Democratic and Popular Republic of Algeria

Ministry of Higher Education and Scientific Research

Mustapha Stambouli University of Mascara

Faculty of Science and Technology

Common Core Department of Science and Technology



Practical Work Handout

Numerical Analysis 02

Instructor: Dr Mokhtar Bouhalouane

This handout is intended for second-year Engineering students in Science and Technology.

Algeria

2026

Foreword

This laboratory manual has been prepared for second year engineering students to introduce them to fundamental numerical methods: polynomial interpolation, numerical integration, and the solution of differential equations. These topics are central to applied mathematics and engineering practice, as they provide the computational tools required to approximate, analyze, and solve problems which often lack an exact analytical solution. The aim of this manual is not only to present the theoretical aspects of these methods, but also to demonstrate their practical implementation through MATLAB codes. Each section is designed to bridge mathematical concepts with algorithmic procedures, allowing students to experiment, verify results, and develop a deeper understanding of the numerical approach. By engaging with these exercises, students will strengthen their analytical skills, and appreciate the role of numerical analysis in modern engineering. This manual is intended to serve as both a a practical guide and a learning companion, inviting curiosity and reflection while providing the solid foundations necessary for further studies.

Contents

Foreword 1

Introduction 3

1 Introduction to Polynomial Interpolation of Data 4

Statement . . . 4

1.1 Lagrange method . . . 4

1.2 Newton's method . . . 5

1.3 Standard Method . . . 13

1.4 Hermite Interpolation . . . 15

2 Calculation of integrals 20

2.1 Composite rectangle method . . . 21

2.1.1 Left rectangle rule . . . 21

2.1.2 Right rectangle rule . . . 21

2.1.3 Midpoint rectangle rule . . . 21

2.2 Trapezoidal rule . . . 22

2.3 Simpson's rule . . . 22

2.4 General Newton–Cotes method . . . 28

3 Numerical solution of first order ordinary differential equations 31

3.1 Explicit Euler method . . . 32

3.2 Implicit Euler method . . . 32

3.3 semi implicit Euler method . . . 33

3.4 Runge-Kutta method of Order 2 . . . 33

3.5 Runge-Kutta method of Order 4 . . . 33

References 40

Annexes 41

Introduction

Solving engineering problems described mathematically relies on numerical methods, which, through algorithms, provide approximate solutions with a certain degree of precision. Since many real world systems lead to equations that cannot be solved analytically, numerical techniques provide a practical way to obtain approximate solutions with a controlled level of accuracy. The growing technological progress in computer power, combined with significant advances in algorithmic efficiency has made these methods increasingly important. The scale of the problems that can now be solved is such that the simulation of real phenomena has become possible. Indeed, the underlying principle of algorithms used is to generate a succession of intermediate approximations $x_0, x_1, \ldots, x_n$ which converge to the exact solution as n approaches infinity. Practically, each iteration refines the previous estimate, reducing the error and improving the fidelity of the computed result. This iterative approach forms the foundation of a wide range of numerical techniques, from interpolation to numerical integration and the numerical solution of differential equations.

1 Introduction to Polynomial Interpolation of Data

As illustrated in the figure below, polynomial interpolation is a method for approximating a function known only at certain points by constructing a polynomial that passes exactly through these points. This polynomial can then be used to estimate the function within the considered interval [1] [2].

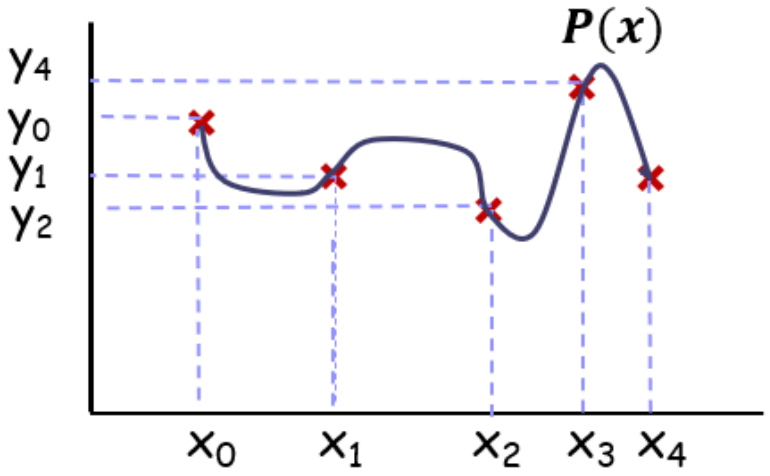


Figure 1: Interpolation principle

Statement

If we have $(n+1)$ distinct points (x_i, y_i) with $i = 0, 1, \ldots, n$ then there exists one and only one polynomial P of degree $\leq n$:

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

which passes exactly through these points, that is, satisfying:

$$\forall i = 0, \ldots, n, \quad P(x_i) = y_i.$$

Various methods exist for constructing and determining the coefficients of this interpolation polynomial [3],[4]. The two principal methods are:

1.1 Lagrange method

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

We define in MATLAB a parametric function `polylag` that takes as parameters two vectors x and y, the abscissae and respectively the ordinates of the points (x_i, y_i), and returns the characteristic polynomial P containing the coefficients of the interpolation polynomial, as well as the matrix L of characteristic vectors of the Lagrange polynomials L_i ($L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x-x_j}{x_i-x_j}$)

```
function [P,L]= polylag(x,y)
if length(x) ~= length(y)     %to ensure data correctness
error('the two vectors must be of the same size ')
elseif length(unique(x)) < length(x)
error(' the elements of the abscissa vector must be distinct')
end
% Lagrange interpolation
n=length(x);
for i=1:n
A=x;
A(i)=[];  % delete the ith element
D=1;       % the denominator of the product L
for j=1:n-1
D=D*(x(i)-A(j));
end
L(i,:)=poly(A)/D; %construction of the Lagrange polynomial Li
end
y=y(:);
P=sum(L.*y);
end
```

1.2 Newton's method

Using the same problem statement, the Newton interpolation polynomial that passes exactly through the $(n+1)$ points is given by:

$$P(x) = \sum_{i=0}^{n} a_i \prod_{j=0}^{i-1} (x - x_j)$$

which can also be written as:

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

or, using divided differences:

$$P(x) = f(x_0) + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \cdots + f[x_0, x_1, \ldots, x_n](x - x_0)\cdots(x - x_{n-1}). \quad (1)$$

Here, the divided differences $f[x_0, x_1, \ldots, x_i]$ are defined recursively as:

$$f[x_i, x_{i+1}] = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i},$$

$$f[x_0, x_1, \ldots, x_i] = \frac{f[x_1, \ldots, x_i] - f[x_0, x_1, \ldots, x_{i-1}]}{x_i - x_0}.$$

We can construct a table of divided differences. An example up to the fourth order is shown below:

x_i	$f(x_i)$	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}, x_{i+4}]$
x_0	$f(x_0)$				
x_1	$f(x_1)$	$f[x_0, x_1]$			
x_2	$f(x_2)$	$f[x_1, x_2]$	$f[x_0, x_1, x_2]$		
x_3	$f(x_3)$	$f[x_2, x_3]$	$f[x_1, x_2, x_3]$	$f[x_0, x_1, x_2, x_3]$	
x_4	$f(x_4)$	$f[x_3, x_4]$	$f[x_2, x_3, x_4]$	$f[x_1, x_2, x_3, x_4]$	$f[x_0, x_1, x_2, x_3, x_4]$

We define in MATLAB a parametric function `polynew` that takes as parameters two vectors x and y, the abscissae and ordinates of the points (x_i, y_i), and returns the characteristic polynomial P containing the coefficients of the interpolation polynomial, as well as the matrix A of divided differences.

```
function [P,A]=polynew(x,y)
if length(x) ~= length(y)     % to ensure data correctness
error('the two vectors must be of the same size ')
elseif length(unique(x)) < length(x)
error(' the elements of the abscissa vector must be distinct')
end
n=length(x);
% Computation of the divided difference matrix
A=zeros(n,n);
A(1:end,1)=y';
l=0;
for j=2:n
l=l+1;
for i=j:n
A(i,j)=(A(i,j-1)-A(i-1,j-1))/(x(i)-x(i-l));
end
end
alpha=diag(A);
% Computation of the  interpolation polynomial coefficients
P=alpha(end);
for i=n-1:-1:1
P=[P  alpha(i)]-[0  P*x(i)];   %Coefficients construction
end
end
```

We now move on to a practical application of these two parametric functions. Let the interpolation support consist of the points with coordinates $(-1, 3)$, $(0, -2)$, and $(3, 7)$. We will determine the coefficients of the Lagrange interpolation polynomial by calling the function `polylag`. Then, we will plot on the same figure the graph of this polynomial together with the interpolation support points. To do so, we will create a MATLAB script called `Laginterp`, in which we define the vector of abscissae x and the vector of ordinates y. This script will then call the function `polylag`, which implements the Lagrange interpolation method.

Script Laginterp

```
close all;clear all; clc;
x=[-1;0;3];
Y= [3;-2;7];
[p, L]=polylag(x,y)
plot(x,y,'*r','MarkerSize',10)  %Plotting interpolation points
a=-10:10;
b=polyval(p,a);
hold on
plot(a,b)         %Plotting the curve of this polynomial
grid
```

The results are as follows, giving the interpolation polynomial and, respectively, the Lagrange polynomials with their coefficients arranged in decreasing order

```
p =
2                  -3              -2
L =
1/4            -3/4           0
-1/3            2/3           1
1/12            1/12          0
```

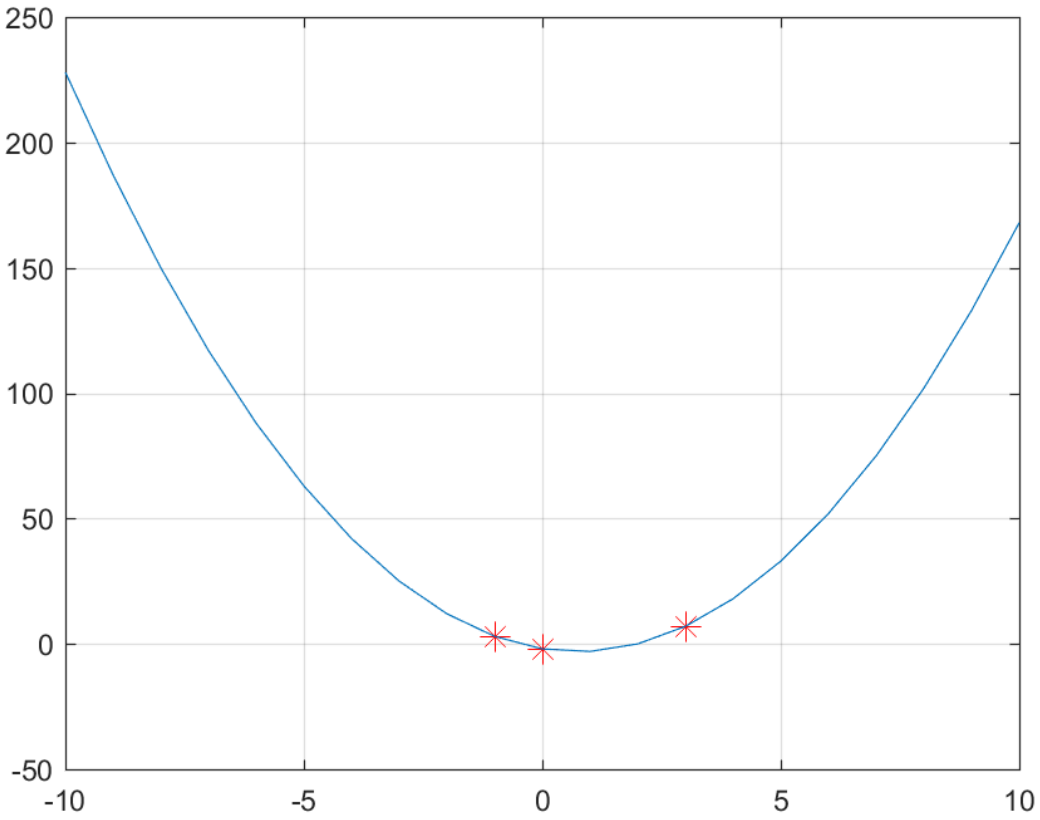


Figure 2 : Lagrange interpolation

In the same way, we will test the implemented Newton interpolation method. Let the interpolation support consist of the following points:
$(-2, -74)$, $(0.2, 4.3411)$, $(-1, 0)$,
$(1, 10)$, $(6, 5950)$, $(2, 42)$.

Script Newinterp

```
close all;clear all;clc;
x=[-2;0.2;-1;1;6;2];
y=[-74;4.3411;0;10;5950;42];
[p,A]=polynew(x,y)
plot(x,y,'*r','MarkerSize',10)   %Plotting the interpolation points
a=-10:0.1:10;
b=polyval(p,a);
hold on
plot(a,b,'LineWidth',1)
grid
```

The results are as follows, giving the interpolation polynomial with its coefficients arranged in decreasing order, and the divided difference table:

```
p =
1        -367489/183744  661478/220493  110248/36749   275617/275616
183743/45936

A =
-74          0           0           0         0      0
2049/472   5199/146      0           0         0      0
0          1646/455   -3999/125      0         0      0
10           5         629/364     281/25      0      0
5950       1188         169        721/25     11/5    0
42         1477         289          40       31/5    1
```

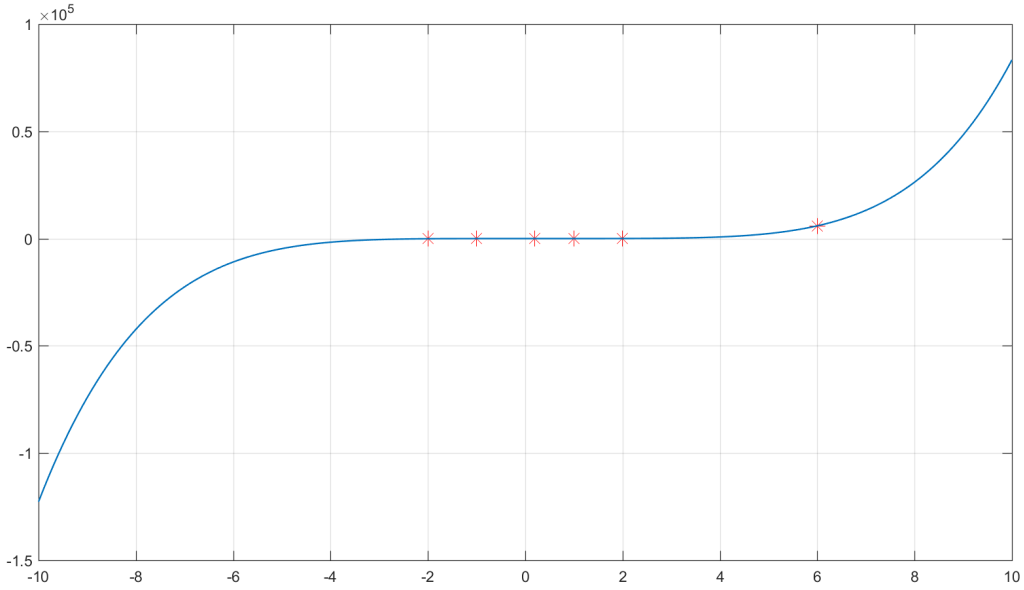


Figure 3 : Newton interpolation

We now attempt to interpolate the function $\ln(x)$ at the points $1, 2, \ldots, 8$, and plot the corresponding curves. The results are given as follow:

```
close all;clear all;clc;
f=@(x)log(x);
x=1:8;          %definition of the vector of abscissas
y=f(x);         %computation of the vector of ordinates
[p,A]=polynew(x,y)
plot(x,y,'*r','MarkerSize',10)  %Plotting interpolation points
a=0.5:0.1:10;
b=polyval(p,a);
y2=f(a);
hold on
plot(a,b,'g',a,y2,'b','LineWidth',1)
grid
```

The results are given as follow:

```
p =
0.0000   -0.0004    0.0063   -0.0543    0.2842   -0.9484    2.1920
-1.4793

A =
0         0         0         0         0         0         0         0
0.6931    0.6931    0         0         0         0         0         0
1.0986    0.4055   -0.1438    0         0         0         0         0
1.3863    0.2877   -0.0589    0.0283    0         0         0         0
1.6094    0.2231   -0.0323    0.0089   -0.0049    0         0         0
1.7918    0.1823   -0.0204    0.0040   -0.0012    0.0007    0         0
1.9459    0.1542   -0.0141    0.0021   -0.0005    0.0002   -0.0001    0
2.0794    0.1335   -0.0103    0.0013   -0.0002    0.0000   -0.0000    0.0000
```

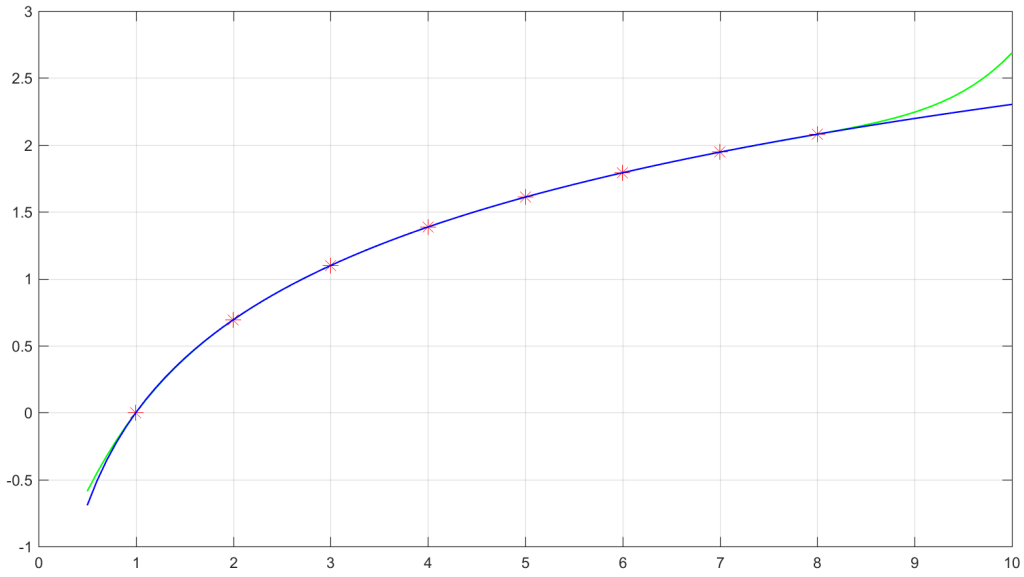


Figure 4 : the Newton interpolation polynomial(in green) and the function (in blue)

In addition to these two main methods ,it is also possible to determine the interpolation polynomial using a standard approach, which consists in solving a linear system of $(n+1)$ equations, corresponding to the number of interpolation points:

1.3 Standard Method

Statement

Let us consider the same setting with the (N+1) interpolation points (x_i, y_i) with $i = 0, 1, \ldots, n$. We then obtain an interpolation polynomial P of degree at most n:

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Satisfying: $\forall i = 0, \ldots, n \quad P(x_i) = y_i \Rightarrow$

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

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

$$\Rightarrow X * A = Y \Rightarrow A = X^{-1} * Y$$

Hence, the solution consists in determining the vector A of coefficients a_i, with X a Vandermonde square matrix of dimension $(N+1)\times(N+1)$. We define in MATLAB a parametric function "standardVan" that takes as input two vectors, x and y, representing respectively the abscissae and ordinates of the interpolation points (x_i, y_i). The function constructs the corresponding Vandermonde matrix, solves the associated linear system, and returns the characteristic vector P containing the coefficients of the interpolation polynomialz

```
function P=standardVan(x,y)
if length(x) ~= length(y)      % to ensure data correctness
error('the two vectors must be of the same size ')
elseif length(unique(x)) < length(x)
error(' the elements of the abscissa vector must be distinct')
end
x = x(:);
%construction of the Vandermonde matrix
V=ones(length(x),length(x));
for k=1:length(x)-1
V(:,k+1)= x.^k;
end
y=y(:);
P=V\y;   %Vector solution
P=flip(P) %interpolation polynomial
end
```

We now return to the same example used previously for the Lagrange method, this time by calling the "standardVan" function within the same script `Newinterp` :

Script stvaninterp

```
close all;clear all;clc;
x=[-1;0;3];
y=[3;-2;7];
p=standardVan(x,y)
plot(x,y,'*r','MarkerSize',10)  %Plotting interpolation points
a=-10:10;
b=polyval(p,a);
hold on
plot(a,b)
grid
```

The results give the same interpolation polynomial and the same plot, as shown below:

```
P =
2
-3
-2
```

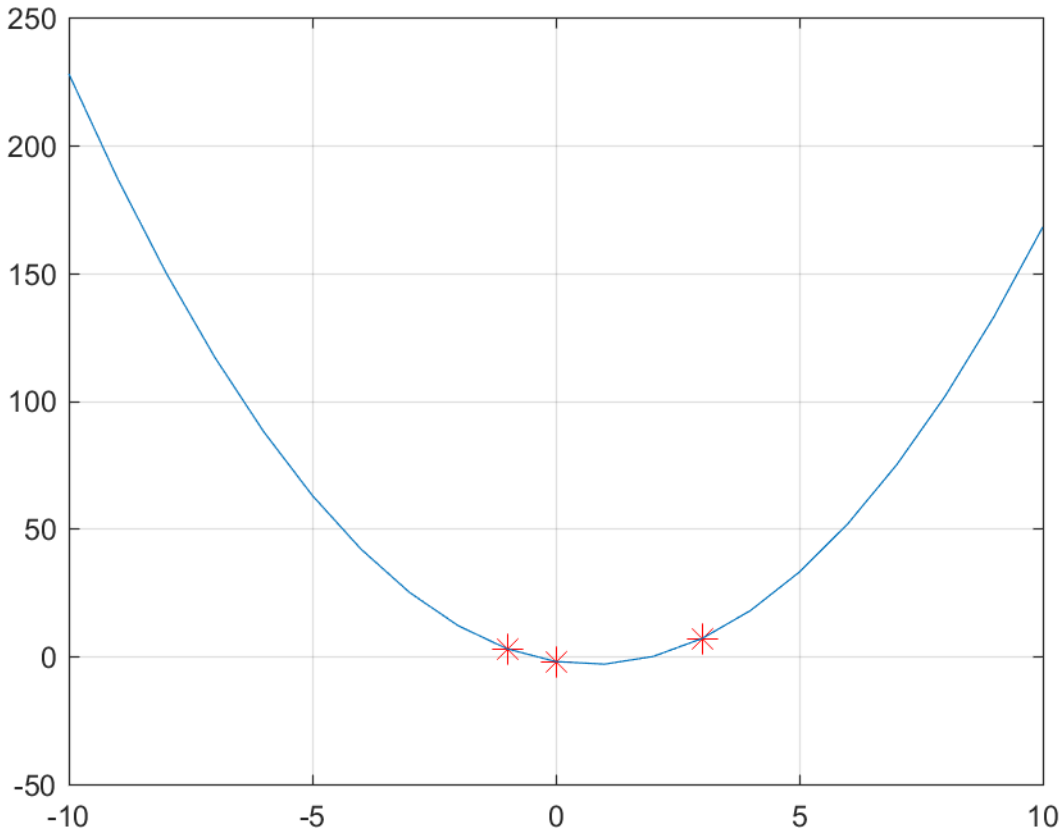


Figure 5 : Standard method

1.4 Hermite Interpolation

Hermite interpolation consists in determining a polynomial that not only interpolates the values of a function at given points, but also interpolates its derivative at those same points. The method is based on the Lagrange basis polynomials $L_i(x)$:

Statement

If we have (n+1) distinct points (x_i, y_i) with $i = 0, 1, \ldots, n$, and (n+1) derivative points $l_i(x)$ with $i = 0, 1, \ldots, n$, then there exists one and only one polynomial P of degree n :

$$p(x) = \sum_{i=0}^{n} \left(y_i \, H_i(x) + z_i \, K_i(x) \right)$$

such that:

$$\begin{cases} H_i(x) = L_i^2(x) \, \left(1 - 2L_i'(x_i) \, (x - x_i) \right), \\ K_i(x) = L_i^2(x) \, (x - x_i) \end{cases}$$

satisfying:

$$\forall i = 0, \ldots, n, \quad \begin{cases} P(x_i) = y_i, \\ P'(x_i) = z_i \end{cases}$$

To implement this method in MATLAB, we define a parametric function `polyHer` that uses the Symbolic Math Toolbox and takes as input three vectors x, y, and z, representing respectively the abscissae, the ordinates, and the derivative values. The function returns the Hermite interpolation polynomial.

```
function P=polyHer(x,y,z)
if (length(x) ~= length(y))||(length(x) ~= length(z))
error('the two vectors must be of the same size ')
elseif length(unique(x)) < length(x)
error(' the elements of the abscissa vector must be distinct')
end
syms t c w;
f=(t-c)/(w-c);
%construction of the Lagrange polynomials
L=subs(f,w,x);
for i=1:length(x)
A=x;
A(i)=[];
L(i)=prod(subs(L(i),c,A));
end
%construction of the Hermite polynomials
for i=1:length(x)
D=subs(diff(L(i)),t,x(i));
H(i)=y(i).*L(i).^2.*(1-2*D*(t-c))+z(i)*L(i).^2.*(t-c);
P(i)=subs(H(i),c,x(i));
end
P=sum(P);
end
```

As a practical application of this method, consider the function:

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

Using a MATLAB script `Herinterp`, which calls the function `polyHer`, we determine the Hermite interpolation polynomial that passes exactly through the support formed by the points of the antiderivative of f at the abscissae $-1, 0$, and 6. We then plot, on the same figure, the graph of this polynomial over the interval $[-1, 6]$, together with the graph of its derivative, while highlighting the interpolation points.

Script Herinterp

```
clear all; clc; close all;
f=@(x) x./sqrt(x.^2+1)-2*sin(2*x);
primf=@(x) sqrt(x.^2+1)+cos(2*x);    % the antiderivative of f
x=[-1 3 6];
y=primf(x);
yp=f(x);
syms t
P=polyHer(x,y,yp)
P=simplify(P)
x1=-1:0.01:6;
y1=subs(P,t,x1);
y2=subs(diff(P),t,x1);
plot(x,y,'K*',x,yp,'r+','MarkerSize',10,'Linewidth',1.01)
hold on
plot(x1,y1,'b',x1,y2,'g','Linewidth',1)  % interpolation polynomial
and its derivative
grid
```

The results yield the following degree 5 interpolation polynomial $P(t)$:

```
P =
(546438891076034197*t^5)/74147264265027846144 - (2242584311240714927*t
^4)/24715754755009282048 + (7324197660619646325*t^3)
/24715754755009282048 + (4619350189893353581*t^2)/74147264265027846144
 - (5816197796999599*t)/110338190870577152 +
7893572247594004031/6178938688752320512
```

The following figure shows the curve of this polynomial in blue, the curve of its derivative in green, as well as the interpolation points of the derivative in red and those of the antiderivative in black.

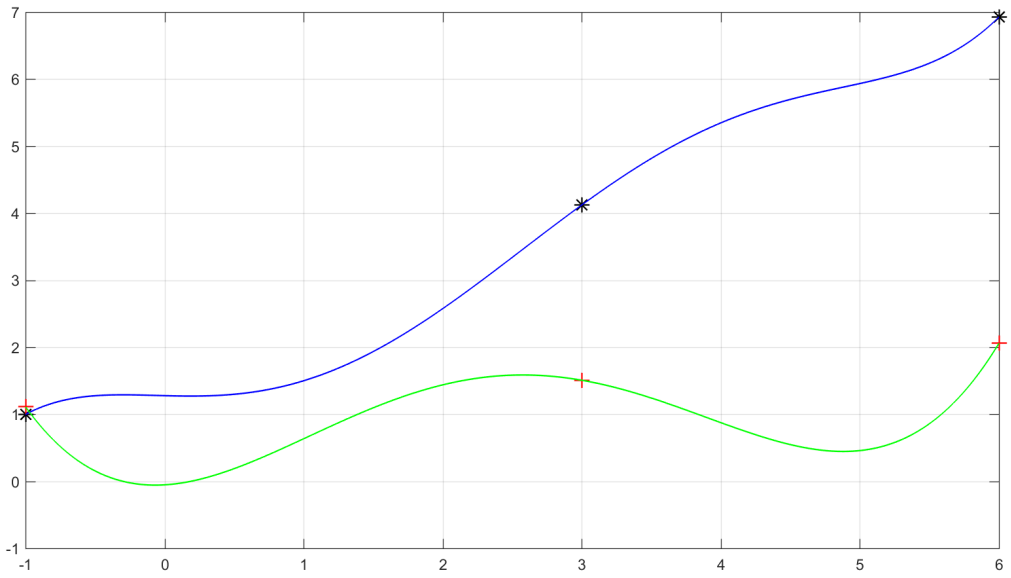


Figure 6 : Interpolation polynomial(in blue) and its derivative(in green)

If we take again the same simple Lagrange interpolation example and add the derivative points $(-1, 1), (0, 2)$, and $(3, 1)$, we obtain the corresponding Hermite interpolation polynomial and the following plots (fig7), as shown after the associated script:

```
clear all; clc; close all;
x=[-1;0;3];
y=[3;-2;7];
yp=[1;2;1];
syms t
P=polyHer(x,y,yp)
P=simplify(P)
x1=-2:0.1:4;
y1=subs(P,t,x1);
y2=subs(diff(P),t,x1);
plot(x,y,'K*',x,yp,'r+','MarkerSize',10,'Linewidth',1.01)
hold on
plot(x1,y1,'b',x1,y2,'g','Linewidth',1)  % interpolation polynomial
and its derivative
grid
```

Results :

```
P =
t^5 - (14*t^4)/3 + (2*t^3)/3 + (40*t^2)/3 + 2*t - 2
```

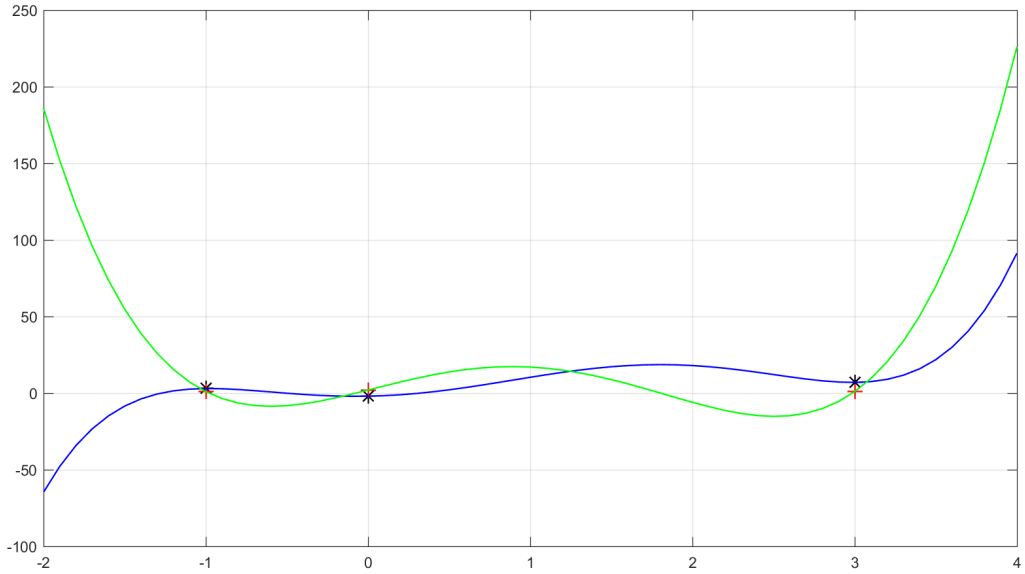


Figure 7: Interpolation polynomial(in blue) and its derivative(in green)

2 Calculation of integrals

In this chapter, we study a numerical integration technique from the class of elementary quadrature methods, specifically the Newton–Cotes method, which is based on approximating a function by an interpolation polynomial on a prescribed interval. The principle of this method can be summarized as follows [5],[4]:

1. the integration interval $[a, b]$ is divided into n subintervals,
2. on each subinterval, the function f to be integrated is approximated by a low-degree polynomial (degree $0, 1, 2, \ldots$),
3. the areas under these polynomials are then summed (as illustrated in the figure below).

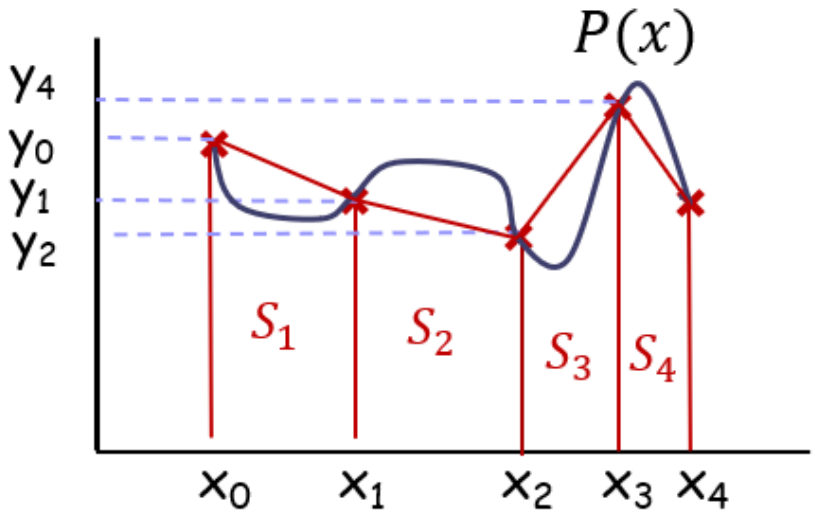


Figure 8 : Example of the composite trapezoidal rule

Under this principle of decomposing an interval into several subintervals, the method is referred to as composite.

2.1 Composite rectangle method

it consists in approximating the function f on each subinterval by a polynomial of degree zero

2.1.1 Left rectangle rule

$$I \approx \sum_{i=0}^{n-1} (x_{i+1} - x_i)\, f(x_i) = \sum_{i=0}^{n-1} h_i\, f(x_i).$$

2.1.2 Right rectangle rule

$$I \approx \sum_{i=1}^{n} (x_{i+1} - x_i)\, f(x_i) = \sum_{i=1}^{n} h_i\, f(x_i).$$

2.1.3 Midpoint rectangle rule

$$I \approx \sum_{i=0}^{n-1} (x_{i+1} - x_i)\, f\left(\frac{x_i + x_{i+1}}{2}\right) = \sum_{i=0}^{n-1} h_i\, f\left(\frac{x_i + x_{i+1}}{2}\right).$$

2.2 Trapezoidal rule

It consists in approximating the function f on each subinterval by a polynomial of degree 1

$$I \approx \sum_{i=0}^{n-1} (x_{i+1} - x_i) \frac{f(x_i) + f(x_{i+1})}{2} = \sum_{i=0}^{n-1} h_i \frac{f(x_i) + f(x_{i+1})}{2}.$$

2.3 Simpson's rule

This method applies to an even number of subintervals (i.e., an even n).It consists in approximating the function f on each subinterval by a polynomial of degree 2. Every three successive points define a polynomial of degree 2, whose integral is computed between x_{i-1} and x_{i+1}

$$\begin{aligned} I &\approx \sum_{\substack{i=1 \\ i \text{ impair}}}^{n-1} \int_{x_{i-1}}^{x_{i+1}} P_i(x)\, dx \\ &= \sum_{\substack{i=1 \\ i \text{ impair}}}^{n-1} \frac{x_{i+1} - x_{i-1}}{6} \left[f(x_{i-1}) + 4f(x_i) + f(x_{i+1}) \right] \\ &= \sum_{\substack{i=1 \\ i \text{ impair}}}^{n-1} \frac{h_i}{6} \left[f(x_{i-1}) + 4f(x_i) + f(x_{i+1}) \right]. \end{aligned}$$

In implementing these two latter techniques (trapezoidal and Simpson's rules) in MATLAB, we will make use of the two parametric functions already implemented in the previous chapter, corresponding to Newton and Lagrange interpolation functions. We will define a parametric function `intRT` implementing the left, right, midpoint rectangle rules, and trapezoidal rule. It takes as input three vectors: a vector of abscissa points defining the subintervals, the corresponding ordinate values, and the vector of midpoint ordinates for the different subintervals, and returns the result of each method.

```
function [IT,IRG,IRD,IMP]=intRT(x,y,ym)
%initialization
IT=0;          %Trapezoidal integral
IRG=0;         %Left rectangle integral
IRD=0;         %Right rectangle integral
IMP=0;         %Midpoint rectangle integral
for i=1: length(x)-1
h=x(i+1)-x(i);
IT=IT+h*(y(i+1)+y(i));
IRG=IRG+h*y(i);
IRD=IRD+h*y(i+1);
IMP=IMP+h*ym(i);
end
IT=IT/2;
end
```

We will define a MATLAB preprocessing function `prep`, that takes as input a function f, the start and end of the integration interval, and the number of subintervals to be constructed. It returns a vector of equally spaced abscissa points, a vector of their corresponding ordinates, and a vector of the midpoint ordinates.

```
function [x,y,ym]=prep(f,start,final,n)
x=linspace(start,final,n+1);
h=(final-start)/n;          %the subinterval step size
y=f(x);
xm=start+h/2:h:final-h/2;    %midpoints
ym=f(xm);
end
```

As a practical application of these methods, let us consider again the same function f as before and attempt to approximate its integral over the interval $[3/2, 3]$. We will discretize this interval using $21, 100$, and 1001 points, and apply the different numerical integration methods: left rectangle, right rectangle, midpoint, and the trapezoidal rule. To do so, we

will define a script named `integration`:

Script integration

```
clear all; clc; close all;
f=@(x) x./sqrt(x.^2+1)-2*sin(2*x);
[x,y,ym]=prep(f,3/2,3,20);
[it,irg,ird,imp]=intRT(x,y,ym);
iexact=integral(f,3/2,3);     %exact value of the integral
etr=abs(iexact-it);         %trapezoidal rule error
erg=abs(iexact-irg);        %left rectangle rule error
erd=abs(iexact-ird);        %right rectangle rule error
emp=abs(iexact-imp);        %midpoint rule error
fprintf('Left Rectangle: %.6f (error is: %.6f)\n',irg,erg);
fprintf('Right Rectangle: %.6f (error is: %.6f)\n',irg,erd);
fprintf('Midpoint : %.6f (error is: %.6f)\n',imp,emp);
fprintf('trapezoidal: %.6f (error is: %.6f)\n',it,etr);
plot(x,y,'k*');
hold on;
fplot(f,[3/2 3])
```

the results show:

```
Left Rectangle: 3.270028 (error is: 0.039637)
Right Rectangle: 3.270028 (error is: 0.032191)
Midpoint : 3.311527 (error is: 0.001862)
trapezoidal: 3.305942 (error is: 0.003723)
```

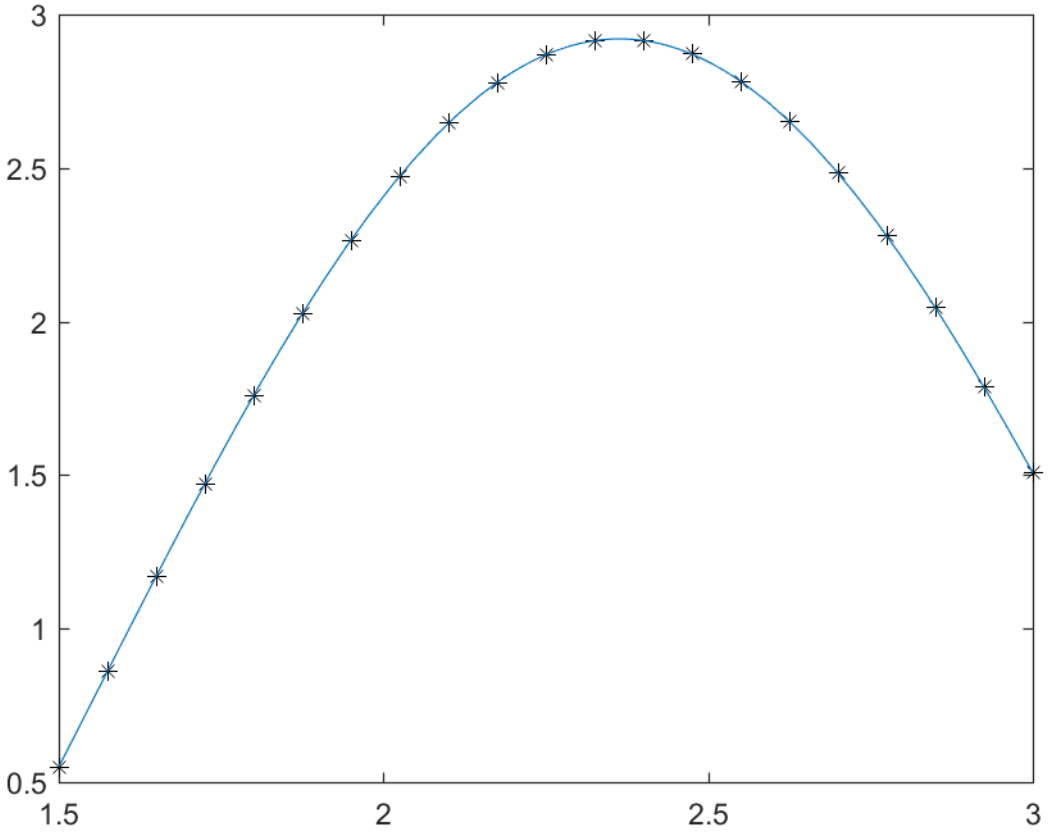


Figure 9 : the curve of the function f and the points of subintervals

Simpson's integration method applies only when the number of subintervals is even. We will define a parametric function `simpsonint`, that implements Simpson's integration rule. It takes as input two vectors: a vector of abscissae corresponding to an even number of subintervals (not necessarily equally spaced), and a vector of their ordinates, and returns the resulting value of the integral. this function will call the `polynew` function implemented earlier.

```
function S=simpsonint(x,y)
%Simpson's integration rule
S=0;           %initialization
if mod(length(x)-1,2)~=0
error('give an even number of subintervals')
end
for i=1:2:length(x)-2
[P,V]=polynew(x(i:i+2),y(i:i+2)); %interpolation polynomial
pint=polyint(P);           % the antiderivative polynomial
S=S+polyval(pint,x(i+2))-polyval(pint,x(i));

end
end
```

As a practical application of this method, we will approximate the integral of the antiderivative of f over the 21 points of the previous interval $[3/2, 3]$, and then over the interval $[-1, 6]$ using 701 equally spaced points, while computing the approximation errors and comparing them with those obtained from the other methods. So, we will define a script named `integration2` as follows:

Script integration2

```
clear all; clc; close all;
f=@(x)x./sqrt(x.^2+1)-2*sin(2*x);
fprintf('for the 21 points in [3/2 3]:\n');
[x,y,ym]=prep(f,3/2,3,20);
s=simpsonint(x,y);
iexact=integral(f,3/2,3);
es=abs(iexact-s);                    % error
fprintf('simpson''s integration rule: %.6f (error is: %.6f)\n',s,es);
fprintf('for the 701 points in [-1 6]:\n');
[x,y,ym]=prep(f,-1,6,700);
s=simpsonint(x,y);
iexact=integral(f,-1,6);
es=abs(iexact-s);                     % error
fprintf('simpson''s integration rule: %.6f (error is: %.9f)\n',s,es);
plot(x,y,'k*');
hold on;
fplot(f,[-1 6])
```

the results show:

```
for the 21 points in [3/2 3]:
simpson's integration rule: 3.309670 (error is: 0.000005)
for the 701 points in [-1 6]:
simpson's integration rule: 5.928550 (error is: 0.000000001)
```

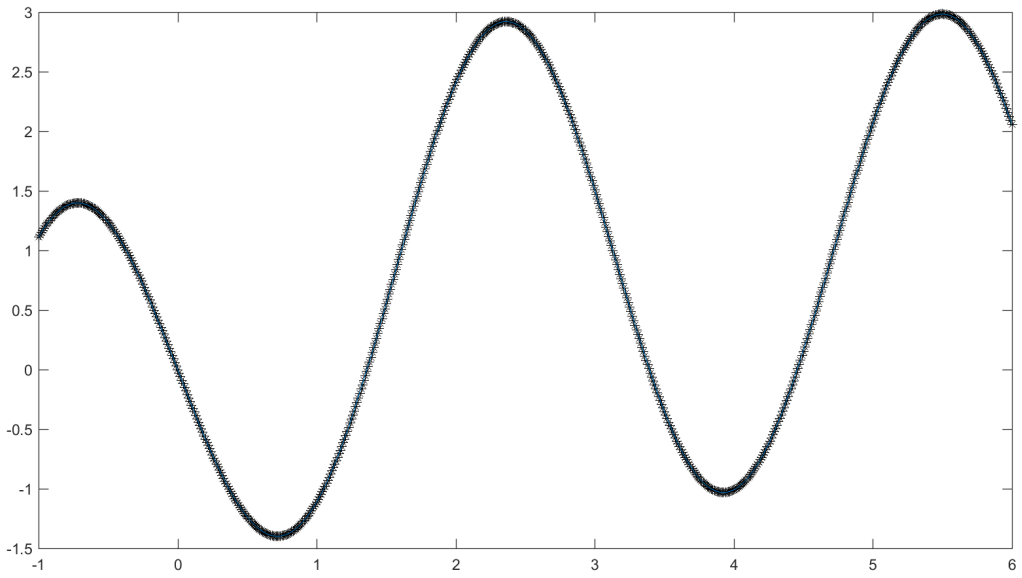


Figure 10 : the curve of the function f and the 701 points of interpolation

We note that, for the 21 points, Simpson's method produces the smallest error among all the methods, and is therefore the most accurate. We will now try to approximate the integral of f, first over the interval $[3/2, 3]$ and then over $[-1, 6]$, using the Lagrange interpolation method instead of Newton's method. To do this, it is sufficient to replace the call to the function `polynew` with "polylag" (line 8: $[P, V] = polylag(x(i : i + 2), y(i : i + 2));$) in the function `simpsonint` When we run the `integration2` script again, the results are exactly the same as those obtained previously:

```
for the 21 points in [3/2 3]:
simpson's integration rule: 3.309670 (error is: 0.000005)
for the 701 points in [-1 6]:
simpson's integration rule: 5.928550 (error is: 0.000000001)
```

2.4 General Newton–Cotes method

We will implement the general Newton–Cotes method as a spline type construction, based on grouping the subinterval points into sets of n interpolation points each. We define a parametric function, `NewtonCotes`, which takes as input a vector of abscissae and a vector

of ordinates together with the number n of points in each interpolation support. It returns the value of the integral of the function over the interval composed of all these points.

NB: for a fully practical and reliable implementation, the total number of points in the integration interval minus one must be a multiple of $n - 1$

```
function S=NewtonCotes(x,y,n)
%general Newton-Cotes integral computed over groups of points
S=0;
R=1:n-1:length(x);
i=1;
while i<length(R)
x1=x(R(i):R(i+1));
y1=y(R(i):R(i+1));
[P,V]=polynew(x1,y1);
pint=polyint(P);
S=S+polyval(pint,max(x1))-polyval(pint,min(x1));
i=i+1;
end
if R(i)~=length(x)
fprintf('The number of points is not consistent with the number of
interpolation supports !!!\n');
x1=x(R(i):end);
y1=y(R(i):end);
[P,V]=polynew(x1,y1);
pint=polyint(P);
S=S+polyval(pint,max(x1))-polyval(pint,min(x1));
end
end
```

As a practical example, if we take the same previous case, the function f on the interval $[-1, 6]$ with 31 equally spaced points and attempt to group them into sets of 4 points, each forming a different interpolation support. Then, we repeat the process using, for instance, 101 points over the same interval, grouped into sets of 5 points.

Script of newton-cotes

```
clear all; clc; close all;
f=@(x)x./sqrt(x.^2+1)-2*sin(2*x);
fprintf('for the 31 points in [-1 6]:\n');
[x,y,ym]=prep(f,-1,6,30);
s=NewtonCotes(x,y,4);
iexact=integral(f,-1,6);
es=abs(iexact-s);    % error
fprintf('Newton-Cotes''s integration method(sets of 4 points): %.6f (
error is: %.9f)\n',s,es);
fprintf('for the 101 points in [-1 6]:\n');
[x,y,ym]=prep(f,-1,6,100);
s=NewtonCotes(x,y,5);
%iexact=integral(f,-1,6);
es=abs(iexact-s);    % error
fprintf('Newton-Cotes''s integration method(sets of 5 points): %.6f (
error is: %.9f)\n',s,es);
plot(x,y,'k*');
hold on;
fplot(f,[-1 6])
```

the results show:

```
for the 31 points in [-1 6]:
Newton-Cotes's integration method(sets of 4 points): 5.929319 (error
is: 0.000769431)
for the 101 points in [-1 6]:
Newton-Cotes's integration method(sets of 5 points): 5.928550 (error
is: 0.000000020)
```

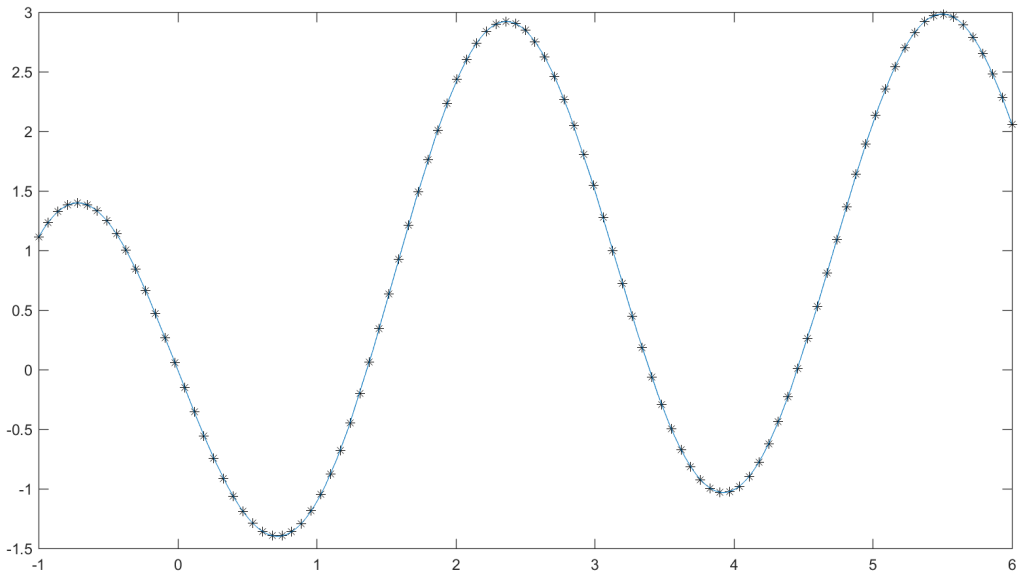


Figure 11 : the 101 points of interpolation

3 Numerical solution of first order ordinary differential equations

In this chapter, we focus on first order differential equations [6][7] of the form shown below, whose numerical solution with a single step method relies on the numerical integration techniques introduced and implemented in the previous chapter:

$$\begin{cases} \dfrac{dy(t)}{dt} = f(t, y(t)) \quad \text{with } t \in [t_0, t_f] \\ y(t_0) = y_0 \end{cases}$$

Solving this problem numerically consists in computing approximate values of $y(t)$ at different values of t $(t_0, t_1, \ldots, t_f)$, using a step size $h = t_{i+1} - t_i$ for $i = 0, 1, \ldots, n-1$.

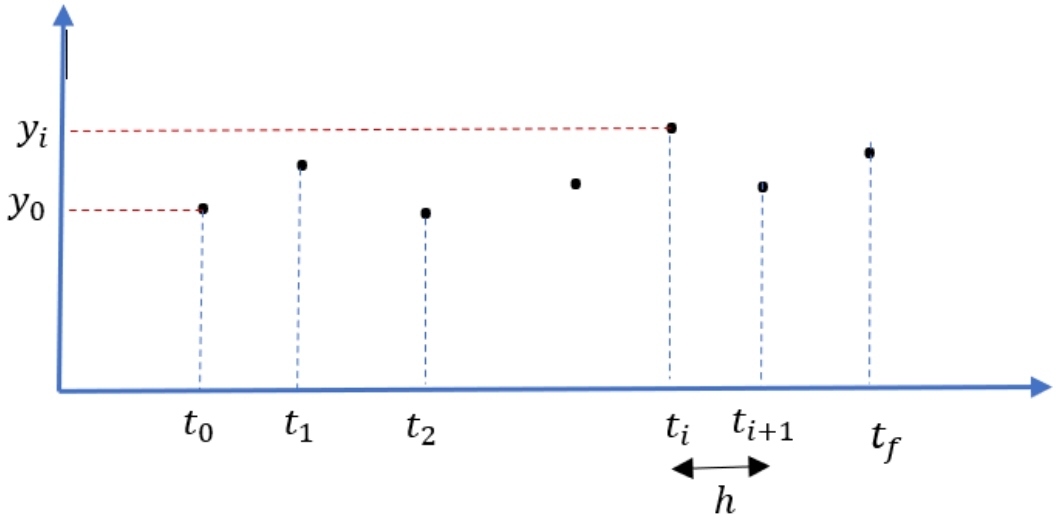


Figure 12 : A few points to approximate

On each subinterval $[t_i, t_{i+1}]$ we have, for $0 \leq i \leq n-1$:

$$\frac{dy(t)}{dt} = f(t, y(t))$$

$$\Longrightarrow \int_{t_i}^{t_{i+1}} \frac{dy(t)}{dt}\, dt = \int_{t_i}^{t_{i+1}} f(t, y(t))\, dt$$

$$\Longrightarrow \big[\, y(t) \,\big]_{t_i}^{t_{i+1}} = \int_{t_i}^{t_{i+1}} f(t, y(t))\, dt$$

$$\Longrightarrow y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y(t))\, dt$$

Knowing $y(t_0)$, various one-step methods described in the literature can be used to approximate $y(t_{i+1})$, depending on the numerical integration technique employed to approximate $\displaystyle\int_{t_i}^{t_{i+1}} f(t, y(t))\, dt$.

3.1 Explicit Euler method

$$y(t_{i+1}) = y(t_i) + h\, f(t_i, y(t_i))$$

3.2 Implicit Euler method

$$y(t_{i+1}) = y(t_i) + h\, f(t_{i+1}, y(t_{i+1}))$$

3.3 semi implicit Euler method

$$y(t_{i+1}) = y(t_i) + h\, f\left(\frac{t_{i+1}+t_i}{2},\, y\left(\frac{t_{i+1}+t_i}{2}\right)\right)$$

$$y(t_{i+1}) = y(t_i) + h\, f\left(t_i + \frac{h}{2},\, y(t_i) + \frac{h}{2} y(t_i)\right)$$

3.4 Runge-Kutta method of Order 2

$$y(t_{i+1}) \approx y(t_i) + \frac{h}{2}(k_1 + k_2), \quad i = 0, 1, \ldots, n-1$$

With:

$$\begin{aligned} k_1 &= f(t_i, y(t_i)) \\ k_2 &= f(t_i + h,\, y(t_i) + h \cdot k_1) \end{aligned}$$

3.5 Runge-Kutta method of Order 4

$$y(t_{i+1}) \approx y(t_i) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \quad i = 0, 1, \ldots, n-1$$

With:

$$\begin{aligned} k_1 &= f(t_i, y(t_i)), \\ k_2 &= f\left(t_i + \tfrac{h}{2},\, y(t_i) + \tfrac{h}{2} k_1\right), \\ k_3 &= f\left(t_i + \tfrac{h}{2},\, y(t_i) + \tfrac{h}{2} k_2\right), \\ k_4 &= f(t_i + h,\, y(t_i) + h \cdot k_3) \end{aligned}$$

We define a parametric MATLAB function, `Eulerfun`, which takes as input the function $f(t, y(t))$, the two endpoints of the integration interval t_0 and t_f, the step size h, and the initial condition $y(t_0)$. It returns a vector of abscissae together with the corresponding ordinate vectors computed using the explicit Euler method, the implicit Euler method, and the semi implicit Euler method.

```
function [t,ye,yi,ysi] = Eulerfun(t0,tf,y0,h,f)
%f is a two dimensional function
t=t0:h:tf;
n=length(t);           % the number of evaluation points
ye=zeros(1,n);
yi=zeros(1,n);
ysi=zeros(1,n);
ye(1)=y0;                % initialization
yi(1)=y0;
ysi(1)=y0;
for i=1:n-1
ye(i+1)=ye(i)+h*f(t(i),ye(i));             % explicit Euler
yi(i+1)=yi(i)+h*f(t(i),yi(i));               % implicit Euler
% semi implicit Euler
ysi(i+1)=ysi(i)+h*f((t(i)+t(i+1))/2,(ysi(i)+h/2*ysi(i)));
end
end
```

We aim to obtain an approximation of the solution of the differential equation below using the three methods, and then plot the curves of their corresponding approximate solutions

$$\begin{cases} \frac{dy}{dx} = -\frac{\sin(x)}{2} - y \quad \text{sur } [0,3] \\ y(0) = 1 \end{cases}$$

To this end, we create a script named `diffsol` in which we define the function $f(t, y(t))$, and which calls the function

Script diffsol

```
clear all; clc; close all;
f=@(t,y)-sin(t)/2-y;
t0=0;
tf=3;
y0=1;
h=0.1;
[t,ye,yi,ysi]=Eulerfun(t0,tf,y0,h,f);
plot(t,ye,'b*',t,yi,'r*',t,ysi,'g*','Linewidth',1)
%exact solution
g=matlabFunction(dsolve('Dy=-sin(t)/2-y','y(0)=1','t'))
y2=g(t);
hold on;
plot(t,y2,'k','Linewidth',1)
```

The simulation yields the following results for a step size $h = 0.1$ where the approximate solutions are shown in different colors, while the curve of the exact solution is plotted in black:

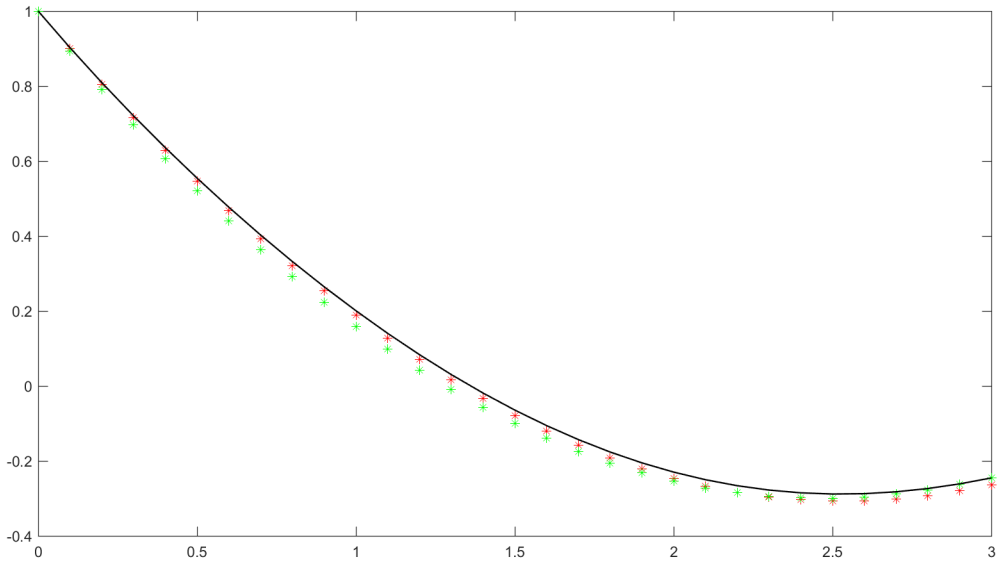


Figure 13 : Explicit Euler(blue). implicit Euler (red). semi-implicit Euler(green). Exact(black)

By reducing the step size to $h = 0.01$, the following results are obtained:

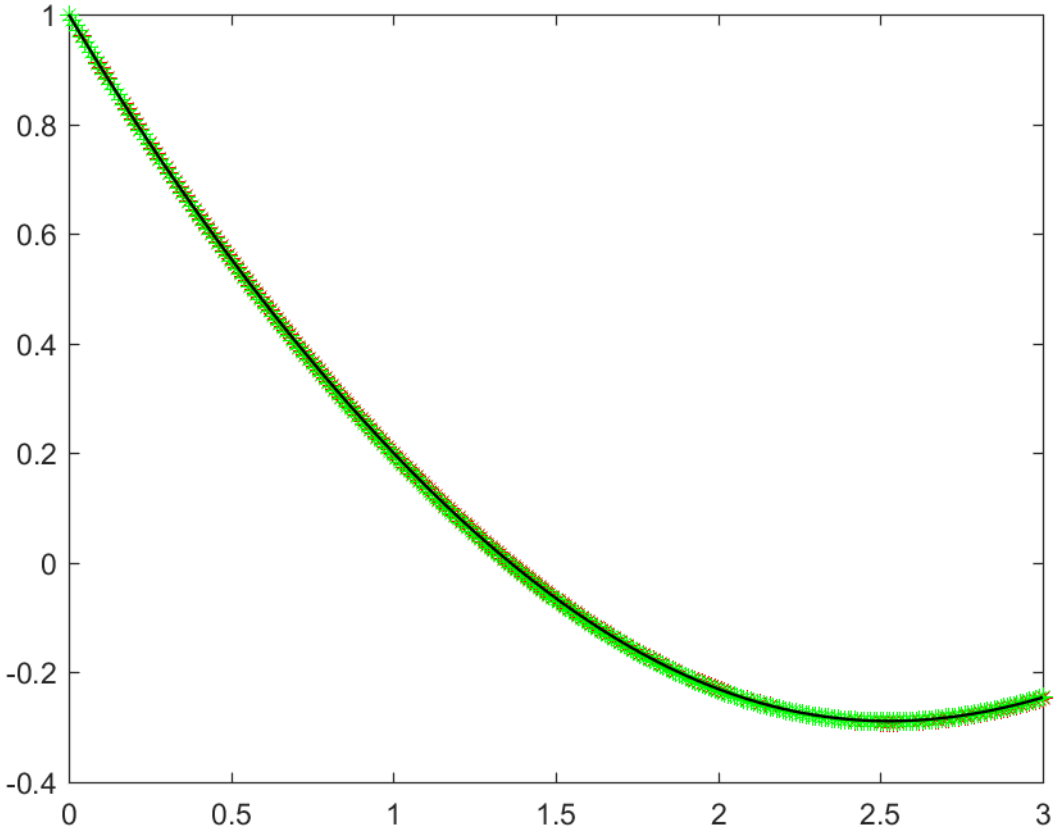


Figure 14 : Explicit Euler(blue). implicit Euler (red). semi-implicit Euler(green). Exact(black)

In the same way, we define two parametric MATLAB functions for the Runge–Kutta method of order 2 and order 4, namely `RK2fun` and `RK4fun`. Each function takes as input the function $f(t, y(t))$, the two endpoints of the integration interval t_0 and t_f, the step size h, and the initial condition $y(t_0)$. It returns a vector of abscissae together with the corresponding ordinate vectors computed using the Runge–Kutta method of order 2 and order 4, respectively.

```
function [x,y]=RK2fun(t0,tf,y0,h,f)
%f is a two dimensional function
t=t0:h:tf;
n=length(t);      %the number of evaluation points
ye=zeros(1,n);
ye(1)=y0;
for i=2:n
K1=f(t(i-1),ye(i-1))
K2=f(t(i-1)+h,ye(i-1)+h*K1)
ye(i)=ye(i-1)+(h/2)*(K1+K2);
end
end
```

```
function [x,y]=RK4fun(t0,tf,y0,h,f)
%f is a two dimensional function
t=t0:h:tf;
n=length(t);      %the number of evaluation points
ye=zeros(1,n);
ye(1)=y0;
for i=2:n
K1=f(t(i-1),ye(i-1));
K2=f(t(i-1)+h/2,ye(i-1)+h/2*K1);
K3=f(t(i-1)+h/2,ye(i-1)+h/2*K2);
K4=f(t(i-1)+h,ye(i-1)+h*K3);
ye(i)=ye(i-1)+h/6*(K1+2*(K2+K3)+K4);
end
end
```

In the same manner, within the same script as before, we call once the function `RK2func` instead of `Eulerfunc`, and once the function `RK4func`, in order to obtain the graphs corresponding to the two methods.

Script diffsol2

```
clear all; clc; close all;
f=@(t,y)-sin(t)/2-y;
t0=0;
tf=3;
y0=1;
h=0.1;
[t,ye1]=RK2fun(t0,tf,y0,h,f);
[t,ye2]=RK4fun(t0,tf,y0,h,f);
plot(t,ye1,'b*',t,ye2,'r*','Linewidth',1)
%exact solution
g=matlabFunction(dsolve('Dy=-sin(t)/2-y','y(0)=1','t'))
y2=g(t);
hold on;
plot(t,y2,'k','Linewidth',1)
```

The simulation yields the following results for a step size $h = 0.1$, where the approximate solutions are shown in different colors, while the curve of the exact solution is plotted in black:

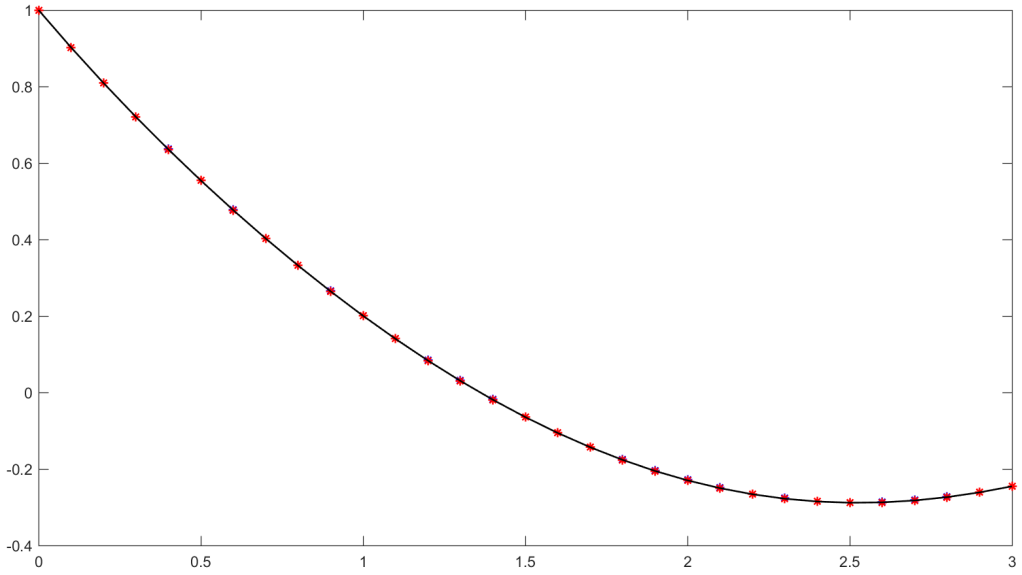


Figure 15 : Runge–Kutta of order 2(blue). Runge–Kutta of order 4(red).Exact(black)

By reducing the step size to $h = 0.01$, the following results are obtained:

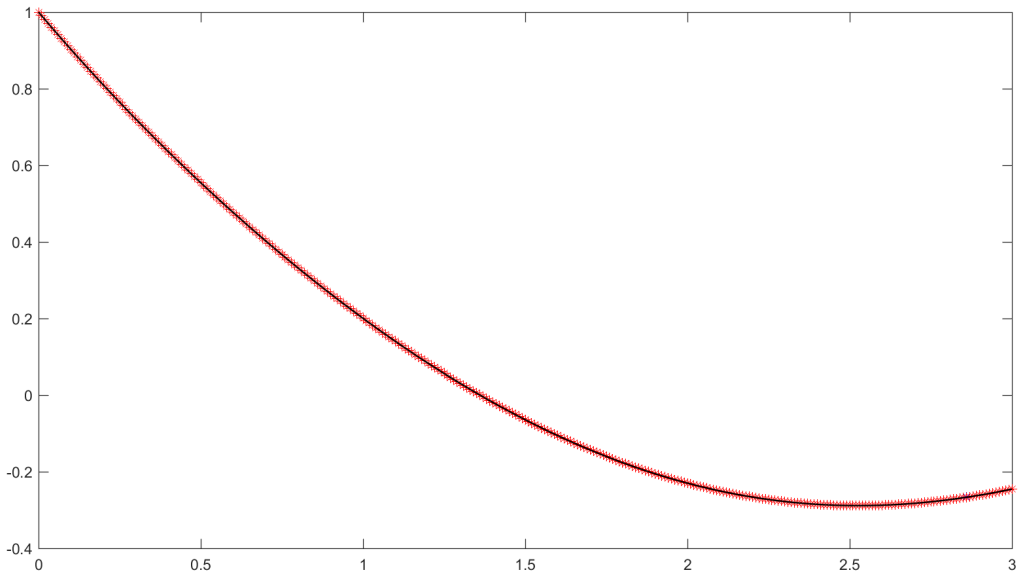


Figure 16 : Runge–Kutta of order 2(blue). Runge–Kutta of order 4(red).Exact(black)

References

[1] SAUER, Timothy. *Numerical Analysis*. 3rd ed. Pearson, 2017.

[2] DEMAILLY, Jean-Pierre. *Analyse numérique et équations différentielles*. Les Ulis : EDP Sciences, 2006.

[3] TREFETHEN, Lloyd N. *Approximation Theory and Approximation Practice, Extended Edition*. SIAM, 2019.

[4] HAMMING, Richard. *Numerical Methods for Scientists and Engineers*. Courier Corporation, 2012.

[5] QUARTERONI, Alfio, SACCO, Riccardo, and SALERI, Fausto. *Numerical Mathematics*. Springer, 2006.

[6] HAIRER, Ernst, WANNER, Gerhard, and NØRSETT, Syvert P. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, 1993.

[7] QUARTERONI, Alfio and VALLI, Alberto. *Numerical Approximation of Partial Differential Equations*. Springer, 1994.

Annexes

The material developed in this handout is derived from my practical worksheets, comprising three laboratory sheets structured according to the main chapters of the course.

Practical Work Sheet No. 1

Introduction to Data Interpolation

Activity:

1) In MATLAB, determine the coefficients of the polynomial whose roots are: $x_0 = -\frac{1}{2},\ x_1 = 2$.

• Evaluate this polynomial at $x = 12$.

2) Consider the interpolation support consisting of the points: $(-1, 3),\ (0, -2),\ (3, 7)$.

• Implement the Lagrange interpolation method in MATLAB, and use it to compute the coefficients of the polynomial that passes exactly through these points. • Plot, on the same figure, the graph of this polynomial together with the interpolation support points. What do you observe?

3) Consider the interpolation support composed of the following points: $(-2, -74),\ (0.2, 4.3411),\ (-1, 0),\ (1, 10),\ (6, 5950),\ (2, 42)$.

• Determine the degree 5 polynomial that passes through all these points. • Plot its graph over the interval $[-10,\ 10]$.

4) Interpolate the function $\ln(x)$ at the points of abscissa $1, 2, \ldots, 8$ and plot the corresponding curves.

5) Using the same interpolation support, implement the Newton interpolation method in MATLAB, and compute the degree 5 polynomial that passes exactly through the support points.

Practical Work Sheet No. 2

Introduction to Numerical Integration

Activity:

1) Let the function

$$f(x) = \frac{x}{\sqrt{x^2+1}} - 2\sin(2x).$$

Plot the Hermite interpolation polynomial, by implementing the method in MATLAB, over the interval $[-1, 6]$. The polynomial must pass exactly through the support consisting of the points of the antiderivative of f with abscissa $-1, 0, 3, 5, 6$.

2) Find an approximation of its integral over the interval $\left[\frac{3}{2}, 3\right]$, using discretizations of 21, 100, and 1000 points, and by implementing the following numerical methods in MATLAB:

- Rectangle rule: left, right, and midpoint
- Trapezoidal rule

3) Using the Simpson method, by implementing it in MATLAB for a set of points that are not necessarily equally spaced, approximate the integral of the antiderivative of f over this support, and then over the interval $[-1, 6]$ using 701 equally spaced points. What do you observe?

4) Approximate the integral of f over the interval $\left[\frac{3}{2}, 3\right]$ using interpolation methods:

- Newton interpolation
- Lagrange interpolation

5) Implement in MATLAB the general Newton–Cotes method, and apply it to compute the required numerical approximations.

Practical Work Sheet No. 3

First Order Differential Equations

Activity:

1) Find an approximation of the solution of the following differential equations by implementing the methods below in MATLAB:

- Explicit Euler method
- Implicit Euler method
- Semi implicit Euler method

$$\frac{dy}{dx} = -\frac{\sin(x)}{y}, \qquad y(-1) = 2, \qquad x \in [-1, 4]$$

$$\frac{dy}{dx} + y = -\frac{\sin(x)}{2}, \qquad y(0) = 1, \qquad x \in [0, 3]$$

$$x + \frac{dy}{dx} - y = 1, \qquad y(0) = 1, \qquad x \in [0, 3]$$

2) Plot the solution curves over the corresponding intervals.

3) Find an approximation of the solution using the Runge–Kutta method.

4) Determine the solution using MATLAB's built-in functions, and plot the corresponding solutions.

5) Find the solution of the differential equation below over the interval [0, 2] :

$$\begin{cases} \dfrac{dy}{dx} = \sin^2(x) - \dfrac{y}{\cos(x) + 1}, \\ y(0) = -\dfrac{1}{2}. \end{cases}$$