People's Democratic Republic of Algeria

Ministry of Higher Education

and Scientific Research

Tissemsilt University

Faculty of Science and Technology
Department of Science and Technology

Course handout

NUMERICAL METHODS

Sector: Mechanical Engineering.

Specialty: Mechanical Construction.

Prepared by : Mr: KHERRAB Mohamed[1].

Dr : SATLA Zouaoui[2]

[1]Assistant teachers class “A”.

[2] Class “B” lecturers.

Tissemsilt - 2022/2023

Introduction

Numerical methods play a crucial role in solving mathematical problems that cannot be easily solved using analytical methods. These methods involve the use of mathematical algorithms and computational techniques to obtain approximate solutions to various mathematical models and equations. This introduction aims to provide an overview of numerical methods, their applications, and their importance in modern scientific and engineering fields.

Numerical methods encompass a wide range of techniques, each tailored to address specific types of problems. Some of the fundamental concepts and techniques in numerical methods include root-finding, interpolation, numerical integration, solving systems of linear equations, and solving ordinary differential equations. These methods are used extensively in diverse fields such as physics, engineering, finance, computer science, and many others.

Accuracy and efficiency are two key considerations in numerical methods. Achieving accurate results is essential to ensure the reliability of the solutions obtained. Efficiency refers to the computational speed and resource utilization of the numerical algorithms. Both accuracy and efficiency are vital factors in determining the effectiveness of numerical methods.

One of the primary challenges in numerical methods is dealing with errors. Numerical computations involve approximations and rounding-off, which introduce errors into the results. Understanding the sources and effects of these errors is crucial for assessing the reliability of the solutions obtained. Backwards error analysis is a technique used to analyze the impact of errors on the accuracy of numerical algorithms.

Floating-point arithmetic is another important aspect of numerical methods. It deals with the representation and manipulation of real numbers in a computer system. Understanding the limitations and characteristics of floating-point arithmetic is crucial for avoiding numerical instabilities and ensuring accurate computations.

Numerical linear algebra is a significant component of numerical methods. It involves the study of algorithms and techniques for solving linear systems of equations and eigenvalue problems. Sparse-matrix/iterative and dense-matrix algorithms are commonly used to solve these problems efficiently.

Numerical methods have seen significant advancements in recent years due to advancements in computer hardware and software. High-performance computing and numerical libraries have enabled the development of more sophisticated algorithms and improved computational efficiency. Additionally, programming languages such as MATLAB and Python provide powerful tools for implementing and executing numerical methods.

In conclusion, numerical methods are essential for solving mathematical problems that cannot be easily solved using analytical methods. These methods provide approximate solutions with a focus on accuracy and efficiency. With their wide range of applications in various fields, numerical methods continue to play a vital role in advancing scientific and engineering research.

Table of Contents

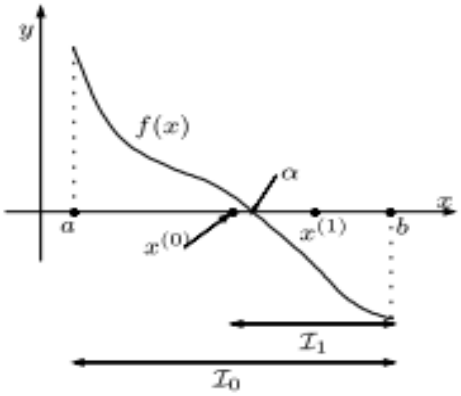
I. Solving nonlinear equations

I.1 Dichotomy method (or bisection)

The dichotomy method is based on the following property:

Let f be an increasing function on:

$$[a,b] \to R, \text{si } f(a)f(b) < 0, So\ \exists\ \alpha \in\ ]a,b[\ \text{ such as } f(\alpha) = 0.$$



Starting from $I_0 = [a,b]$, the dichotomy method produces a sequence of subintervals

$I_0 = [a_n, b_n], n \geq 0$, such as $f(a_n)f(b_n) < 0$

Formally, we put

$a_0 = a\ and\ b_0 = b$ So for $n \geq 0$

If $f(x_n) \leq 0, So\ a_{n+1} = c_n\ and\ b_{n+1} = b$.

Otherwise, $a_{n+1} = a\ and\ b_{n+1} = x_n$.

We notice $c_n = (a_n + b_n)/2$

I.2 Newton's method also called Newton-Raphson method

Suppose $f \in R\ and\ \hat{f}(\alpha) = 0$ (i.e. α is a simple root of). by asking

$$qk = f'(x^{(n)}) \qquad \forall n \geq 0$$

and taking the initial value(0) , we obtain Newton's method (also called Newton-Raphson or tangent method)

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \forall n \geq 0$$

At the n-th iteration, Newton's method requires the evaluation of the two Functions f and f' at point (n) .

I.1 Fixed point method

We give in this section a general procedure to find the roots of a nonlinear equation. The method is based on the fact that it is always possible, for $f: [a, b] \rightarrow R$, to transform the problem $f(x) = 0$ en an equivalent problem $x - \varphi(x) = 0$, or auxiliary function $\varphi: [a, b] \rightarrow R$ was chosen so that $\varphi(\alpha) = \alpha$ When $f(\alpha) = 0$. Approaching the zeros of f therefore comes down to the problem of determining the fixed points of φ, which is done using the following iterative algorithm:

Given (0) , we set

$$x^{(k+1)} = \varphi(x^k), \qquad k \geq 0$$

We are given $x(0)$ and we consider the sequence $x(k+1) = \varphi(x(k))$, for $k \geq 0. Si$

1. $\forall\, x \in [a, b], \varphi(x) \in [a, b]$,

2. $\varphi \in C1\,([a, b])$,

3. $\exists\, K < 1: |\, \varphi(x)\,| \leq K\ \forall\, x \in [a, b]$,

Alors φ a un unique point fixe α dans[a,b]et la suite $\{\, x(k)\,\}$ converge vers α

pour tout choix de $x(0) \in [a, b]$.De plus,on a

Then φ has a unique fixed point α in $[a, b]$ and the sequence $\{\, x(k)\,\}$ converges to α for any choice of $x(0) \in [a, b]$. Moreover, we have

I.4 Applications

1. Implement the dichotomy method in Matlab to solve the function $f(x)$, taking $x \in [-1.5, -0.5]$

   $f(x)$=x.^2-2.*x-3

2. Write the algorithm for solving the equation $g\,(x)$ using Newton's method. We will make mistakes = 10-5.

   g (x) = $x^3 - 10x^2 + 29x - 20 = 0$ giving x0=5.

3. We consider the function H defined by :H (x)=$x^2 - x - 1$

- Determine the fixed point of H.
- Establish a Matlab program to solve H (x)=0 by using the fixed point method.

Algorithm. Bisection Method

```
close all
clear all
clc
xl=2,
xu=3;
myfunc=@(x)x.^2-6;
x=linspace(xl,xu,100);
fun1=myfunc(x);
fig=figure();
set(fig,'color','white')
plot(x,fun1)
grid on
hold on
funmin=min(fun1);
funmax=max(fun1);
plot([xu xu],[funmin funmax],'k')
plot([xl xl],[funmin funmax],'k')
err=5;
%fun=myfunc(xn);
while err>1e-3

xn=xl;
fun=myfunc(xn);
delx=(xu+xl)/2;
xn=delx;
ystar=myfunc(xn);
if sign(ystar)~=sign(fun)
xl=delx;
else
 xu=delx;
end

err=abs(ystar)

%fprintf('la racine est %g/n',xn)
end
%xn
%fun
%fprintf('la racine est %g/n',xn)
```

Algorithm. Fixed point method

```
function out=ptfix(x)
close all
clear all
clc

f=inline('x.^2-2.*x-3');
p=[1 -2 -3];
roots(p);
g=inline('(x.^2-3)/2');

x=-2:0.1:2;
figure (1)
plot(x,x,'k-','linewidth',2)
hold on
plot(x,g(x),'b-','linewidth',2)
hold on
plot(x,f(x),'r-','linewidth',2)
x1=0;%x2=g(x1)
err=1;
while err > 1e-3

    x2=g(x1)
    err=abs(x2-x1);
    x1=x2

end

%x1
```

Algorithm. Newton-Raphson method

```
clear all
clc
syms p
x=[0,2,4,5,8,10];
y=[-1,1,6,0,2,5];
n=size(x,2)
d=zeros(n,n);
d(:,1)=y;
for j=2:n
for i=j:n
d(i,j)=(d(i-1,j-1)-d(i,j-1))/(x(i-j+1)-x(i));
end
end
Df=1;
for j = 1 : n-1
   Df=(p - x(j)).*Df;
```

```
  c(j)=Df.*d(j+1,j+1);
end
   P=sum(c) plot(x,f(x),'r-','linewidth',2)
x1=0;%x2=g(x1)
err=1;
while err > 1e-3

    x2=g(x1)
    err=abs(x2-x1);
    x1=x2

end

%x1
```

I Polynomial interpolation

II.1 Introduction

In numerical problems, we very often substitute a known function $f(x)$ at a finite number of points by a simpler and easily computable function $P(x)$: this is the approximation. In this chapter polynomial interpolations and their implementation in Matlab is addressed.

II.2 Definition

Consider n+1 couples (x_i, y_i). The problem is to find a polynomial P_n called interpolation polynomial or interpolating polynomial which allowed to calculate y for any value of x, such as:

$$P_n(x_i) = a_m x_i^m + \cdots + a_1 x_i + \alpha_0 = y_i \qquad i = 0, \dots, n. \tag{II.1}$$

The points x_i are called interpolation nodes.

II.3 Condition d'interpolation polynomiale

The interpolation polynomial $P_n(x_i)$ must check the condition $y_i = P_n(x_i) \forall\, i = 0, \dots, n$, that is, the polynomial must pass through all the points $M_i(x_i, y_i)$.

II.4 Lagrange interpolation

II.2.1 Theorem.

Given n+1 distinct points $x_0, \dots, x_n$ and n+1 corresponding values $y_0, \dots, y_n$, there exists a unique polynomial P such that

$$P(x_i) = y_i \text{ for } i = 0, \dots, n. \tag{II.2}$$

II.4.3 Demonstration.

To prove the existence, we will explicitly construct P_n . Let's pose

$$l_i \in P_n : l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \qquad i = 0, \dots, n. \tag{II.3}$$

Polynomials$\{ l_i , i = 0, \dots, n \}$ forming a basis of P_n. By breaking down P_n on this basis, we have

$$P_n(x) = \sum_{i=0}^{n} b_i l_i(x), \quad \text{d'où} \tag{II.4}$$

$$P_n(x_i) = \sum_{i=0}^{n} b_i l_i(x_i) = y_i, \qquad i = 0, \dots, n. \tag{II.5}$$

The coefficients $l_i(x_i)$have the following property:

$$l_i(x_i) = \delta_{ij} = \begin{cases} 1 & si\ i = j \\ 0 & si\ i \neq j \end{cases} \quad \delta_{ij} \text{ being Kronecker's symbol} \tag{II.6}$$

We instantly determine that $b_i = y_i$.

Therefore, the interpolation polynomial exists and is written in the following form

$$P(x) = \sum_{j=0}^{n} y_i l_i(x) \tag{II.7}$$

II.4.4 **Matlab**.

A program called Lagrange makes it possible to carry out a polynomial interpolation based on the Lagrange algorithm is implemented under Matlab. Given at the program input two vectors pointx and pointy contain respectively the interpolation variables and the corresponding function values. The output is the calculated polynomial. After entering data, two for loops and an inequality condition are used to obtain the polynomial, at the end of the program a graph is proposed to validate the results algorithm

Algorithm. Lagrange Lagrange interpolation

```
function Poly = Lagrange (pointx,pointy)
n=length(pointx);
syms x
n=length(pointy);
L=1;
Poly=0;
for i=1:n
   for j=1:n
      if i~=j
        L=L*((x-pointx(j))/(pointx(i)-pointx(j)));
      end
```

```
    end
    plot(pointx,eval(subs(L,pointx)))
    hold on
    Poly=Poly+pointy(i)*L;
    L=1;
end
    plot(pointx, subs(Poly,pointx),'*',pointx,pointy)
end
```

II.4.5 Example

- We give the function $f(x) = 1 + sin(3x)$ defines on the interval$]0,2[$, to construct the corresponding polynomial using the given Matlab Lagrange program, the interval is devised on 20 nodes. The results obtained are plotted in Figure II.4. The function f is presented in black line and the polynomial with stars.

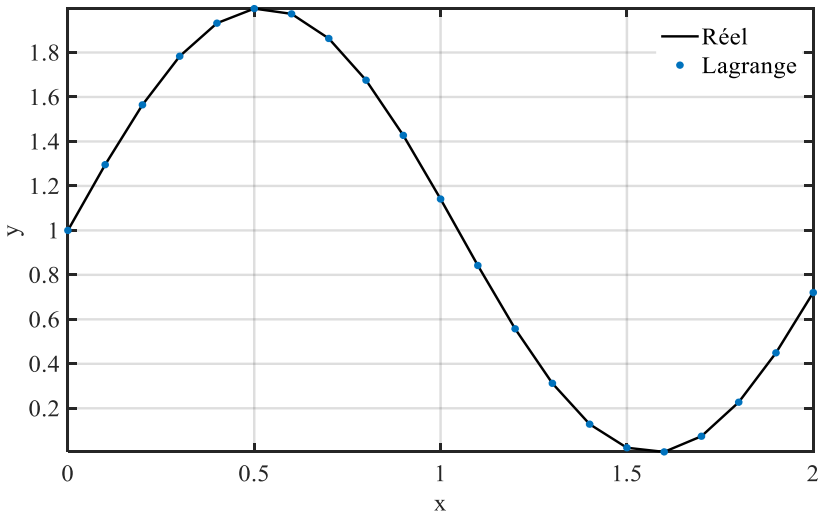


Figure II.4. Lagrange interpolation of the function$f(x) = 1 + sin(3x)$

- Another example is presented here, this time we consider the following table giving the values ν $(m^2.s-1)$ the kinematic viscosity of water as a function of temperature T (℃) :

T (℃)	15	16	17	18	19	20	21	22	23	24	25	26	27	28
ν $(m^2.$ $-1)$	1.14	1.11	1.08	1.06	1.03	1.01	0.983	0.960	0.938	0.917	0.896	0.876	0.857	0.839

Figure II.5 shows the graphs of the data (solid line) and of the corresponding Lagrange polynomial (Stars)

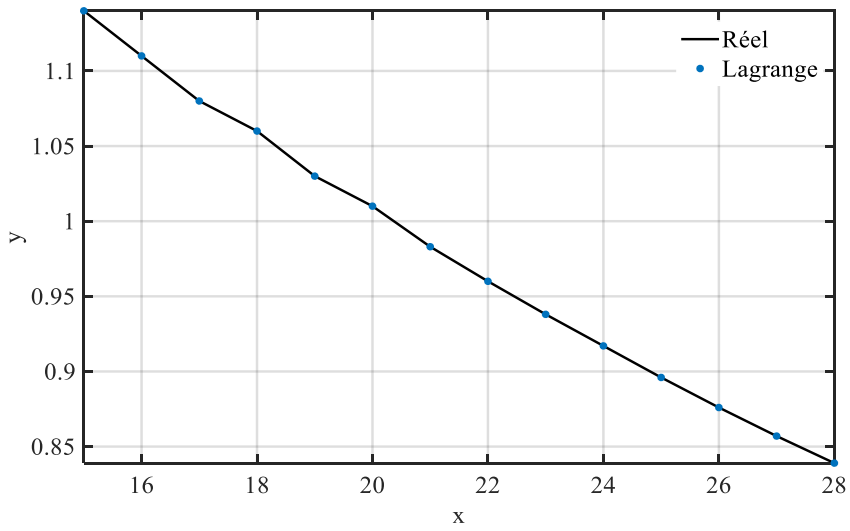


Figure II.5. Lagrange interpolation of Data

II.5 Hermit interpolation

II.5.1 Theorem.

giving $(x_i, y^{(k)}(x_i))$ for $i = 0,1, \dots, n, k = 0,1, \dots, m_i$ where $m_i \in N$, There is a unique polynomial P de degré $\leq m$ p called the Hermite interpolation polynomial such that:

$$P_m = \sum_{i=0}^{n} \sum_{j=0}^{k_i} y^{(j)}(x_i) h_{ij}(x) \tag{II.8}$$

II.5.2 Demonstration.

Polynomials h_{ij} are given by the recurrence relations defined for any $j = 0,1, \dots, k_i - 1$

$$h_{ij}(x) = \frac{(x - x_i)^j}{j!} q_i(x) - \sum_{k=j+1}^{k_i} C_k^j q_i^{(k-j)}(x_i) h_{ik}(x) \tag{II.9}$$

and

$$h_{ik_i}(x) = \frac{(x - x_i)^{k_i}}{k_i!} q_i(x) \tag{II.10}$$

Where

$$q_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \left(\frac{x - x_j}{x_i - x_j} \right)^{k_i+1} \quad \text{(II.11)}$$

In the case where k is constant and equal to 1, we have the following expressions :

$$P_m(x) = \sum_{i=0}^{n} r_i(x) y(x) + s_i(x) y'(x) \quad \text{(II.12)}$$

With

$$r_i(x) = (1 - 2(x - x_i) l_i'(x)) l_i^2(x) \quad \text{(II.13)}$$

and

$$s_i(x) = (x - x_i) l_i^2(x) \quad \text{(II.14)}$$

where l_i is the Lagrange polynomial.

II.5.3 **Matlab**.

We propose a Program called Hermit in which an implementation of the Hermit formula is carried out in order to obtain the interpolation polynomial. In pose as input, three vectors pointx, pointy, and pointyd respectively contain the interpolation variables, the corresponding function values and its derivative. The interpolation polynomial is returned as output. Two for loops and a Lagrange inequality condition are used to produce the functions needed for the interpolation. A after a summation of all the iterations is set up to have the polynomial, and finalizes it by drawing the validation curve.

Algorithm. Hermit Interpolation of Hermit

```
function y = Hermit (pointx,pointy,pointyd)
 n=length(pointx);
y=0;
L=1;
syms x
for i=1:n
for j=1:n
    if (i~=j)
```

```
        L=L*(x-pointx(j))/(pointx(i)-pointx(j));
        dL=L*(1/(pointx(i)-pointx(j)));
    end
end
Ri=(1-2*(x-pointx(i))*dL)*L^2;
si=(x-pointx(i))*L^2;
y=y+ pointy(i)*Ri + pointyd(i)*si;
L=1;
end
plot(pointx,subs(y,pointx),'*',pointx,pointy)
end
```

II.6.3 Examples

1. We give the function

   x = 1:.2:2; y ;

$f(x) = 1/x$ Set to interval]1,2[

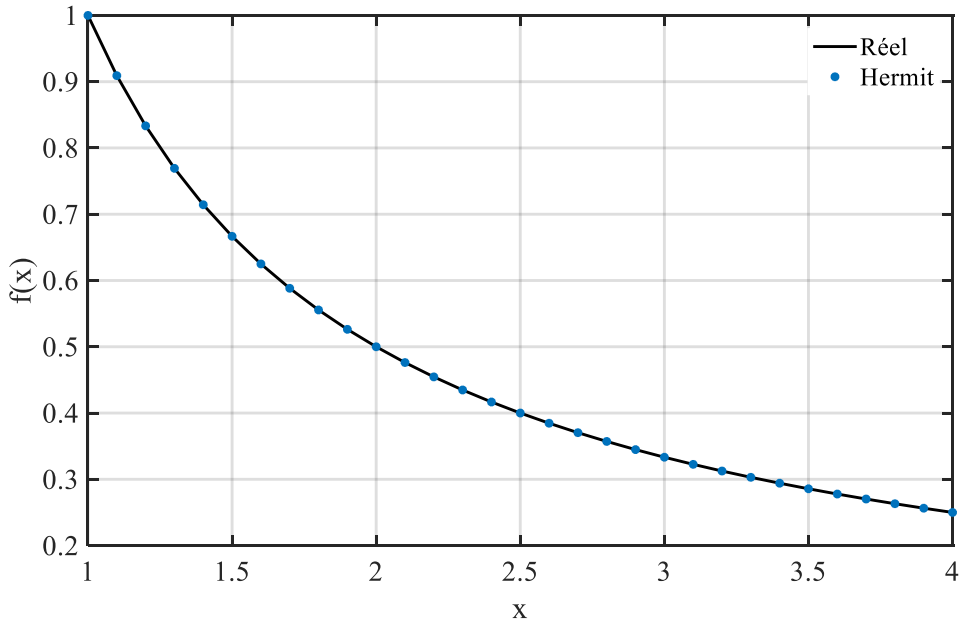


Figure II.4. Hermit's interpolation of the function $f(x) = 1/x$

II.6 Newton's interpolation

II.6.1 Definition.

Let the operator ∇ be defined by: $y = f(x) \rightarrow \nabla y = f(x + \nabla x) - f(x)$. this operator has the following properties :

$$\begin{gathered} \forall\, u\ et\ v,\ \ \nabla(u + v) = \nabla u + \nabla v \\ \nabla(cu) = c\nabla u \ \ ou\ \ c = constant \\ \nabla^n(\nabla^m u) = \nabla^{(n+m)} \\ \nabla^0(y) = y \end{gathered} \tag{II.15}$$

II.6.2 Theorem.

According to Newton's method, The interpolation polynomial of degree n which passes through the $n + 1$ points $(x_0, y_0), (x_0, y_0), \dots, (x_{n-1}, y_{n-1}), (x_n, y_n)$ where the x_i are distinct, is unique and given by:

$$\begin{gathered} P_n(x) = b_0 + b_1 (x - x_0) + b_1 (x - x_0)(x - x_2) + \cdots \\ + b_n(x - x_0) \dots (x - x_{n-1}) \end{gathered} \tag{II.16}$$

II.6.3 **Démonstration**.

The coefficients of P_n are determined by forcing the polynomial through each data point:

$y_i = P_{n-1}(x), i = 1, \dots, n$. This gives the following equations:

$$\begin{cases} b_0 = y(x_0) \\ b_1 = \dfrac{y(x_1) - y(x_0)}{x_1 - x_0} = \nabla^1 y_1(x_1, x_0) \\ b_2 = \dfrac{\dfrac{y(x_2) - y(x_1)}{x_2 - x_1} - \dfrac{y(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0} = \nabla^2 y_2(x_2, x_1, x_0) \\ \vdots \\ b_n = \dfrac{\dfrac{y(x_n) - y(x_{n-1})}{x_2 - x_1} - \dots \dfrac{y(x_1) - y(x_0)}{x_1 - x_0}}{x_n - x_0} = \nabla^n y_n(x_n, \dots, x_0) \end{cases} \tag{II.17}$$

It is more convenient to work with the following format:

$$\begin{array}{llllll} x_0 & y(x_0) & & & & \\ x_1 & y(x_1) & \nabla^1 y_1(x_1, x_0) & & & \\ x_2 & y(x_2) & \nabla^1 y_1(x_2, x_1) & \nabla^2 y_2(x_2, x_1, x_0) & & \\ . & . & . & . & & \\ . & . & . & . & & \\ . & . & . & . & & \\ & & & & . & \\ x_n & y(x_n) & \nabla^1 y_1(x_n, x_{n-1}) & \nabla^2 y_1(x_n, x_{n-1}, x_{n-2}) & . & \nabla^n y_1(x_n, \dots, x_0) \\ & & & & . & \end{array} \qquad \text{(II.18)}$$

II.6.4 Matlab.

Newton's interpolation algorithm is implemented in the New_interp program. The code allows to have the polynomial as an output. We note that the input vectors pointx, pointy are vectors containing respectively the interpolation variables, and the corresponding function values. Two for loops are used to fill a matrix with the derivative operators, and using another loop the polynomial was created. The code will end with a validation plot.

Algorithm. New_interp Newton interpolation

```
function y = New_interp (pointx, pointy)
   m  = length(pointx);
   Df = zeros(m, m);
   Df(:, 1) = pointy';
 for j=2:m
   for i=j:m
      Df(i,j)=(Df(i-1,j-1)-Df(i,j-1))/(pointx(i-j+1)-pointx(i))
    end
end
syms x
   b=1;
   for j = 1 : m-1
  b=(x - pointx(j)).*b;
  c(j)=b.*Df(j+1,j+1);
   end
  y=pointy(1)+sum(c);
plot(pointx,subs(y,pointx),'*',pointx,pointy)
end
```

II.6.5 Example

We give the function

$f(x) = 1 + sin(3x)$ Set to interval]0,2[

- Calculate The values of f and the deferences divided over the interval]0.2[with a step of 0.2
- Corresponding to the function Write Newton's polynomial $P(x)$
- Evaluate $P(x)$ for $x = 0.4\,, x = 0\,.8, and\ x = 1.2$
- Draw the figures $f(x)\ and\ P(x)$

II.7 Chebyshev interpolation

II.7.1 Definition.

Tchebychev interpolation, also called Lagrange interpolation at Tchebychev points, is a Lagrange interpolation performed at particular points defined by

$$xi = cos\left(\frac{2(n-i)+1}{2n+2}\right)\pi \quad and\ \ 0 < i < n$$

II.8 Spline

Spline interpolation is a piecewise approximation method that is done in several steps to construct a representative polynomial.

II.8.1 Definition.

Let $\{x_0\,, \ldots, x_n\}, n\ +\ 1$ interpolation points of a function f on $[a, b]$. We are looking for a polynomial $Sp_k(x_i)$ of degree (k) on each elementary interval $[x_i, x_{i+1}[$ verified

$$Sp_{k,i}(x_i) = Sp_{k,[x_i x_{i+1}]} \in P_k,\ \text{for } i = 0,1, \ldots, n \qquad Sp_k \in C^{k-1} \tag{II.19}$$

i-th, polynomial $Sp_{k,i}(x_i) = Sp_{k,[x_i x_{i+1}]}$composing the spline has order (k) and can be written Under the form

$$Sp_{k,i} = \sum_{j=1}^{k} Sp_{h,i}(x - x_i)^k \qquad \text{(II.20)}$$

II.8.2 **Linear Spline**

Giving $(x_0, y_1), (x_2, y_2), \dots, (x_n, y_n),$, in [a,b], Fit the linear splines to the data. It simply involves forming the consecutive data by straight lines. So if the above data is given in ascending order, the linear splines are given by the following functions

$$Sp(x) = \begin{cases} Sp_0(x) = a_0 x + b_0 \quad x \in [x_0, x_1] \\ Sp_1(x) = a_1 x + b_1 \quad x \in [x_1, x_2] \\ \vdots \\ Sp_{n-1}(x) = a_{n-1} x + b_{n-1} \quad x \in [x_{n-1}, x_n] \end{cases} \qquad \text{(II.21)}$$

Checking the interpolation condition:

$$Sp(x_i) = y_i \qquad \text{(II.22)}$$

The determination of $Sp(x)$ requires the evaluation of the coefficients $a_i \; and \; b_i$ for $i = 0, \dots, n-1$, To be able to solve the problem, we have:

$$\begin{cases} Sp(x_i) = y_i \\ y_i = a_i x_i + b_i \end{cases}$$

$$\begin{cases} Sp(x_{i+1}) = y_{i+1} \\ y_{i+1} = a_i x_{i+1} + b_i \end{cases} \qquad \text{(II.23)}$$

We therefore deduce for$i = 0,1, \dots, n$ $Sp(x_i)$ can be written in the form

$$Sp_i(x) = Sp(x_i) + \frac{Sp(x_{i+1}) - Sp(x_i)}{x_{i+1} - x_i}(x - x_i) \qquad Where \; x_i < x < x_{i+1} \qquad \text{(II.24)}$$

II.8.2.1 **Matlab**.

The Matlab code spline_linear allows to perform interpolation by linear splines. The code uses the input vectors pointx, pointy, which are vectors containing the x and y coordinates of the function respectively. As for loop is used for the evaluation of the polynomial coefficients. The result is compared with the exact result by a validation plot.

Algorithm. spline_linear Linear Splines

```
function poly = spline_linear (pointx, pointy)

%  pointx: les coordonnées x
%  pointy: les coordonnées y
%  M=(y-y0)/(x-x0)
%  Sp=a(x-x0)+y0

pointx=(0:0.2:2)';
pointy=1+sin(3*pointx);
n  = length(pointx);

for i=1:n-1

M=(pointy(i+1)-pointy(i))/(pointx(i+1)-pointx(i))
xspline=linspace(pointx(i),pointx(i+1),10)
Sp=pointy(i)+M*(xspline-pointx(i))
plot(xspline,Sp)
pause

end

scatter(pointx,pointy,50,'r','filled')
grid on;
xlim([min(pointx) max(pointx)]);
ylim([min(pointy) max(pointy)]);
xlabel('x');
ylabel('y');
title('Linéar Spline')
```

II.8.3 Quadratic spline

Consider in in [a,b], n+1 distinct points $x_0, \ldots, x_n$ and n+1 corresponding values $y_0, \ldots, y_n$. The interpolation by quadratic splines is equivalent to the approximation on each elementary interval $[x_i , x_{i+1}]$ by a polynomial of the second degree in the form:

$$Sp(x) = \begin{cases} Sp_0(x) = a_0x^2 + b_0x + c_0 & x \in [x_0, x_1] \\ Sp_1(x) = a_1x^2 + b_1x + c_1 & x \in [x_1, x_2] \\ \vdots & \\ Sp_{n-1}(x) = a_{n-1}x^2 + b_{n-1}x + c_{n-1} & x \in [x_{n-1}, x_n] \end{cases} \quad \text{(II.25)}$$

The determination of $Sp(x)$ requires the evaluation of the coefficients a_i , b_i and c_i for $i = 0, \dots, n-1$, To be able to solve the problem, we know that Each quadratic spline passes through two consecutive points, this gives us 2n equations:

$$\begin{cases} Sp_i(x_i) = y_i \\ y_i = a_i x_i^2 + b_i x_i + c_i \end{cases} \qquad \text{(II.26)}$$

$$\begin{cases} Sp_i(x_{i+1}) = y_{i+1} \\ y_{i+1} = a_i x_{i+1}^2 + b_i x_{i+1} + c_i \end{cases}$$

We also know that the first derivatives of consecutive splines are continuous and equal in common points, which can give us (n-1) equations:

$$\begin{cases} \grave{Sp}_i(x_{i+1}) = \grave{Sp}_{i+1}(x_{i+1}) \\ 2a_i x_i + b_i = 2a_{i+1} x_{i+1} + b_{i+1} \end{cases} \qquad \text{(II.27)}$$

We are now left with only one equation to solve the problem, we can assume that the first spline is linear, then: a_1=0

II.1.1 **Matlab**.

Quadratic spline interpolation is implemented in the program Quad_spline. The code calculates the polynomial coefficients. We note that the input vectors pointx, pointy are vectors containing respectively the interpolation variables, and the corresponding function values. Two for loops are used for the spline continuity and derivative continuity condition, the results are compared with the exact results by a validation plot.

Algorithm. Quad_spline Quadratic Spline

```
function [coef] = Quad_spline (pointx, pointy)

%  pointx: les coordonnées x
%  pointy: les coordonnées y
n  =  length(pointx)-1;
% Nécessite  l'évaluation de  3*n Spécifient a_i ,b_i  et c_i
Spi  =  [0;zeros(3*n-1,1)];
Mat  =  zeros(length(Spi),length(Spi));
```

```
% Initialisation de première spline
% la première spline est linéaire
Mat(1,1)=1;
% la condition de continuité de spline
idx=0;
count=1;
for i=2:2:2*n
   idx=idx+1
   Mat(i,count:count+2)       =   [pointx(idx)^2 pointx(idx) 1];
   Mat(i+1,count:count+2)     =   [pointx(idx+1)^2 pointx(idx+1) 1];
   Spi(i)                =   pointy(idx);
   Spi(i+1)              =   pointy(idx+1);
   count                 =   count+3;

end
%  continuité de la dérivée de spline
jdx=1;
fdx=2;
for i=2*n+2:3*n

   Mat(i,[jdx jdx+1 jdx+3 jdx+4]) =   [2*pointx(fdx) 1 -2*pointx(fdx) -1];

   jdx              =   jdx+3;
   fdx              =   fdx+1;
end

% calcul des coefficients
coef      =      Mat\Spi;
j=1;
hold on;
for i=1:n

   xspline=linspace(pointx(i),pointx(i+1),10)
   Splq=coef(j)*xspline.^2+coef(j+1)*xspline+coef(j+2)
   plot(xspline,Splq)
   pause
   hold on
   j=j+3;
end
scatter(pointx,pointy,50,'r','filled')
grid on;
xlim([min(pointx) max(pointx)]);
ylim([min(pointy) max(pointy)]);
xlabel('x');
ylabel('y');
title('Quadratic Spline')

end
```

II. Question

Exercice. 1 : write a Matlab program that evaluates the Lagrange, Newton and Hermite interpolation of the function $f(x) = sin(4\pi x)$ on the interval [0,1]

III. numerical integration

III.1. Trapeze method

The approximate method of integration, known as the trapezium method, consisting in replacing the initial function by a staircase approximation. We will therefore here also tabulate $f(x)$ at n+1 points $f_0, f_1, \dots, f_n$ in the integration interval $[x_0, x_n]$. We will then interpolate linearly (polynomial of order 1, i.e. a straight line) to approximate f(x) between two tabulated points. We will therefore replace the arcs x_i, x_{i+1} by their chords.

Integration will then be done by summing the area of the trapezoids, i.e.

$$\int_{x_{i=1}}^{n} f(x)dx = \frac{1}{2}\sum_{i=1}^{n}(f_{i-1} + f_i)\,\Delta x_i \qquad \text{III.1}$$

We assume that all the points are regularly spaced:

$$\Delta x_i = \Delta x = \frac{b-a}{n} \qquad \text{III.2}$$

Thus, the previous integral becomes:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2}[(f_0 + f_1) + (f_1 + f_2) + \cdots + (f_{n-2} + f_{n-1}) + (f_{n-1} + f_n)] \qquad \text{III.3}$$

We can finally deduce

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2}[f_0 + f_n + 2\sum_{i=1}^{n-1} f_i] \qquad \text{III.4}$$

III.1.1 **Matlab**.

This algorithm calculates the integral "I" of any function via the trapezoid rule in the interval [a, b] with n + 1 equidistant points

Algorithm. trap_int trapeze integration

```
functionint= trap_int
a=input (‘ insérer le borne inferieur de l’intervalle)
b=input (‘ insérer le borne supérieur de l’intervalle’)
m=input (‘ insérer le nombre de devision’)
fun=@(x) (insérer la function ici)
x=[a:h:b];
dim=length(x);
y=eval (fun);
if size(y) ==1
y= diag(ones(dim))*y;
```

```
end
int=h*(0.5*y(1)+sum (y(2:m))+0.5*y(m+1));
```

III.2 Simpson's method

Simpson's method allows the approximate calculation of an integral with the following formula:

$$\int_a^b f(x)dx \approx \frac{(b-a)}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)] \quad \text{(III.5)}$$

In this formula, one can wonder where the coefficients $1/6$ and $2/3$ (which appears as $4/6$) come from.

To obtain Simpson's formula, we will carry out an interpolation with a polynomial of degree 2. A polynomial being a function very easy to integrate, we approximate the integral of the function f over the interval [a,b], by l 'integral of the polynomial at the nodes $x_0 = a, x_1 = \frac{a+b}{2}$ and $x_2 = b$.

$$\int_a^b f(x) \approx \int_a^b P(x) = \int_a^b Ax^2 + Bx + C\ dx \quad \text{(III.6)}$$

To facilitate the calculations, we will use the interval $[-h, h]$ such as$h = \frac{b-a}{2}$

After integration:

$$\int_{-h}^{h} P(x) = \frac{A}{3}x^3 + \frac{B}{2}x^2 + Cx|_a^b \quad \text{(III.7)}$$

We evaluate the integral over the interval$[-\boldsymbol{h}, \boldsymbol{h}]$ we obtain

$$\int_{-h}^{h} P(x) = \frac{A}{3}(2h^3) + C2h \quad \text{(III.8)}$$

We take h as a common factor, the expression becomes:

$$\int_a^b P(x) = \frac{h}{3}(2Ah^2 + 6C) \quad \text{(III.9)}$$

Now, we evaluate the polynomial P on three points in the interval $[-h, h]$

$$\begin{cases} P(-h) = Ah^2 - Bh + C \cong f(-h) \\ P(h) = Ah^2 + Bh + C \approx f(h) \\ P(0) = C \approx f(0) \end{cases} \quad \text{(III.10)}$$

Adding the terms of the previous equation together, we get:

$$f(-h) + f(h) = 2Ah^2 + 2C \quad \text{(III.11)}$$

We decompose 6C into two values

$$\int_{-h}^{h} P(x) = \frac{h}{3}(2Ah^2 + 2C + 4C) \quad \text{(III.12)}$$

we replace the equation in the expression

$$\int_{-h}^{h} P(x) = \frac{h}{3}(f(-h) + f(h) + 4f(0)) \quad \text{(III.13)}$$

Finally, we can see that

$$\int_{a}^{b} f(x) = \frac{(b-a)}{6}(f(a) + f(b) + 4f\left(\frac{b+a}{2}\right)) \quad \text{(28)}$$

The previous expression can be rewritten for several sub-intervals in the form:

$$\int_{a}^{b} f(x)dx \cong \frac{h}{3}[f(x_0) + f(x_n) + 2\sum_{j=1}^{\frac{n}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1})] \quad \text{(III.15)}$$

$h = (b - a)/n$ is the length of these subintervals

$x_i = a + ih$ for $i = 0,1, \ldots, n-1, n$

III.3 **Matlab**.

The present Matlab algorithm calculates the integral "I" via Simpson's rule in the interval [a, b] with n + 1 equidistant points

Algorithm. simp_int simpson integration

```
functionint= simp_int
a=input (' insérer le borne inferieur de l'intervalle)
b=input (' insérer le borne supérieur de l'intervalle')
```

```
m=input (‘ insérer le nombre de devision’)
fun=@(x) (insérer la function ici)
h=(b-a)/n;
x=a:h:b;
   I= h/3*(fun(x(1))+2*sum(fun(x(3:2:end-2)))+4*sum(fun(x(2:2:end)))+fun(x(end)));
```

III.1 Applications

We give the function

$$f(x)=\text{x.\^{}2-2.*x-3}$$

- Calculate the integral of the function $f(x)$, taking $\in [-1,2]$ using Simpson's and trapezium methods.
- Implement the two methods on Matlab in order to solve the same function.

IV. Method of direct solution of systems of linear equations

IV.1 Gaussian method

Consider the following system of linear equations:

$$A\ x = b$$

Where A is a matrix of coefficients, x is a vector of unknowns, and b is a constant vector.

Gauss's method consists of performing operations on the rows of the matrix A in order to transform it into an upper triangular shape. More precisely, we perform operations of the form:

$$L_i = L_i + c * L_j$$

Where L_i and L_j are rows of matrix A and c is a constant coefficient. This operation consists of adding a multiple of line $L_j$to line L_i. We perform these operations so as to have a non-zero element in the first column of the matrix, then a non-zero element in the second column, and so on until we have an upper triangular matrix

Then, one can solve the system of equations using upward substitution. This involves raising the upper triangular matrix using the following equation for each unknown $\_i$:

$$x\_i = (b\_i - sum(j = i + 1 \text{ à } n, A\_ij * x\_j)) / A\_ii$$

Where n is the number of unknowns. This equation uses the known values of x_j for j greater than i to find the value of x_i. We can thus find the value of all the unknowns x by going up the upper triangular matrix.

IV.1 The LU (Lower-Upper) factorization method

LU is a method for solving a system of linear equations by decomposing the coefficient matrix into two lower and upper triangular matrices. It can be mathematically described as follows:

Consider the following system of linear equations:

$$A\ x = b$$

Where A is a matrix of coefficients, x is a vector of unknowns, and b is a constant vector.

The LU factorization method consists in finding two matrices L and U such that:

$$A = L\ U$$

Where L is a lower triangular matrix (with 1s on the diagonal) and U is an upper triangular matrix.

Once the L and U matrices have been found, the system of equations can be solved in two steps:

Solve the system of equations $L\ y = b$ using backward substitution.

Solve the system of equations $U\ x\ =\ y$ using upward substitution.

The LU factorization method is often faster than the Gaussian method for large systems of equations because it requires fewer substitution and elimination operations. However, the LU factorization method can be more complex to implement and can fail if the matrix of coefficients is singular.

IV.3 **Matlab**.

Algorithm. Gaussian method

```
function x = Gauss(A, b)
% Function to solve the system of linear equations Ax = b
% using the Gaussian method.
% A: matrix of coefficients
% b: constant vector
% x: solution vector
[n,n] = size(A);
x = zeros(n,1);

% Étape 1 : Elimination
for k = 1 : n-1
   for i = k+1 : n
      factor = A(i,k) / A(k,k);
      A(i,k+1:n) = A(i,k+1:n) - factor * A(k,k+1:n);
      b(i) = b(i) - factor * b(k);
   end
end
% Step 2: Upward Substitution
x(n) = b(n) / A(n,n);
for i = n-1 : -1 : 1
   x(i) = (b(i) - A(i,i+1:n) * x(i+1:n)) / A(i,i);
end
```

Algorithm. *LU* method

```
function x = LU(A, b)
% Function to solve the system of linear equations Ax = b
% using the LU factorization method.
% A: matrix of coefficients
% b: constant vector
% x: solution vector

[n,n] = size(A);
x = zeros(n,1);
L = eye(n);

% Step 1: Factorization LU
for k = 1 : n-1
   for i = k+1 : n
```

```
        factor = A(i,k) / A(k,k);
        L(i,k) = factor;
        A(i,k+1:n) = A(i,k+1:n) - factor * A(k,k+1:n);
    end
end
U = triu(A);

% Step 2: Solve the system
y = L \ b;
x = U \ y;
```

V. Solving Differential Equations

V.1 Euler's method

Euler's method is a numerical method for solving ordinary differential equations (ODE) of the type:

$$dy/dt = f(t,y)$$

Où y est la variable dépendante et t est le temps.

La méthode d'Euler consiste à approcher la solution y(t) en utilisant une série de points discrets $(t\_0, y\_0), (t\_1, y\_1), \ldots, (t\_n, y\_n)$ sur la courbe. À partir d'un point initial (t_0, y_0), la méthode d'Euler utilise la dérivée approximative de la courbe à ce point pour déterminer le prochain point (t_1, y_1).

Where y is the dependent variable and t is time.

Euler's method consists of approximating the solution $y(t)$ using a series of discrete points $(t\_0, y\_0), (t\_1, y\_1), \ldots, (t\_n, y\_n)$ on the curve. Starting from an initial point (t_0, y_0), Euler's method uses the approximate derivative of the curve at that point to determine the next point (t_1, y_1).

Mathematically, Euler's method can be formulated as follows:

$$y\_\{i+1\} = y\_i + h * f(t\_i, y\_i)$$

Where h is the time step and i is a time index. Euler's method can be implemented in a loop by considering successive values of t and y until the final time is reached.

Euler's method is simple to implement and can be used to quickly solve simple differential equations, but it can give inaccurate results for more complex ODEs due to the approximate nature of the method. There are more accurate methods for solving ODEs, such as the Runge-Kutta method or the modified form method.

V.2 The Runge-Kutta method

The Runge-Kutta method is a numerical method for solving ordinary differential equations (ODE) of the type:

$$dy/dt = f(t,y)$$

Where y is the dependent variable and t is time.

The Runge-Kutta method uses a more advanced approach to approximating the derivative of the curve by using multiple intermediate points instead of a single point. There are several forms of the Runge-Kutta method, but the best-known form is the Runge-Kutta method of order 4 (RK4).

The mathematical formula for the Runge-Kutta method of order 4 is given by::

$$k1 = h * f(t\_i, y\_i)\ k2 = h * f(t\_i + h/2, y\_i + k1/2)\ k3 = h * f(t\_i + h/2, y\_i + k2/2)\ k4 = h * f(t\_i + h, y\_i + k3)\ y\_\{i+1\} = y\_i + (k1 + 2k2 + 2k3 + k4)/6$$

Where h is the time step and i is a time index. The Runge-Kutta method can be implemented in a loop by taking into account the successive values of t and y until the final time is reached.

Runge-Kutta's method is more accurate than Euler's method in solving ODEs because it uses multiple intermediate points to approximate the derivative of the curve. However, the Runge-Kutta method is also more complex to implement and requires more computations for each iteration.

Matlab.

Algorithm. Euler method

```
function [t, y] = euler(f, tspan, y0, h)
% EULER Solve ordinary differential equation using Euler's method
% [T, Y] = EULER(F, TSPAN, Y0, H) with TSPAN = [T0, T1, ..., TFINAL]
% solves the ODE y' = f(t,y) from time T0 to TFINAL with initial value
% Y0 using step size H. The function F(T, Y) must return a column vector
% corresponding to f(t,y). Each row in the solution array Y corresponds
% to a time in T.

t = tspan(1):h:tspan(end);
y = zeros(length(t), length(y0));
y(1,:) = y0;

for i = 2:length(t)
y(i,:) = y(i-1,:) + h * f(t(i-1), y(i-1,:)).';
end
```

Algorithm. Runge Kutta method

```
function [t, y] = rk4(f, tspan, y0, h)
% RK4 Solve ordinary differential equation using Runge-Kutta method
% [T, Y] = RK4(F, TSPAN, Y0, H) with TSPAN = [T0, T1, ..., TFINAL]
% solves the ODE y' = f(t,y) from time T0 to TFINAL with initial value
% Y0 using step size H. The function F(T, Y) must return a column vector
% corresponding to f(t,y). Each row in the solution array Y corresponds
% to a time in T.
t = tspan(1):h:tspan(end);
y = zeros(length(t), length(y0));
y(1,:) = y0;
for i = 2:length(t)
k1 = h * f(t(i-1), y(i-1,:)).';
k2 = h * f(t(i-1) + h/2, y(i-1,:) + k1/2).';
k3 = h * f(t(i-1) + h/2, y(i-1,:) + k2/2).';
k4 = h * f(t(i), y(i-1,:) + k3).';
y(i,:) = y(i-1,:) + (k1 + 2k2 + 2k3 + k4)/6;
```

```
end
```

Reference

1. Jaan, K. (2009). numerical methods in engineering with MATLAB.
2. Esfandiari, R. S. (2017). *Numerical methods for engineers and scientists using MATLAB®*. Crc Press.
3. Chapra, S. (2011). *EBOOK: Applied Numerical Methods with MATLAB for Engineers and Scientists*. McGraw Hill.
4. Mathews, J. H., & Fink, K. D. (2004). *Numerical methods using MATLAB* (Vol. 4). Upper Saddle River, NJ: Pearson prentice hall.
5. Valentine, D. T., & Hahn, B. (2022). *Essential MATLAB for engineers and scientists*. Academic Press.
6. Quarteroni, A., Sacco, R., & Saleri, F. (2008). Méthodes Numériques: Algorithmes, analyse et applications. Springer Science & Business Media.
7. Quarteroni, A. M., Sacco, R., & Saleri, F. (2000). Méthodes numériques pour le calcul scientifique: programmes en MATLAB. Springer Science & Business Media.
8. Grivet, J. P. (2021). Méthodes numériques appliquées. In Méthodes numériques appliquées. EDP sciences.