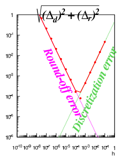
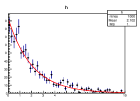
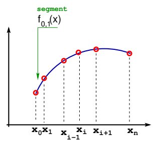
Computational Physics

numerical methods with C++ (and UNIX)







Fernando Barao
Instituto Superior Tecnico, Dep. Fisica
email: barao at lip.pt

Numerical methods

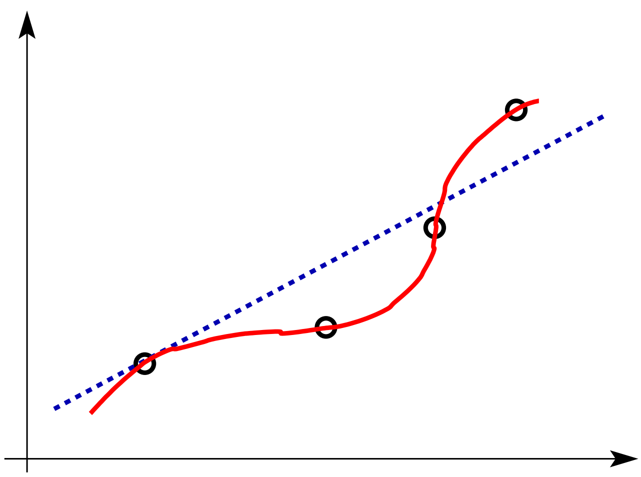
- ✔ System of linear equations
  - ► Gauss elimination
  - ► LU decomposition
  - ► Gauss-Seidel method
- ✔ Interpolation
  - ► Lagrange interpolation
  - ► Newton method
  - ► Neville method
  - ► Cubic spline
- ✔ Numerical derivatives
  - ► First derivative $\mathcal{O}(h^2)$, $\mathcal{O}(h^4)$
  - ► Second derivative $\mathcal{O}(h^2)$, $\mathcal{O}(h^4)$
  - ► Derivative by interpolation
- ✔ Numerical integration
  - ► Newton-Cotes: trapezoidal and Simpson rules
  - ► Gaussian quadrature
- ✔ Monte-Carlo methods

Numerical methods

- ✔ System of linear equations
  - ► Gauss elimination
  - ► LU decomposition
  - ► Gauss-Seidel method
- ✔ Interpolation
  - ► Lagrange interpolation
  - ► Newton method
  - ► Neville method
  - ► Cubic spline
- ✔ Numerical derivatives
  - ► First derivative $\mathcal{O}(h^2)$, $\mathcal{O}(h^4)$
  - ► Second derivative $\mathcal{O}(h^2)$, $\mathcal{O}(h^4)$
  - ► Derivative by interpolation
- ✔ Numerical integration
  - ► Newton-Cotes: trapezoidal and Simpson rules
  - ► Gaussian quadrature
- ✔ Monte-Carlo methods

Data interpolation

- ✔ Having a set of discrete data points (x_i, y_i), **data interpolation** is the way of getting a continuous description passing through the data points



Lagrange interpolation

- ✔ Lagrange interpolation relies on the fact that in a finite interval a function $\mathbf{f}(\mathbf{x})$ can allways be represented by a polynomial $\mathbf{P}(\mathbf{x})$
- ✔ **Linear interpolation:** polynomial of **degree one** passing through data points $(\mathbf{x_1}, \mathbf{y_1})$ and $(\mathbf{x_2}, \mathbf{y_2})$

$$\boxed{P(x) = P_0 + P_1 x}$$

System to be solved:

$$\begin{cases} y_1 = & P_0 + P_1 x_1 \\ y_2 = & P_0 + P_1 x_2 \end{cases} \Rightarrow \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \end{pmatrix} \begin{pmatrix} P_0 \\ P_1 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

$$\boxed{\begin{cases} P_1 = \frac{y_2 - y_1}{x_2 - x_1} \\ P_0 = y_2 - P_1 x_1 \end{cases} \qquad P(x) = P_0 + P_1 x = y_1 \frac{x - x_2}{x_1 - x_2} + y_2 \frac{x - x_1}{x_2 - x_1}}$$

Lagrange interpolation (cont.)

- ✔ **second-degree polynomial interpolation:** polynomial of **degree two** passing through data points $(\mathbf{x_1}, \mathbf{y_1})$, $(\mathbf{x_2}, \mathbf{y_2})$ and $(\mathbf{x_3}, \mathbf{y_3})$

$$\boxed{P(x) = P_0 + P_1 x + P_2 x^2}$$

System to be solved:

$$\begin{cases} y_1 = & P_0 + P_1 x_1 + P_2 x_1^2 \\ y_2 = & P_0 + P_1 x_2 + P_2 x_2^2 \\ y_3 = & P_0 + P_1 x_3 + P_2 x_3^2 \end{cases} \Rightarrow \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{pmatrix} \begin{pmatrix} P_0 \\ P_1 \\ P_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

$$\boxed{P(x) = y_1 \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} + y_2 \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} + y_3 \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}}$$

Lagrange interpolation (cont.)

✔ **n polynomial interpolation:** polynomial of **degree n** passing through $(\mathbf{n+1})$ data points $(\mathbf{x_0},\mathbf{y_0}),\ (\mathbf{x_1},\mathbf{y_1}),\ \cdots,\ (\mathbf{x_n},\mathbf{y_n})$

$$\boxed{P(x) = P_0 + P_1 x + P_2 x^2 + \cdots + P_n x^n}$$

$$\begin{aligned} P_n(x) &= \textstyle\sum_{i=0}^{n}\ y_i\ \ell_i(x) \\ &= y_0\ \ell_0(x) + y_1\ \ell_1(x) + \\ &\quad \cdots + y_n\ \ell_n(x) \end{aligned}$$

$$\ell_i(x) = \prod_{\substack{j=0\\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j} \qquad (i = 0, 1, 2, \cdots, n)$$

```
// n = polynomial degree

// n+1 = nb of data points

// x,y = abcissa and values

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

// loop on data points (0...n)

   for (int i=0; i<n+1; i++) {

// we need a second loop for
// the product

      for (...) {

      }

   }
}
```


Interpolation: C++ class scheme

```
 ----------------------
 |  class DataPoints  |
 ----------------------   class DataPoints {
    / \      / \            public:
     |        |                  ...
     |        |                  virtual double Interpolate(double x);
     |        |                  virtual void Draw();
     |        |                  virtual void Print();
     |        |             protected:
     |        |                  int N; //nb data points
     |        |                  double *x, *y; //x and y values
     |        |           };
     |        |
     |        |
     |        |
     |        |
     |    ----------------------------
     |    |  class LagrangeInterpol  |
     |    ----------------------------   class LagrangeInterpol : public DataPoints {
     |                                     public:
     .                                          ...
     .                                          double Interpolate(double x);
     .                                          void Draw();
  (other interpolation                          void Print();
   classes)                                private:
                                                ? //specific data to class
                                         };
```

Newton method

- ✔ The Newton method provides a better computational procedure to get an interpolating polynomial of degree n passing through $(n+1)$ data points

$$
\begin{aligned}
x_i &= x_0, x_1, \cdots, x_n \\
y_i &= y_0, y_1, \cdots, y_n \\
a_i &= a_0, a_1, \cdots, a_n
\end{aligned}
$$

$$\boxed{P(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \cdots + a_n(x - x_0)(x - x_1)\cdots(x - x_{n-1})}$$

- ✔ This polynomial can be written in an efficient computational way:

$$\boxed{P(x) = a_0 + (x - x_0)\ [a_1 + (x - x_1)\ [a_2 + (x - x_2)[\cdots[a_{n-1} + (x - x_{n-1})a_n]\ldots]}$$

- ✔ The coefficients are determined by imposing the polynomial to pass through the data points:

$$
\begin{aligned}
(x_0, y_0): &\quad y_0 = a_0 \\
(x_1, y_1): &\quad y_1 = a_0 + a_1(x_1 - x_0) \\
(x_2, y_2): &\quad y_2 = a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1) \\
&\vdots \\
(x_n, y_n): &\quad y_n = a_0 + a_1(x_n - x_0) + \cdots + a_n(x_n + x_0)(x_n - x_1)\cdots(x_n - x_{n-1})
\end{aligned}
$$

Newton method

- ✔ **Coefficients:**

$$
\begin{aligned}
a_0 &= y_0 \\
a_1 &= \frac{y_1 - y_0}{x_1 - x_0} \equiv \nabla y_1 \\
a_2 &= \nabla^2 y_2 \\
a_3 &= \nabla^3 y_3 \\
a_4 &= \nabla^4 y_4 \\
&\vdots \\
a_n &= \nabla^n y_n
\end{aligned}
$$

Divided diferences:

$$
\begin{aligned}
\nabla y_i &= \frac{y_i - y_0}{x_i - x_0} && (i = 1, 2, .., n) \\
\nabla^2 y_i &= \frac{\nabla y_i - \nabla y_1}{x_i - x_1} && (i = 2, 3, .., n) \\
\nabla^2 y_i &= \frac{\nabla^2 y_i - \nabla^2 y_2}{x_i - x_2} && (i = 3, 4, .., n) \\
&\vdots \\
\nabla^n y_n &= \frac{\nabla^{n-1} y_n - \nabla^{n-1} y_{n-1}}{x_n - x_{n-1}}
\end{aligned}
$$

The diagonal terms of the table are the coefficients of the polynomial

	0th	1st	2nd	3rd	4th
x_0	y_0				
x_1	y_1	∇y_1			
x_2	y_2	∇y_2	$\nabla^2 y_2$		
x_3	y_3	∇y_3	$\nabla^2 y_3$	$\nabla^3 y_3$	
x_4	y_4	∇y_4	$\nabla^2 y_4$	$\nabla^3 y_4$	$\nabla^4 y_4$

Computing the interpolated value at x with the polynomial computed in a recursive way:

$$
\begin{aligned}
P_0(x) &= a_n \\
P_1(x) &= a_{n-1} + (x - x_{n-1})P_0(x) \\
P_2(x) &= a_{n-2} + (x - x_{n-2})P_1(x) \\
&\vdots \\
P_k(x) &= a_{n-k} + (x - x_{n-k})P_{k-1}(x) \quad (k = 1, 2, ..., n)
\end{aligned}
$$

Newton method: algorithm

Coefficients:

```
// degree n polynomial
// n+1 data points
//
// For computing the coefficients
// we can use a one-dimensional
// array a[n+1]
//

 1) make array a[n+1];

 2) copy contents of Y[] data to array a[]

 3) compute divided differences and
    store them in the one dimensional
    array a[]

    loop on k=1; k<n+1; k++

       loop on i=k; i<n+1; i++

          a[i] = (a[i] - a[k-1]) /
                 (x[i] - x[k-1])
```

Polynomial:

```
// degree n polynomial
// n+1 data points
//
// For computing the polynomial at
// a point x
// we use the recurrence existing
// after factorizing the polynomial
//
// We assume having already the
// coefficients
// computed in the array a[n+1]
//

 1) init the last polynomial P

    P = a[n];

 2) loop on k=1; k<n+1; k++

       P = a[n-k] + (x - x[n-k])*P
```


Neville method

✔ The Neville algorithm is still better by computing standards for finding the n degree polynomial because does not require to a computation in two steps

✔ It uses linear interpolations between successive iterations: one point needed at $0th$ order, two points at $1st$ order, three points at $2nd$ order, ..., $n+1$ points at nth order

0th order: $P_0[x_0] = y_0, \; \cdots \; P_n[x_n] = y_n$

1st order (linear): $P_1[x_0, x_1] = C_0 + C_1 x = \frac{y_1(x-x_0) - y_0(x-x_1)}{x_1 - x_0} = \frac{(x-x_0)\, P[x_1] - (x-x_1)\, P[x_0]}{x_1 - x_0}$

2nd order: $P_2[x_0, x_1, x_2] = \frac{(x-x_2)\, P[x_0,x_1] - (x-x_0)\, P[x_1,x_2]}{x_0 - x_2}$

3rd order: $P_3[x_0, x_1, x_2, x_3] = \frac{(x-x_3)\, P[x_0,x_1,x_2] - (x-x_0)\, P[x_1,x_2,x_3]}{x_0 - x_3}$

... ...

x values	0th order	1st order	2nd order	3rd order	*...order*
x_0	$P_0(x_0) = y_0$				
x_1	$P_0(x_1) = y_1$	$P_1[x_0, x_1]$			
x_2	$P_0(x_2) = y_2$	$P_1[x_1, x_2]$	$P_2[x_0, x_1, x_2]$		
x_3	$P_0(x_3) = y_3$	$P_1[x_2, x_3]$	$P_2[x_1, x_2, x_3]$	$P_3[x_0, x_1, x_2, x_3]$	
x_4	$P_0(x_4) = y_4$	$P_1[x_3, x_4]$	$P_2[x_2, x_3, x_4]$	$P_3[x_1, x_2, x_3, x_4]$	
$\cdots$	$\cdots$	$\cdots$	$\cdots$		
x_n	$P_0(x_n) = y_n$	$P_1[x_{n-1}, x_n]$	$P_2[x_{n-2}, x_{n-1}, x_n]$	$P_3[x_{n3}, x_{n-2}, x_{n-1}, x_n]$	

Neville method: algorithm?

```
1) We can try to work with only one array
   (1-dim) y[] containing the 0th order
   polynomials passing by the values

2) loop on the order of the polynomials:
   i=0, i<n+1

3) loop on every column to compute the
   different polynomials

4) the interpolant calculated at the
   coordinate x, corresponds to the last
   value
```


Numerical methods

- System of linear equations
  - Gauss elimination
  - LU decomposition
  - Gauss-Seidel method
- Interpolation
  - Lagrange interpolation
  - Newton method
  - Neville method
  - Cubic spline
- Numerical derivatives
  - First derivative $\mathcal{O}(h^2)$, $\mathcal{O}(h^4)$
  - Second derivative $\mathcal{O}(h^2)$, $\mathcal{O}(h^4)$
  - Derivative by interpolation
- Numerical integration
  - Newton-Cotes: trapezoidal and Simpson rules
  - Gaussian quadrature
- Monte-Carlo methods

Numerical methods

- System of linear equations
  - Gauss elimination
  - LU decomposition
  - Gauss-Seidel method
- Interpolation
  - Lagrange interpolation
  - Newton method
  - Neville method
  - Cubic spline
- Numerical derivatives
  - First derivative $\mathcal{O}(h^2)$, $\mathcal{O}(h^4)$
  - Second derivative $\mathcal{O}(h^2)$, $\mathcal{O}(h^4)$
  - Derivative by interpolation
- Numerical integration
  - Newton-Cotes: trapezoidal and Simpson rules
  - Gaussian quadrature
- Monte-Carlo methods