Interpolation by Lagrange Polynomial

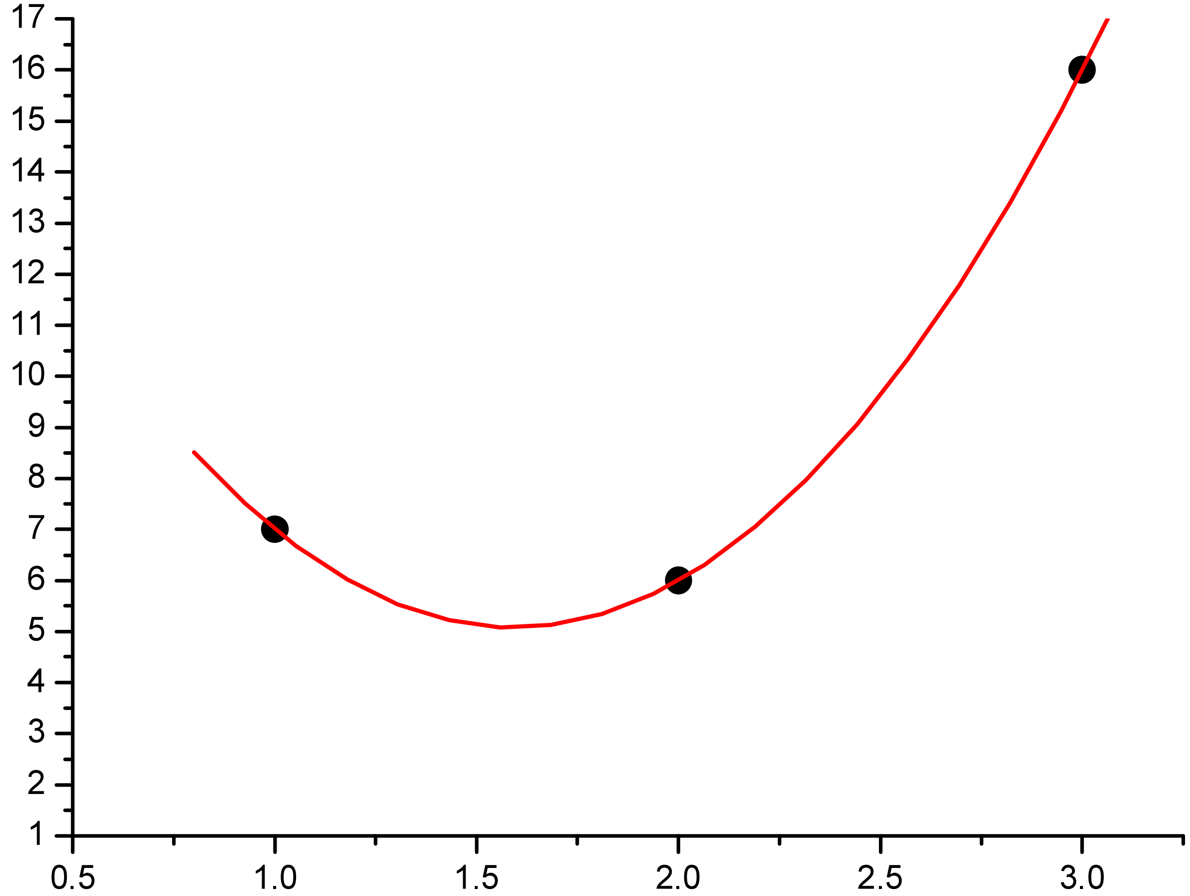
- **Task** (simple formulation): There is a set of "**experimental**" points (x_0, y_0), (x_1, y_1),..., (x_n, y_n), $x_0 < x_1 < ... < x_n$.

x_0	x_1	x_2	...	x_n
y_0	y_1	y_2	...	y_n

We want to construct an analytical function that passes through all these points. We want to estimate the values of the function in any point $x \in [x_0, x_n]$ (**interpolation**) or even outside of interval $[x_0, x_n]$ (**extrapolation**).

- **Task** (mathematical formulation): There is an *unknown* function $f(x)$. We only know its values $f(x_0)$, $f(x_1)$,.. $f(x_n)$ at certain points $x_0 < x_1 < ... < x_n$.
  We want to construct an **approximation** $P(x)$ to the function $f(x)$ such that $P(x_i) = f(x_i)$.

- The idea of the Lagrange interpolation: we always can aproximate $f(x)$ by a polynomial $P_n(x)$ of order n that has the following property: $y_i = P_n(x_i)$, i.e., that connects all these points.



- **Lagrange polynomial construction:**

For any point x_i we define the function $p_i(x)$:

$$p_i(x) = \prod_{j=0,\, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

Note that $p_i(x)$ is a polynomial of degree n with the properties:

$$p_i(x_i) = \prod_{j=0,\, j\neq i}^{n} \frac{x_i - x_j}{x_i - x_j} = \prod_{j=0,\, j\neq i}^{n} 1 = 1$$

$$p_i(x_k) = \prod_{j=0,\, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} = \prod_{j=0,\, j\neq i,\, j\neq k}^{n} \frac{x_k - x_j}{x_i - x_j} \cdot \frac{x_k - x_k}{x_i - x_k} = 0 \quad (k \neq i)$$

Thus, $p_i(x_k) = \delta_{ik}$. Now we combine the polynomials $p_i(x)$:

$$P_n(x) = \sum_{i=0}^{n} p_i(x) y_i = \sum_{i=0}^{n} \prod_{j=0,\, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} y_i \qquad \text{— a polynomial of degree } n \text{ (or less)}$$

$$\Rightarrow P_n(x_k) = \sum_{i=0}^{n} p_i(x_k) y_i = \sum_{i=0}^{n} \delta_{ik} y_i = y_k \qquad \Rightarrow \quad P_n(x_k) = y_k \quad \forall k$$

Theorem

There exists a **unique polynomial** $P_n(x)$ of degree less than or equal to n, such that $y_k = P_n(x_k)$ $\forall$ i =0...n:

$$P_n(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

The ***existence*** is proved by the construction of the Lagrange polynomial. Each summand in the Lagrange polynomial is a product of exactly *n* expressions ($x - x_j$). Opening the parentheses, we obtain a polynomial of degree *n*. The sum of such polynomials is also a polynomial of degree not greater than *n*.

The ***uniqueness*** is proved by *reduccio ad absurdum*. Suppose that there exist two different polynomials:

$$A(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

$$B(x) = \sum_{i=0}^{n} b_i x^i = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n$$

Then,

$$\begin{cases} A(x_k) = y_k \\ B(x_k) = y_k \end{cases} \quad \text{for all } k = 0, 1, \ldots, n \qquad (n+1) \text{ values in total}$$

$$A(x_k) = \sum_{i=0}^{n} a_i x_k^i = a_0 + a_1 x_k + a_2 x_k^2 + \ldots + a_n x_k^n = y_k$$

$$B(x_k) = \sum_{i=0}^{n} b_i x_k^i = b_0 + b_1 x_k + b_2 x_k^2 + \ldots + b_n x_k^n = y_k$$

Subtracting these equations:

$$A(x_k) - B(x_k) = \sum_{i=0}^{n} a_i x_k^i - \sum_{i=0}^{n} b_i x_k^i = \sum_{i=0}^{n} \left(a_i - b_i\right) x_k^i \Rightarrow$$

$$\Rightarrow \left(a_0 - b_0\right) + \left(a_1 - b_1\right) x_k + \left(a_2 - b_2\right) x_k^2 + \ldots + \left(a_n - b_n\right) x_k^n = 0$$

Then we have an algebraic equation of degree n (or less than n) that has $(n+1)$ distinct roots. This contradicts the *Fundamental Theorem of Algebra* (any polynomial of degree n has exactly n roots (real or complex, taking into account their multiplicity).

Programming

- To calculate the interpolated values at any point x, we can use directly the Lagrange formula:

$$P(x) = \sum_{i=0}^{n} \prod_{j=0,\, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} y_i$$

Obviously, two nested loops must be organized: one to program the product and the other one for summation. Programming the product, we must ensure that $j \neq i$.

- Sometimes we need the coefficients of the polynomical explicitly:

$$P(x) = \sum_{k=0}^{n} a_k x^k = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

Algorithm A (a maximum of 8 points will be given)**:**

To find $a_0, a_1, ..., a_n$, a system of linear equation can be constructed:

$$\begin{cases} a_0 + a_1 x_0 + a_2 x_0^2 + ... + a_n x_0^n = y_0 \\ a_0 + a_1 x_1 + a_2 x_1^2 + ... + a_n x_1^n = y_1 \\ .... \\ a_0 + a_1 x_n + a_2 x_n^2 + ... + a_n x_n^n = y_n \end{cases}$$

where $a_0, a_1, ..., a_n$ are unknowns. Thus, the coefficient matrix (***Vandermonde matrix* Z** $= [x_i^j],\ i, j = 0, n$) must be constructed as follows:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & ... & x_0^n \\ 1 & x_1 & x_1^2 & ... & x_1^n \\ ... & & & & \\ 1 & x_n & x_n^2 & ... & x_n^n \end{pmatrix} . \begin{pmatrix} a_0 \\ a_1 \\ ... \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ ... \\ y_n \end{pmatrix}$$

***Hint*.** • *The Vandermonde matrix* construction is easy. The i-th column can be computed by multiplying the $(i{-}1)$-th column by the x vector.

• Do not use explicit **`x(i)**j`** , as it will be highly inefficient.

***Hint*.** • *A programming trick*. In the way explained, the Vandermonde matrix is defined as $[x_i^j]$, with indices running from 0 to n.

```
double precision, dimension(0:n,0:n) :: Z
```

• However, linear equation solvers usually use matrices from 1 to n:

```
subroutine GAUSS(A,B,X,m)
integer :: m
double precision, dimension(1:m,1:m) :: A
double precision, dimension(1:m) :: B, X
```

• The following call serves to overcome this problem:

```
call GAUSS(Z,Y,A,n+1)
```

where **`n+1`** is the *size* of the matrix (and vector). No modification of the subroutine is needed is this case.

Algorithm B (a bit tricky, but more efficient;up to 10 points will be given)**:**

- It is also possible to obtain the coefficients a_0, a_1, ..., a_n directly from the Lagrange formula:

$$P(x) = \sum_{i=0}^{n} \prod_{j=0,\, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} y_i$$

Let us consider finding the coefficients b_0, b_1, ..., b_n of the following product:

$$\prod_{j=0,\, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} y_i = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n$$

We should initially construct array **b** as follows:

$$b_0 = y_i,\ b_1 = b_2 = \ldots = 0.$$

Then, within a loop over j ($j \neq i$) should calculate new coefficients b_0, b_1, ..., b_n produced by multiplication by $(x - x_j)$. Let's now see what happens with coefficients of a polynomial $b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n$ upon multiplication by $(x - x_j)$.

- Let's now see what happens with coefficients of a polynomial $b_0 + b_1 x + b_2 x^2 + \dots + b_n x^n$ upon multiplication by $(x - x_j)$:

$$\left(b_0 + b_1 x + b_2 x^2 + \dots + b_n x^n\right) \cdot (x - x_j) =$$
$$= b_0 x + b_1 x^2 + b_2 x^3 + \dots + b_{n-1} x^n + b_n x^{n+1} -$$
$$- x_j b_0 - x_j b_1 x - x_j b_2 x^2 - \dots - x_j b_n x^n =$$
$$= -x_j b_0 + (b_0 - x_j b_1) x + (b_1 - x_j b_2) x^2 + \dots + (b_{n-1} - x_j b_n) x^n + b_n x^{n+1}$$

Thus, the following changes occur with the coefficients:

- $b_0' = -x_j b_0$
- $b_k' = b_{k-1} - x_j b_k \qquad k = 1, \dots, n$
- $b_{n+1}' = x_j b_n$

Afterwards, all the coefficients b_0, b_1, ..., b_n must be divided by $(x - x_j)$.

- Upon finishing the loop over j, the coefficients b_0, b_1, ..., b_n must be summed.

Programming task

- ***Input***:
  - The number of "experimental" points n;
  - The points themselves (X's and Y's by pairs).

- ***Output***:
  - Values of the coefficients $a_0, .., a_n$ obtained;
  - Values of x_i and corresponding y_i calculated using the coefficients $a_0, a_1, ..., a_n$

- ***Note***:
  - The program must be general, i.e, work for any n.

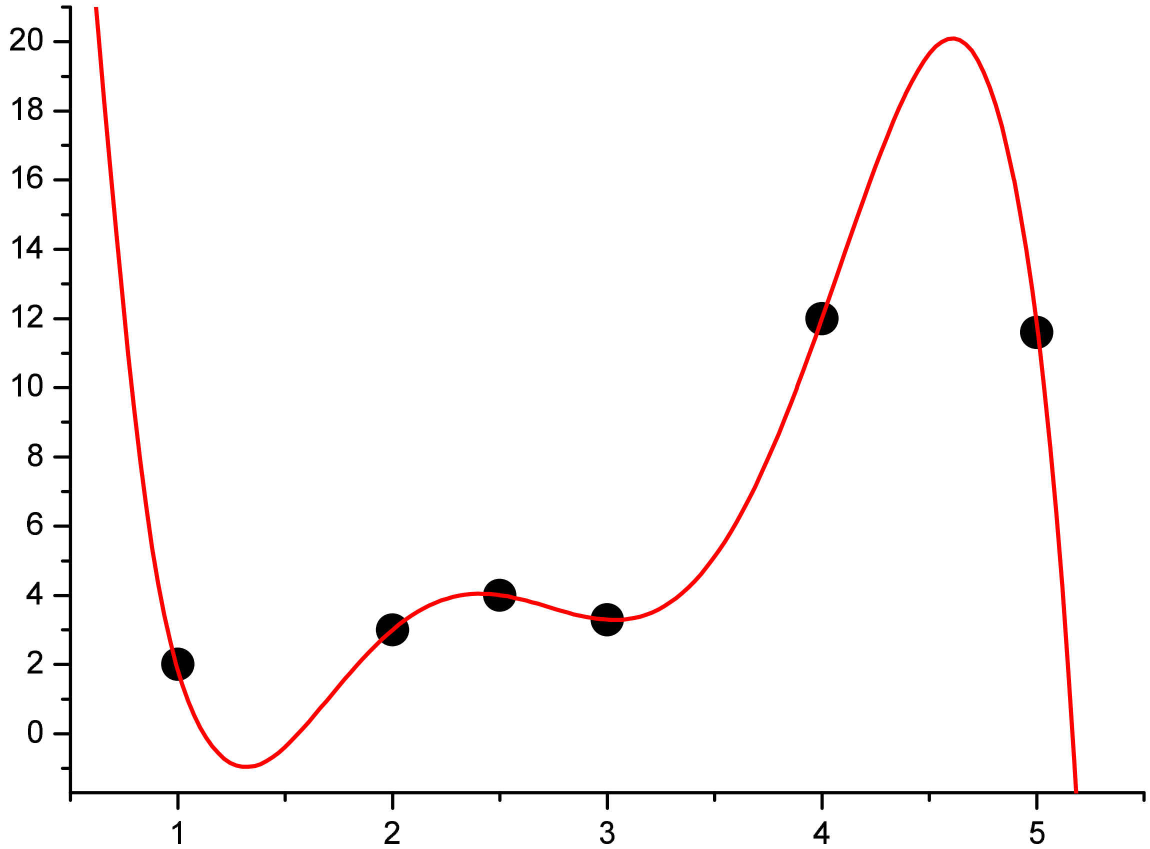
Hint

- A polynomial $P(x) = \sum_{k=0}^{N} a_k x^k$ is *never* calculated according to this formula directly since it requires about $n^2/2$ multiplications.
- The easiest way to implement it efficiently corresponds to the following formula:

$$P(x) = \left(\left(a_N x + a_{N-1}\right) x + a_{N-2}\right) x \ldots + a_0$$

Warnings

- The Lagrange polynomial is a poor approximation. It should **not** be used for practical interpolation.



- The Vandermonde matrices are always non-singular (invertible), but often ill-conditioned (“almost singular”). A good Gauss subroutine (with double precision) is required to resolve them.