People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

Ibn Khaldoun University – Tiaret

Faculty of Material Sciences

Department of Physics

Course Notes

Numerical Physics

Intended for students

3rd Year (LMD System)

Field of Study : Physics

Specialization : Fundamental Physics

The Author:

Dr. Benabdellah Ghlamallah

2025/2026

Preface

This course handout « Numerical Physics » is intended for third-year L.M.D. students, majoring in Fundamental Physics (L5). It consists of four chapters, following the framework of the L.M.D. academic bachelor program (2014–2015). The objective is to design and study numerical methods for solving certain mathematical problems in physics, generally arising from the modeling of real-world problems, where the solution is sought through computer calculations.

Each chapter includes a series of exercises with solutions, as well as practical sessions (Lab), enabling the reader to gradually learn the theory of selected numerical methods and to apply the acquired theoretical knowledge in writing computer programs that implement the studied methods.

Table des matières

Constantes universelles

Speed of light in vacuum	$c_0 = 2.997\,924\,58 \times 10^{8}\,\mathrm{m\,s^{-1}}$
Electron charge	$e = 1.602\,05 \times 10^{-19}\,\mathrm{C}$
Rest mass of the electron	$m_0 = 9.1083 \times 10^{-31}\,\mathrm{kg}$
Rest mass of the neutron	$m_n = 1.676 \times 10^{-27}\,\mathrm{kg}$
Rest mass of the proton	$m_p = 1.679 \times 10^{-27}\,\mathrm{kg}$
Mass ratio	$m_p/m_0 = 1836.1$
Planck's constant	$h = 6.6245 \times 10^{-34}\,\mathrm{J\,s}$
Boltzmann's constant	$k = 1.3803 \times 10^{-23}\,\mathrm{J\,K^{-1}}$
Avogadro's number (molecules per mole)	$N_A = 6.024\,86 \times 10^{23}\,\mathrm{mol^{-1}}$

General Introduction

The experimental physicist performs an experiment to highlight a phenomenon and obtains numerical measurement results, whereas the theoretician proposes a theory using approximations in order to find a model that admits an analytical solution to the problem and predicts the experimental results. However, these approximations can be sources of errors or deviations from the experimental results. The computational physicist transforms the model into computer code and executes it on a computer to obtain simulated results, which the theoretician can then compare with theoretical predictions.

With the advent of computers and the progress of numerical analysis, it is no longer necessary to search for linearized equations or to reduce the number of variables. Instead, the goal is to find the appropriate computational scheme (the algorithm) to solve real problems within an acceptable computation time. This field (specialty) is known as « Numerical Physics ».

Numerical physics, which can be defined as the intersection of physical science, computer science, and mathematics, more generally allows :

— obtaining numerical solutions to known equations that do not possess an analytical solution ;

— verifying the validity of approximations proposed by theoreticians ;

— carrying out experiments that are not feasible in practice ;

— designing experiments before attempting them in reality (reducing costs).

The practice of numerical physics requires :

— a solid knowledge of the physics of the relevant domains ;

— an equally solid understanding of numerical analysis methods ;

— practical experience in algorithms and in one or more programming languages.

Chapitre 1

Polynomial Interpolation

1.1 Introduction

In physics, many problems rely on experimental data or numerical simulations that provide results only at discrete points. However, physical quantities are usually continuous functions of variables such as space, time, or energy. Interpolation provides a mathematical tool to construct continuous approximations of these quantities from a finite set of known data points.

The importance of interpolation in the field of physics can be summarized as follows :

— It enables the estimation of intermediate values of physical quantities when direct measurement or computation is not available.

— It allows the comparison of experimental data with theoretical models defined on different grids or domains.

— It serves as a foundation for numerical methods, such as numerical integration, differentiation, and the solution of differential equations.

— It helps in reducing the cost of experiments or simulations by requiring fewer sampled points while still providing accurate approximations.

Therefore, interpolation is not only a practical technique for handling data but also a fundamental component in numerical physics, bridging the gap between discrete data and continuous physical laws.

1.2 Polynomial Interpolation

In physics, experiments typically yield pairs of measurement values (x_i, y_i), where one quantity depends on the other through a function f, i.e., $y_i = f(x_i)$. Since the function $f(x)$ is known only at a finite number of points x_0, x_1,..., x_n, it is often approximated by a unique interpolation polynomial $P_n(x)$ of degree n, with coefficients $a_0, a_1, \ldots, a_n$. The polynomial $P_n(x)$ is defined on the interval $[x_0, x_n]$ and

satisfies the following linear system :

$$\begin{cases} a_0 + a_1\, x_0 + a_2\, x_0^2 + \cdots + a_n\, x_0^n = y_0, \\ a_0 + a_1\, x_1 + a_2\, x_1^2 + \cdots + a_n\, x_1^n = y_1, \\ \vdots \\ a_0 + a_1\, x_n + a_2\, x_n^2 + \cdots + a_n\, x_n^n = y_n. \end{cases} \tag{1.1}$$

Solving this linear system for the unknown coefficients a_i determines the interpolation polynomial $P_n(x)$. Interpolation requires that $P_n(x_i) = f(x_i)$ for all $i = 0, 1, \ldots, n$. Once $P_n(x)$ is obtained, one may need to evaluate $f(x)$ at values of x **not among the given nodes**. If x lies within $[x_0, x_n]$, the process is called *interpolation* ; if x is outside, it is called *extrapolation*.

Polynomial approximation is widely used, since the error can be made arbitrarily small by increasing the polynomial degree [3].

1.3 Lagrange Polynomial Interpolation

1.3.1 Lagrange Polynomial

For a polynomial of degree **n = 1** :

$$P_1(x) = a_0 + a_1 x \tag{1.2}$$

The system of equations associated with this polynomial can be written as :

$$\begin{cases} a_0 + a_1\, x_0 = y_0, \\ a_0 + a_1\, x_1 = y_1. \end{cases} \tag{1.3}$$

We obtain :

$$\begin{aligned} a_0 &= y_0 - \frac{y_1 - y_0}{x_1 - x_0}\, x_0, \\ a_1 &= \frac{y_1 - y_0}{x_1 - x_0}. \end{aligned}$$

Therefore, the polynomial (1.2) can be written as :

$$\begin{aligned} P_1(x) &= y_0 - \frac{y_1 - y_0}{x_1 - x_0} x_0 + \frac{y_1 - y_0}{x_1 - x_0} x \\ &= \frac{y_0(x_1 - x) + y_1(x - x_0)}{x_1 - x_0} \\ &= y_0 \left(\frac{x - x_1}{x_0 - x_1} \right) + y_1 \left(\frac{x - x_0}{x_1 - x_0} \right) \\ &= y_0 L_0(x) + y_1 L_1(x). \end{aligned} \tag{1.4}$$

Similarly, for a polynomial of degree $n = 2$:

$$P_2(x) = a_0 + a_1 x + a_2 x^2 \tag{1.5}$$

$$\begin{aligned} P_2(x) &= y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} \\ &\quad + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \\ &= y_0 L_0(x) + y_1 L_1(x) + y_2 L_2(x). \end{aligned} \tag{1.6}$$

In general, the **Lagrange** [1] polynomial is written as :

$$\begin{aligned} P_n(x) &= \sum_{i=0}^{n} y_i \prod_{j=0; j \neq i}^{n} \frac{x - x_j}{x_i - x_j} \\ &= \sum_{i=0}^{n} y_i \cdot L_i(x). \end{aligned} \tag{1.7}$$

1.3.2 Lagrange Polynomial Algorithm

1. Joseph-Louis Lagrange, was borne in Italy, 1736-1813

Algorithm 1.3.1 Lagrange Polynomial Interpolation Algorithm

Require: Arrays *xvalues*(i) and *yvalues*(i), interpolation point z

function LAGRANGE_POLYNOME_INTERPOLATION(*xvalues*, *yvalues*, z)
  $p \leftarrow \text{size}(xvalues)$
  $n \leftarrow p - 1$
  $y \leftarrow 0$
  for $i \leftarrow 1$ to $n + 1$ **do**
    $L \leftarrow 1$
    for $j \leftarrow 1$ to $n + 1$ **do**
      if $i \neq j$ **then**
        $L \leftarrow L \times \dfrac{z - xvalues(j)}{xvalues(i) - xvalues(j)}$
      end if
    end for
    $y \leftarrow y + yvalues(i) \times L$
  end for
  return y
end function

Python Programme :

```
def lagrange_interpolation(xi, yi, z, n ):
    poly = 0
    p=n+1
    for i in range(p):
        L = 1
        for j in range(p):
            if j != i:
                L *= (xi[j]- z) / (xi[j]-xi[i])
        poly += yi[i] * L
    return poly
```

- **Example 1.3.1.** ***:***

Let the function $f(x)$ pass through the points given in the following table :

t_i (s)	1.0	1.5	2.0
y_i (m)	0.0000	0.6082	1.3863

1. *Find the approximate value of $f(1.75)$ by Lagrange polynomial interpolation.*
2. *Knowing that the true function is $f(x) = x\ln(x)$, compute the absolute error.*

Solution :

The Lagrange interpolation polynomial for three points ($n = 2$) is

$$P_2(t) = y_0 L_0(t) + y_1 L_1(t) + y_2 L_2(t),$$

with the basis polynomials

$$L_0(t) = \frac{(t-t_1)(t-t_2)}{(t_0-t_1)(t_0-t_2)}, \quad L_1(t) = \frac{(t-t_0)(t-t_2)}{(t_1-t_0)(t_1-t_2)}, \quad L_2(t) = \frac{(t-t_0)(t-t_1)}{(t_2-t_0)(t_2-t_1)}.$$

Evaluate at $t = 1.75$:

$$L_0(1.75) = \frac{(1.75-1.5)(1.75-2.0)}{(1.0-1.5)(1.0-2.0)} = 0.000000,$$
$$L_1(1.75) = \frac{(1.75-1.0)(1.75-2.0)}{(1.5-1.0)(1.5-2.0)} \approx 0.749999,$$
$$L_2(1.75) = \frac{(1.75-1.0)(1.75-1.5)}{(2.0-1.0)(2.0-1.5)} \approx 0.250001.$$

Hence

$$P_2(1.75) = y_0 L_0(1.75) + y_1 L_1(1.75) + y_2 L_2(1.75)$$
$$P_2(1.75) = 0.0000 \cdot 0 + 0.6082 \cdot 0.749999 + 1.3863 \cdot 0.250001$$
$$P_2(1.75) \approx 0.9760125.$$

The exact value (given $f(x) = x \ln x$) is

$$f(1.75) = 1.75 \ln(1.75) \approx 0.9793276.$$

Therefore the absolute error is

$$\left|P_2(1.75) - f(1.75)\right| \approx |0.9760125 - 0.9793276| \approx 0.0033151.$$

the interpolant underestimates the true value here, so the signed error $P_2(1.75) - f(1.75) \approx -0.0033151$.

Remark 1.3.2. *:*

1. *The Lagrange formula depends only on the abscissas x_i, $i = 0, 1, \ldots, n$.*
2. *The Lagrange interpolation method has two major drawbacks :*
   - *The approximation error does not necessarily decrease when the number of interpolation points increases.*
   - *The method is not recursive : it is impossible to deduce the interpolation polynomial P_{n+1} from P_n when a new interpolation point (x_{n+1}, y_{n+1}) is added.*

Indeed, introducing a new point x_{n+1} requires recomputing all the Lagrange polynomials L_i at the points x_i, $i = 0, 1, \ldots, n+1$.

3. *What is the most appropriate choice of the polynomial degree? The answer is not straightforward :*
   - *A polynomial of excessively high degree may lead to strong oscillations between the known points, which distorts the interpolation.*
   - *A degree higher than 3 is only justified if supported by reliable reference data (for example, additional experimental measurements).*

1.4 Newton Polynomial Interpolation

1.4.1 Divided Differences

Let $x_0, x_1, \ldots, x_n$ be $(n+1)$ distinct points in the interval $[a, b]$, and let $y_i = f(x_i)$ where f is a function defined on $[a, b]$.

The divided differences of successive orders $0, 1, 2, \ldots, n$ are defined as follows :

$$\begin{aligned}
\text{Order } 0 : \delta^0 y_i &= y_i = [x_i] \\
\text{Order } 1 : \delta^1 y_i &= \frac{\delta^0 y_{i+1} - \delta^0 y_i}{x_{i+1} - x_i} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = [x_i, x_{i+1}] \\
\text{Order } 2 : \delta^2 y_i &= \frac{\delta^1 y_{i+1} - \delta^1 y_i}{x_{i+2} - x_i} = [x_i, x_{i+1}, x_{i+2}] \\
&\vdots \\
\text{Order } n : \delta^n y_i &= \frac{\delta^{n-1} y_{i+1} - \delta^{n-1} y_i}{x_{i+n} - x_i} = [x_i, x_{i+1}, x_{i+2}, \ldots, x_{n+1}]
\end{aligned}$$

Construction of a divided difference table :

1.4.2 Newton's Polynomial Formula

Newton's polynomial [2] formula is written using divided differences :

$$\begin{aligned}
P_n &= y_0 + \delta^1 y_0 (x - x_0) + \delta^2 y_0 (x - x_0)(x - x_1) + \ldots + \delta^n y_0 (x - x_0)\ldots(x - x_{n-1}) \\
&= y_0 + \sum_{i=1}^{n} \delta^i y_0 \prod_{j=0}^{i-1} (x - x_j)
\end{aligned} \tag{1.8}$$

2. Sir Isaac Newton, anglais, 1643-1727

i	x_i	$y_i = \delta^0 y_i$	$\delta^1 y_i$	$\delta^2 y_i$	$\delta^3 y_i$
0	x_0	y_0	$\delta^1 y_0 = \frac{y_1 - y_0}{x_1 - x_0}$	$\delta^2 y_0 = \frac{\delta^1 y_1 - \delta^1 y_0}{x_2 - x_0}$	$\delta^3 y_0 = \frac{\delta^2 y_1 - \delta^2 y_0}{x_3 - x_0}$
1	x_1	y_1	$\delta^1 y_1 = \frac{y_2 - y_1}{x_2 - x_1}$	$\delta^2 y_1 = \frac{\delta^1 y_2 - \delta^1 y_1}{x_3 - x_1}$	
2	x_2	y_2	$\delta^1 y_2 = \frac{y_3 - y_2}{x_3 - x_2}$		
3	x_3	y_3			

Remark 1.4.1. *:*

1. *Renumber the points in such a way that the first point is the one closest to the interpolation point, and so on.*
2. *Compute the absolute error. If this error is of the order 10^{-6}, stop the calculation; otherwise, continue the interpolation until the table is exhausted (until convergence).*
3. *The interpolation polynomial P_n can be defined by the following recurrence relation :*

$$\begin{cases} P_0(x) = f(x_0) \\ P_n(x) = P_{n-1}(x) + \delta^n y_0 \prod_{j=0}^{n-1}(x - x_j); \quad n > 1 \end{cases} \tag{1.9}$$

1.4.3 Algorithm of Newton Polynomial Interpolation :

The algorithm of divided differences and Newton's interpolation polynomial, known as the « Horner's Algorithm », which uses the recurrence relation (1.9) :

Algorithm 1.4.1 Newton Polynomial Interpolation Algorithm

Require: Arrays $x(i)$ and $y(i)$, interpolation point z
Ensure: Approximated value $yp \approx f(z)$
$p \leftarrow$ number of points x_i
$n \leftarrow p - 1$ ▷ degree of the interpolation polynomial
Initialize table D
for $i \leftarrow 1$ **to** $n + 1$ **do**
    $D(i, 1) \leftarrow y(i)$
    for $j \leftarrow 1$ **to** $i - 1$ **do**
        $D(i, j+1) \leftarrow \dfrac{D(i, j) - D(i-1, j)}{x(i) - x(i-j)}$
    end for
end for
$diag(:) \leftarrow$ diagonal of $D(i, j)$
$yp \leftarrow y(1)$
for $i \leftarrow 1$ **to** n **do**
    $r \leftarrow 1$
    for $j \leftarrow 1$ **to** i **do**
        $r \leftarrow r \times (z - x(j))$
    end for
    $yp \leftarrow yp + r \times diag(i + 1)$
end for
return yp

Python Programme :

```python
import numpy as np

def newton_interpolation(xi, yi, z, n):
    p = len(xi)
    #----- Divided Difference Table----
    table = np.zeros((p, p))
    for i in range(p):
        table[i, 0] = yi[i]
        for j in range(1, i+1):
            table[i, j] = (table[i, j-1] - table[i-1, j-1]) / (xi[i
    ↪ ] - xi[i-j])
    coef = np.diag(table)
    #----evaluate Newton polynomial order (n) at z ----
    poly = yi[0]
    for i in range(1, n+1):
        r = 1
        for j in range(i):
            r *= (z - xi[j])
        poly += r * coef[i]
    return  table , poly
```

- **Example 1.4.2. :**

The time t required for a car to accelerate to a velocity v from an initial velocity of 8 m/s is given in the following table :

$v(m/s)$	8	11	15	20
$t(s)$	0.0	1.6	3.2	4.8

— *Estimate the time required for the car to reach a velocity of* 18 m/s *using a second-order Newton interpolation polynomial (ensure the highest possible accuracy).*

Solution :

The Newton polynomial of degree $n = 2$, with $x = v$ and $y = t$(see formula (1.8)) is

$$P_2(v) = t_0 + \delta^1 t_0\,(v - v_0) + \delta^2 t_0\,(v - v_0)(v - v_1).$$

For $n = 2$ we need only three points. The points must be ordered by increasing distance from the abscissa $v = 18$.

We therefore take the three abscissas $v = 20,\ 15,\ 11$. From the divided-difference table is :

i	v	t	δt	$\delta^2 t$
0	20	**4.8**		
			0.32	
1	15	3.2		**-0.0089**
			0.40	
2	11	1.6		

Divided-difference table of order 2 ($n = 2$)

Thus

$$P_2(v) = 4.8 + 0.32\,(v - 20) - 0.0089\,(v - 20)(v - 15),$$
$$P_2(18) = 4.8 + 0.32\,(18 - 20) - 0.0089\,(18 - 20)(18 - 15).$$

Numerical evaluation gives

$$t(18) \simeq P_2(18) = 4.2134\ s.$$

Rounding may produce slight variations (e.g. 4.2133 *or* 4.2128 *depending on the number of significant digits used in intermediate divided differences). The value above is consistent with the coefficients shown in the table to four decimal places (10^{-4}).*

1.4.4 Equidistant Points case

In the case where the interpolation points x_i are equally spaced with a distance h between two consecutive points x_i and x_{i+1}, h is called the « interpolation step » :

$$x_{i+1} = x_i + h.$$

a)- Newton's forward interpolation formula

The forward finite differences (denoted by the operator Δ_+) are defined as : $\Delta_+ y_i = y_{i+1} - y_i = f(x_{i+1}) - f(x_i)$

$$\begin{aligned}
&\textit{Order } 0: \quad \Delta_+^0 y_i = y_i \quad ; \quad i = 0, 1, .., n \\
&\textit{Order } 1: \quad \Delta_+^1 y_i = y_{i+1} - y_i \quad ; \quad i = 0, 1, .., n-1 \\
&\textit{Order } 2: \quad \Delta_+^2 y_i = \Delta_+^1 y_{i+1} - \Delta_+^1 y_i \quad ; \quad i = 0, 1, .., n-2 \\
&\qquad\qquad \vdots \\
&\textit{Order } k: \quad \Delta_+^k y_i = \Delta_+^{k-1} y_{i+1} - \Delta_+^{k-1} y_i \quad ; \quad i = 0, 1, .., n-k
\end{aligned}$$

The table of forward finite differences is :

x_i	y_i	$\Delta_+^1 y_i$	$\Delta_+^2 y_i$	$\Delta_+^3 y_i$	..	..	$\Delta_+^n y_i$
x_0	y_0						
		$\Delta_+ y_0$					
x_1	y_1		$\Delta_+^2 y_0$				
		$\Delta_+ y_1$		$\Delta_+^3 y_0$			
x_2	y_2		$\Delta_+^2 y_1$		..	..	
		$\Delta_+ y_2$			..	..	
x_3	y_3	.		.	..	..	
.	.	.	.	.	..	..	
.	.	.	.	.	..	..	$\Delta_+^n y_0$
.	.	.	.	.	..		
.	.	.	.	$\Delta_+^3 y_{n-3}$			
.	.	.	$\Delta_+^2 y_{n-2}$				
.	.	$\Delta_+ y_{n-1}$					
x_n	y_n						

Let $r = \frac{x-x_0}{h}$, then we obtain :

$$\begin{aligned} x - x_0 &= r\,h \\ x - x_1 &= x - (x_0 + h) = h(r-1) \\ x - x_2 &= x - (x_0 + 2h) = h(r-2) \\ &\vdots \\ x - x_{n-1} &= x - (x_0 + (n-1)h) = h(r-n+1) \end{aligned}$$

The forward Newton interpolation polynomial formula, according to equation (1.8), is :

$$\begin{aligned} P_n(x) &= y_0 + r\Delta_+ y_0 + \frac{r(r-1)}{2!}\Delta_+^2 y_0 + \frac{r(r-1)(r-2)}{3!}\Delta_+^3 y_0 \\ &\quad + \ldots + \frac{r(r-1)(r-2)\ldots(r-n+1)}{n!}\Delta_+^n y_0 \end{aligned} \tag{1.10}$$

In the forward **Gregory–Newton** formula, the interpolation points are taken to the right of the point under consideration. In this case, $r = \frac{x-x_0}{h} > 1$.

- **Example 1.4.3. :**

When measuring the flow velocity of water in a cylindrical pipe, the following data were obtained :

$t(s)$	0	10	20	30
$v(m/s)$	2.00	1.89	1.72	1.44

— *Find the velocity at $t = 5\,s$ using the Newton forward interpolation polynomial of degree 3.*

Solution :

We have $v_0 = 2.00$ m/s and we wish to evaluate at $t = 5$ s. Note that the step size is constant, $h = 10$ s. (We set $x = t$ and $y = v$.) The table of forward finite differences is :

i	t_i	v_i	$\Delta_+^1 v_i$	$\Delta_+^2 v_i$	$\Delta_+^3 v_i$
0	0	**2.00**			
			-0.11		
1	10	1.89		**-0.06**	
			-0.17		**-0.05**
2	20	1.72		-0.11	
			-0.28		
3	30	1.44			

We set : $r = \frac{t-t_0}{h} = \frac{5-0}{10} = 0.5$
According to the formula (1.10) :

$$\begin{aligned} v(t) &= P_3(t) \\ &= v_0 + r\Delta_+ v_0 + \frac{r(r-1)}{2!}\Delta_+^2 v_0 + \frac{r(r-1)(r-2)}{3!}\Delta_+^3 v_0 \\ v(5) &= 2.0 + (0.5)(-0.11) + \frac{(0.5)(0.5-1)}{2}(-0.06) + \frac{(0.5)(0.5-1)(0.5-2)}{6}(-0.05) \\ v(5) &= 2.0 - 0.055 + 0.0075 - 0.003125 = 1.949375 \end{aligned}$$

Thus, the flow velocity of the water at $t = 5$ s *is approximately* 1.949 m/s.

b)- Newton's Backward interpolation formula

We define the backward finite differences (denoted by the operator Δ_-) as :
$\Delta_- y_i = y_i - y_{i-1} = f(x_i) - f(x_i - h)$

$$\begin{aligned} \textit{Order } 0: \quad \Delta_-^0 y_i &= y_i && ; \quad i = 0, 1, .., n \\ \textit{Order } 1: \quad \Delta_-^1 y_i &= y_i - y_{i-1} && ; \quad i = 1, 2.., n \\ \textit{Order } 2: \quad \Delta_-^2 y_i &= \Delta_-^1 y_i - \Delta_-^1 y_{i-1} && ; \quad i = 2, 3.., n \\ &\vdots \\ \textit{Order } k: \quad \Delta_-^k y_i &= \Delta_-^{k-1} y_i - \Delta_-^{k-1} y_{i-1} && ; \quad i = k, .., n \end{aligned}$$

The table of backward finite differences is :

x_i	y_i	$\Delta_-^1 y_i$	$\Delta_-^2 y_i$	$\Delta_-^3 y_i$	..	..	$\Delta_-^n y_i$
x_0	y_0						
		$\Delta_- y_1$					
x_1	y_1		$\Delta_-^2 y_2$				
		$\Delta_- y_2$		$\Delta_-^3 y_3$			
x_2	y_2		$\Delta_-^2 y_3$		..	..	
		$\Delta_- y_3$			..	..	
x_3	y_3	.		.	..	..	
.	.	.	.	.	..	..	
.	.	.	.	.	..	..	$\Delta_+^n y_n$
.	.	.	.	.	..		
.	.	.	.	$\Delta_-^3 y_n$			
.	.	.	$\Delta_-^2 y_n$				
.	.	$\Delta_- y_n$					
x_n	y_n						

We set : $r = \frac{x-x_n}{h}$, thus we obtain :
The backward Newton interpolation polynomial, according to formula (1.8), is :

$$\begin{aligned} P_n(x) &= y_n + r\Delta_- y_n + \frac{r(r+1)}{2!}\Delta_-^2 y_n + \frac{r(r+1)(r+2)}{3!}\Delta_-^3 y_n \\ &\quad + \ldots + \frac{r(r+1)(r+2)\ldots(r+n-1)}{n!}\Delta_-^n y_n \end{aligned} \tag{1.11}$$

In the **Gregory–Newton** backward formula, the interpolation points are taken to the left of the considered point. In this case, $r = \frac{x-x_n}{h} < 1$.

- **Example 1.4.4.** *:*

Use the data from ***Example (1.4.3)*** *to find the velocity at $t = 25\,s$ using the Newton backward interpolation polynomial of degree 3.*

Solution :

We have $v_{(n=3)} = 1.44$ m/s at $t_{(n=3)} = 30$ s, and we wish to evaluate at $t = 25$ s.

Note that the step size is constant, $h = 10$ s. (We set $x = t$ and $y = v$.) The table of backward finite differences is :

i	t_i	v_i	$\Delta_-^1 v_i$	$\Delta_-^2 v_i$	$\Delta_-^3 v_i$
0	0	2.00			
			-0.11		
1	10	1.89		-0.06	
			-0.17		**-0.05**
2	20	1.72		**-0.11**	
$\rightarrow$			**-0.28**		
3	**30**	**1.44**			

We set : $r = \frac{t-t_3}{h} = \frac{25-30}{10} = -0.5$
According the formula (1.11) :

$$\begin{aligned} v(t) &= P_3(t) \\ &= v_3 + r\Delta_- v_3 + \frac{r(r+1)}{2!}\Delta_-^2 v_3 + \frac{r(r+1)(r+2)}{3!}\Delta_-^3 v_3 \\ v(25) &= 1.44 + (-0.5)(-0.28) + \frac{(-0.5)(-0.5+1)}{2}(-0.11) \\ &\quad + \frac{(-0.5)(-0.5+1)(-0.5+2)}{6}(-0.05) \\ v(25) &= 1.44 + 0.14 + 0.01375 + 0.003125 = 1.5969 \end{aligned}$$

Thus, the flow velocity of the water at $t = 25$ s is approximately 1.5969 m/s.

1.5 Exercises with solutions

1.5.1 Exercises

- **Exercise 1.1.** *:*

A small bead of mass m is released from rest at the origin of a vertical axis $(O, \overrightarrow{k})$ directed downward. The following table gives the measured distance traveled as a function of time :

t_i (s)	1	2	3
y_i (m)	5	20	45

1. Determine the Lagrange interpolation polynomial of degree 2.
2. Compute the distance y at $t = 1.5$ s.

Solution : 1.1

- **Exercise 1.2.** *:*

A projectile of mass m is launched from the origin O of a Cartesian frame $(O, \overrightarrow{i}, \overrightarrow{j})$ with an initial velocity $\overrightarrow{v_0}$ making an angle α with the horizontal axis ox. The projectile passes through the following positions :

x_i (m)	1	1.5	2.5	4.4
y_i (m)	1.532	2.148	3.080	3.749

1. Construct the table of divided differences of order 3.
2. Find the Newton interpolation polynomial of degree 2.
3. Compute an approximate value of the ordinate y corresponding to $x = 1.3$.
4. Deduce the value of the launch **angle** α and the **initial speed** v_0, given that the projectile trajectory in a uniform gravitational field (with $g = 10\ \mathrm{m\,s^{-2}}$) is described by :

$$y(x) = -\frac{g}{2v_0^2 \cos^2 \alpha}\, x^2 + x \tan \alpha.$$

Solution : 1.2

- **Exercise 1.3.** *:*

The measured voltage across a dipole yielded the following values :

t [s]	0	3	4
U [V]	-8	4	0

It is assumed that the voltage varies slowly enough to be well approximated by a low-degree polynomial.

Estimate, from these data, the time t at which the voltage reaches its **maximum**, and the voltage U at that maximum.

- **Exercise 1.4.** *:*

The following table gives the heat capacity of *methylcyclohexane* as a function of temperature [2] :

T [K]	150	160	170	180	190	210	230	250	270
C_p [kJ kg^{-1} K^{-1}]	1.426	1.447	1.469	1.492	1.516	1.567	1.627	1.696	1.770

1. Estimate the heat capacity of methylcyclohexane at 179 K using a Lagrange polynomial of degree 1.
2. Estimate the heat capacity of methylcyclohexane at 179 K using a Lagrange polynomial of degree 2 passing through the first three points of the table.
3. Compare the results obtained in (1) and (2). Comment.

Answer : $C_p^{(1)}(179) = 1.4897, \qquad C_p^{(2)}(179) = 1.489655.$

- **Exercise 1.5.** *:*

The following table gives the thermal conductivity of acetone vapor as a function of temperature [2] :

T (°F)	k (Btu/(hr ft °F))
32	0.0057
115	0.0074
212	0.0099
363	0.0147

1. Estimate the thermal conductivity of acetone at 300 °F using the Lagrange polynomial of degree 2 passing through the last three points of the table.
2. Estimate the temperature in °F that corresponds to the thermal conductivity $k = 0.008$ Btu/(hr ft °F) using a Lagrange polynomial of degree 2 passing through the last three points of the table.

Answer : $k(300) = P_2(300) = 1.2563 \times 10^{-2}$ Btu/(hr ft °F);
$T(0.008) = P_2^{-1}(0.008) = 1.3943 \times 10^{-2}$ °F.

1.5.2 Solutions

- **solution 1.1.** *:*

1) The Lagrange interpolation polynomial of degree 2 is

$$P_2(t) = L_0(t) + L_1(t) + L_2(t),$$

with

$$L_0(t) = 5\frac{(t-2)(t-3)}{(1-2)(1-3)} = 2.5\,(t^2 - 5t + 6),$$

$$L_1(t) = 20\frac{(t-1)(t-3)}{(2-1)(2-3)} = -20\,(t^2 - 4t + 3),$$

$$L_2(t) = 45\frac{(t-1)(t-2)}{(3-1)(3-2)} = 22.5\,(t^2 - 3t + 2).$$

Summing the three terms yields

$$P_2(t) = 5t^2.$$

2) Therefore

$$y(1.5) = P_2(1.5) = 5\,(1.5)^2 = 5 \times 2.25 = 11.25 \text{ m}.$$

Interpretation : The interpolating polynomial $P_2(t) = 5t^2$ coincides exactly with the theoretical law of free fall under constant acceleration $g = 10$ m/s^2, starting from rest :

$$y(t) = \tfrac{1}{2}gt^2.$$

Thus, the interpolation recovers the physical equation of motion of the falling object, confirming that the experimental data are consistent with uniform gravitational acceleration.

Exercise 1.1

- **solution 1.2.** *:*

1. Table of divided differences of order 3 :

   The divided differences are defined by $\delta^n y_i = \dfrac{\delta^{n-1}y_{i+1} - \delta^{n-1}y_i}{x_{i+n} - x_i}$ for $n = 1, 2, 3$.

i	x_i	y_i	$\delta^1 y_i$	$\delta^2 y_i$	$\delta^3 y_i$
0	1.0000	**1.5320**			
1	1.5000	2.1480	**1.2320**		
2	2.5000	3.0800	0.9320	**-0.2000**	
3	4.4000	3.7490	0.3521	-0.2000	0.0000

2. Newton polynomial of degree 2 ($y = P_2(x)$) :

$$P_2(x) = y_0 + \delta^1 y_0\,(x - x_0) + \delta^2 y_0\,(x - x_0)(x - x_1).$$

Substituting the numeric values gives

$$P_2(x) = 1.5320 + 1.232\,(x - 1.0) + (-0.2)\,(x - 1.0)(x - 1.5).$$

Expanding,

$$P_2(x) = -0.2\,x^2 + 1.732\,x.$$

3. For $x = 1.3$ we obtain

$$y(1.3) = P_2(1.3) = -0.2\,(1.3)^2 + 1.732\,(1.3) \approx 1.9136 \text{ m}.$$

4. Determination of α and v_0 :
Compare term-by-term the fitted quadratic

$$y(x) = -0.2\,x^2 + 1.732\,x$$

with the theoretical projectile trajectory

$$y(x) = -\frac{g}{2v_0^2 \cos^2\alpha}\,x^2 + x\,\tan\alpha,$$

with $g = 10\ \text{m}\,\text{s}^{-2}$. This yields the system

$$\begin{cases} \tan\alpha = 1.732, \\ -\dfrac{g}{2v_0^2\cos^2\alpha} = -0.2. \end{cases}$$

Hence

$$\alpha = \arctan(1.732) = 60^\circ,$$

and from the second equation

$$\frac{g}{2v_0^2\cos^2\alpha} = 0.2 \quad \Rightarrow \quad v_0^2 = \frac{g}{2\cdot 0.2\cos^2\alpha} = \frac{10}{0.4\cos^2 60^\circ}.$$

Since $\cos 60^\circ = 0.5$ (so $\cos^2 60^\circ = 0.25$),

$$v_0^2 = \frac{10}{0.4\times 0.25} = \frac{10}{0.1} = 100,$$

thus

$$v_0 = 10\ \mathrm{m/s}.$$

Exercise 1.2

Chapitre 2

Best Approximation and Least Squares Sense

2.1 Introduction

The method of least squares is one of the most fundamental tools in error theory and estimation. Its purpose is to determine the *best-fitting curve* (or line) for a given set of experimental data points, which are typically affected by measurement errors. The principle is to minimize the sum of the Squared deviations (residuals) between the observed values and the values predicted by the curve. In this way, the influence of experimental uncertainties is reduced by « adding information » through the fitting process.

This procedure is closely related to *regression analysis*, where the quantitative relationship between two or more variables is modeled. Curve fitting, as a special case of regression, provides approximate equations that represent the underlying trend in raw data. Since the fitting of a curve to a given data set is not necessarily unique, the least squares method ensures the selection of the curve with the minimal overall deviation from the measurements, thereby yielding the optimal or « best » approximation.

Unlike probabilistic approaches, the least squares method is purely algebraic. It was independently formulated by **L**egendre and **G**auss at the beginning of the 19^{th} century and has since become one of the most widely used techniques in all observational sciences ([6]).

2.2 Polynomial Best Approximation and Least Squares Sense

When a function f is known only at certain experimental points x_i, it is often desirable to estimate its values for arbitrary x. To achieve this, the function f is replaced by a simpler function, whose evaluation is more straightforward, such as a polynomial or a trigonometric function. In other words, the experimental data points (x_i, y_i) are fitted by a function $f(x)$ using the method of *polynomial approximation in the least squares sense*.

2.2.1 Discrete Case

For a given set of experimental data (or measurements) (x_i, y_i),$i = 1, \ldots, n$, the polynomial approximation in the least squares sense aims to determine a polynomial of degree m :$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m$ that minimizes the following functional J :

$$J = \sum_{i=1}^{n} \big(P(x_i) - y_i\big)^2. \tag{2.1}$$

The minimum is attained when the partial derivatives with respect to the coefficients a_k, for $k = 0, 1, \ldots, m$, vanish :

$$\frac{\partial J}{\partial a_k} = 2 \sum_{i=1}^{n} \left(a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m - y_i\right) x_i^k = 0.$$

That is,

$$\begin{aligned} \sum_{i=1}^{n} \left(a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m - y_i\right) x_i^k &= 0, \\ a_0 \sum_{i=1}^{n} x_i^k + a_1 \sum_{i=1}^{n} x_i^{k+1} + a_2 \sum_{i=1}^{n} x_i^{k+2} + \cdots + a_m \sum_{i=1}^{n} x_i^{k+m} &= \sum_{i=1}^{n} y_i x_i^k. \end{aligned}$$

Let us denote

$$S_p = \sum_{i=1}^{n} x_i^p, \quad v_k = \sum_{i=1}^{n} y_i x_i^k$$

Then, the system (2.2.1) is equivalent to the following linear system :

$$\begin{bmatrix} S_0 & S_1 & \cdots & S_m \\ S_1 & S_2 & \cdots & S_{m+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_m & S_{m+1} & \cdots & S_{2m} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_m \end{bmatrix}. \tag{2.2}$$

Solving this system of linear equations yields the coefficients $a = (a_0, a_1, \ldots, a_m)$ of the least squares polynomial :

$$P_m(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_m x_i^m \tag{2.3}$$

Residuals and Standard Deviation in Polynomial Least Squares Fit

For a polynomial of degree m fitted to data points (x_i, y_i), $i = 1, \ldots, n$, the fitted values are : $\hat{y}_i = P_m(x_i)$

Residuals : The residuals are the differences between the observed values y_i and the fitted values $\hat{y}_i$ ($i = 1, \ldots, n$.) :

$$e_i = y_i - \hat{y}_i \tag{2.4}$$

Sum of Squared Residuals (SSE) : The sum of squared residuals is :

$$\begin{aligned} \text{SSE} &= \sum_{i=1}^{n} e_i^2 \\ &= \sum_{i=1}^{n} \left(y_i - P_m(x_i)\right)^2 \end{aligned} \tag{2.5}$$

Residual Variance and Standard Deviation : An unbiased estimate of the residual variance is :

$$\hat{\sigma}^2 = \frac{\text{SSE}}{n - (m+1)} \tag{2.6}$$

where $m + 1$ is the number of polynomial coefficients. The standard deviation of the residuals is

$$\hat{\sigma} = \sqrt{\hat{\sigma}^2} \tag{2.7}$$

Coefficient of Determination R^2 : The coefficient of determination measures the proportion of the variance in the dependent variable explained by the model :

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}}. \tag{2.8}$$

where The total Sum of Squares :

$$\mathrm{SST} = \sum_{i=1}^{n} (y_i - \bar{y})^2 \tag{2.9}$$

and the mean of y_i for n points is :

$$\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i \tag{2.10}$$

Python Programme : Linear fit(with simple program)

```
# Linear fit(least squares): P(x)=a*x +b
# sgma_r  Standard deviation of residuals
#
import numpy as np

def moindres_carres(xi, yi):
    n = len(xi)
    sx = np.sum(xi)
    sxc = np.sum(xi ** 2)
    sy = np.sum(yi)
    sxy = np.sum(xi * yi)
    a = (n * sxy - sx * sy) / (n * sxc - sx ** 2)
    b = (sxc * sy - sx * sxy) / (n * sxc - sx ** 2)
    sgma_r = np.sqrt(1 / (n - 2) * np.sum((yi - a * xi - b) ** 2))
    return a, b, sgma_r
```

Python Programme :Polynomial fitting (least squares) with order n

```
import numpy as np
# Polynomial fitting (least squares)
# to find the coefficients of a polynomial of order n
coef = np.polyfit(xi, yi, n)
```

Physical Interpretation.

— Each residual e_i represents the deviation of an experimental measurement from the polynomial approximation.

— The standard deviation $\hat{\sigma}$ indicates the typical magnitude of these deviations in the same units as y.

— A smaller $\hat{\sigma}$ implies that the polynomial closely represents the experimental data, providing a quantitative measure of the goodness of fit.

— It can also be used to assess the uncertainty in predicting y values for new x inputs using the fitted polynomial.

— A value of R^2 close to 1 indicates a strong agreement between the polynomial model and the data.

- **Example 2.2.1.** *:*

Fit the given points using the least squares method with a function of the form $y = a_0 + a_1 x$ *and compute the sum of squared errors (SSE) and the root-mean-square residual.*

x_i	y_i
0.53	*5.19*
0.62	*5.49*
1.13	*6.88*
1.36	*7.41*
1.68	*8.20*

We have $n = 5$ *points :* (x_i, y_i); $i = 1, 2, \ldots, 5$
The least squares expression is given by : $J = \sum_{i=1}^{i=5}(y_i - a_0 - a_1 x_i)^2$

$$\begin{cases} \frac{\partial J}{\partial a_0} = 0 \Leftrightarrow -2\sum(y_i - a_0 - a_1 x_i) = 0 \\ \frac{\partial J}{\partial a_1} = 0 \Leftrightarrow -2\sum[x_i(y_i - a_0 - a_1 x_i)] = 0 \end{cases}$$

$$\begin{cases} n a_0 + a_1 \sum x_i = \sum y_i \\ a_0 \sum x_i + a_1 \sum x_i^2 = \sum x_i y_i \end{cases}$$

$$a_1 = \frac{\sum x_i \sum y_i - n \sum x_i y_i}{(\sum x_i)^2 - n \sum x_i^2} \quad ; \quad a_0 = \frac{\sum y_i - a_1 \sum x_i}{n}$$

	x_i	y_i	x^2	$x_i y_i$
	0.53	*5.19*	*0.2809*	*2.7507*
	0.62	*5.49*	*0.3844*	*3.4038*
	1.13	*6.88*	*1.2769*	*7.7744*
	1.36	*7.41*	*1.8496*	*10.0776*
	1.68	*8.20*	*2.8224*	*13.776*
Sum :	*5.32*	*33.17*	*6.6142*	*37.7825*
mean=sum/5		$\bar{y}$*=6.634*		

$$\begin{cases} 5a_0 + 5.32a_1 = 33.17, & (1) \\ 5.32a_0 + 6.6142a_1 = 37.7825, & (2) \end{cases} \Leftrightarrow \begin{cases} a_0 = 3.8565 \\ a_1 = 2.6104 \end{cases}$$

The fitted linear model is :

$$\hat{y}(x) = a_0 + a_1 x = 3.8565 + 2.6104\, x.$$

The residuals are :

$$e_i = y_i - \hat{y}(x_i).$$

Numerically,

i	x_i	y_i	$\hat{y}(x_i) = P_1(x_i)$	$e_i = y_i - \hat{y}(x_i)$	e_i^2
1	0.53	5.19	5.2400	-0.0500	$2.50E-03$
2	0.62	5.49	5.4749	0.0151	$2.28E-04$
3	1.13	6.88	6.8062	0.0738	$5.45E-03$
4	1.36	7.41	7.4066	0.0034	$1.16E-05$
5	1.68	8.20	8.2420	-0.0420	$1.16E-05$
sum	//	//	//	//	$9.95E-03$

The sum of squared residuals (SSE) is

$$\text{SSE} = \sum_{i=1}^{5} e_i^2 \approx 9.95 \times 10^{-3}.$$

An unbiased estimator of the residual variance uses the degrees of freedom $n-2$ (two parameters estimated : a_0, a_1) :

$$\hat{\sigma}^2 = \frac{\text{SSE}}{n-2} \approx \frac{9.95 \times 10^{-3}}{3} \approx 3.32 \times 10^{-3}.$$

Thus the standard error (root-mean-square residual) is

$$\hat{\sigma} = \sqrt{\hat{\sigma}^2} \approx 5.76 \times 10^{-2}.$$

A measure of goodness-of-fit is the coefficient of determination R^2 :

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}},$$

where The total Sum of Squares $\text{SST} = \sum_{i=1}^{n}(y_i - \bar{y})^2$ *the mean of y_i for $n = 5$ points is :* $\bar{y} = \sum_{i=1}^{n} y_i$

For these data SST ≈ 6.5089, *hence*

$$R^2 \approx 1 - \frac{9.95 \times 10^{-3}}{6.5089} \approx 0.9985.$$

Physical interpretation :

— *The standard error* $\hat{\sigma} \approx 0.057$ *(in the same units as* y*) measures the typical deviation of measured values from the fitted line.*

— *A small* $\hat{\sigma}$ *and a value* R^2 *close to 1 (here* $R^2 \approx 0.9985$*) indicate that the linear model explains nearly all the variance in the data : the fit is very good.*

— *Physically,* $\hat{\sigma}$ *can be interpreted as the combination of measurement noise and modelling error (i.e., deviation of the true relation from a perfect straight line).*

2.2.2 Continuous Case

Let f be a continuous function on the interval I. We seek a least-squares approximation polynomial $P_n(x)$ of degree less than or equal to n, such that the following functional J_c is minimized :

$$J_c = \int_I \big(P(x) - f(x)\big)^2 \, dx \tag{2.11}$$

The minimum is attained when the partial derivative with respect to the polynomial coefficients a_k, for $k = 0, 1, \ldots, m$, vanishes.

- **Example 2.2.2. :**

Use the least-squares method to find the best approximation of the function $f(x) = e^x$ *by a quadratic polynomial on the interval* $I = [0, 1]$.

Given : $\int xe^x \, dx = (x-1)e^x$, $\int x^2 e^x \, dx = (x^2 - 2x + 2)e^x$

We want to approximate the function $f(x) = e^x$ *on the interval* $I = [0, 1]$ *by the quadratic polynomial :* $P_2(x) = a_0 + a_1 x + a_2 x^2$

$$J_c = \int_I (P(x) - f(x))^2 \, dx = \int_0^1 (a_0 + a_1 x + a_2 x^2 - e^x)^2 dx$$

Let us set the partial derivatives equal to zero :

$$\begin{cases} \frac{\partial J_c}{\partial a_0} = 0 \\ \frac{\partial J_c}{\partial a_1} = 0 \\ \frac{\partial J_c}{\partial a_2} = 0 \end{cases} \Leftrightarrow \begin{cases} -2\int_0^1 (a_0 + a_1 x + a_2 x^2 - e^x) dx = 0 \\ -2\int_0^1 [x(a_0 + a_1 x + a_2 x^2 - e^x)] dx = 0 \\ -2\int_0^1 [x^2(a_0 + a_1 x + a_2 x^2 - e^x)] dx = 0 \end{cases}$$

$$\Leftrightarrow \begin{cases} \int_0^1 (a_0 + a_1 x + a_2 x^2) dx = \int_0^1 e^x dx \\ \int_0^1 (a_0 x + a_1 x^2 + a_2 x^3) dx = \int_0^1 x e^x dx \\ \int_0^1 (a_0 x^2 + a_1 x^3 + a_2 x^4) dx = \int_0^1 x^2 e^x dx = 0 \end{cases}$$

$$\Leftrightarrow \begin{cases} a_0 + \frac{1}{2} a_1 + \frac{1}{3} a_2 = 1.7183 \\ \frac{1}{2} a_0 + \frac{1}{3} a_1 + \frac{1}{4} a_2 = 1 \\ \frac{1}{3} a_0 + \frac{1}{4} a_1 + \frac{1}{5} a_2 = 0.7183 \end{cases}$$

System of linear equations in matrix form : $Mx = b$

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1.7183 \\ 1 \\ 0.7183 \end{bmatrix}$$

This system of linear equations can be solved using one of the methods discussed in Chapter 3. From the three equations above, we obtain :

$$\Leftrightarrow \begin{cases} a_0 = 1.013 \\ a_1 = 0.8511 \\ a_2 = 0.8392 \end{cases}$$

Therefore, on the interval $I = [0, 1]$ *we can write :*

$$e^x = 1.013 + 0.8511x + 0.8392x^2$$

2.3 Best Trigonometric Approximation of a Periodic Function

A trigonometric series is defined as a series of functions $\sum u(x)$ whose general term $u(x)$ has the form :

$$\begin{cases} u_0(x) = \frac{1}{2} a_0, & \text{if } n = 0, \\ u_n(x) = a_n \cos(nx) + b_n \sin(nx), & \text{otherwise.} \end{cases} \tag{2.12}$$

The trigonometric series of a function f is given by the partial sums of the trigonometric series (2.12) in the form :

$$S_p(f(x)) = \tfrac{1}{2}a_0 + \sum_{k=1}^{p} \big(a_k \cos(kx) + b_k \sin(kx)\big), \tag{2.13}$$

where we set $b_0 = 0$.

For a 2π-periodic function, the Fourier coefficients of f are the real numbers a_n and b_n, defined by :

$$\begin{aligned} a_k &= \frac{1}{\pi}\int_0^{2\pi} f(x)\cos(kx)\,dx, \\ b_k &= \frac{1}{\pi}\int_0^{2\pi} f(x)\sin(kx)\,dx. \end{aligned} \tag{2.14}$$

Another representation of trigonometric series is the complex exponential form, known as the *Fourier series* :

$$S_n(f(x)) = \sum_{k=-n}^{n} c_k e^{ikx}, \tag{2.15}$$

where the Fourier coefficients are defined by :

$$c_k = \frac{1}{2\pi}\int_0^{2\pi} f(x)e^{ikx}\,dx. \tag{2.16}$$

In practice, the integral (2.16) is not always computable ; therefore, we introduce the discrete Fourier transform :

$$c_k = \frac{1}{2n+1}\sum_{j=0}^{2n} f(x_j)e^{ikx_j}, \qquad k = -n,\ldots,n, \tag{2.17}$$

with $x_j = \frac{2\pi j}{2n+1}$.

We observe that the approximation (2.15) is an interpolation, since the relation

$$\frac{1}{2n+1}\sum_{j=0}^{2n} e^{ikx_j} = \begin{cases} 1, & \text{if } k \in (2n+1)\mathbb{Z}, \\ 0, & \text{otherwise.} \end{cases} \tag{2.18}$$

implies that $S_n(f(x_j)) = f(x_j)$ for $j = 0, 2, \ldots, 2n$.

2.4 Exercises with solutions

2.4.1 Exercises

- **Exercise 2.1.** :

The table below shows the variation of the voltage across a charged capacitor of capacitance C during discharge through an ohmic conductor of resistance $R = 100\ \Omega$:

t(s)	2	4	6	8
U (volt)	6.032	4.043	2.710	1.817

The theoretical expression for the voltage across the capacitor during discharge is : $U(t) = U_0\, e^{-\frac{t}{RC}}$.

a)- Fit the given data points by the least-squares method using a function of the form $U(t) = A\, e^{Bt}$.

b)- Deduce the values of the capacitance C and of U_0.

solution : (2.1)

- **Exercise 2.2.** :

In the laboratory we perform the simple pendulum experiment with very small oscillations. We vary the pendulum length, denoted L, and record the period of oscillation, denoted T. The measurement results are given in the table below :

$L_i(m)$	1.00	1.50	1.75	2.00	2.50
$T_i(s)$	2.006	2.457	2.654	2.837	3.172

The theoretical relation between T and L has the form $T = aL^b$, where a and b are constants.

1. Find the values of a and b by performing a least-squares fit of the data to the function $T = aL^b$.
2. Given $a = \dfrac{2\pi}{\sqrt{g}}$, calculate the value of the gravitational acceleration g in the laboratory.

Answer : $a = 2.006,\ b = 0.501\,;\ g = 9.81$

- **Exercise 2.3.** :

Let $f(x) = \sin(\frac{1}{3}x)$ for $x \in I = [0, 1]$. Find the best degree-1 polynomial (in the least-squares sense) approximating $f(x)$ on the interval I (give results to three decimal places).

Answer : $0.306x + 0.012$

- **Exercise 2.4.** *:*

Develop in a Fourier series the function given on a half-period on the segment $[0,2]$ by $f(x) = x - \frac{1}{2}x^2$, extending to $[-2,0]$ as an odd function.

Answer *:* $f(x) = \sum_{k=1}^{\infty} \frac{8}{\pi^3 k^3}\left[1-(-1)^k\right]\sin\left(\frac{1}{2}k\pi x\right)$

2.4.2 Solution

- **solution 2.1.** *:*

a) Least-squares fit :

We consider the approximation function $U(t) = A\,e^{Bt}$.

$\ln(U) = \ln(A) + Bt$.

Set $y = \ln(U)$, $a_0 = \ln(A)$ and $a_1 = B$.

The least-squares expression is $S = \sum_{i=1}^{4}(y_i - a_0 - a_1 t_i)^2$.

$$\begin{cases} \dfrac{\partial S}{\partial a_0} = 0 \\ \dfrac{\partial S}{\partial a_1} = 0 \end{cases} \iff \begin{cases} -2\sum(y_i - a_0 - a_1 t_i) = 0, \\ -2\sum[t_i(y_i - a_0 - a_1 t_i)] = 0. \end{cases}$$

$$\iff \begin{cases} 4a_0 + a_1 \sum t_i = \sum y_i, \\ a_0 \sum t_i + a_1 \sum t_i^2 = \sum t_i y_i. \end{cases}$$

$$a_1 = \frac{\sum t_i \sum y_i - 4\sum t_i y_i}{(\sum t_i)^2 - 4\sum t_i^2}, \quad ; \quad a_0 = \frac{\sum y_i - a_1 \sum t_i}{4}.$$

	t_i	U	$y_i = \ln(U)$	t_i^2	$t_i y_i$
	2	6.032	1.797	4	3.594
	4	4.043	1.396	16	5.584
	6	2.71	0.996	36	5.976
	8	1.817	0.597	64	4.776
Sum :	20	/	4.786	120	19.93

$$\begin{aligned} a_1 &= \frac{20 \cdot 4.786 - 4 \cdot 19.93}{(20)^2 - 4 \cdot 120} = -0.2, \\ a_0 &= \frac{4.786 - (-0.2) \cdot 20}{4} = 2.1972. \end{aligned}$$

$$A = e^{a_0} = 9, \quad B = a_1 = -0.2.$$

Finally : $U(t) = 9\,e^{-0.2t}$.

b) The values of the capacitance C and U_0.

By comparing the obtained equation with $U(t) = U_0\,e^{-\frac{t}{RC}}$, we deduce : $U_0 = 9$ V and $C = 0.05$ F.

exercise :2.1

Chapitre 3

Solution of Systems of Linear Equations

3.1 Introduction

Systems of linear equations frequently arise in physics problems, such as the solution of the one-dimensional heat diffusion equation, or the fitting of experimental data using the method of least squares (see Chapter 2).

For systems of linear equations of order greater than 3 (i.e., three equations with three unknowns), one should not attempt to solve them manually by elimination, as is typically done with small systems. Instead, it is more practical to use numerical methods with the aid of a computer.

Numerical stability issues quickly become critical, and therefore it is advisable to rely on numerical library subroutines provided with the compiler in use. These subroutines are pre-written, extensively tested, and compiled programs whose strengths and weaknesses are well known, and which are assembled in the compiler's library.

The methods for solving systems of linear equations can be classified into two categories :

— **Direct methods :** These yield the exact solution in a finite number of operations. Examples include **G**aussian elimination, **G**auss–**J**ordan elimination, LU decomposition, and the **C**holesky method.

— **Iterative methods :** These construct a sequence of vectors X^k that converges to the desired solution X. The iteration is stopped after a finite number n of steps, chosen such that X^k is sufficiently close to X. Examples include the **J**acobi method, the **G**auss–**S**eidel method, and relaxation methods.

In this chapter, we present only the **G**aussian elimination and **G**auss–**J**ordan methods (direct methods), as well as the **J**acobi and **G**auss–**S**eidel methods (iterative methods).

3.2 Linear Systems of Equations

A linear system of order n ($n \in \mathbb{N}^*$) can be written as follows :

$$\begin{cases} a_{11}x_1 + a_{12}\,x_2 + ... + a_{1n}\,x_n = b_1 \\ a_{21}x_1 + a_{22}\,x_2 + ... + a_{2n}\,x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}\,x_2 + ... + a_{nn}\,x_n = b_n \end{cases} \tag{3.1}$$

where the unknowns x_i are to be determined.
The problem can be reformulated in matrix terms as follows :

$$Ax = b \tag{3.2}$$

where $A = (a_{ij})$ is an $n \times n$ square matrix, $b = (b_i)$ is the column vector, and $x = (x_i)$ is the column vector of the unknowns of the system.

3.3 Direct Methods

3.3.1 Gaussian Elimination Method

The purpose of the Gaussian elimination method is to transform the system $Ax = b$ into an equivalent system of the form $A^{(n)}x = b^{(n)}$, where $A^{(n)}$ is an upper triangular matrix derived from $A = A^{(1)}$, and $b^{(n)}$ is a suitably modified right-hand side.

Let A be a square matrix of order $n = 4$. The corresponding system can be written as :

$$\begin{cases} a_{11}x_1 + a_{12}\,x_2 + a_{13}x_3 + a_{14}\,x_4 = b_1 \\ a_{21}x_1 + a_{22}\,x_2 + a_{23}x_3 + a_{24}\,x_4 = b_2 \\ a_{31}x_1 + a_{32}\,x_2 + a_{33}x_3 + a_{34}\,x_4 = b_3 \\ a_{41}x_1 + a_{42}\,x_2 + a_{43}x_3 + a_{44}\,x_4 = b_4 \end{cases} \tag{3.3}$$

The Gaussian elimination method : $(A^{(1)}, b^{(1)}) \underrightarrow{\text{transformation}} (A^{(n)}, b^{(n)})$

— **Step 1 :** The system can be written in matrix form (without modifying the index (1)) :

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} & a_{24}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} & a_{34}^{(1)} \\ a_{41}^{(1)} & a_{42}^{(1)} & a_{43}^{(1)} & a_{44}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1^{(1)} & \cdots & (L_1^{(1)}) \\ b_2^{(1)} & \cdots & (L_2^{(1)}) \\ b_3^{(1)} & \cdots & (L_3^{(1)}) \\ b_4^{(1)} & \cdots & (L_4^{(1)}) \end{bmatrix}$$

— **Step 2 :** If $a_{11}^{(1)} \neq 0$ (otherwise we perform a row permutation), we create zeros in the first column below the main diagonal :

$$\begin{array}{l} Pivot(1) : Pi(1) = a_{11}^{(1)} \\ L_2^{(2)} \leftarrow L_2^{(1)} - L_1^{(1)} \frac{a_{21}^{(1)}}{Pi(1)} \\ L_3^{(2)} \leftarrow L_3^{(1)} - L_1^{(1)} \frac{a_{31}^{(1)}}{Pi(1)} \\ L_4^{(2)} \leftarrow L_4^{(1)} - L_1^{(1)} \frac{a_{41}^{(1)}}{Pi(1)} \end{array} \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & a_{24}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} & a_{34}^{(2)} \\ 0 & a_{42}^{(2)} & a_{43}^{(2)} & a_{44}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1^{(1)} & \cdots & (L_1^{(1)}) \\ b_2^{(2)} & \cdots & (L_2^{(2)}) \\ b_3^{(2)} & \cdots & (L_3^{(2)}) \\ b_4^{(2)} & \cdots & (L_4^{(2)}) \end{bmatrix}$$

— **Step 3 :** If $a_{22}^{(2)} \neq 0$ (otherwise we perform a row permutation), we create zeros in the second column below the main diagonal :

$$\begin{array}{l} Pivot(2) : Pi(2) = a_{22}^{(2)} \\ L_3^{(3)} \leftarrow L_3^{(2)} - L_2^{(2)} \frac{a_{32}^{(2)}}{Pi(2)} \\ L_4^{(3)} \leftarrow L_4^{(2)} - L_2^{(2)} \frac{a_{42}^{(2)}}{Pi(2)} \end{array} \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & a_{24}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & a_{34}^{(3)} \\ 0 & 0 & a_{43}^{(3)} & a_{44}^{(3)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1^{(1)} & \cdots & (L_1^{(1)}) \\ b_2^{(2)} & \cdots & (L_2^{(2)}) \\ b_3^{(3)} & \cdots & (L_3^{(3)}) \\ b_4^{(3)} & \cdots & (L_4^{(3)}) \end{bmatrix}$$

— **Step 4 :** If $a_{33}^{(3)} \neq 0$ (otherwise we perform a row permutation), we create zeros in the third column below the main diagonal :

$$\begin{array}{l} Pivot(3) : Pi(3) = a_{33}^{(3)} \\ L_4^{(4)} \leftarrow L_4^{(3)} - L_3^{(3)} \frac{a_{43}^{(3)}}{Pi(3)} \end{array} \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & a_{24}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & a_{34}^{(3)} \\ 0 & 0 & 0 & a_{44}^{(4)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1^{(1)} & \cdots & (L_1^{(1)}) \\ b_2^{(2)} & \cdots & (L_2^{(2)}) \\ b_3^{(3)} & \cdots & (L_3^{(3)}) \\ b_4^{(4)} & \cdots & (L_4^{(4)}) \end{bmatrix} \tag{3.4}$$

The final solution is obtained : by solving the upper triangular system (3.4) using the backward substitution method, that is, we first determine x_4, then x_3, x_2,

and finally x_1 ; hence the relations :

$$\begin{cases} x_4 = b_4^{(4)} / a_{44}^{(4)} \\ x_3 = (b_3^{(3)} - a_{34}^{(3)}\, x_4) / a_{33}^{(3)} \\ x_2 = (b_2^{(2)} - a_{23}^{(2)}\, x_3 - a_{24}^{(2)}\, x_4) / a_{22}^{(2)} \\ x_1 = (b_4^{(1)} - a_{12}^{(1)}\, x_2 - a_{13}^{(1)}\, x_3 - a_{14}^{(1)}\, x_4) / a_{11}^{(1)} \end{cases} \tag{3.5}$$

Remark 3.3.1. *:*

1. *To avoid the propagation of rounding errors and to ensure the stability of computations, the* ***largest pivot*** *in absolute value must be chosen. Therefore, the pivot is selected by row permutation (rearrangement of the rows), which is called the Gauss method with pivoting.*
2. *The Gauss method without row permutation is called the "****ordinary Gauss method or method without pivoting****."*
3. *The Gauss method also makes it possible to compute the determinant of the matrix A as :*

$$dét(A) = (-1)^p \prod_{i=1}^{n} a_{ii}^{(i)} \tag{3.6}$$

where p is the number of row permutations.

3.3.2 Algorithm of the Gaussian Elimination Method

To transform $(A^{(1)}; b^{(1)})$ into $(A^{(n)}; b^{(n)})$ using the Gaussian elimination method without row permutations (ordinary Gauss method), we apply the following formulas :

1. To obtain the upper triangular matrix and the modified right-hand side :

$$\begin{cases} a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} a_{kj}^{(k)} \\ b_i^{(k+1)} = b_i^{(k)} - \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} b_k^{(k)} \end{cases} \tag{3.7}$$

   where $1 \leq k \leq (n-1)$, $(k+1) \leq i \leq n$, and $1 \leq j \leq n$.

2. To find the solution from the upper triangular matrix :

$$\begin{cases} x_n = \frac{b_n^{(n)}}{a_{nn}^{(n)}} \\ x_i = \frac{1}{a_{ii}} (b_i^{(i)} - \sum_{j=i+1}^{n} a_{ij}^{(i)} x_j) \end{cases} \tag{3.8}$$

where $i = n-1, ..., 1$ et $j = i+1, ..., n$

These relations can be expressed in algorithmic form (3.3.1) :

Algorithm 3.3.1 Gaussian elimination method for solving a linear system of order $n : Ax = b$

Require: Dimension n, matrix $A(n,n)$, vector $b(n)$
Ensure: Upper triangular matrix A, solution vector $x(n)$

```
1:  procedure GAUSSIAN_ELIMINATION(A, b, n)
2:      for k ← 1 to n − 1 do
3:          for i ← k + 1 to n do
4:              t ← A(i,k) / A(k,k)
5:              b(i) ← b(i) − t × b(k)
6:              for j ← 1 to n do
7:                  A(i,j) ← A(i,j) − t × A(k,j)
8:              end for
9:          end for
10:     end for                                   ▷ Back substitution
11:     for i ← n downto 1 do
12:         S ← 0                                 ▷ initialize sum
13:         for j ← i + 1 to n do
14:             S ← S + x(j) × A(i,j)
15:         end for
16:         x(i) ← (b(i) − S) / A(i,i)
17:     end for
18:     return x
19: end procedure
```

Remark 3.3.2. *:Using the Gauss method without pivoting (ordinary method), the number of operations required to compute the solution of the system $Ax = b$ of order n is given by :*

$$\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$$

including $\frac{1}{6}n(n-1)(2n+5)$ additions, $\frac{1}{6}n(n-1)(2n+5)$ multiplications, and $\frac{1}{2}n(n+1)$ divisions.

- **Example 3.3.3.** *:*

Consider the linear system (3.9) given by :

$$\begin{cases} x_1 + 2x_2 + 3x_3 + 4x_4 = 1 \\ 2x_1 + 3x_2 + 4x_3 + x_4 = 2 \\ 3x_1 + 4x_2 + x_3 + 2x_4 = 3 \\ 4x_1 + x_2 + 2x_3 + 3x_4 = 4 \end{cases} \tag{3.9}$$

1. *Write the linear system in matrix form* $Ax = b$.
2. *Solve the system using the Gaussian elimination method (apply the augmented matrix notation* $(A|b)$*).*
3. *Deduce the determinant of the matrix* A *(i.e.,* $dét(A)$*).*

1. *The linear system (3.9) in matrix form* $Ax = b$ *can be written as :*

$$\underbrace{\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}}_{b} \begin{matrix} \cdots & (L_1) \\ \cdots & (L_2) \\ \cdots & (L_3) \\ \cdots & (L_4) \end{matrix}$$

2. *Solving the system using the Gaussian elimination method : We start from the augmented matrix* $(A|b)$ *:*

$$\left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 1 \\ 2 & 3 & 4 & 1 & 2 \\ 3 & 4 & 1 & 2 & 3 \\ 4 & 1 & 2 & 3 & 4 \end{array}\right] \begin{matrix} \cdots & (L_1) \\ \cdots & (L_2) \\ \cdots & (L_3) \\ \cdots & (L_4) \end{matrix}$$

$$\begin{array}{l} Pivot(1) : Pi(1) = 1 \\ L_2^{'} \leftarrow L_2 - 2L_1 \\ L_3^{'} \leftarrow L_3 - 3L_1 \\ L_4^{'} \leftarrow L_4 - 4L_1 \end{array} \left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 1 \\ 0 & -1 & -2 & -7 & 0 \\ 0 & -2 & -8 & -10 & 0 \\ 0 & -7 & -10 & -13 & 0 \end{array}\right] \begin{matrix} \cdots & (L_1) \\ \cdots & (L_2^{'}) \\ \cdots & (L_3^{'}) \\ \cdots & (L_4^{'}) \end{matrix}$$

$$\begin{array}{l} Pivot(2) : Pi(2) = -1 \\ \\ L_3^{''} \leftarrow L_3^{'} - 2L_2^{'} \\ L_4^{''} \leftarrow L_4^{'} - 7L_2^{'} \end{array} \left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 1 \\ 0 & -1 & -2 & -7 & 0 \\ 0 & 0 & -4 & 4 & 0 \\ 0 & 0 & 4 & 36 & 0 \end{array}\right] \begin{matrix} \cdots & (L_1) \\ \cdots & (L_2^{'}) \\ \cdots & (L_3^{''}) \\ \cdots & (L_4^{''}) \end{matrix}$$

$$\begin{array}{l} Pivot(2) : Pi(3) = -4 \\ \\ \\ L_4^{'''} \leftarrow L_4^{''} + L_3^{''} \end{array} \left[\begin{array}{cccc|c} 1 & 2 & 3 & 4 & 1 \\ 0 & -1 & -2 & -7 & 0 \\ 0 & 0 & -4 & 4 & 0 \\ 0 & 0 & 0 & 40 & 0 \end{array}\right] \begin{matrix} \cdots & (L_1) \\ \cdots & (L_2^{'}) \\ \cdots & (L_3^{''}) \\ \cdots & (L_4^{'''}) \end{matrix}$$

Finally, by rewriting this last matrix in the form of system (3.5), we obtain the solution :

$$x_4 = x_3 = x_2 = 0 \quad and \quad x_1 = 1$$

3. *The determinant of the matrix* A*,* $dét(A)$ *: since no row permutations were performed, we have :*

$$det(A) = (-1)^0 \prod_{i=1}^{4} a_{ii} = 1 \cdot (-1) \cdot (-4) \cdot (40) = 160$$

where a_{ii} are the diagonal elements of the upper triangular matrix obtained after Gaussian elimination.

3.4 Iterative Methods

Direct methods are not recommended for solving a system of linear equations when the order of the system is quite large, because the number of operations to be performed will be very large, which leads to the propagation of rounding errors. We therefore resort to iterative methods, which consist of generating, from an initial vector $X^{(0)}$, a sequence of vectors $X^{(n)}$ converging towards the solution X.

In this part of the course, we will explain the Jacobi and Gauss-Seidel iterative methods using the following linear system of order $(n = 3)$:

$$\begin{cases} a_{11}x_1 + a_{12}\,x_2 + a_{13}x_3 = b_1 & (1) \\ a_{21}x_1 + a_{22}\,x_2 + a_{23}x_3 = b_2 & (2) \\ a_{31}x_1 + a_{32}\,x_2 + a_{33}x_3 = b_3 & (3) \end{cases}$$

We extract x_1, x_2, and x_3 from (1), (2), and (3), assuming that the coefficients $(a_{ii})_{i=1,\dots,3}$ are nonzero. We obtain :

$$\begin{cases} x_1 = \frac{1}{a_{11}}(b_1 - a_{12}\,x_2 - a_{13}x_3) \\ x_2 = \frac{1}{a_{22}}(b_2 - a_{21}\,x_1 - a_{23}x_3) \\ x_3 = \frac{1}{a_{33}}(b_3 - a_{31}\,x_1 - a_{32}x_2) \end{cases} \Leftrightarrow \begin{cases} x_1 = f(x_2, x_3) \\ x_2 = g(x_1, x_3) \\ x_3 = h(x_1, x_2) \end{cases} \tag{3.10}$$

Let $X^{(0)} = \begin{pmatrix} x_1^{(0)} \\ x_2^{(0)} \\ x_3^{(0)} \end{pmatrix}$ be the initial vector, which is chosen arbitrarily.

3.4.1 Jacobi Method

The Jacobi method consists of the following iterative process (replace $X^{(0)}$ in system (3.10) to obtain $X^{(1)}$, and so on) :

1st step

$$\begin{cases} x_1^{(1)} = f(x_2^{(0)}, x_3^{(0)}) \\ x_2^{(1)} = g(x_1^{(0)}, x_3^{(0)}) \\ x_3^{(1)} = h(x_1^{(0)}, x_2^{(0)}) \end{cases} \rightarrow X^{(1)} = \begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{pmatrix}$$

2nd step

$$\begin{cases} x_1^{(2)} = f(x_2^{(1)}, x_3^{(1)}) \\ x_2^{(2)} = g(x_1^{(1)}, x_3^{(1)}) \\ x_3^{(2)} = h(x_1^{(1)}, x_2^{(1)}) \end{cases} \rightarrow X^{(2)} = \begin{pmatrix} x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \end{pmatrix}$$

$\vdots$

kth step

$$\begin{cases} x_1^{(k)} = f(x_2^{(k-1)}, x_3^{(k-1)}) \\ x_2^{(k)} = g(x_1^{(k-1)}, x_3^{(k-1)}) \\ x_3^{(k)} = h(x_1^{(k-1)}, x_2^{(k-1)}) \end{cases} \rightarrow X^{(k)} = \begin{pmatrix} x_1^{(k)} \\ x_2^{(k)} \\ x_3^{(k)} \end{pmatrix}$$

Thus, we continue in the same way until the stopping criterion is satisfied (Remark 3.4.1).

3.4.2 Gauss-Seidel Method

The Gauss-Seidel method is based on the following iterative process :

1st step

$$\begin{cases} x_1^{(1)} = f(x_2^{(0)}, x_3^{(0)}) \\ x_2^{(1)} = g(x_1^{(1)}, x_3^{(0)}) \\ x_3^{(1)} = h(x_1^{(1)}, x_2^{(1)}) \end{cases} \rightarrow X^{(1)} = \begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{pmatrix}$$

2nd step

$$\begin{cases} x_1^{(2)} = f(x_2^{(1)}, x_3^{(1)}) \\ x_2^{(2)} = g(x_1^{(2)}, x_3^{(1)}) \\ x_3^{(2)} = h(x_1^{(2)}, x_2^{(2)}) \end{cases} \rightarrow X^{(2)} = \begin{pmatrix} x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \end{pmatrix}$$

And so on, until the stopping criterion is satisfied. $\vdots$

k^{th} **step** $\vdots$

$$\begin{cases} x_1^{(k)} = f(x_2^{(k-1)}, x_3^{(k-1)}) \\ x_2^{(k)} = g(x_1^{(k)}, x_3^{(k-1)}) \\ x_3^{(k)} = h(x_1^{(k)}, x_2^{(k)}) \end{cases} \rightarrow X^{(k)} = \begin{pmatrix} x_1^{(k)} \\ x_2^{(k)} \\ x_3^{(k)} \end{pmatrix}$$

Remark 3.4.1. *:*

1. ***Stopping criterion :*** *In general, the iterative method is stopped based on the relative error between two successive iterations $X^{(k)}$ and $X^{(k+1)}$. That is, the iterative process stops if :*

$$\frac{\|X^{(k+1)} - X^{(k)}\|}{\|X^{(k+1)}\|} < \varepsilon, \tag{3.11}$$

where ε is a prescribed tolerance.

$$\frac{\|X^{(k+1)} - X^{(k)}\|}{\|X^{(k+1)}\|} < \epsilon$$

where ϵ is a positive real number chosen sufficiently small.

2. *The iterative method is said to converge if $X^{(k)} \to X^*$ (the solution of the system) as $k \to +\infty$.*

3. *The iterative process of the method converges if the following condition is satisfied :*

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|$$

- **Example 3.4.2.** *:*

Let the linear system (3.9) be given by :

$$\begin{cases} 10x_1 + 3x_2 - 2x_3 = 57 \\ 2x_1 + 8x_2 - x_3 = 20 \\ x_1 + x_2 + 5x_3 = -4 \end{cases} \tag{3.12}$$

1. *Write the linear system in the matrix form $Ax = b$.*
2. *Solve the system using the Jacobi and Gauss-Seidel iterative methods (3 iterations) with the initial vector $X^{(0)} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$.*

The system can be written as :

$$\begin{cases} x_1 = \frac{1}{10}(57 - 3x_2 + 2x_3) \\ x_2 = \frac{1}{8}(20 - 2x_1 + x_3) \\ x_3 = \frac{1}{5}(-4 - x_1 - x_2) \end{cases}$$

Jacobi Method :

1st iteration

$$\begin{cases} x_1^{(1)} = \frac{1}{10}(57 - 3 + 2) = 5.6 \\ x_2^{(1)} = \frac{1}{8}(20 - 2 + 1) = 2.375 \\ x_3^{(1)} = \frac{1}{5}(-4 - 1 - 1) = -1.2 \end{cases}$$

2nd iteration

$$\begin{cases} x_1^{(2)} = \frac{1}{10}(57 - 3(2.375) + 2(-1.2)) = 4.7475 \\ x_2^{(2)} = \frac{1}{8}(20 - 2(5.6) + (-1.2)) = 0.95 \\ x_3^{(2)} = \frac{1}{5}(-4 - 5.6 - (2.375)) = -2.395 \end{cases}$$

3rd iteration

$$\begin{cases} x_1^{(3)} = \frac{1}{10}(57 - 3(0.95) + 2(-2.395)) = 4.936 \\ x_2^{(3)} = \frac{1}{8}(20 - 2(4.7475) + (-2.395)) = 1.0137 \\ x_3^{(3)} = \frac{1}{5}(-4 - 4.7475 - (0.95)) = -1.9395 \end{cases}$$

========== Jacobi method ==========

iter	x_1	x_2	x_3
0	1	1	1
1	5.6000	2.3750	-1.2000
2	4.7475	0.9500	-2.3950
3	4.9360	1.0137	-1.9395
4	5.0080	1.0236	-1.9900
5	4.9949	0.9992	-2.0063
6	4.9990	1.0005	-1.9988
7	5.0001	1.0004	-1.9999
8	4.9999	0.9999	-2.0001
9	5.0000	1.0000	-2.0000

solution:

==========

$x_1 = 5.0000; x_2 = 1.0000 \ et \ x_3 = -2.0000$

==========

Gauss-Seidel Method :

1st iteration

$$\begin{cases} x_1^{(1)} = \frac{1}{10}(57 - 3 + 2) = 5.6 \\ x_2^{(1)} = \frac{1}{8}(20 - 2(5.6) + 1) = 1.225 \\ x_3^{(1)} = \frac{1}{5}(-4 - 5.6 - 1.225) = -2.165 \end{cases}$$

2nd iteration

$$\begin{cases} x_1^{(2)} = \frac{1}{10}(57 - 3(1.225) + 2(-2.165)) = 4.8995 \\ x_2^{(2)} = \frac{1}{8}(20 - 2(4.8995) + (-2.165)) = 1.0045 \\ x_3^{(2)} = \frac{1}{5}(-4 - 4.8995 - (1.0045)) = -1.9808 \end{cases}$$

3rd iteration

$$\begin{cases} x_1^{(3)} = \frac{1}{10}(57 - 3(4.8995) + 2(-1.9808)) = 5.0025 \\ x_2^{(3)} = \frac{1}{8}(20 - 2(5.0025) + (-1.9395)) = 1.0018 \\ x_3^{(3)} = \frac{1}{5}(-4 - 5.0025 - (1.0018)) = -2.0009 \end{cases}$$

========== Gauss-Siedel method ============

iter	x_1	x_2	x_3
0	*1*	*1*	*1*
1	*5.6000*	*1.2250*	*-2.1650*
2	*4.8995*	*1.0045*	*-1.9808*
3	*5.0025*	*1.0018*	*-2.0009*
4	*4.9993*	*1.0001*	*-1.9999*
5	*5.0000*	*1.0000*	*-2.0000*

solution :

==========

$x_1 = 5.0000; x_2 = 1.0000$ *et* $x_3 = -2.0000$

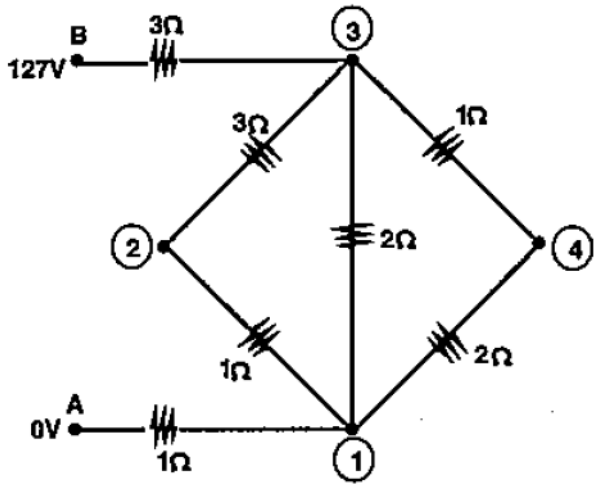
==========

3.5 Exercises with Solutions

3.5.1 Exercises

- **Exercise 3.1.** *:*

Consider the following electrical circuit :



- Calculate the voltages at nodes 1, 2, 3, and 4 with respect to node A ($V_A = 0$), using Kirchhoff's law : "The algebraic sum of currents at a node is zero."

Solution : (3.1)

- **Exercise 3.2.** *:*

Consider the following system of linear equations :

$$\begin{cases} 2x_1 + x_2 + 0 + 4x_4 = 2 \\ -4x_1 - 2x_2 + 3x_3 - 7x_4 = -9 \\ 4x_1 + x_2 - 2x_3 + 8x_4 = 2 \\ 0 - 3x_2 - 12x_3 - x_4 = 2 \end{cases}$$

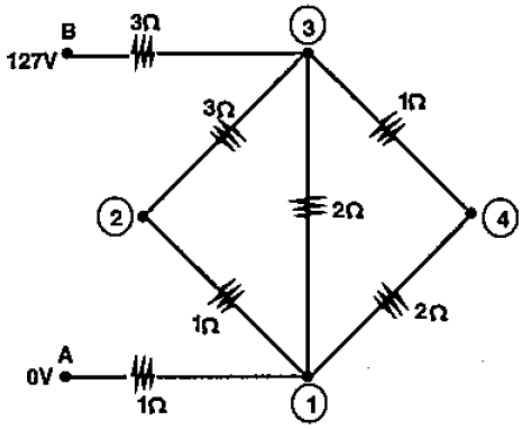
1. Write the linear system in matrix form $Ax = b$.
2. Solve the system using the Gaussian elimination method (using the augmented matrix notation $(A|b)$).
3. Deduce the determinant of the matrix A (i.e., dét(A)).

answer : (3.2)

3.5.2 Solutions

- **solution 3.1.** *:*

According to Kirchhoff's law, "The algebraic sum of currents at a node is zero," we have, based on the circuit diagram :



(1) : $I_{A1} + I_{21} + I_{31} + I_{41} = 0$

(2) : $I_{12} + I_{32} = 0$

(3) : $I_{13} + I_{23} + I_{43} + I_{B3} = 0$

(4) : $I_{14} + I_{34} = 0$

Using Ohm's law : $I_{AB} = \frac{V_A - V_B}{R_{AB}}$

(1) : $\frac{V_A - V_1}{1} + \frac{V_2 - V_1}{1} + \frac{V_3 - V_1}{2} + \frac{V_4 - V_1}{2} = 0$

(2) : $\frac{V_1 - V_2}{1} + \frac{V_3 - V_2}{3} = 0$

(3) : $\frac{V_1 - V_3}{2} + \frac{V_2 - V_3}{3} + \frac{V_4 - V_3}{1} + \frac{V_B - V_3}{3} = 0$

(4) : $\frac{V_1 - V_4}{2} + \frac{V_3 - V_4}{1} = 0$

Simplifying these equations, we obtain the following system of linear equations :

$$\begin{cases} -6V_1 + 2V_2 + V_3 + V_4 = 0 & (1) \\ 3V_1 - 4V_2 + V_3 + 0 = 0 & (2) \\ 3V_1 + 2V_2 - 13V_3 + 6V_4 = -254 & (3) \\ V_1 + 0 + 2V_3 - 3V_4 = 0 & (4) \end{cases}$$

$$\begin{bmatrix} -6 & 2 & 1 & 1 \\ 3 & -4 & 1 & 0 \\ 3 & 2 & -13 & 6 \\ 1 & 0 & 2 & -3 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -254 \\ 0 \end{bmatrix} \begin{matrix} \cdots (L_1) \\ \cdots (L_2) \\ \cdots (L_3) \\ \cdots (L_4) \end{matrix}$$

Using the Gaussian elimination method to solve this system of equations :

```
      -----Pivot numero 1 -----
|-6   2       1        1        |  0    |
|0    -3      1.5      0.5      |  0    |
|0    3       -12.5    6.5      |  -254 |
|0    0.3333  2.1667   -2.8333  |  0    |
      -----Pivot numero 2-----
|-6   2   1       1        |  0    |
|0    -3  1.5     0.5      |  0    |
|0    0   -11     7        |  -254 |
|0    0   2.3333  -2.7778  |  0    |
The final upper triangular matrix [A|B] (pivot number 3):
|-6   2   1.   1        |  0        |
|0    -3  1.5  0.5      |  0        |
|0    0   -11  7        |  -254     |
|0    0   0    -1.2929  |  -53.8788 |
      Solution of the system:
====================================
V1 = 25.7969; V2 = 31.75, V3 = 49.6094 and V4 = 41.6719 ( Volt)
====================================
```

exercise :3.1

- **solution 3.2.** *:*

The system in matrix form :

$$\begin{bmatrix} 2 & 1 & 0 & 4 & 2 \\ -4 & -2 & 3 & -7 & -9 \\ 4 & 1 & -2 & 8 & 2 \\ 0 & -3 & -12 & -1 & 2 \end{bmatrix}$$

pivot————————-1——

$$\begin{bmatrix} 2 & 1 & 0 & 4 & 2 \\ 0 & 0 & 3 & 1 & -5 \\ 0 & -1 & -2 & 0 & -2 \\ 0 & -3 & -12 & -1 & 2 \end{bmatrix}$$

Pivot ——— 2 —- (row swap between row 2 and row 3) –

$$\begin{bmatrix} 2 & 1 & 0 & 4 & 2 \\ 0 & -1 & -2 & 0 & -2 \\ 0 & 0 & 3 & 1 & -5 \\ 0 & 0 & -6 & -1 & 8 \end{bmatrix}$$

pivot————————-3——

$$\begin{bmatrix} 2 & 1 & 0 & 4 & 2 \\ 0 & -1 & -2 & 0 & -2 \\ 0 & 0 & 3 & 1 & -5 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix}$$

We obtain the system of equations :

$$\begin{cases} x_4 = -2/1 \\ x_3 = (-5 - x_4)/3 \\ x_2 = (-2 + 2x_3)/(-1) \\ x_1 = (2 - x_2 - 4x_4)/2 \end{cases}$$

Solution : $x_1 = 3.; x_2 = 4.; x_3 = -1.; x_4 = -2.$

Determinant (single permutation, $p = 1$)

$$det(A) = (-1)^1 \prod_{i=1}^{4} a_{ii}^{(i)} = (1-)(2 \cdot (-1) \cdot 3 \cdot 1) = 6 \tag{3.13}$$

exercise :3.2

Chapitre 4

Numerical Solution of Differential Equations

4.1 Introduction

Most numerical methods for solving differential equations apply to problems of the Cauchy type. Such differential equations can be encountered in fluid mechanics, heat transfer, or electronics.

Analytical solutions exist only for certain types of differential equations, but there is a large class of differential equations that cannot be solved analytically. Therefore, numerical solutions are essential in these cases.

4.2 Differential Equations

A differential equation is any equation relating a function $y(t)$ and its successive derivatives $(y^{(m)})$:

$$F[t, y(t), y^{(1)}(t), y^{(2)}(t), \ldots, y^{(m)}(t)] = 0 \tag{4.1}$$

where $y^{(m)} = \frac{d^m y(t)}{dt^m}$.

The order of this equation is determined by its highest-order derivative. Therefore, equation (4.1) is of order m. The solution of the problem consists in finding a function $y(t)$.

Cauchy Problem :

A first-order differential equation with an initial condition, which can be written in the following form (equation 4.2), is known as a **Cauchy problem**.

$$\begin{cases} y'(t) = f(t, y(t)), & t \in [t_0; T] \\ y(t_0) = y_0, & \text{initial condition at } t_0. \end{cases} \tag{4.2}$$

4.3 Numerical Solution of Differential Equations

4.3.1 General Form of the Analytical Solution

Let the first-order differential equation be :

$$\frac{d}{dt}y(t) = f(t, y(t)) \tag{4.3}$$

The formal solution of this equation is :

$$\int_{y(t_0)}^{y(t)} dy = \int_{t_0}^{t} f(s, y(s))ds$$

Thus, the solution $y(t)$ can be written as :

$$y(t) - y_0 = \int_{t_0}^{t} f(s, y(s))ds \tag{4.4}$$

The problem with this formal solution is that the unknown $y(t)$ appears under the integral, and it is not always computable.

4.3.2 Picard's Iterative Method

In Picard's method, one can iterate equation (4.4) to generate a series of approximations to $y(t)$, denoted as ${}_1y(t)$, ${}_2y(t)$, ..., ${}_ky(t)$:

Given $y(t_0) = y_0$, the series of approximations to $y(t)$ can be written as :

$$\begin{aligned}
{}_1y(t) &= y_0 + \int_{t_0}^{t} f(s, y_0)ds \\
{}_2y(t) &= y_0 + \int_{t_0}^{t} f(s, {}_1y(s))ds \\
{}_3y(t) &= y_0 + \int_{t_0}^{t} f(s, {}_2y(s))ds \\
&\vdots \\
{}_ky(t) &= y_0 + \int_{t_0}^{t} f(s, {}_ky(s))ds
\end{aligned} \tag{4.5}$$

Picard's method can be particularly useful when it is possible to perform an analytical integration of the sequence $\int_{t_0}^{t} f(s, {}_ky(s))\, ds$.

- **Example 4.3.1.** *:*

Compute, using Picard's method, the third series of approximations for the function $y(t)$ defined by the following differential equation :

$$\begin{cases} y' = t + 1 - y, \\ y(0) = 1, \qquad \textit{initial condition at } t = 0. \end{cases} \tag{4.6}$$

We have : $y(t_0) = y(0) = 1$ and $f(s,y) = -y + t + 1$, so the series of approximations to $y(t)$ can be written as :

$$\begin{aligned}
{}_1y(t) &= 1 + \int_0^t (s + 1 - y_0)ds \\
&= 1 + \int_0^t (s + 1 - 1)ds \\
&= 1 + \frac{1}{2}t^2 \\
{}_2y(t) &= 1 + \int_0^t (s + 1 - {}_1y)ds \\
&= 1 + \int_0^t (s + 1 - (1 + \frac{1}{2}s^2)ds \\
&= 1 + \frac{1}{2}t^2 - \frac{1}{6}t^3 \\
{}_3y(t) &= 1 + \int_0^t (s + 1 - {}_2y)ds \\
&= 1 + \int_0^t (s + 1 - (1 + \frac{1}{2}t^2 - \frac{1}{6}t^3)ds \\
&= 1 + \frac{1}{2}t^2 - \frac{1}{6}t^3 + \frac{1}{24}t^4 \\
&\vdots \\
{}_ky(t) &= 1 + \sum_{k=1}^{n} (-1)^{(k+1)} \frac{t^{(k+1)}}{(k+1)!}
\end{aligned}$$

The approximate solution (for $k = 3$) is : $y(t) \simeq 1 + \frac{1}{2}t^2 - \frac{1}{6}t^3 + \frac{1}{24}t^4$

========== Picard method : $y' = t + 1 - y$ ==========

${}_ky \backslash t$	0	0.1	0.2	0.3	0.4
${}_1y$	1	1.00500	1.02000	1.04500	1.08000
${}_2y$	1	1.00483	1.01866	1.04049	1.06933
${}_3y$	1	1.00483	1.01873	1.04083	1.07040
${}_4y$	1	1.00483	1.01873	1.04081	1.07031
${}_5y$	1	1.00483	1.01873	1.04081	1.07032
exact Sol. y(t)	1	1.00483	1.01873	1.04081	1.07032

==========

4.3.3 Taylor Series-Based Methods

Taylor series-based methods consist of subdividing the interval $[t_0, T]$ into N subintervals of equal length h, and we denote the subdivision points by t_n :

$$t_1 = t_0 + h$$
$$\text{and} \quad t_n = t_0 + nh \quad \text{ou } n = 0, 1, .., N$$

The Taylor series expansion of $y(t_{n+1})$ up to order m around the point t_n can be written as :

$$y(t_{n+1}) = y(t_n) + h\,y^{(1)}(t_n) + \frac{h^2}{2!}y^{(2)}(t_n) + \cdots + \frac{h^m}{m!}y^{(m)}(t_n) + 0(h^{m+1}). \tag{4.7}$$

a)- Euler's Method

If we truncate the Taylor series (4.7) at order **1**, we obtain :

$$y(t_{n+1}) = y(t_n) + h\,y'(t_n) + o(h^2) \tag{4.8}$$

The application of this formula to compute the values $y(t_n)$ is known as Euler's method. It consists of finding an approximation to the solution of equation (4.2) such that :

$$y(t_{n+1}) \simeq y_{n+1} = y_n + hf(t_n, y_n) \tag{4.9}$$

b)- Euler's Algorithm

Given a step size h, an initial condition (t_0, y_0), and a maximum number of iterations N, the iterative formula of Euler's method can be written as :

$$\begin{cases} t_{n+1} = t_n + h & \text{for } 0 \leq n \leq N \\ y_{n+1} = y_n + hf(t_n, y_n) \end{cases} \tag{4.10}$$

- **Example 4.3.2. :**

Algorithm 4.3.1 Algorithm for solving a differential equation (Cauchy problem) using Euler's method.

Require: Step size h, number of iterations n, initial conditions t_0, y_0
Ensure: Approximated values of y at each step

```
procedure EULERMETHOD(h, n, t_0, y_0)
    t ← t_0
    y ← y_0
    for i ← 1 to n do
        y ← y + h · f(t, y)
        t ← t + h                          ▷ Solution at step i : (t, y)
        print t, y
    end for
    return y
end procedure
```

1. *Using Euler's method, find the first five values of the function $y(t)$ defined by the following differential equation for step sizes $h = 0.1$ and $h = 0.2$:*

$$\begin{cases} y' = -y + t + 1, \\ y(0) = 1, \qquad \textit{initial condition at } t = 0. \end{cases} \tag{4.11}$$

2. *Compare the obtained results with those calculated from the exact solution :*
   $y(t) = t + exp(-t)$

1. *The approximate solution of the differential equation :*
   We have Euler's formula : $y_{i+1} = y_i + h\,F(t_i, y_i)$
   where $F(t, y) = y' = -y + t + 1$
   - *for a step* $h = 0.1$ *:*
     ***)** *at* $t = t_0 = 0$ *on a* $y(0) = 1$*(initial condition) : so* $F(t_0, y_0) = 0$
     ***)** *at* $t = t_1 = 0.1 \Rightarrow y(t_1 = 0.1) = y_1$
     By substituting into Euler's formula :
     $y(0.1) = y_1 = y_0 + 0.1F(x_0, y_0) = 1 + 0.1 * (0) = 1$
     ***)** *at* $t = t_2 = 0.2 \Rightarrow y(t_2 = 0.2) = y_2$
     $y_2 = y_1 + 0.1F(t_1, y_1) = 1 + 0.1 * (-1 + 0.1 + 1) = 1.01$
     and so on
   - *for* $h = 0.1$ *:*
     ***)** *at* $t = t_0 = 0$ *we have* $y(0) = 1$*(initial condition) :* $F(t_0, y_0) = 0$
     ***)** *at* $t = t_1 = 0.2 : \Rightarrow y(t_1 = 0.2) = y_1$
     By substituting into Euler's formula : :

$y(0.2) = y_1 = y_0 + 0.2F(t_0, y_0) = 1 + 0.2 * (0) = 1$

***)** *at* $t = t_2 = 0.4 \Rightarrow y(t_2 = 0.4) = y_2$

$y_2 = y_1 + 0.2F(t_1, y_1) = 1 + 0.2 * (-1 + 0.2 + 1) = 1.04$

and so on

========== *Euler method for* $h = 0.1$ ==========

i	t	y_i *Euler*	$y(t) = t + e^{-t}$	*Absolute error*
0	0	1	1	0
1	0.1000	1.0000	1.0048	0.0048
2	0.2000	1.0100	1.0187	0.0087
3	0.3000	1.0290	1.0408	0.0118
4	0.4000	1.0561	1.0703	0.0142
5	0.5000	1.0905	1.1065	0.0160

==========

========== *Euler method for* $h = 0.2$ ==========

i	t	y_i *Euler*	$y(t) = t + e^{-t}$	*Absolute error*
0	0	1	1	0
1	0.2000	1.0000	1.0187	0.0187
2	0.4000	1.0400	1.0703	0.0303
3	0.6000	1.1120	1.1488	0.0368
4	0.8000	1.2096	1.2493	0.0397

==========

Remark 4.3.3. *:The error of Euler's method occurs at each step (iteration). It is clear that the accuracy of this method can be improved by decreasing the step size h.*

c)- Second-Order Taylor Methods

By increasing the number of terms included in the Taylor series (equation 4.7), one can expect an improvement in the numerical solution obtained by Euler's method.

For the second-order Taylor method, we need the first and second derivatives.

$$y(t_{n+1}) = y(t_n) + h\,y'(t_n) + \frac{h^2}{2!}y''(t_n) + o(h^3) \tag{4.12}$$

The second-order derivative can be obtained by differentiating the differential equation (4.2) :

$$y(t_{n+1}) = y(t_n) + h\,f(t_n, (y(t_n)) + \frac{h^2}{2!}f'(t_n, y(t_n)) + o(h^3) \tag{4.13}$$

The derivative of f is :

$$\begin{aligned} f'(t,y(t)) &= \frac{\partial f(t,y(t))}{\partial t} + \frac{\partial f(t,y(t))}{\partial y} y'(t) \\ &= \frac{\partial f(t,y(t))}{\partial t} + \frac{\partial f(t,y(t))}{\partial y} f(t,y(t)) \end{aligned}$$

The solution $y(t_{n+1}) \simeq y_{n+1}$ can be written as :

$$y_{n+1} = y_n + h\, f(t_n, y_n) + \frac{h^2}{2}[\frac{\partial f(t_n,y_n)}{\partial t} + \frac{\partial f(t_n,y_n)}{\partial y} f(t_n,y_n)] \tag{4.14}$$

- **Example 4.3.4. :**

Using the second-order Taylor method do the example (4.3.2)

We have the second-order Taylor formula : $y_{i+1} = y_i + h\,F(t_i,y_i) + \frac{h^2}{2}F'(t_n,y_n)$

where $F(t,y) = y' = -y + t + 1$

and its derivative : $F'(t,y) = \frac{\partial F(t,y(t))}{\partial t} + \frac{\partial F(t,y(t))}{\partial y} f(t,y(t)) = 1 - f(t,y(t))$

- *for* $h = 0.1$ *:*

  ***)** *at* $t = t_0 = 0$ *on a* $y(0) = 1$ *(initial condition) :* $F(t_0,y_0) = 0$ *and* $F'(t_0,y_0) = 1$

  ***)** *at* $t = t_1 = 0.1 \Rightarrow y(t_1 = 0.1) = y_1$

  By substituting into the Taylor formula :

  $y(0.1) = y_1 = y_0 + 0.1F(x_0,y_0) + \frac{0.1^2}{2} * F'(t_0,y_0) = 1 + 0.1 * (0) + \frac{0.1^2}{2} = 1.005$

  ***)** *at* $t = t_2 = 0.2 \Rightarrow y(t_2 = 0.2) = y_2$ *:*

  $F(t_1,y_1) = -y_1 + t_1 + 1 = -1.005 + 0.1 + 1 = 0.095$

  and $F'(t_1,y_1) = 1 - 0.095 = 0.905$

  $y_2 = y_1 + 0.1F(t_1,y_1) + \frac{0.1^2}{2}F'(t_0,y_0)$

  $= 1.005 + 0.1 * (0.095) + \frac{0.1^2}{2} * 0.905 = 1.019025$

========== *Taylor order 2 method for* $h = 0.1$ ==========

i	t	y_i *Tay2*	$y(t) = t + e^{-t}$	*Absolute error*
0	0.1	1.0050	1.0048	0.0002
1	0.2	1.0190	1.0187	0.0003
2	0.3	1.0412	1.0408	0.0004
3	0.4	1.0708	1.0703	0.0005
4	0.5	1.1071	1.1065	0.0005
5	0.6	1.1494	1.1488	0.0006

==========

Remark 4.3.5. *:It can be observed that the error is smaller with the second-order Taylor method than with Euler's method.*

4.4 Exercises with solutions

4.4.1 Exercises

- **Exercise 4.1.** *:*

The differential equation related to the velocity of the ball (y) during its vertical fall in a fluid is given with the initial condition by the following formula :

$$\begin{cases} y' = -y + 2 & (1) \\ y(0) = 0 \qquad \text{at } t = 0 \end{cases}$$

1. Verify that the exact solution of the differential equation (1) is :

$$y(t) = 2(1 - e^{-t})$$

2. Use the Euler method with a step $h = 0.1$ to determine the approximate value of the velocity for $t = 0.1$ and $t = 0.2$, then complete the following table :

i	x_i	y_i(Euler method)	$y(x) = 2(1 - e^{-x})$	$y(x) - y_i$
0	0.0	0	0	0
1	0.1			
2	0.2			
3	0.3			
4	0.4			

(provide results with 4 decimal places).

solution : (4.1)

- **Exercise 4.2.** *:*

A glass ball is dropped, ***without initial velocity***, at the surface of a vertical tube containing castor oil. The ball is under the action of three forces : weight, buoyant force (Archimedes), and a friction force. From the study of the video recording of the fall, we obtained the experimental values of the velocity v_{exp} :

t (s)	0	0.2	0.4	0.6	0.8	1.0
v_{exp}.	0	0.4375	0.6841	0.7762	0.8308	0.8505
$v(m/s)$	0				0.8400	
Δv	0				0.0092	

To model the motion of the ball, we assumed that ***the friction force*** is proportional to v^2. By applying Newton's second law, we obtained the differential equation of motion of the ball :

$$\frac{dv}{dt} + 2.89v^2 = 2.24 \qquad (1)$$

1. Write equation (1) as a Cauchy problem.
2. Solve the differential equation (1) using the modified Euler method (Taylor of order 2), and complete the table above.
3. Draw a conclusion regarding the choice of the friction force.

solution : (4.2)

- **Exercise 4.3.** *:*

Undamped heavy pendulum of mass m, suspended at O by a weightless string of length l, displaced by an angle $\theta_0 = \frac{\pi}{3}$ from the equilibrium position and then released without initial velocity $\theta'(0) = 0$.

The study of the pendulum motion, using the fundamental law of dynamics, allows us to find the following nonlinear second-order differential equation :

$$\begin{cases} \theta''(t) = \frac{g}{l} sin(\theta), \\ \theta(0) = \frac{\pi}{3}, \qquad\qquad \text{at } t = 0. \end{cases}$$

Where $g = 10m.s^{-2}$ and $l = 0.5m$

Find, using the Euler method, four values of the angle θ and the angular velocity θ' by choosing a step $h = 0.1$.

4.4.2 Solutions

- **solution 4.1.** *:*

1. Verification of the exact solution of : $y' = -y + 2$

$$\begin{cases} y(t) = 2(1 - e^{-t}) \\ y'(t) = 2e^{-t} \end{cases}$$

$$\begin{aligned} -y + 2 &= -2(1 - e^{-t}) + 2 \\ &= -2 + 2e^{-t} + 2 \\ &= 2e^{-t} \\ &= y' \end{aligned}$$

2. The approximate solution of the differential equation : $y' = F(t, y) = -y + 2$

   Given $h = 0.1$ and the initial condition at $t_0 = 0 : y_0 = y(0) = 0$ and $F(t_0, y_0) = 2$

   Replacing in Euler's formula : $y_{i+1} = y_i + h\,F(t_i, y_i)$

   ***)** $y_1 = y_0 + 0.1F(t_0, y_0) = 0 + 0.1 * (-0 + 2) = 0.2$

   For $t = 0.1$ **we have** $y(0.1) = 0.2$

   ***)** $y_2 = y_1 + 0.1F(t_1, y_1) = 0.2 + 0.1 * (-0.2 + 2) = 0.38$

- Similarly :

i	t_i	y_i(Euler method)	$y(t) = 2(1 - e^{-t})$	$y(t) - y_i$
0	0.0	0	0	0
1	0.1	0.200	0.1903	0.0097
2	0.2	0.380	0.3625	0.0175
3	0.3	0.542	0.5184	0.0236

exercise :4.1

- **solution 4.2.** *:*

The initial velocity is zero : $v(0) = 0$ and $\frac{dv}{dt} + 2.89v^2 = 2.24$

1- Cauchy problem : $\begin{cases} \dot{v} = 2.24 - 2.89v^2 \\ v(0) = 0 \end{cases}$

2- Modified Euler method (Taylor of order 2) :

$$v_{n+1} = v_n + h.f(t_n, v_n) + \frac{h^2}{2}\left(\frac{\partial f}{\partial t} + f(t_n, v_n)\frac{\partial f}{\partial v}\right)$$

$$\text{Where : } \begin{cases} t_{n+1} = t_n + h = t_n + 0.2 \\ f(t_n, v_n) = 2.24 - 2.89v_n^2 \\ \frac{\partial f}{\partial t} = 0 \\ \frac{\partial f}{\partial v} = -5.78v_n. \end{cases}$$

$$\text{At } t_0 = 0 : \begin{cases} v_0 = 0 \\ f(t_0, v_0) = 2.24 \\ (\frac{\partial f}{\partial v})_{t_0} = -5.78v_0 = 0 \end{cases}$$

$$\text{For } t_1 = 0.2 : \begin{cases} v_1 = v_0 + 0.2.f(t_0, v_0) + 0.02f(t_0, v_0)(\frac{\partial f}{\partial v})_{t_0} = 0.4480 \\ f(t_1, v_1) = 2.24 - 2.89v_1^2 = 1.6600 \\ (\frac{\partial f}{\partial v})_{t_1} = -5.78v_1 = -2.5894 \end{cases}$$

$$\text{For } t_2 = 0.4 : \begin{cases} v_2 = v_1 + 0.2.f(t_1, v_1) + 0.02f(t_1, v_1)(\frac{\partial f}{\partial v})_{t_1} = 0.6940 \\ f(t_2, v_2) = 2.24 - 2.89v_1^2 = 0.8480 \\ (\frac{\partial f}{\partial v})_{t_2} = -5.78v_1 = -4.0113 \end{cases}$$

t (s)	0	0.2	0.4	0.6	0.8	1.0
$v_{exp.}$	0	0.4375	0.6841	0.7762	0.8308	0.8505
$v.$	0	0.4480	0.6940	0.7956	0.8400	0.8606
Δv	0	0.0105	0.0099	0.0194	0.0092	0.0101

3- Conclusion : The choice of the friction force $f_r = k.v^2$ is appropriate since the absolute error is of order ± 0.01 compared to the measured results.

exercise :4.2

Annexe A

Practical Work

A.1 TP : Introduction to Python and NumPy

a)- Matrices and Vectors

Python with `NumPy` provides functionality for working with vectors and matrices. Even scalar variables are treated as 1×1 arrays.

Defining a matrix in Python :

Use `np.array()` to define vectors or matrices. Rows are separated by commas, and each row is enclosed in square brackets : $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ python syntax :

```
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6]])
```

There are also built-in functions to create standard vectors/matrices :

- `np.zeros(n)` creates a vector of zeros
- `np.ones(n)` creates a vector of ones
- `np.eye(n)` creates an identity matrix
- `np.diag(...)` creates a diagonal matrix
- `np.linspace(a,b,n)` creates n evenly spaced elements between a and b

- **Exercise 1. :**

1. Construct the matrices :

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 3 & 2 \\ 0 & -1 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

2. Using `np.zeros`, create a row vector x of 5 zeros.
3. Using `np.ones`, create a column vector y of 5 ones.
4. Using `np.eye`, create the 4×4 identity matrix I.
5. Using `np.linspace`, create a row vector t with 10 elements evenly spaced between -1 and 1.

b)- Accessing Elements of a Matrix

For a vector, the i-th component is obtained with `x[i]` in Python (0-based index). For a matrix, the element at row i and column j is `A[i,j]`.

- **Exercise 2.** *:*
  1. Define the matrix :
  $$F = \begin{pmatrix} 8 & 1 & 6 & 5 \\ 3 & 5 & 7 & 4 \\ 4 & 9 & 2 & 1 \\ 5 & 3 & 6 & 2 \end{pmatrix}$$
  2. Extract element F_{23} : `F[1,2]`.
  3. Extract the first row (`F[0,:]`), first column (`F[:,0]`), row 1 columns 2-4 (`F[0,1:4]`).
  4. Extract submatrix :
  $$\begin{pmatrix} 1 & 6 \\ 5 & 7 \\ 9 & 2 \end{pmatrix}$$
  using Python slicing.

c)- Matrix Operations

Python/NumPy supports standard matrix operations.

- **Exercise 3.** *:*
$$A = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 3 & 2 \\ 0 & -1 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$
  1. With the defined matrices A and B, compute $A + B$, $A * B$ and $2 * A$.

d)- Visualization Example (Optional)

```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(-1, 1, 10)
plt.plot(t, np.sin(t))
plt.title("Example plot with t")
plt.xlabel("t")
plt.ylabel("sin(t)")
plt.grid(True)
plt.show()
```

A.2 Polynomial Interpolation

A.2.1 Objective

— Implement the Lagrange and Newton polynomial interpolation algorithms in Python.

— Test the influence of the polynomial order on the interpolation results.

- **Exercise 4.** **:**Consider the function :

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

1. Write your own Python program that calculates the Lagrange polynomial $p_n(x)$ associated with the points : $(-1, f(-1)), (0, f(0))$, and $(1, f(1))$. Compare your results $p_2(-0.5)$ and $p_2(0.5)$ with the exact values $f(-0.5)$ and $f(0.5)$ respectively.
   Example program Python (*lagrange.py*) :

```
import numpy as np

def f(x):
    return 1 / (1 + x**2)

def lagrange_poly(x, xi, yi):
    n = len(xi)
    L = 0
    for i in range(n):
        li = 1
        for j in range(n):
            if i != j:
                li *= (x - xi[j]) / (xi[i] - xi[j])
        L += yi[i] * li
    return L
```

2. complete the program and insert the following values in the programme *lagrange.py* :

$$xi = [-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0],$$
$$yi = [0.2, 0.3077, 0.5, 0.8, 1.0, 0.8, 0.5, 0.3077, 0.2]$$

3. Execute the program while varying the polynomial order from 2 to 8.
4. Plot the original function $f(x)$ and its polynomial interpolation on the same figure. Draw your conclusion on the choice of polynomial order.
   Example Python plotting :

```
import matplotlib.pyplot as plt

x_plot = np.linspace(-2, 2, 200)
plt.plot(x_plot, f(x_plot), label='f(x)')

for n in range(2, 9):
    xi_subset = xi[:n+1]
    yi_subset = yi[:n+1]
    y_poly = [lagrange_poly(x, xi_subset, yi_subset) for x in x_plot]
    plt.plot(x_plot, y_poly, label=f'Lagrange order {n}')

plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Function and Lagrange Interpolation')
plt.grid(True)
plt.show()
```

A.2.2 Lab : Polynomial Interpolation

A.2.3 Lagrange Polynomial Interpolation

Python Programme :

```
#=====================================
#       Course Notes
#     Numerical Physics
# program :
#    Lagrange Polynomial Interpolation
#=====================================

import numpy as np
import matplotlib.pyplot as plt
#-----------Function to interpolate
def funct(x):
    return 1/( 1 + x**2 )

#-----------Lagrange interpolation---------
def lagrange_interpolation(xi, yi, z, order ):
    poly = 0
    p=order+1
    for i in range(p):
        L = 1
        for j in range(p):
            if j != i:
                L *= (xi[j]- z) / (xi[j]-xi[i])
        poly += yi[i] * L
    return poly
#-----------Data points--------------------
xi = np.array([-2., -1.5,-1., -0.5 , 0.0 ,  0.5 , 1.0 , 1.5,  2.])
yi = np.array([0.2,0.3077, 0.5,0.80, 1., 0.8 , 0.5, 0.3077, 0.2])
z = -1.75

#-----------User input---------------------
num_points = len(xi)
n = int(input("Enter the order (n) of the Lagrange polynomial: "))

if n >= num_points:
    print("Polynomial order is higher than the number of points")
else:
    yp = lagrange_interpolation(xi, yi,z, n)
    yf = funct(z)
    delta = np.abs(yf - yp)

    print("     n      z       P(z)      f(z)       delta")
    print("-------------------------------------------------")
    print(f"{n:6d}  {z:8.6f}  {yp:8.6f} {yf:8.6f}  {delta:8.6f}")

    #-------Generate data for plotting--------------------
```

```
    x = np.linspace(np.min(xi), np.max(xi), 200)
    y = [lagrange_interpolation(xi, yi, val, n) for val in x]

    #----------- Plot the results-------------------------
    plt.plot(x, y, label=f"Lagrange Pn(x) (order n={n})")
    plt.plot(x, funct(x), color='green', label="Function:f(x)")
    plt.scatter(xi, yi, color='red', label="Data points")
    plt.scatter(z,yp,color='purple',label=f"interpolate point {z}")
    plt.title("Lagrange Interpolation for f(x) = 1/(1+x^2)")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend()
    plt.grid(True)
    plt.savefig(f"plot_lag_{n}.png", dpi=300)
    #plt.show()
```

Execution Result :

```

Enter the order (n) of the Lagrange polynomial: 2

     n      z       P(z)      f(z)       delta
-----------------------------------------------
     2  -1.750000  0.243275 0.246154  0.002879
```

A.2.4 Newton Polynomial Interpolation

Python Programme :

```
#=====================================
#       Course Notes
#     Numerical Physics
# program :
#    Newton Polynomial Interpolation
#=====================================

import numpy as np
import matplotlib.pyplot as plt

#-----------Function to interpolate
def funct(x):
    return 1/( 1 + x**2 )
#-----------Newton interpolation
def newton_interpolation(xi, yi, z, n):
    p = n+1
    #----- Divided Difference Table----
    table = np.zeros((p, p))
    for i in range(p):
        table[i, 0] = yi[i]
        for j in range(1, i+1):
```

```
            table[i, j] = (table[i, j-1] - table[i-1, j-1]) / (xi[i ] - xi[i-j])
    coef = np.diag(table)
    #----evaluate Newton polynomial order (n) at z ----
    poly = yi[0]
    for i in range(1, n+1):
        r = 1
        for j in range(i):
            r *= (z - xi[j])
        poly += r * coef[i]
    return  table , poly

#-----------Data points--------------------
xi = np.array([-2., -1.5,-1., -0.5 , 0.0 ,  0.5 , 1.0 , 1.5,  2.])
yi = np.array([0.2,0.3077, 0.5,0.80, 1., 0.8 , 0.5, 0.3077, 0.2])
z = -1.75

#-----------User input--------------------
num_points = len(xi)
n = int(input("Enter the order (n) of Newton polynomial: "))
if n >= num_points:
    print("Polynomial order is greater than number of points \n")
else:
    table, yp = newton_interpolation(xi, yi, z, n)
    yf = funct(z)
    delta = np.abs(yf - yp)
    print("========= Divided Difference Table =========")
    print("  xi         yi        D1f        D2f       D3f ...")
    for i in range(n + 1):
        row = f"{xi[i]:8.4f}"
        row += "  " + "  ".join(f"{table[i, j]:8.4f}" for j in range(i + 1))
        print(row)

    print("========= interpolation info =========")

    print("     n      z       P(z)      f(z)       delta")
    print("-------------------------------------------------")
    print(f"{n:6d}  {z:8.6f}  {yp:8.6f} {yf:8.6f}  {delta:8.6f}")

    #-------Generate data for plotting--------------------
    x = np.linspace(np.min(xi), np.max(xi), 100)
    y = []
    for w in x:
        table, poly = newton_interpolation(xi, yi, w, n)
        y.append(poly)
    #----------- Plot the results-------------------------
    plt.plot(x, y, label=f"Newton Pn(x) (order n={n})")
    plt.plot(x, funct(x), color='green', label="Function:f(x)")
    plt.scatter(xi, yi, color='red', label="Data points")
    plt.scatter(z,yp,color='purple',label=f"interpolate point {z}")
```

```
    plt.title("Newton Interpolation for f(x) = 1/(1+x^2)")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend()
    plt.grid(True)
    plt.savefig(f"plot_new_{n}.png", dpi=300)
    #plt.show()
```

Execution Result :

```

Enter the order (n) of Newton polynomial: 2
========= Divided Difference Table =========
  xi          yi         D1f         D2f        D3f ...
 -2.0000    0.2000
 -1.5000    0.3077    0.2154
 -1.0000    0.5000    0.3846    0.1692

========= interpolation info =========
     n      z       P(z)      f(z)       delta
-----------------------------------------------
     2  -1.750000  0.243275 0.246154  0.002879
```

A.3 Lab : Polynomial Approximation

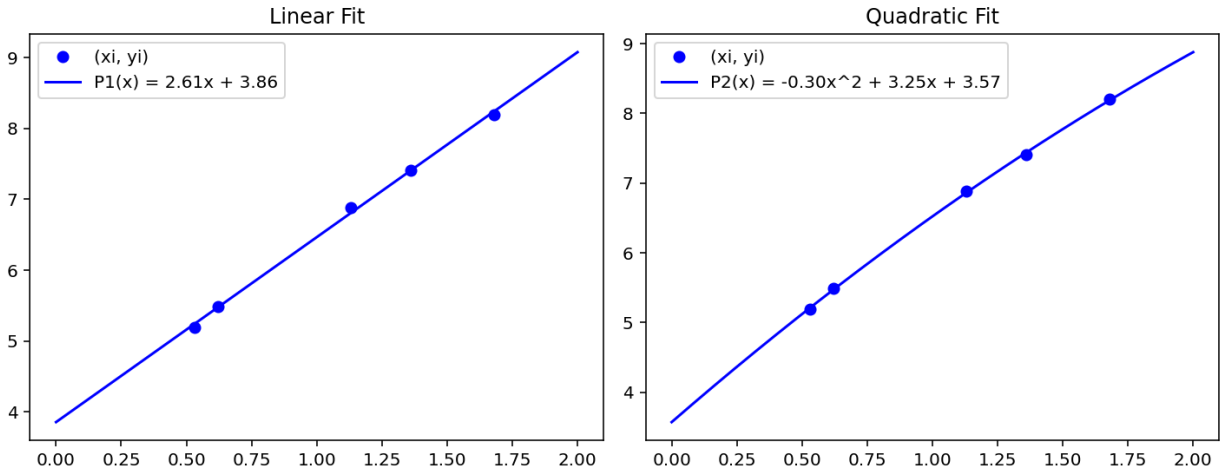
A.3.1 Discrete Case

Use Numpy's built-in function ***polyfit*** to verify the result of example (2.2.1). The least-squares approximation polynomial in Numpy is expressed as :

$$Pn = a_1 x^n + a_2 x^{n-1} + ... + a_n x + a_{n+1}$$

Python Programme :

```python
import numpy as np
import matplotlib.pyplot as plt

# Data points
xi = np.array([0.53, 0.62, 1.13, 1.36, 1.68])
yi = np.array([5.19, 5.49, 6.88, 7.41, 8.2])

# Polynomial fitting (least squares)
a1 = np.polyfit(xi, yi, 1)  # Linear fit
a2 = np.polyfit(xi, yi, 2)  # Quadratic fit
print(a1)
print(a2)
# Generate points for plotting
x = np.arange(0, 2.01, 0.01)

# Compute fitted values
y1 = np.polyval(a1, x)
y2 = np.polyval(a2, x)

# Plot results
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(xi, yi, 'bo', label='(xi, yi)')
plt.plot(x, y1, 'b', label=f'P1(x) = {a1[0]:.2f}x + {a1[1]:.2f}')
plt.legend()
plt.title('Linear Fit')

plt.subplot(1, 2, 2)
plt.plot(xi, yi, 'bo', label='(xi, yi)')
plt.plot(x, y2, 'b', label=f'P2(x) = {a2[0]:.2f}x^2 + {a2[1]:.2f}x
    ↪ + {a2[2]:.2f}')
plt.legend()
plt.title('Quadratic Fit')

plt.tight_layout()
plt.savefig("moindre_carre.png")
```



A.3.2 Continuous Case

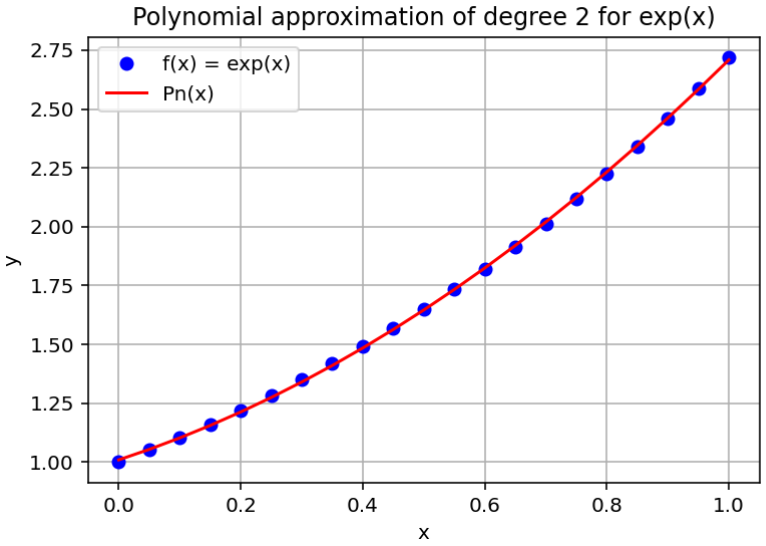
Verify the result of example (2.2.2) : **Python Programme :**

```
# ------------------------------------------------------------
# Polynomial approximation using the least squares method
# with NumPy's built-in polyfit and polyval functions
# Approximating f(x) = e^x with a polynomial of degree n
# over the interval [0, 1]
# ------------------------------------------------------------

import numpy as np
import matplotlib.pyplot as plt

# Define the function
def funct(x):
    return np.exp(x)

# Sample points
xi = np.array([0, 0.1, 0.2, 0.3, 0.6, 0.7, 0.8, 0.9, 1])
yi = funct(xi)

# Ask the user for the polynomial degree
n = int(input("Enter the degree (n) of the polynomial: "))

# Fit the polynomial of degree n
a = np.polyfit(xi, yi, n)

# Generate points for smooth plotting
t = np.arange(0, 1.05, 0.05)

# Evaluate the polynomial at points t
pn = np.polyval(a, t)

```

```
# Display results
print('\nCoefficients "a0, a1, a2, ..., an" of the polynomial of degree n:')
print('Pn(x) = a0*x^n + a1*x^(n-1) + ... + an')
print(a)

# Plot the function and its polynomial approximation
plt.plot(t, funct(t), 'bo', label='f(x) = exp(x)')
plt.plot(t, pn, 'r', label='Pn(x)')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Polynomial approximation of degree {n} for exp(x)')
plt.grid(True)
plt.savefig("mc_continu.png")
```

Execution Result :

```
Enter the degree (n) of the polynomial: 2

Coefficients "a0, a1, a2, ..., an" of the polynomial of degree n:
Pn(x) = a0*x^n + a1*x^(n-1) + ... + an
[0.85145635 0.84730751 1.00990061]
```



A.4 Lab : Numerical Solution of Differential Equations

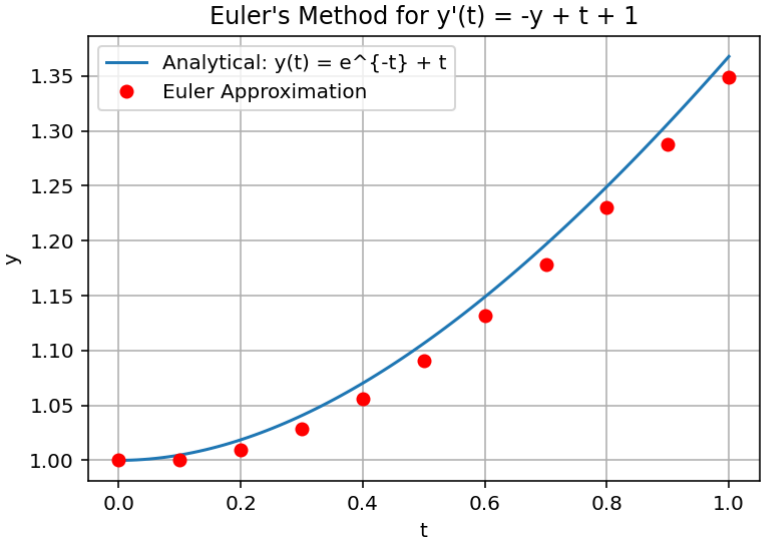
A.4.1 Euler Method

Python Programme :

```python
# ------------------------------------------------------------
# Euler method to solve a Cauchy problem (initial value problem)
# Differential equation:
#           y'(t) = f(t, y),   y(t0) = y0
# Example: f(t, y) = -y + t + 1
# Analytical solution: g(t) = exp(-t) + t
# ------------------------------------------------------------

import numpy as np
import matplotlib.pyplot as plt
# Define the differential equation
def f(t, y):
    """Differential equation y'(t) = f(t, y)."""
    return -y + t + 1
# Define the analytical (exact) solution
def g(t):
    """Exact solution y(t) = exp(-t) + t."""
    return np.exp(-t) + t

# Initial conditions
y0 = 1.0      # initial value y(t0)
t0 = 0.0      # initial time
h = 0.1       # step size
tf = 1.0      # final time

# ------------------------------------------------------------
# Time grid
t = np.arange(t0, tf + h, h)
n = len(t)
y = np.zeros(n)
y[0] = y0

# Euler method iteration
for i in range(n - 1):
    y[i + 1] = y[i] + h * f(t[i], y[i])
# ------------------------------------------------------------
# Display results
print(f"{'t':>8} {'y_i':>10} {'y(t)':>12} {'|delta|':>10}")
print('-' * 44)
for i in range(n):
    exact = g(t[i])
    error = abs(y[i] - exact)
    print(f"{t[i]:8.2f} {y[i]:10.5f} {exact:12.5f} {error:10.5f}")

```

```
# ------------------------------------------------------------
# Plot results
t_fine = np.linspace(0, 1, 100)
plt.plot(t_fine, g(t_fine), '-', label='Analytical: y(t) = e^{-t} +
    ↪  t')
plt.plot(t, y, 'ro', label='Euler Approximation')
plt.xlabel('t')
plt.ylabel('y')
plt.legend()
plt.title("Euler's Method for y'(t) = -y + t + 1")
plt.grid(True)
plt.savefig("euler.png")
```

```
        t         y_i          y(t)        |delta|
-----------------------------------------------
    0.00     1.00000       1.00000     0.00000
    0.10     1.00000       1.00484     0.00484
    0.20     1.01000       1.01873     0.00873
    0.30     1.02900       1.04082     0.01182
    0.40     1.05610       1.07032     0.01422
    0.50     1.09049       1.10653     0.01604
    0.60     1.13144       1.14881     0.01737
    0.70     1.17830       1.19659     0.01829
    0.80     1.23047       1.24933     0.01886
    0.90     1.28742       1.30657     0.01915
    1.00     1.34868       1.36788     0.01920
```



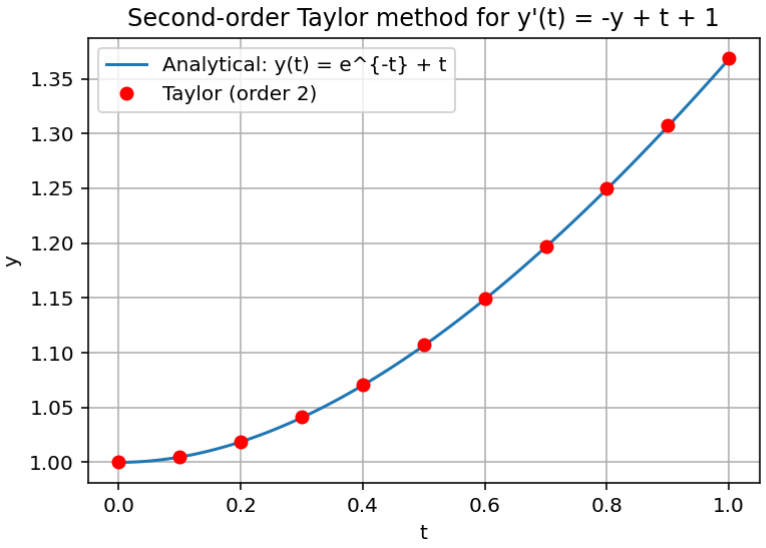
A.4.2 Taylor Method of Order 2

Python Programme :

```
# ------------------------------------------------------------
# Taylor method of order 2 for solving a Cauchy problem
# Differential equation:
#           y'(t) = f(t, y),  y(t0) = y0
# Example: f(t, y) = -y + t + 1
# Analytical solution: g(t) = exp(-t) + t
# ------------------------------------------------------------

import numpy as np
import matplotlib.pyplot as plt

# Define the differential equation
def f(t, y):
    """The function f(t, y) = y'(t)."""
    return -y + t + 1

# Partial derivatives of f
def dfdt(t, y):
    """Partial derivative df/dt."""
    return 1

def dfdy(t, y):
    """Partial derivative df/dy."""
    return -1

# Analytical solution
def g(t):
    """Exact solution y(t) = exp(-t) + t."""
    return np.exp(-t) + t
# ------------------------------------------------------------
# Initial conditions and parameters
y0 = 1.0       # initial value y(t0)
t0 = 0.0       # initial time
h = 0.1        # step size
tf = 1.0       # final time

# ------------------------------------
# Time grid and initialization
t = np.arange(t0, tf + h, h)
n = len(t)
y = np.zeros(n)
y[0] = y0

# ------------------------------------
# Taylor method of order 2
for i in range(n - 1):
    dt = dfdt(t[i], y[i])
```

```
    dy = dfdy(t[i], y[i])
    y[i + 1] = y[i] + h * f(t[i], y[i]) + (h ** 2 / 2) * (dt + dy * f(t[i], y[i]))

# ------------------------------------------------------------
# Display results
print(f"{'t':>8} {'y_i':>10} {'y(t)':>12} {'|delta|':>10}")
print('-' * 44)
for i in range(n):
    exact = g(t[i])
    error = abs(y[i] - exact)
    print(f"{t[i]:8.2f} {y[i]:10.5f} {exact:12.5f} {error:10.5f}")

# ----------------------------------------
# Plot results
t_fine = np.linspace(0, 1, 100)
plt.plot(t_fine, g(t_fine), '-', label='Analytical: y(t) = e^{-t} + t')
plt.plot(t, y, 'ro', label='Taylor (order 2)')
plt.xlabel('t')
plt.ylabel('y')
plt.legend()
plt.title("Second-order Taylor method for y'(t) = -y + t + 1")
plt.grid(True)
plt.savefig("taylor.png")
```

```
       t        y_i         y(t)       |delta|
--------------------------------------------
    0.00    1.00000      1.00000    0.00000
    0.10    1.00500      1.00484    0.00016
    0.20    1.01902      1.01873    0.00029
    0.30    1.04122      1.04082    0.00040
    0.40    1.07080      1.07032    0.00048
    0.50    1.10708      1.10653    0.00055
    0.60    1.14940      1.14881    0.00059
    0.70    1.19721      1.19659    0.00062
    0.80    1.24998      1.24933    0.00065
    0.90    1.30723      1.30657    0.00066
    1.00    1.36854      1.36788    0.00066
```



Bibliographie

[1] Ph. DEPONDT. *La boîte à outils de la PHYSIQUE NUMERIQUE*. Université Pierre et Marie Curie Paris-6 - ENS-Cachan, 2008-2009. URL : depondt@insp.jussieu.fr.

[2] Donatien N'Dri & Steven DUFOUR. *Exercices pour les cours de calcul scientifique pour ingénieurs MTH2210A*. Édition du 19 novembre 2018. École Polytechnique de Montréal, 2009.

[3] Franck JEDRZEJEWSKI. *Introduction aux méthodes numériques*. Deuxième édition. OCLC : 248181784. Paris : Springer, 2001. ISBN : 978-2-287-59711-4. URL : https://link.springer.com/content/pdf/bfm\%3A978-2-287-28199-0\%2F1.pdf.

[4] Karima MEBARKI. *Cours Analyse Numérique*. Université Abderrahmane Mira de Béjaia Faculté des sciences exactes Département de mathématiques. URL : mebarqi_karima@hotmail.fr.

[5] Mazen SAAD. *ANALYSE NUMERIQUE*. Ecole Centrale de Nantes Dépt. Info/Math, 2011-2012. URL : Mazen.Saad@ec-nantes.fr.

[6] Jean-Jacques SAMUELI. « Legendre et la méthode des moindres carrés ». In : *Bibnum [En ligne], Mathématiques* (2020). URL : http://journals.openedition.org/bibnum/580.

[7] Brian STOUT. *Méthodes numériques de résolution d'équations différentielles*. Université de Provence Institut Fresnel, Case 161 Faculté de St Jérôme Marseille, France, Fevrier 2007. URL : brian.stout@fresnel.fr.

[8] HARNANE YAMINA. *Cours Méthodes numériques I Méthodes des différences finies*. Université Larbi Ben M'Hidi - Oum-El-Bouaghi Faculté des Sciences et Sciences Appliquées Département de Génie Mécanique, 2015-2016.