - x_i$, $i=1,2,...,(n-1)$

By integrating *m-1* times the relation (7.52) we obtain:

$$S_i^{(m-2)}(x) = \frac{D_{i+1}(x-x_i)^2 - D_i(x_{i+1}-x)^2}{2h_i} + C_{1i} \tag{7.53}$$

$$S^{(m-3)}(x) = \frac{D_{i+1}(x-x_i)^3 + D_i(x_{i+1}-x)^3}{6h_i} + C_{1i}x + C_{2i} \tag{7.54}$$

$$S_i'(x) = \frac{D_{i+1}(x-x_i)^{m-1} + (-1)^{m-2}(x_{i+1}-x)^{m-1}D_i}{(n-1)!h_i} + \frac{C_{1i}x^{m-3}}{(m-3)!} + \frac{C_{2i}x^{m-4}}{(m-4)!} + ... + C_{m-1,i} \tag{7.55}$$

$$S_i(x) = \frac{D_{i+1}(x-x_i)^m + (-1)^{m-1}(x_{i+1}-x)^{m-1}D_i}{n!h_i} + \frac{C_{1i}x^{m-2}}{(m-2)!} + \frac{C_{2i}x^{m-3}}{(m-3)!} + ... + C_{m-2,i}x + C_{m-1,i} \tag{7.56}$$

For the entire interval $[a,b]$ a linear system results by imposing both the condition to have $S_i(x_i) = y_i$, $i=1,2,..,n$ and the continuity for the *(m-1)* equations in all the points x_i. In the extremes x_1 and x_2 we write the LAGRANGE polynomial, we derive it to the *(m-1)th* order and find the values for the derivatives in x_1 and x_n. We obtain the unknowns: D_i, D_{i+1}, C_{1i},..., $C_{m-1,i}$ for each interval. In this case there are $n+(m-1)+(n-1)$ unknowns and $n+(m-1)+(n-1)$ equations. Being a linear system, it can be solved using one of the methods described in workshop *#3 - Numerical resolution of the systems of equations*.

The algorithm and program below were implemented taking into account the restrictions S_i, which are polynomials of degree 3.

In this case:

$$S_i(x) = A_i x^3 + B_i x^2 + C_i x + E_i \tag{7.57}$$

$$S_i''(x) = \frac{D_{i+1}(x-x_i) + D_i(x_{i+1}-x)}{h_i}$$

$$S'(x) = \frac{D_{i+1}(x-x_i)^2 - D_i(x_{i+1}-x)^2}{2h_i} + C_{1i}$$

$$S_i(x) = \frac{D_{i+1}(x-x_i)^3 - D_i(x_{i+1}-x)^3}{6h_i} + C_{1i}x + C_{2i} \tag{7.58}$$

From $S(x_i) = y_i$ *and* $S(x_{i+1}) = y_{i+1}$ we get:

$$C_{1i} = \frac{y_{i+1} - y_i}{h_i} - \frac{D_{i+1} - D_i}{6} h_i$$

$$C_{2i} = \frac{x_{i+1} y_i - x_i y_{i+1}}{h_i} - \frac{D_i x_{i+1} - D_{i+1} x_i}{6} h_i$$

By identifying relations (7.57) and (7.58) we get:

$$A_i = \frac{D_{i+1} + D_i}{6h_i}; B_i = \frac{D_i x_{i+1} - D_{i+1} x_i}{2h_i} \tag{7.59}$$

$$C_i = \frac{D_{i+1} x_i^2 - D_i x_{i+1}^2}{2h_i} + \frac{y_{i+1} - y_i}{h_i} - \frac{D_{i+1} - D_i}{6} h_i$$

$$E_i = \frac{x_{i+1}^3 D_i - x_i^3 D_{i+1}}{6h_i} + \frac{y_i x_{i+1} - y_{i+1} x_i}{h_i} - \frac{D_i x_{i+1} - D_{i+1} x_i}{6} h_i \tag{7.60}$$

On account of the first derivative's continuity in point x_i, $S_{i-1}^{'}(x_i) = S_i^{'}(x_i)$ we obtain:

$$\frac{h_{i-1}}{6} D_{i-1} + 2(h_i + h_{i-1}) D_i + h_i D_{i+1} = \left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right) 6 \tag{7.61}$$

Considering the first degree derivative in points x_1 and x_2 equal to:

$$y_1' = \frac{y_2 - y_1}{x_2 - x_1} - \frac{y_3 - y_2}{x_3 - x_2} + \frac{y_3 - y_1}{x_3 - x_1} \tag{7.62}$$

respectively

$$y_n' = -\frac{y_{n-1} - y_{n-2}}{x_{n-1} - x_{n-2}} + \frac{y_n - y_{n-1}}{x_n - x_{n-1}} + \frac{y_n - y_{n-2}}{x_n - x_{n-2}} \tag{7.63}$$

then we get a three-diagonal system in D_i, $i = 1,2,3,...,n$

$$\begin{cases} 2h_1 D_1 + h_1 D_2 = 6\left(\dfrac{y_2 - y_1}{h_1} - y_1'\right) \\ h_1 D_1 + 2(h_1 + h_2) D_2 + h_2 D_3 = 6\left(\dfrac{y_4 - y_3}{h_3} - \dfrac{y_3 - y_2}{h_3}\right) \\ \dots\dots\dots\dots\dots\dots \\ h_{i-1} D_{i-1} + 2(h_{i-1} + h_i) D_i + h_i D_{i+1} = 6\left(\dfrac{y_{i+1} - y_i}{h_i} - \dfrac{y_i - y_{i+1}}{h_i}\right) \\ \dots\dots\dots\dots\dots\dots \\ h_{n-2} D_{n-2} + 2(h_{n-2} + h_{n-1}) D_{n-1} + h_{n-1} D_n = 6\left(\dfrac{y_n - y_{n-1}}{h_{n-1}} - \dfrac{y_{n-1} - y_{n-2}}{h_{n-1}}\right) \\ h_{n-1} D_{n-1} + 2h_{n-1} D_n = 6\left(y_n' - \dfrac{y_n - y_{n-1}}{h_i}\right) \end{cases} \tag{7.64}$$

where y_1' şi y_n are given by expressions (7.62) and (7.63).

From system (7.64) the values of D_i, $i = 1,2,...,n,$ result. From (7.59) we obtain the coefficients for the restrictions on each interval, restrictions that approximate the given function. If $\bar{x}$ in which the function needs to be computed is given, we establish the interval in which $\bar{x}$ is located, and we compute the value of the function restriction on this interval in point $\bar{x}$.

7.5.1. Algorithm 7.6. SPLINE functions

```
Arguments
(
   n: number of given points in function, integer ;
   x: horizontal coordinate of given points, array ;
   y: vertical coordinates , array ;
   xp: interpolation point, real ;
   A,B,C: array of diagonal elements in the tri-diagonal system;
   TL: array of free terms in the tri-diagonal system;
)
{
   h: array of horizontal steps of points;
   S: array for solutions of the tri-diagonal system;
   derx1: derivative in x1, real;
   derx2: derivative in x2, real;
   num: the interval number in which xp is to be found, integer;
   valint: retains punctual 3rd degree polynomial value;
   xp: the unknown abscissa to interpolate, real;
   i, j: counters, integer;
   for i = 1 to n-1   compute
```

$$h_i = x_{i+1} - x_i;$$

```
   compute
```

$$derx1 = \frac{y_2 - y_1}{x_2 - x_1} - \frac{y_3 - y_2}{x_3 - x_2} + \frac{y_3 - y_1}{x_3 - x_1};$$

compute
$$derxn = -\frac{y_{n-1} - y_{n-2}}{x_{n-1} - x_{n-2}} + \frac{y_n - y_{n-1}}{x_n - x_{n-1}} + \frac{y_n - y_{n-2}}{x_n - x_{n-2}};$$

build Three-diagonal system

$$\begin{cases} 2.h_1.D_1 + h_1.D_2 + 0.D_3 + \text{-------------------} + 0.D_n = 6(\frac{y_2 - y_1}{h_1} - derx1) \\ 0.D_1 + h_1.D_2 + 2(h_1 + h_2)D_2 + h_2 D_3 + 0.D_4 + \text{--------} + 0.D_n = 6(\frac{y_3 - y_2}{h_2} - \frac{y_2 - y_1}{h_{i-1}}) \\ \text{---} \\ 0.D_1 + 0.D_2 + 0.D_3 + \text{--------------} + h_{n-1}D_{n-1} + 2.h_{n-1}.D_n = 6(derxn - \frac{y_n - y_{n-1}}{h_{n-1}}) \end{cases}$$

```
  Solve the system using algorithm (3.8), workshop #3.
   Compute S = Tridiagonal(A,B,C,TL);

  /* Search for interval containing interpolation point */
  i=1;
  repeat
      i=i+1;
  until xp<x[i];
```

With formula (6.59) compute A_i, B_i, C_i, E_i for i found;

compute $val\,\mathrm{int} = A_s.xp^3 + B_s.xp^2 + C_s.xp + E_s;$

```
  return valint;
}
```

7.5.2. Implementation of Algorithm 7.6

```
/* Function to implement interpolation through cubic Splines.
 * Function returns:
 * ~> 1 if value found;
 * ~> 0 if tri-diagonal system can be solved;
 * ~> 2 if interpolation value is not within the
 *definition interval;
*/
int Spline(  int ord,
             double xi[],
             double yi[],
             double point,
             double *valoare
          )
{
static double
a[NrMax],b[NrMax],c[NrMax],d[NrMax],der2[NrMax];
int i,cod,loc;
double d1,d2;

if( (point<xi[1])||(point>xi[ord]) )return 2;

/*Prepare tri-diagonal system for 2nd derivative */
/*Compute fist derivative in ends */
d1=    (yi[2]-yi[1])/(xi[2]-xi[1])-
```

```
       (yi[3]-yi[2])/(xi[3]-xi[2])+
       (yi[3]-yi[1])/(xi[3]-xi[1]);

d2=    -(yi[ord-1]-yi[ord-2])/(xi[ord-1]-xi[ord-2])+
       (yi[ord]-yi[ord-1])/(xi[ord]-xi[ord-1])+
       (yi[ord]-yi[ord-2])/(xi[ord]-xi[ord-2]);

/*Compute first line coefficients */
a[1]=0;
b[1]=2*(xi[2]-xi[1]);
c[1]=xi[2]-xi[1];
d[1]=6*( (yi[2]-yi[1])/(xi[2]-xi[1])-d1 );
/*Compute coefficients for lines 2...ord-1*/
for(i=2;i<=ord-1;i++)
       {
       a[i]=xi[i]-xi[i-1];
       b[i]=2*(xi[i+1]-xi[i-1]);
       c[i]=xi[i+1]-xi[i];
       d[i]= 6*( (yi[i+1]-yi[i])/(xi[i+1]-xi[i])-
             (yi[i]-yi[i-1])/(xi[i]-xi[i-1]) );
       }


/* Compute last line coefficients */
a[ord]=xi[ord]-xi[ord-1];
b[ord]=2*(xi[ord]-xi[ord-1]);
c[ord]=0;
d[ord]=6*(d2-(yi[ord]-yi[ord-1])/(xi[ord]-xi[ord-1]));
cod=SolveTridi(ord,a,b,c,d,der2);
if(cod==0)return 0;

/* Compute coefficients for local SPLINE
 * Use the same variables as for the three-diagonal
 * system.
 */
for(i=1; i<=ord-1;i++)
{
 a[i]=(der2[i+1]-der2[i])/( 6*(xi[i+1]-xi[i]));
 b[i]=(der2[i]*xi[i+1]-der2[i+1]*xi[i])/(2*(xi[i+1]-xi[i]));
 c[i]= (der2[i+1]*pow(xi[i],2)-der2[i]*pow(xi[i+1],2))/
       (2*(xi[i+1]-xi[i]))+
       (yi[i+1]-yi[i])/(xi[i+1]-xi[i])-a[i]*pow(xi[i+1]-xi[i],2);

 d[i]=(der2[i]*pow(xi[i+1],3)-der2[i+1]*pow(xi[i],3))/
       (6*(xi[i+1]-xi[i]))+(yi[i]*xi[i+1]-yi[i+1]*xi[i])/
       (xi[i+1]-xi[i])-
       b[i]*pow((xi[i+1]-xi[i]),2)/3;
}

/* Locate value to interpolate to find what functions to
 * apply.
 */
loc = 1;
while (point < xi[loc]) loc++;
*valoare =    a[loc]*pow(point,3) + b[loc]*pow(point,2)+
              c[loc]*point+d[loc];
return 1;
}
```

7.6. INTERPOLATION OF PERIODIC FUNCTIONS

Let us consider the periodic function $f:[a,b]\to R$ with the property that $f(a)=f(b)$ where $T=b-a$ is the *function's period* (or function's *cycle*).

Experimentally, we obtain *n* values of the function *f(x)* on period $[a,b]$ tabulated in Table 7.5. We want to know the ordinate's value of the function $f(x)$ in an unknown point $x_p \in [a,b]$, where $x_p \neq x_i$, $i=1,2,...,n$.

Table 7.5

x	$x_1 = a$	x_2	x_3	----	$x_{2n} = b$
y	y_1	y_2	y_3	----	y_{3n}

In order to determine these values we compute an interpolation polynomial for the periodic function.

Definition 7.5. We call a *fundamental system of periodic functions* the set of functions φ, *i*=0,1,2,...,*n* defined on $[a,b]$ if:

- they are continuous,
- they are linearly independent on $[a,b]$, $\varphi(a)=\varphi(b)$ for any i =1,2,...,*n*, and
- the function $H(x)=\sum_{i=0}^{n} a_i\varphi_i$, with:

$$\sum_{i=0}^{n} a_i^2 > 0 \tag{7.65}$$

has at most *n* roots in $[a,b]$.

We make the transformation:

$$t=\frac{2\pi(x-a)}{b-a} \tag{7.66}$$

to transform the interval $[a,b]$ into [0, 2π]. In this case we consider the fundamental system of periodic functions on $[a,b]$:

$$1, \cos x, \sin x, \cos 2x, \sin 2x, \ldots, \cos nx, \sin nx \tag{7.67}$$

Proposition 1*:* Let there be some periodic function *f:[0,2π]→***R** and the points $(x_i, f(x_i))$, *i=0,1,...,2n* where $x_i \in [0,2\pi]$, $x_i \neq x_j$. Then the interpolation polynomial is:

$$I_n=\sum_{i=0}^{2n}\left(f(x_i)\cdot\prod_{\substack{j=0\\ j\neq i}}^{n}\left(\frac{\sin\left(\frac{x-x_j}{2}\right)}{\sin\left(\frac{x_i-x_j}{2}\right)}\right)\right), \text{ with } j=0,\ldots,n,\ j\neq i \tag{7.68}$$

We notice an analogy between the LAGRANGE interpolation polynomial and the periodic function interpolation polynomial (7.67).

For the periodic function on *[a, b]* we explicitly state the equation of *x* from formula (7.66):

$$x = \frac{b-a}{2\pi} + a \tag{7.69}$$

The interpolation polynomial has the following format

$$I_n = \sum_{i=0}^{2n} f(x_i) \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{\sin \frac{(b-a)}{4\pi}(x - x_j)}{\sin \frac{(b-a)}{4\pi}(xi - x_j)} \tag{7.70}$$

7.7.1. Algorithm 7.7. Periodic functions' interpolation

```
Arguments
{
    a: interval left limit, real ;
    b: interval right limit  , real ;
    n: number of known function values, integer ;
    x: horizontal coordinates for known points, array ;
    y: vertical coordinates for known points, array;
    x̄ : interpolation point, real;
)
{
 i,j: counters, integer ;
 sum: variable to retain sums, real;
 prod: variable to retain products, real;

 sum = 0;
 for i= 0 to n
   {  prod = 1;
      for j=0 to n
         if          j!=i          then       compute
```

$$prod = prod \cdot \frac{\sin\left[\frac{(b-a)}{4\pi} \cdot (x - x_j)\right]}{\sin\left[\frac{(b-a)}{4\pi} \cdot (x_i - x_j)\right]};$$

```
      compute sum=sum + y[i]*prod;
   }
 the interpolated value is sum
}
```

7.7.2. Implementation of Algorithm 7.7

```
/* Implementation for periodic function interpolation.
 * Returns the interpolated value.
 */
double IFPer
```

```
(
    int n,
    double x[],
    double y[],
    double T,
    double point
)
{
    int i,j;
    double sum=0,prod;

    for(i=0; i<=n; i++)
    {
     prod=1;
     for(j=0;j<=n;j++)
        if(j!=i)
           prod *=  sin(((b-a)/(4* pi))*(point-x[j]))
                    / sin(((b-a)/(4* pi))*(x[i]-x[j]));
     sum += y[i]*prod;
    }
    return sum;
}
```

7.8. THE INTERPOLATION OF MULTIDIMENSIONAL FUNCTIONS

Let us consider a function with two variables f : $\mathbf{E}\to\mathbf{R}$ where $\mathbf{E}\subset\mathbf{R}^2$ under the format $f = g(x, y)$. Let us consider the following known points $(x_i, y_i, f(x_i, y_i))$ for $i = 0,1,2,...,n$ and $j = 0,1,...,n$.

We want to find out a nth degree polynomial to satisfy the conditions $P_n(x_i, y_i) = f(x_i, y_i)$ with $i, j = 0,1,2,...,n$. Let the nth degree polynomial be:

$$P_n(x, y) = a_{00} + a_{10}x + a_{01}y + a_{11}xy + .. + a_{1n}xy^n \tag{7.68}$$

The polynomial coefficients can be placed under the following matrix format:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} & \cdots & a_{0(n-1)} & a_{0n} \\ a_{10} & a_{11} & a_{12} & a_{13} & \cdots & a_{1(n-1)} & 0 \\ --- & --- & --- & --- & --- & --- & --- \\ a_{(n-2)0} & a_{(n-2)1} & a_{(n-2)2} & 0 & \cdots & 0 & 0 \\ a_{(n-1)0} & a_{(n-1)1} & 0 & 0 & \cdots & 0 & 0 \\ a_{n0} & 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix} \tag{7.69}$$

To find these coefficients we need $\frac{(n+1)(n+2)}{2}$ equations. These equations can be obtained from the given points date $(x_i, y_i, f(x_i, y_i))$ that need to be also a number of $\frac{(n+1)(n+2)}{2}$.

As such, the points in which the function is known will give a triangular form:

$$\begin{pmatrix} (x_0, y_0) & (x_0, y_1) & (x_0, y_2) & \dots & (x_0, y_n) \\ (x_1, y_0) & (x_1, y_1) & (x_1, y_2) & \dots & 0 \\ - - - & - - - & - - - & - - - & - - - \\ (x_{n-1}, y_0) & (x_{n-1}, y_1) & 0 & \dots & 0 \\ (x_n, y_0) & 0 & 0 & \dots & 0 \end{pmatrix} \tag{7.70}$$

In this case the points aren't attached on a nth degree curve which leads to a non-null determinant Δ != 0, so to a unique coefficient computation in (7.68). We look for an nth degree polynomial:

$$\begin{aligned} P_n(x, y) = a_{00} + a_{10}(x - x_0)^{[1]} + a_{01}(y - y_0)^{[1]} + a_{20}(x - x_0)^{[2]} + \\ + C_{11}(x - x_0)^{[1]}(y - y_0)^{[1]} + C_{02}(y - y_0)^{[2]} + \dots + C_{0n}(y - y_0)^{[n]} \end{aligned} \tag{7.71}$$

By applying the finite differences for two variable functions:

$$\Delta^{i+j}_{x^i y^j} f(x_0, y_0) = \Delta^{i+j}_{x^i y^j} P_n(x_0, y_0) = a_{ij} h^i k^j i! j! \tag{7.72}$$

Taking into account the formula (7.72) in expression (7.71) we get:

$$a_{ij} = \frac{\Delta^{i+j}_{x^i y^j} f(x_0, y_0)}{h^i k^j i! j!}$$

Introducing the computed coefficients in the polynomial's expression (7.71) we get the interpolation polynomial for a two variable function:

$$P_n(x, y) = \sum_{\substack{i, j=0 \\ i+j \le n}}^{n} \frac{\Delta^{i+j}_{x^i y^j} f(x_0, y_0)}{h^i k^j i! j!} (x - x_0)^{[i]} (y - y_0)^{[j]} \tag{7.73}$$

The formula can be extended to multidimensional functions, but we must be careful to how we select the points through which the function is chosen to make the interpolation polynomial's coefficients calculus possible.

In practical applications, for two variable functions we usually use the extended LAGRANGE's polynomial. Let us consider $(n+1)(m+1)$ distinct points in which the function is $f(x_i, y_j), i = 0,1,\dots,n; j = 0,1,\dots,n.$ We look for the polynomial $P(x, y)$ with a degree as high as n in x and m in y so as to have:

$$P(x_i, y_i) = f(x_i, y_i), i = 0,1,\dots,n; \; j = 0,1,2,\dots,n.$$

The LAGRANGE polynomial for functions of two variables is:

$$L(x, y) = \sum_{i=0}^{n} \sum_{j=0}^{m} f(x_i, y_j) \prod_{\substack{k=0 \\ k \neq i}}^{n} \frac{x - x_k}{x_i - x_k} \prod_{\substack{k=0 \\ k \neq j}}^{m} \frac{y - y_k}{y_j - y_k} \qquad (7.74)$$

For this polynomial we present the algorithm and C source implementation.

7.8.1. Algorithm 7.8. Two-variables functions' interpolation

```
Arguments
(
   n,m: number of given points on Ox and Oy, integers;
   x: array of point coordinates on Ox;
   y: array of point coordinates on Oy;
   z: function value matrix;
   pcx: Ox interpolation point coordinate;
   pcy: Oy interpolation point coordinate;
)
{
   i, j, k: counters, integer;
   prodx, prody: partial products;
   value: partial sums;
   value = 0;
   for i=0,…, n
   {
      prodx = 1;
      for k = 0,…,n
         if k!=i
            compute prodx = (pcx-x[k])/(x[i]-x[k]);
         for j=0,…,m
         {
            prody=1;
            for k=0,…,m
               if k!=j
                  compute prody = (pcy-y[k])/(y[j]-y[k]);
            compute value = z(i,j)*prodx*prody;
         }
   }
Returns interpolation's result kept in 'value'
}
```

7.8.2. Implementation for Algorithm 7.8

```
double Lagrange2
(
   int n,
   int m,
   double x[],
   double y[],
   double z[][NMAX],
   double pcx,
   double pcy
)
{
   int i, j, k;
   double prodx, prody, value;
```

```
    value = 0;
    for(i=0; i<=n; i++)
    {
        prodx = 1;
        for(k=0; k<=n; k++)
                if(k!=i) prodx *= (pcx-x[k])/(x[i]-x[k]);
        for(j=0; j<=m; j++)
        {
            prody = 1;
            for(k=0; k<=m; k++)
                if(k!=j) prody *= (pcy-y[k])/(y[j]-y[k]);
            value += z[i][j]*prodx*prody;
        }
      return value;
}
```

7.9. APPLICATIONS

1. Let us consider the experimental data obtained for a temperature coefficient characteristic α [mV/^0C], function of voltage, for a Zener diode, in the following table:

Table 7.6

U [mV]	10	20	55	60	70	115	150	280	300	435
I [mA]	0.5	1.0	1.5	2.0	2.3	2.0	1.5	0.5	0.3	1.0

It must be determined points between the given values for a most-precise graphical representation.

The value of function for `U=90mV` has been determined, resulting in the following values (up to 5 decimals):

- for LAGRANGE's method α=`81.82715`;
- for NEWTON's 1st kind method α=`81.82715`;
- for NEWTON's 2nd kind method α=`81.82715`;
- for AITKEN's method α=`85.80118`;