Worksheet 6 : Coursework Assesment

Raphael Garnaud, student number : U 17612997

September 5, 2018

Contents

1 Root finding **5**
- 1.0.1 a 5
- 1.0.2 b 7
- 1.0.3 c 11

2 Polynomial interpolation revisited **13**
- 2.0.1 a 13
- 2.0.2 b 15
- 2.0.3 c 15
- 2.0.4 d 15
- 2.0.5 e 16
- 2.0.6 f 17

3 Gaussian quadrature **19**
- 3.0.1 a 19
- 3.0.2 b 20
- 3.0.3 c 20
- 3.0.4 d 20
- 3.0.5 e 21
- 3.0.6 f 22
- 3.0.7 g 23
- 3.0.8 h 24
- 3.0.9 i 26

4 Runge-Kutta methods **27**
- 4.0.1 a 27
- 4.0.2 b 27
- 4.0.3 c 28

5 Rock-scissors-paper **32**
- 5.0.1 a 32
- 5.0.2 b 32
- 5.0.3 c 32
- 5.0.4 d 33
- 5.0.5 e 34
- 5.0.6 f 36

I Annexe **38**

List of Figures

1.1 Bracket search for finding the roots . . . 7
1.2 Bracket search for finding the roots . . . 9
1.3 Bracket search for finding the roots and the derivative . . . 10
1.4 Bracket search for finding the root and methods comparison . . . 11

2.1 Comparison of the Hermite method and the one saw in the course with 3 and 8 data points . . . 13
2.2 Lagrange polynomial for $N = 10$ . . . 15
2.3 Comparison between the Chebyshev method and the linearly spaced points data . . . 17
2.4 The function $g(x)$ and the interpolation with linearly spaced data and Chebyshev against time . . . 18

3.1 Comparison of efficency between the Gaussian's and Simpson's methods . . . 21
3.2 Probability density function of χ_k^2 against t . . . 22
3.3 Finding the minimum error for $P(X > 3.3)$ with $k = 1$ . . . 24
3.4 Finding the minimum error for $P(X > 3.3)$ with $k = 4$ . . . 25

4.1 Absolute and relative error comparison of efficiency between the fourth method of Runge-Kutta . . . 29

5.1 x, y and z against t for $h = 0.02$ and up to $t = 10$ . . . 33
5.2 x, y and z against t for $h = 0.02$ and up to $t = 100$ . . . 33
5.3 x, y and z against t for $h = 0.02$ and up to $t = 100$ . . . 34
5.4 $\frac{d(x+y+z)}{dt}$ and $\frac{d(xyz)}{dt}$ against t for $h = 0.02$ . . . 35
5.5 Graph of x, y, z, $x + y + z$, $\frac{d(x+y+z)}{dt}$ and $\frac{d(xyz)}{dt}$ against t for $h = 0.002$ . . . 36
5.6 Graph of x, y, z, $x + y + z$, xyz against t comptued with $ode45$ . . . 37

List of Tables

1.1 Number of steps 8
1.2 Values of the root 8
1.3 Values of the root 9
1.4 Maximum of the function and number of steps required 10
1.5 Absolute error 11

2.1 Values of each Lagrange polynomial 15
2.2 Maximum error between $g(x)$ and the different interpolation 17

3.1 Roots and weights of the Gaussian Quadrature 20
3.2 Value of $P(X > 3.3)$ computed with Midpoint 23
3.3 Value of $P(X > 3.3)$ for $k = 1$ and $k = 4$ 24

4.1 Butcher Talbeau 27
4.2 Butcher Talbeau 27
4.3 Approximation of $y(1)$ according to the different mthods of Runge-Kutta 27

Chapter 1

Root finding

1.0.1 a

For this question, I actually choosed to write 3 functions, because depending on the use it was easier.

Code

I will give here only the most important function, where there is everything that could be in the other functions. But for some questions, instead of using this function I will use shorter ones (theses are available at the end).

```
function [aerror,rerr,nbsteps, value]=rootfinding2(f,fprime,a,b,n,err,cerr,cm)
%aerr: absolute error storage
%rerr : realtive error storage
%value : corresponds to the value of the nearest root function
%f: is the function
%fprime: derivative of the function
%a,b : is the bracket
%n: number maximum of steps used
%err: relative and/or absolute error tolerances
%cerr: corresponds to the choice of the error
%cm: correponds to the method we want to use
%rerru : is the relative error used for the tolerance

if(b-a<0)
   error("the values for a and be you've entered is not a bracket")
end

if(err<0)
   error("The error tolerance can't be negative")
end

if(cerr=='a')
   rerru=500;
   aerru=b-a;
else
   rerru=50;
   aerru=500;
end

x0=(a+b)/2;
if(cm=='nr')
```

```
x=x0;
c=0;
x2=x0+2*err;
fp=fprime(x);
while(c<n & aerru>err & rerru>err)
    x2=x;
    t=x-f(x)./fprime(x);
    if(t>b |t<a)
       error('There is a convergence problem')
    end
    x=t;
    c=c+1;
    aerror(c)=abs(x2-x);
    rerr(c)=abs((x2-x)/x);
    if(cerr=='a')
        aerru=aerror(c);
    else
        rerru=rerr(c);
    end
end
value=x;
nbsteps=c;
end
if(cm=='se')
c=0;x=a;x2=b;
while(c<n & aerru>err & rerru>err)
    t=x-((x-x2)/(f(x)-f(x2)))*f(x);
    x2=x;
    if(t>b |t<a)
       error('There is a convergence problem')
    end
    x=t;
    c=c+1;
    aerror(c)=abs(x2-x);
    rerr(c)=abs((x2-x)/x);
    if(cerr=='a')
        aerru=aerror(c);
    else
        rerru=rerr(c);
    end
end
value=x;
nbsteps=c;
end
if(cm=='rd')
    x1=a;x2=b;c=0;
    x4=(x1+x2)/2;
    xp=x4+2*err;
while(c<n & aerru>err & rerru>err)
    x3=(x1+x2)/2;
    xp=x4;
    x4=x3+(x3-x1)*(sign(f(x1)-f(x2))*f(x3))./sqrt(f(x3).^2-f(x1).*f(x2));
    if(x4>x2 |x4<x1)%this is impossible, we don't need it
       error('There is a convergence problem')
    end
    if(f(x1)*f(x4)<0)
        x2=x4;
    elseif(f(x2)*f(x4)<0)
```

```
        x1=x4;
    else
        break;
    end
    if(x1>x2)
        t=x1;
        x1=x2;
        x2=t;
    end
    c=c+1;
    aerror(c)=abs(xp-x4);
    rerr(c)=abs((xp-x4)/x4);
    if(cerr=='a')
        aerru=aerror(c);
    else
        rerru=rerr(c);
    end
end
x=x4;
value=x;
nbsteps=c;
end
end
```

comments on the code

We chose to use a loop "while" instead of a loop "for" because the loop while do not loose time in continuing computing values after having found the solution with the tolerance required.

In order to initialize the value relative error tolerance, we can't use the formula because we might divide by zero. So we choose a big value, indeed it will be initialize in the loop "while".

Source for the Ridder Formula : https://www-sop.inria.fr/coprin/logiciels/ALIAS/ALIAS-C++/node65.html

1.0.2 b

i

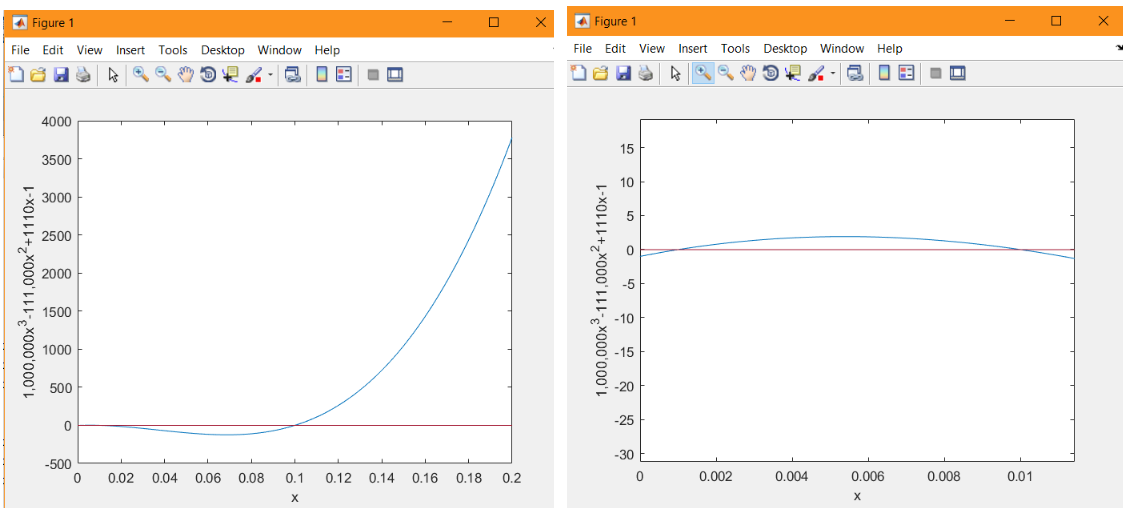

Figure 1.1: Bracket search for finding the roots

We can observe that there is three roots. Each of the roots are between two values that we can evaluate throughout the pictures.

	First root	Second root	Third root
Newton-Raphson	4	1	10
Secant	6	6	error
Ridder's method	5	0	133

Table 1.1: Number of steps

	First root	Second root	Third root	Third root for Ridder's method
Values	$9.999 \cdot 10^{-4}$	0.01	0.1	0.09999991

Table 1.2: Values of the root

As a conclusion, the both most efficient methods are the Newton-Raphson methods and Ridder's.

```
N=1000;
x=linspace(0,0.2,N);
y=1e6*x.^3-1.11e5*x.^2+1110*x-1;
plot(x,y,x,zeros(N));

format long
%i
%first root

[nbsteps1,value1]=rootfinding(@(x) 1e6*x.^3-1.11e5*x.^2+1110*x-1,@(x)
    3e6*x.^2-2.22e5*x+1110,0,0.004,20,1e-8,'nr')
[nbsteps2,value2]=rootfinding(@(x) 1e6*x.^3-1.11e5*x.^2+1110*x-1,@(x)
    3e6*x.^2-2.22e5*x+1110,0,0.004,20,1e-8,'se')
[nbsteps3,value3]=rootfinding(@(x) 1e6*x.^3-1.11e5*x.^2+1110*x-1,@(x)
    3e6*x.^2-2.22e5*x+1110,0,0.004,20,1e-8,'rd')

%second root
[nbsteps1,value1]=rootfinding(@(x) 1e6*x.^3-1.11e5*x.^2+1110*x-1,@(x)
    3e6*x.^2-2.22e5*x+1110,0.008,0.012,20,1e-8,'nr')
[nbsteps2,value2]=rootfinding(@(x) 1e6*x.^3-1.11e5*x.^2+1110*x-1,@(x)
    3e6*x.^2-2.22e5*x+1110,0.008,0.012,20,1e-8,'se')
[nbsteps3,value3]=rootfinding(@(x) 1e6*x.^3-1.11e5*x.^2+1110*x-1,@(x)
    3e6*x.^2-2.22e5*x+1110,0.008,0.012,5,1e-8,'rd')

%third
[nbsteps1,value1]=rootfinding(@(x) 1e6*x.^3-1.11e5*x.^2+1110*x-1,@(x)
    3e6*x.^2-2.22e5*x+1110,0.06,1,50,1e-8,'nr')
%[nbsteps2,value2]=rootfinding(@(x) 1e6*x.^3-1.11e5*x.^2+1110*x-1,@(x)
    3e6*x.^2-2.22e5*x+1110,0.06,0.12,50,1e-8,'se')%has a problem of convergence probably
    beacause of the braket
[nbsteps3,value3]=rootfinding(@(x) 1e6*x.^3-1.11e5*x.^2+1110*x-1,@(x)
    3e6*x.^2-2.22e5*x+1110,0.06,1,500,1e-8,'rd')%takes a very long time to converge
```

ii

On the first graph, we can observe that the curve cut the horizontal axises but these points aren't always roots. Actually, these are points where the function tends to the infinity. On these different

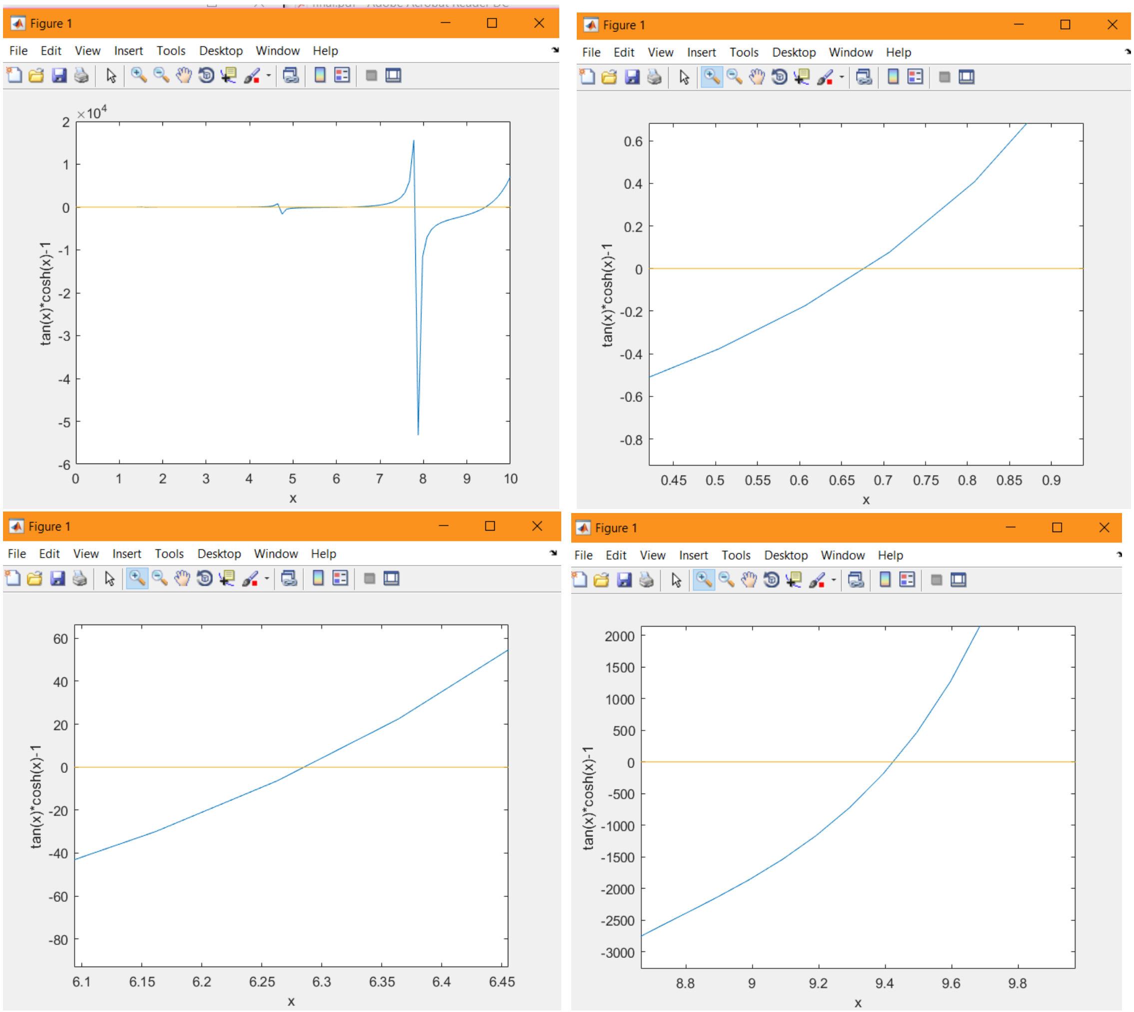

Figure 1.2: Bracket search for finding the roots

graphs, we can also observe the three positive roots, indeed we can find brackets that we are going to use to determine the roots.

So, the three last roots are summarize in this table.

	First root	Second root	Third root
Values	0.67888664	6.28690629	9.42493933
Number of steps	3	4	4

Table 1.3: Values of the root

```
N=100;
x=linspace(0,10,N);
y=tan(x).*cosh(x)-1;
yp=(1+(tan(x).^2)).*cosh(x)+tan(x).*sinh(x);
plot(x,y,x,zeros(N));

format long
%first root
[nbsteps3,value3]=rootfinding(@(x) tan(x).*cosh(x)-1,@(x)
```

```
    (1+(tan(x).^2)).*cosh(x)+tan(x).*sinh(x),0.4,0.9,20,1e-8,'rd')

%second root [6,6.4]
[nbsteps3,value3]=rootfinding(@(x) tan(x).*cosh(x)-1,@(x)
    (1+(tan(x).^2)).*cosh(x)+tan(x).*sinh(x),6,6.4,20,1e-8,'rd')

%second root [9,9.8]
[nbsteps3,value3]=rootfinding(@(x) tan(x).*cosh(x)-1,@(x)
    (1+(tan(x).^2)).*cosh(x)+tan(x).*sinh(x),9,9.8,20,1e-8,'rd')
```

iii

Here actually, we are looking for a maximum of a function. In order to find this maximum, knowing this isn't this infinity, we have to find the points where the derivative of this function reach zero. So we first compute this derivative.
$g'(x) = -\frac{-5\cdot x^4\cdot(\exp\frac{1}{x}-1)+x^3\cdot\exp\frac{1}{x}}{x^5\cdot(\exp\frac{1}{x}-1)}$

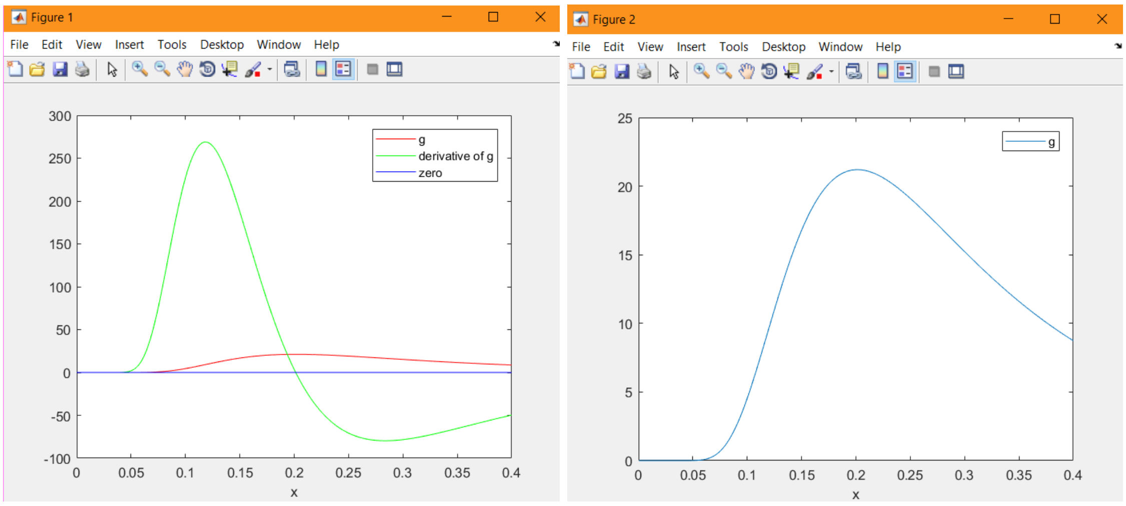

Figure 1.3: Bracket search for finding the roots and the derivative

We can easily find a bracket where the root of the derivative is contained in.

Value	Number of steps for the Ridder's method
0.20128401	2

Table 1.4: Maximum of the function and number of steps required

So we can find the point where the derivative is zero and at this point the function, when we see its shape, is clearly the maximum.

```
N=1000;
x=linspace(0,0.4,N);
y=x.^-5.*((exp(1./x)-1)).^-1;
ypt3=(-5*x.^4.*(exp(1./x)-1)+x.^3.*exp(1./x))./(x.^5.*(exp(1./x)-1)).^2;
ypc=myDiff(x,y,'cd');
plot(x,y,x,ypt,x,zeros(N),'y');
```

```
legend('y','der','zero')
plot(x,y)
legend('y')

format long
%first root
[nbsteps,value]=ridder(@(x)
    (-5*x.^4.*(exp(1./x)-1)+x.^3.*exp(1./x))./(x.^5.*(exp(1./x)-1)).^2,0.15,0.3,20,8)
```

1.0.3 c

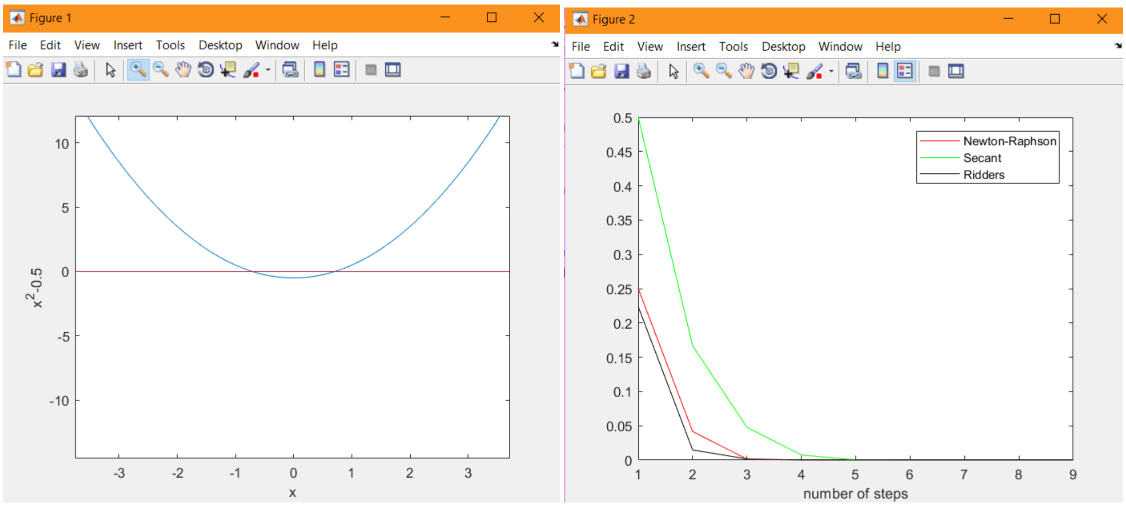

Figure 1.4: Bracket search for finding the root and methods comparison

On the first figure, we observe that the first root is between -1 and 0.

Ridder's	0.2	0.01	0.001	0.0002	0.00001	0.000002	$2{\cdot}10^{7}$	$3\cdot10^{8}$	$3\cdot10^{9}$
Newton-Raphson	0.2	0.04	0.001	0.000001	$7{\cdot}10^{13}$				
Secant	0.5	0.1	0.04	0.007	0.0002	0.000001	$1{\cdot}10^{10}$		

Table 1.5: Absolute error

We can observe that the most efficient methods are, in decreasing order, Ridder's, Newton-Raphson and Secant. But, there is something strange that Ridder's method finaly takes more steps to find the solution even if it is more efficient (cf the curve).

Code

```
N=1000;
x=linspace(-10,10,N);
y=x.^2-0.5;
plot(x,y,x,zeros(N))
figure
format long
[aerror1,rerror1,nbsteps1,value1]=rootfinding2(@(x) x.^2-0.5,@(x)
    2*x,-1,0,20,1e-8,'a','nr')
[aerror2,rerror2,nbsteps2,value2]=rootfinding2(@(x) x.^2-0.5,@(x)
    2*x,-1,0,20,1e-8,'a','se')
```

```
[aerror3,rerror3,nbsteps3,value3]=rootfinding2(@(x) x.^2-0.5,@(x) 2*x,-1,0,20,1e-8,'a','rd')%we could have used only one variable for error, and weather we choose absolute or relative error, the answer would correpond to it.
plot([1:nbsteps1],aerror1,'r',[1:nbsteps2],aerror2,'g',[1:nbsteps3],aerror3,'k')
legend('Newton-Raphson','Secant','Ridders')
```

Chapter 2

Polynomial interpolation revisited

2.0.1 a

For this question, we will use a function done in the previous worksheet using the Vandermonde Matrix. And actually because I didn't understand at first the question, I have also programmed a version with the Hermite Matrix. And so, in this question, we will compare the method using the Vandermonde Matrix and the Hermite one. And moreover, we will also compare the different kind of data points used : between lineraly spaced and Chebyshev.

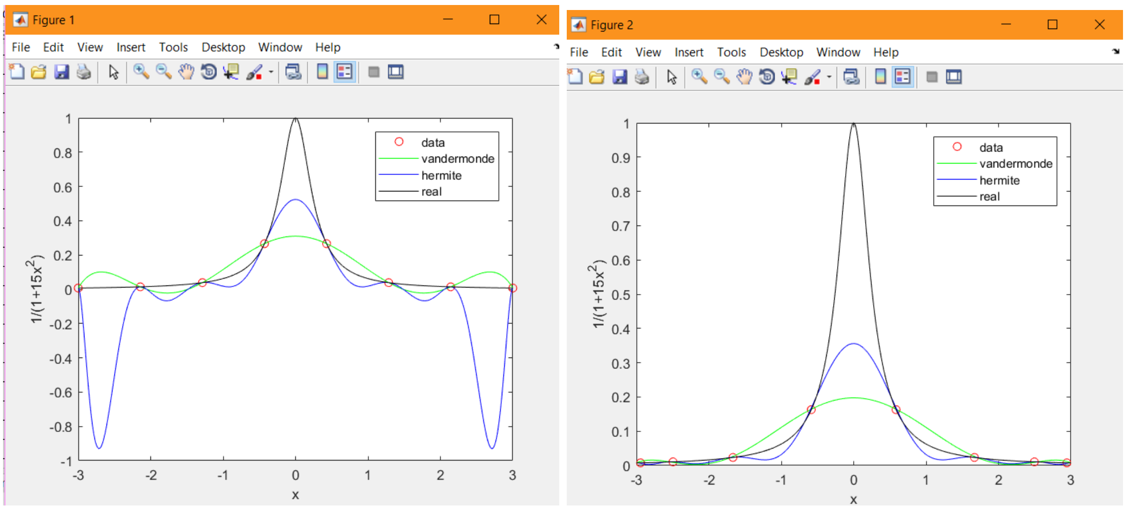

Figure 2.1: Comparison of the Hermite method and the one saw in the course with 3 and 8 data points

We can observe that with a very small amount of data, the Hermite method is more efficient because it uses two times each data point. Bu the downside of the Hermite method is that for a larger amount of data, for example 8, there are much more "‘noise"’. Actually, the reason why we could take Chebyshev point method instead of the linear one is to reduce this noise.
Another solution, in order to improve the Hermite method could be to apply a spline method to it, indeed we would have all the advantages of the Hermite method but without its downsides.

Code

The interpolation method explained in the lecture :

```
function [E,y]=interb(f,N3,xdata,ydata)
```

```
M=vander(xdata);
a=M\(ydata');
N=length(xdata);
x=linspace(min(xdata),max(xdata),N3);
y=polyval(a,x);
E=abs(f(x(1))-y(1));
for i=1:N3
    if(E<abs(f(x(i))-y(i)))
        E=abs(f(x(i))-y(i));
    end
end
plot(xdata,ydata,'ro',x,y,'g-');
legend('data','p(x)');
end
```

The *hermite* function is :

```
function [E,y]=hermite(f,N3,xdata,ydata)
format long
x=xdata;
n2=length(xdata);
t=ones(n2);
M1=x'.*t;%we put in t one, then we place x in the right direction
M1=[M1 M1];
M=[M1.^(2*n2-[1:2*n2]);(2*n2-[1:2*n2-1]).*M1(:,1:end-1).^(2*n2-[2:2*n2]),zeros(n2,1)];
a=M\(ydata');
x=linspace(min(xdata),max(xdata),2*N3);
y=polyval(a,x);
E=abs(f(x(1))-y(1));
end
```

```
a=-3;b=3;Nv=8;
N=[1:Nv];

xdata=linspace(-3,3,Nv);
ydata=1./(1+15*xdata.^2);
ydatader=-(30*xdata)./((1+15*xdata.^2).^2);
N2=600;
x=linspace(a,b,N2);
y=1./(1+15*x.^2);
[E,y1]=interb(@(x) 1/(1+15*x.^2),N2,xdata,ydata);
[E,y2]=hermite(@(x) 1/(1+15*x.^2),N2,xdata,[ydata,ydatader]);
x2=linspace(a,b,2*N2);
plot(xdata,ydata,'or',x,y1,'g',x2,y2,'b',x,y,'k')
legend('data','vandermonde','hermite','real')
figure
xdata=(a+b)/2+((b-a)/2)*cos(((2*N-1)*pi)./(2*Nv))
ydata=1./(1+15*xdata.^2);
ydatader=-(30*xdata)./((1+15*xdata.^2).^2);
N2=600;
x=linspace(a,b,N2);
y=1./(1+15*x.^2);
[E,y1]=interb(@(x) 1/(1+15*x.^2),N2,xdata,ydata);
[E,y2]=hermite(@(x) 1/(1+15*x.^2),N2,xdata,[ydata,ydatader]);
x2=linspace(a,b,2*N2);
plot(xdata,ydata,'or',x,y1,'g',x2,y2,'b',x,y,'k')
legend('data','vandermonde','hermite','real')
```

2.0.2 b

2.0.3 c

Lets take the definition of L_i given in the question b, $L_i(x) = \frac{\prod_{j=1,j\neq i}^{N}(x-x_j)}{\prod_{j=1,j\neq i}^{N}(x_i-x_j)} = \prod_{j=1,j\neq i}^{N}\left(\frac{x-x_j}{x_i-x_j}\right)$.
So $L_i(x_j) = \frac{(x_j-x_j)}{(x-i-x_j)} \cdot \prod_{k=1,k\neq i,k\neq j}^{N}\left(\frac{x_j-x_k}{x_i-x_j}\right) = 0$ and $L_i(x_i) = \prod_{j=1,j\neq i}^{N}\left(\frac{x_i-x_j}{x_i-x_j}\right) = \prod_{j=1}^{N} 1 = 1$.
Indeed, $p(x_k) = \sum_{j=1}^{N} x_j L_j(x) = y_k L_k + \sum_{j=1,k\neq j}^{N} x_j L_j(x) = y_k$ Actually, we find the same equality for the interpolant constants saw in the course $p(x_k) = a_{n-1}x_k^{n-1} + a_{n-2}x_k^{n-2} + ... + a_1 x_1 + a_0 = y_k$. And from this equality, we will deduce the matrix of the interpolation explained in the lecture. So this definition of the interpolation is equivalent to the one explained in the lecture.

2.0.4 d

The values of the polynomials $L_i(x)$ depending on the precision of the interpolation. We choose $N = 10$.
For this precision criteria, we obtain.

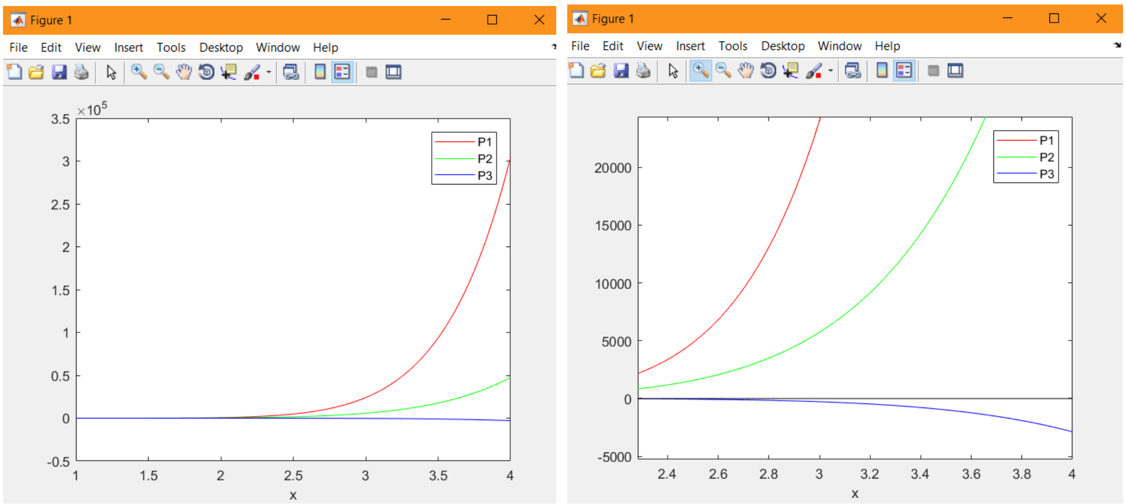

Figure 2.2: Lagrange polynomial for $N = 10$

	$x = 1$	$x = 2$	$x = 4$
P_1	0.5556	682	30493.5
P_2	6.6667	342.66	4746.7
P_3	2.7778	-1.6666	-2877.2

Table 2.1: Values of each Lagrange polynomial

Code

The function *lagrange* is :

```
function [L,res]=lagrange(nb,x,y)
t=linspace(min(x),max(x),nb);
N=length(x);
n=length(t);
```

```
for j=1:n
for i=1:N
L(i,j)=prod(t(j)*ones(1,N-1)-x([1:i-1,i+1:N]))./prod(x(i)*ones(1,N-1)-x([1:i-1,i+1:N]));
end
end
for j=1:n
res(j)=sum(y*L(:,j));
end
end
```

```
a=-3;b=3;Nv=9;N=[1:Nv];
xdata=[1 2 4];
n=length(xdata);
ydata=1./(1+15*xdata.^2);
N=10;
t=linspace(min(xdata),max(xdata),N);
[L,y1]=lagrange(N,xdata,ydata);
P=zeros(n,300);
x=linspace(min(xdata),max(xdata),300);
for i=1:n
   P(i,:)=polyval(L(i,:),x);
end
plot(x,P(1,:),'r',x,P(2,:),'g',x,P(3,:),'b')
legend('P1','P2','P3')
format long
for i=1:3
    for j=1:3
        Q(i,j)=polyval(L(i,:),xdata(j));
    end
end
Q
```

2.0.5 e

We can observe that the Chebyshev points method reduces the noise compared to when the points are linearly spaced.

Code

The function *barycentric* is :

```
function res=barycentric(nb,x,y,cip)
t=linspace(min(x),max(x),nb);
N=length(x);
n=length(t);
if(cip=='li')
    for i=1:N
        w(i)=(-1)^(i-1).*nchoosek(N-1,i-1);
    end
elseif(cip=='ch')
    for i=1:N
        w(i)=(-1)^(i-1)*sin(((2*i-1)*pi)/(2*N));
    end
end
tempo=sum((w./(t'-x)).*y*ones(N,1),2)./sum(w./(t'-x),2);
```

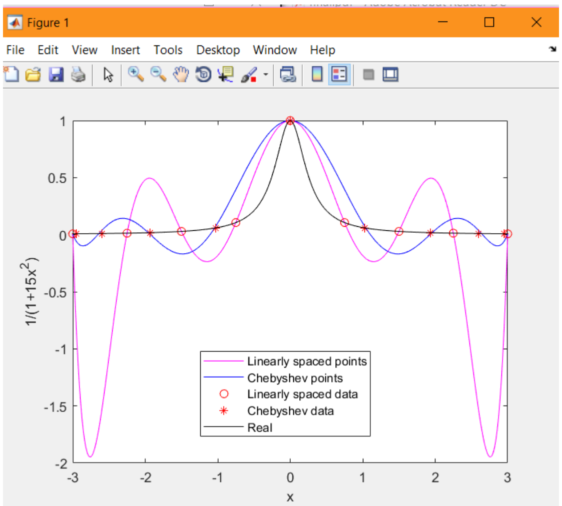

Figure 2.3: Comparison between the Chebyshev method and the linearly spaced points data

```
tempo(1)=y(1);tempo(end)=y(end);%because we know the value of the function at these
    points
res=tempo';
end
```

```
a=-3;b=3;Nv=9;N=[1:Nv];
xdata1=linspace(-3,3,Nv);
ydata1=1./(1+15*xdata1.^2);
xdata2=(a+b)/2+((b-a)/2)*cos(((2*N-1)*pi)./(2*Nv));
ydata2=1./(1+15*xdata2.^2);
n=500;
t=linspace(min(xdata1(1),xdata2(1)),max(xdata1(end),xdata2(end)),n);
y1=barycentric(n,xdata1,ydata1,'li');
y2=barycentric(n,xdata2,ydata2,'ch');
plot(t,y1,'m',t,y2,'b',xdata1,ydata1,'or',xdata2,ydata2,'*r',t,1./(1+15*t.^2),'k');
legend('Linearly spaced points','Chebyshev points','data1','data2','Real')
```

2.0.6 f

	Lagrange	Barycentric linear spaced	Barycentric Chebyshev
Absolute error	$3.99 \cdot 10^{-5}$	$3.99 \cdot 10^{-5}$	$2.1 \cdot 10^{-3}$

Table 2.2: Maximum error between $g(x)$ and the different interpolation

We can observe that, in this case, the use of the Chebyshev points do not increase the efficiency of the algorithm. It is actually the opposite, indeed it seems that the Chebyshev allow to reduce the noise, but when there is none, it is better to use the linearly spaced method.

Code

```
xdata1=[-1:1/8:1];
ydata1=1./(1+xdata1.^2);
a=-1;b=1;Nv=17;N=[1:Nv];%the value of N is 17 because the length of xdata is 17
```

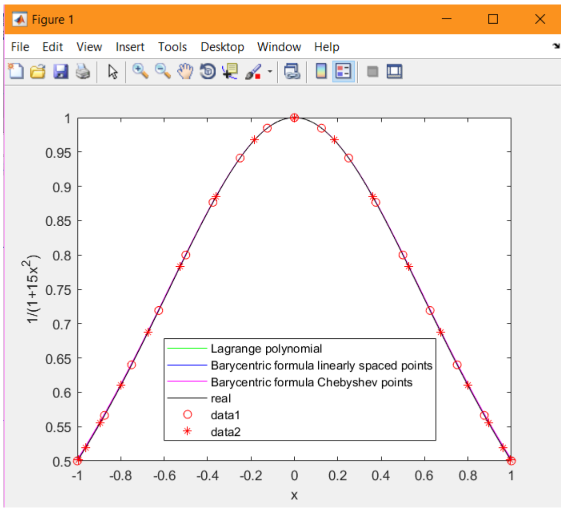

Figure 2.4: The function $g(x)$ and the interpolation with linearly spaced data and Chebyshev against time

```
xdata2=(a+b)/2+((b-a)/2)*cos(((2*N-1)*pi)./(2*Nv));
ydata2=1./(1+xdata2.^2);
n=500;
[L,y1]=lagrange(n,xdata1,ydata1);
y2=barycentric(n,xdata1,ydata1,'li');
y3=barycentric(n,xdata2,ydata2,'ch');
t=linspace(min(xdata1(1),xdata2(1)),max(xdata1(end),xdata2(end)),n);
r=1./(1+t.^2);
plot(t,y1,'g',t,y2,'b',t,y3,'m',t,r,'k',xdata1,ydata1,'or',xdata2,ydata2,'*r');
legend('Lagrange polynomial','Barycentric formula linearly spaced points','Barycentric
    formula Chebyshev points','real','data1','data2')
m1=max(abs(r-y1))
m2=max(abs(r-y2))
m3=max(abs(r-y3))
```

Chapter 3

Gaussian quadrature

3.0.1 a

In order to compute the Legendre polynomials we need to use this kind of matrix.

$$\begin{pmatrix} 0 & 0 & \cdots & \cdots & 0 \\ 1 & 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & \ddots & 0 & 0 \\ 0 & \cdots & \cdots & 1 & 0 \end{pmatrix}$$

For $n = 2$, we use this one.

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$$P_2(x) = \left(\frac{4-1}{2}\right) \cdot \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot P_1(x) - \left(\frac{2-1}{2}\right) P_0(x)$$

$$P_2(x) = \frac{3}{2} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} - \frac{1}{2} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} - \frac{1}{2} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -\frac{1}{2} \\ 0 \\ \frac{3}{2} \\ 0 \end{pmatrix}$$

Indeed, with this matrix, we can program, in a general case the Lengendre polynomial $P_n(x)$.

Code

```
function P=gaussian(n)
P=zeros(n,n+1);
P(1,end)=1;P(2,end-1)=1;
x=zeros(n+1,n+1);
for i=2:n+1
    x(i,i-1)=1;
end
for i=2:n
    P(i+1,:)=((2*i-1)/i)*P(i,:)*x-((i-1)/i)*P(i-1,:);%we use i+1, because with matlab we start at 1 and not 0
end
end
```

3.0.2 b

So we create a function that will compute the roots (using the function root of matlab) and the weight obtained with the formula given in the course.

	$N=2$	$N=3$	$N=4$
x	$\{-0.6, 0.6\}$	$\{-0.8, 0, 0.8\}$	$\{-0.8, -0.3, 0.3, 0.8\}$
w	$\{1, 1\}$	$\{0.55, 0.8, 0.56\}$	$\{0.35, 0.6, 0.6, 0.35\}$

Table 3.1: Roots and weights of the Gaussian Quadrature

Indeed, these values will be useful to compute the Lagrange polynomial because they remain the same no matter the function.

Code

The function weight

```
function [root,w]=weight(n)
P=gaussian(n+2);
root= roots(P(n+1,:));
%root=sort(root);
for i=1:n
w(i)=2*(1-root(i)^2)/((n+1)^2*(polyval(P(n+2,:),root(i)))^2);
end
end
```

And its application.

```
[root2,w2]=weight(2)
[root3,w3]=weight(3)
[root4,w4]=weight(4)
```

3.0.3 c

We are searching to compute $\int_a^b f(t)dt$.
Let define $t = \frac{b-a}{2}x + \frac{a+b}{2}$ and then $dt = \frac{b-a}{2}dx$.
Let also define ϕ such like $x = \left(t - \frac{a+b}{2}\right)\frac{2}{b-a} = \frac{2t-a-b}{b-a} = \phi(t)$.
So $\phi(a) = \frac{2a-a-b}{b-a} = -1$ and $\phi(b) = \frac{2b-a-b}{b-a} = 1$.
Hence $\int_a^b f(t)dt = \int_{-1}^1 f(\phi(x))\phi'(x)dx = \left(\frac{b-a}{2}\right)\int_{-1}^1 f\left(\left(\frac{b-a}{2}\right)x + \frac{a+b}{2}\right)dx$.

3.0.4 d

So using the roots and the weight computed in the question b and applying the formula discovered in the question c, we can write a function that compute the integral of a function f between a and b with a number of panels n.

Code

```
function res=inte(f,a,b,n)
[x,w]=weight(n);
if(imag(x)~=0)
   disp("Some of the roots have an imaginary part")
end
res=((b-a)/2)*sum(w.*f(((b-a)/2)*x'+(a+b)/2));
```

```
end
```

3.0.5 e

In this question, we will compare the Gaussian's method and the Simpon's method using that the solution is found with $erf(0.75)$.

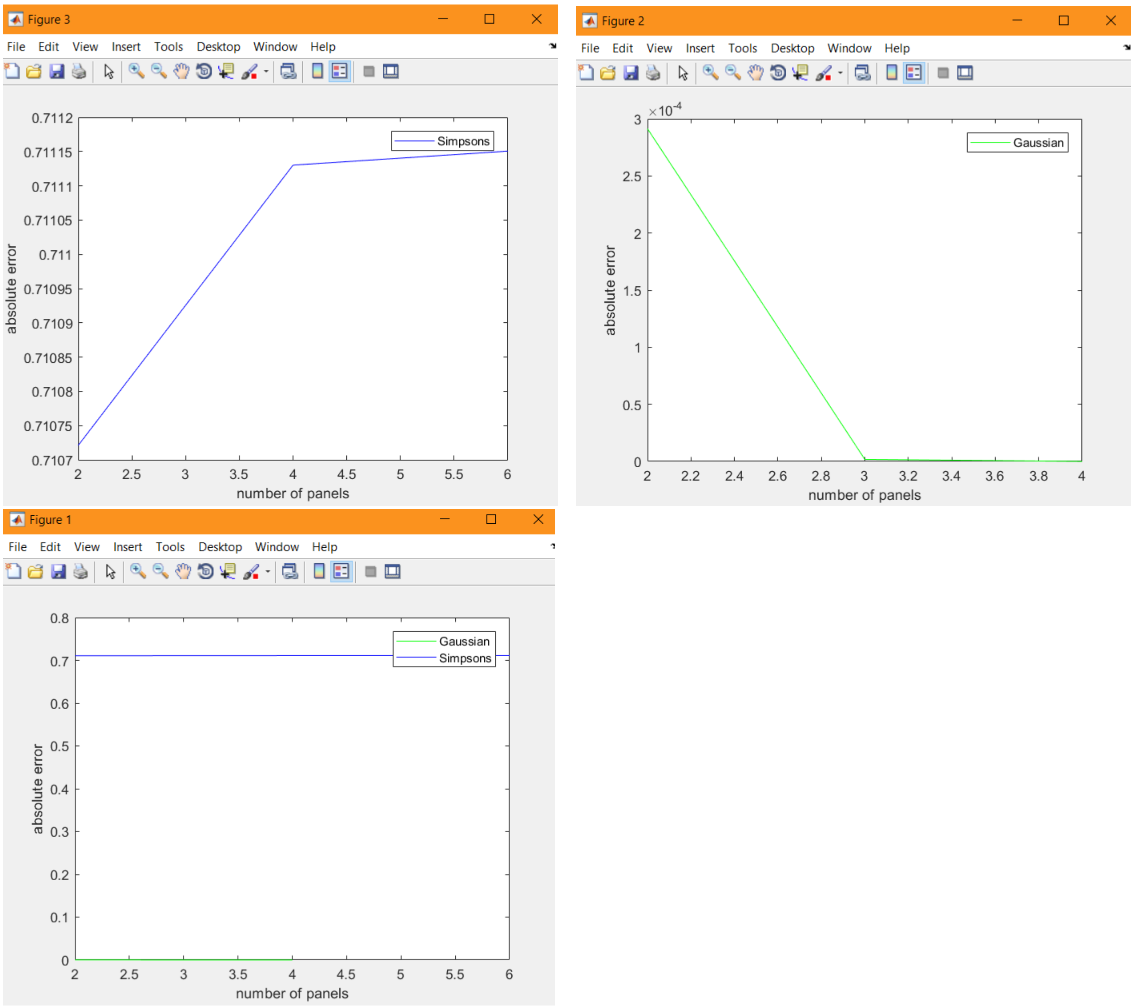

Figure 3.1: Comparison of efficency between the Gaussian's and Simpson's methods

We can observe that the Gaussian's method is more precise and with less panels. Actually the improvements of each method is absolutely not of the same kind. When the Gaussian's method tries to find the best places where to evaluate the function to make the integral most accurate as possible, the Simpson's method tries to use as best as possible the values of the points already given. Indeed, we could try to combinate both methods in order to improve the accuracy of the integral.

Code

```
format long
%Gaussian
```

```
res(1)=(2/sqrt(pi))*inte(@(x) exp(-x.^2),0,0.75,2);
res(2)=(2/sqrt(pi))*inte(@(x) exp(-x.^2),0,0.75,3);
res(3)=(2/sqrt(pi))*inte(@(x) exp(-x.^2),0,0.75,4);
%Simpson's
res(4)=(2/sqrt(pi))*myQuad(@(x) exp(-x.^2),0,0.75,2,'sip');
res(5)=(2/sqrt(pi))*myQuad(@(x) exp(-x.^2),0,0.75,4,'sip');
res(6)=(2/sqrt(pi))*myQuad(@(x) exp(-x.^2),0,0.75,6,'sip');
err=abs(res-erf(0.75))
plot([2:4],err(1:3),'g',[2:2:6],err(4:6),'b')
legend('Gaussian','Simpsons')
xlabel('number of panels')
ylabel('absolute error')
figure
plot([2:4],err(1:3),'g')
legend('Gaussian')
xlabel('number of panels')
ylabel('absolute error')
figure
plot([2:2:6],err(4:6),'b')
xlabel('number of panels')
ylabel('absolute error')
legend('Simpsons')
```

3.0.6 f

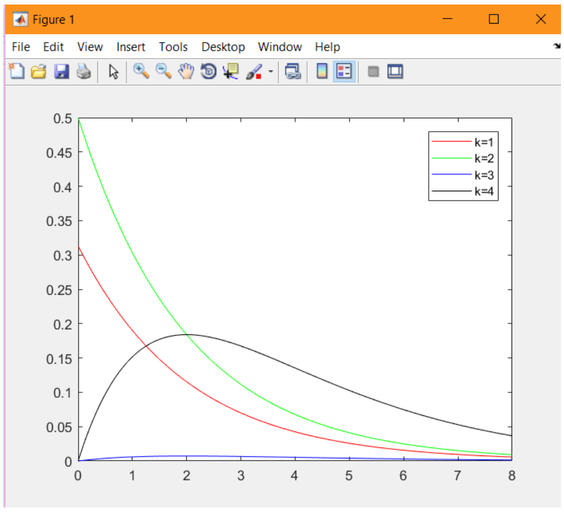

Figure 3.2: Probability density function of χ_k^2 against t

We can observe that f_1 seems to diverge for $x \to 0$.
$lim_{x\to 0} f_1(x) = \frac{1}{2^{1/2}\Gamma(1/2)} x^{-1/2} \exp^{-x/2}$
$lim_{x\to 0} f_1(x) = A \cdot \frac{1}{\sqrt{x}} \exp^{-x/2}$ with $A = 1/(\sqrt{2} \cdot \sqrt{\pi}\frac{(2n!)}{2^{2n} n!})$.
So $lim_{x\to 0} f_1(x) = \infty$ because $lim_{xx\to 0} exp^{-x/2} = 1$.

Code

```
N=300;
x=linspace(0,8,N);
```

```
y=zeros(4,N);
y(1,:)=fk(x,1,1e300,30);
y(2,:)=fk(x,2,1e300,30);
y(3,:)=fk(x,3,1e300,30);
y(4,:)=fk(x,4,1e300,30);
plot(x,y(1,:),'r',x,y(2,:),'g',x,y(3,:),'b',x,y(4,:),'k')
legend('k=1','k=2','k=3','k=4')
```

For this function, we tried to use the definition of Γ with the integral, but it didn't work out, this is why there are the infty and n parameters.

```
function y=fk(x,k,infty,n)
if(x>=0 & mod(k,2)==0)
    y=(1/(2.^(k/2)).*factorial(k/2-1)).*x.^(k/2-1).*exp(-x/2);
elseif(x>=0 & mod(k,2)==1)
    y=1/(2.^(k/2)).*sqrt(pi)*factorial(k/2-1/2)./(2.^(2*k)*factorial(k/2-1/2)).*x.^(k/2-1/2).*exp(-x/2);
else
    y=0;
end
end
```

3.0.7 g

In order to compute the probability, we use this formula $P(X > \epsilon) = 1 - P(X < \epsilon) = 1 - \int_0^\epsilon f(x)$. The problem that have all the methods (except midpoint) is that they use the first born of the bracket, here it is 0 where $f_1 \to \infty$. But the midpoint do not use the same data $x_i = a + (i - \frac{1}{2})h$ (for trapezoid and Simpson's rule the definition of x is $x_i = a + ih$), indeed, when $a = 0$, the function do not diverge because $x_1 = 0 + \frac{h}{2} = \frac{h}{2}$(for trapezoid and Simpson's rule $x_1 = 0$). However, even if the probability do not diverge (as it happens with Trapezoid and Simposon's rule), the approximation is very bad, and strangely the approximation is better as the number of panels decrease. For example, the better approximation is when there is only one panel. This can be explained by the fact that as the function diverge, more we evaluate the function away from zero, less it is big and then better the value is.

Number of panels	1	10	100
Midpoint method	0.39	0.6	0.698
Trapezoid rule	$-\infty$	$-\infty$	$-\infty$
Simpson's rule	$-\infty$	$-\infty$	$-\infty$

Table 3.2: Value of $P(X > 3.3)$ computed with Midpoint

Code

```
close all; clear all; path(pathdef); clc;
k=1;infty=1e10;n=[1 10 100];
if(mod(k,2)==0)
    y=@(x)(1/(2.^(k/2)).*factorial(k/2-1)).*x.^(k/2-1).*exp(-x/2);
elseif(mod(k,2)==1)
    y=@(x)1/(2.^(k/2)).*sqrt(pi)*factorial(k/2-1/2)./(2.^(2*k)*factorial(k/2-1/2)).*x.^(k/2-1).*exp(-x/2);
end
format long
for i=1:3
    p1=1-myQuad(y,0,3.3,n(i),'mip');
    p2=1-myQuad(y,0,3.3,n(i),'sip');
```

```
    p3=1-myQuad(y,0,3.3,n(i),'tra');
    p=1-integral(y,0,3.3,'AbsTol',1e-16);
    e(1,i)=abs(p1-p);
    e(2,i)=abs(p2-p);
    e(3,i)=abs(p3-p);
end
e
```

3.0.8 h

However, the problem we had with the previous methods do not occurs here because. Actually, this can be explained by the fact the value computed by the function f_1 is equal to $\frac{3.3-0}{2}0 + \frac{a+b}{2}$ and not 0 for $x = 0$ and with 49 panels.

In order to find the value of $P(X > 3.3)$ correct to 6 decimal places, we compare it with the value computed with the function "integral" of matlab. For this comparison we prefer to plot the graph of the error against time because throughout different tries, we have observed that the error can increase over time.

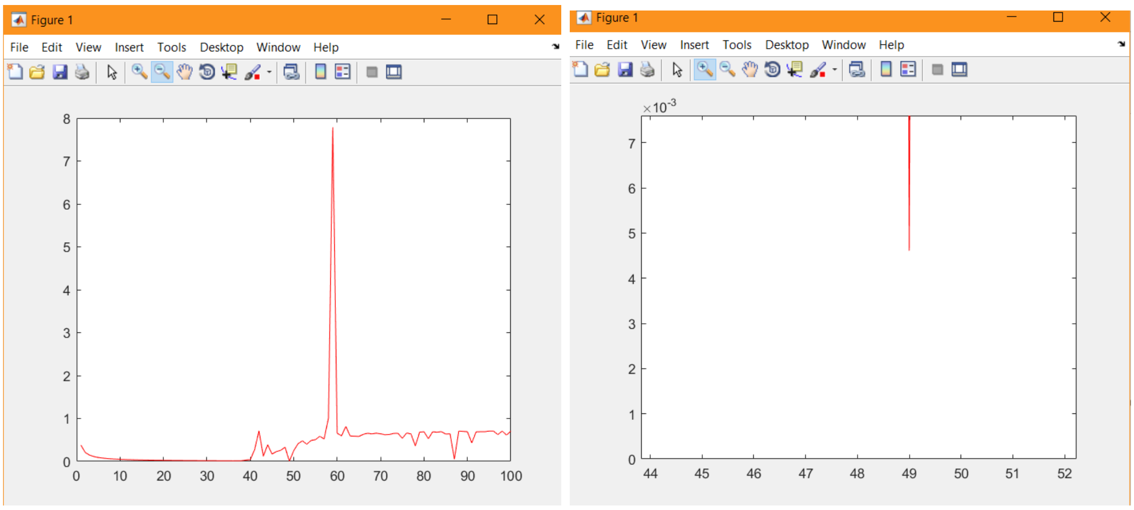

Figure 3.3: Finding the minimum error for $P(X > 3.3)$ with $k = 1$

We can observe that the minimal error is around 50, and on the second figure, we discover that it is 49. So the best we could find up to a hundred panels is for 49 panels and the error is 10^{-2}.

For k=4, we don't really need to go as far as for the previous one : with only four panels we reached an error around 10^{-7}.

	Value	Absolute error	Number of panels
$P(X > 3.3)$ for $k = 1$	0.27	10^{-3}	49
$P(X > 3.3)$ for $k = 4$	0.508932	10^{-7}	4

Table 3.3: Value of $P(X > 3.3)$ for $k = 1$ and $k = 4$

Code

```
k=1;infty=1e10;n=30;
if(mod(k,2)==0)
```

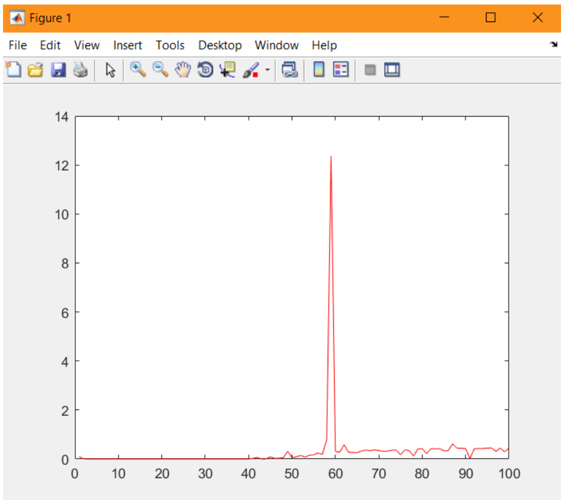

Figure 3.4: Finding the minimum error for $P(X > 3.3)$ with $k = 4$

```
    y=@(x)(1/(2.^(k/2)).*factorial(k/2-1)).*x.^(k/2-1).*exp(-x/2);
elseif(mod(k,2)==1)
    y=@(x)1/(2.^(k/2)).*sqrt(pi)*factorial(k/2-1/2)./(2.^(2*k)*factorial(k/2-1/2)).*x.^(k/2-1).*exp(-x/2);
end
format long
p1=1-inte(y,0,3.3,49)
p=1-integral(y,0,3.3,'AbsTol',1e-12)
e=abs(p-p1)

d=1;f=100;
mi=10;
for i=d:f
    pt=1-inte(y,0,3.3,i);
    et(i)=abs(pt-p);
    if(et(i)<mi)
        it=i;
        mi=et(i);
    end
end
mi% this is the value of the minimum error
it% this is the number of panels where the error is minimal
plot([1:length(et)],et,'r')

k=4;infty=1e10;n=30;
if(mod(k,2)==0)
    y=@(x)(1/(2.^(k/2)).*factorial(k/2-1)).*x.^(k/2-1).*exp(-x/2);
elseif(mod(k,2)==1)
    y=@(x)1/(2.^(k/2)).*sqrt(pi)*factorial(k/2-1/2)./(2.^(2*k)*factorial(k/2-1/2)).*x.^(k/2-1).*exp(-x/2);
end
format long
p1=1-inte(y,0,3.3,4)
p=1-integral(y,0,3.3,'AbsTol',1e-12)
e=abs(p-p1)
```

Comments on code

In order to determine, how many panels are the best, we use a loop and searching for the minimum error.

3.0.9 i

The only built-in comand that I know in order to compute the probability for k=4 accurate to 6 decimal. And acutally, this is the one I've used for the last two questions.

$P(X > 3.3) = 0.508932$

Code

```
k=4;
if(mod(k,2)==0)
    y=@(x)(1/(2.^(k/2)).*factorial(k/2-1)).*x.^(k/2-1).*exp(-x/2);
elseif(mod(k,2)==1)
    y=@(x)1/(2.^(k/2)).*sqrt(pi)*factorial(k/2-1/2)./(2.^(2*k)*factorial(k/2-1/2)).*x.^(k/2-1).*exp(-x/2);
end

p=1-integral(y,0,3.3,'AbsTol',1e-6)
```

Chapter 4

Runge-Kutta methods

4.0.1 a

A Butcher tableau allow us to find the coefficients for the different Runge-Kutta methods.

$$
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}
$$

Table 4.1: Butcher Talbeau

But the general tableau have few constrains.
$\sum_{i=1}^{s} b_i = 1$
$\sum_{j=1}^{s} a_{i,j} = c_i$

For example for the RK4, the corresponding tableau is :

$$
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{1}{2} & 0 & \frac{1}{2} & & \\
1 & 0 & 0 & 1 & \\
\hline
 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}
\end{array}
$$

Table 4.2: Butcher Talbeau

And we can verify that : $\frac{1}{2} = \frac{1}{2}$, $1 = 0 + 0 + 1$ and $\frac{1}{6} + \frac{1}{3} + \frac{1}{3} + \frac{1}{6} = 1$.

4.0.2 b

	RK4	Midpoint method	Heun's	Kutta's$\frac{3}{8} - rule$	Real
$y(1)$	0.74329	0.7428	0.7439	0.7432939	0.7432941
$Error$	10^{-7}	10^{-4}	10^{-4}	10^{-6}	

Table 4.3: Approximation of $y(1)$ according to the different mthods of Runge-Kutta

Code

```
a=0;b=1;h=0.1;y0=1;
N=(b-a)/h;
[t,yk4]=rk(@(x,t)-(x*t)/(1+t^2),a,b,y0,h,'4');
[t,ym]=rk(@(x,t)-(x*t)/(1+t^2),a,b,y0,h,'m');
[t,yH]=rk(@(x,t)-(x*t)/(1+t^2),a,b,y0,h,'H');
[t,yK]=rk(@(x,t)-(x*t)/(1+t^2),a,b,y0,h,'K');
yreal=1./sqrt(t.^2+1);
format long
yk4(end)
ym(end)
yH(end)
yK(end)
yreal(end)
e=[abs(yreal(end)-yk4(end)) abs(yreal(end)-ym(end)) abs(yreal(end)-yH(end))
    abs(yreal(end)-yK(end))]
```

4.0.3 c

Compute of the real value of $y(1)$

We have that $\frac{\dot{y(t)}}{y(t)} = -\frac{t}{1+t^2} \Leftrightarrow ln(y(t)) = -\frac{1}{2}ln(1+t^2)$. So $y(t) = \exp(ln((1+t^2)^{-1/2}) = \frac{1}{\sqrt{1+t^2}}$. Hence, $y(1) = \frac{1}{\sqrt{2}}$

Absolute error

The idea to compute the value of $y(1)$ is to look at a bracket and the value we want is at the end of the bracket (we can take $[0,1]$).
Indeed, as the previous value have been computed in detail thanks to a good value of h, the last value is $y(1)$. A problem if we take a too big bracket is a truncation error that could increase over the number of points studied. So the best to do is to take h as small as possible and the same for the bracket.

With the first figure (absolute error), we can verify that the efficiency of Runge-Kutta 4 is respected : from 10^{-1} to 10^{-3} the absolute error have been improved from 10^{-7} to 10^{-15}, so the global error is around $O(h^4)$ (which is the value maximum in the course). But after 10^{-15}, the error increase instead of decreasing whereas, the less efficient methods (Heun's and Midpoint) continue to improve. This can be explained by the truncation error, because each time we compute $m1, m2, m3$ and $m4$, there is a small truncation error that add, and at the end this is why the error increase again. But why this doesn't it happen with Heun's and Midpoint method ? This is probably because we don't compute $m1, m2, m3$ indeed there are many less calculus with truncation error. We can compare this fact with the Hermite Method, which optimize the use of the data, but where the 'noise' (here it is the truncation error) increase.

From the graph we can establish that the most efficient methods in decreasing order are : the fourth Runge-Kutta, Kutta $\frac{3}{8}$, Midpoint method and finally Heun's method. But the good point of the Kutta $\frac{3}{8}$ is that, the error caused by the truncation error increase less fast than the RK4 method.

Code

Function Runge-Kutta

```
function [t,y]=rk(f,a,b,y0,h,cm)
N=(b-a)/h;
y(:,1)=y0;
```

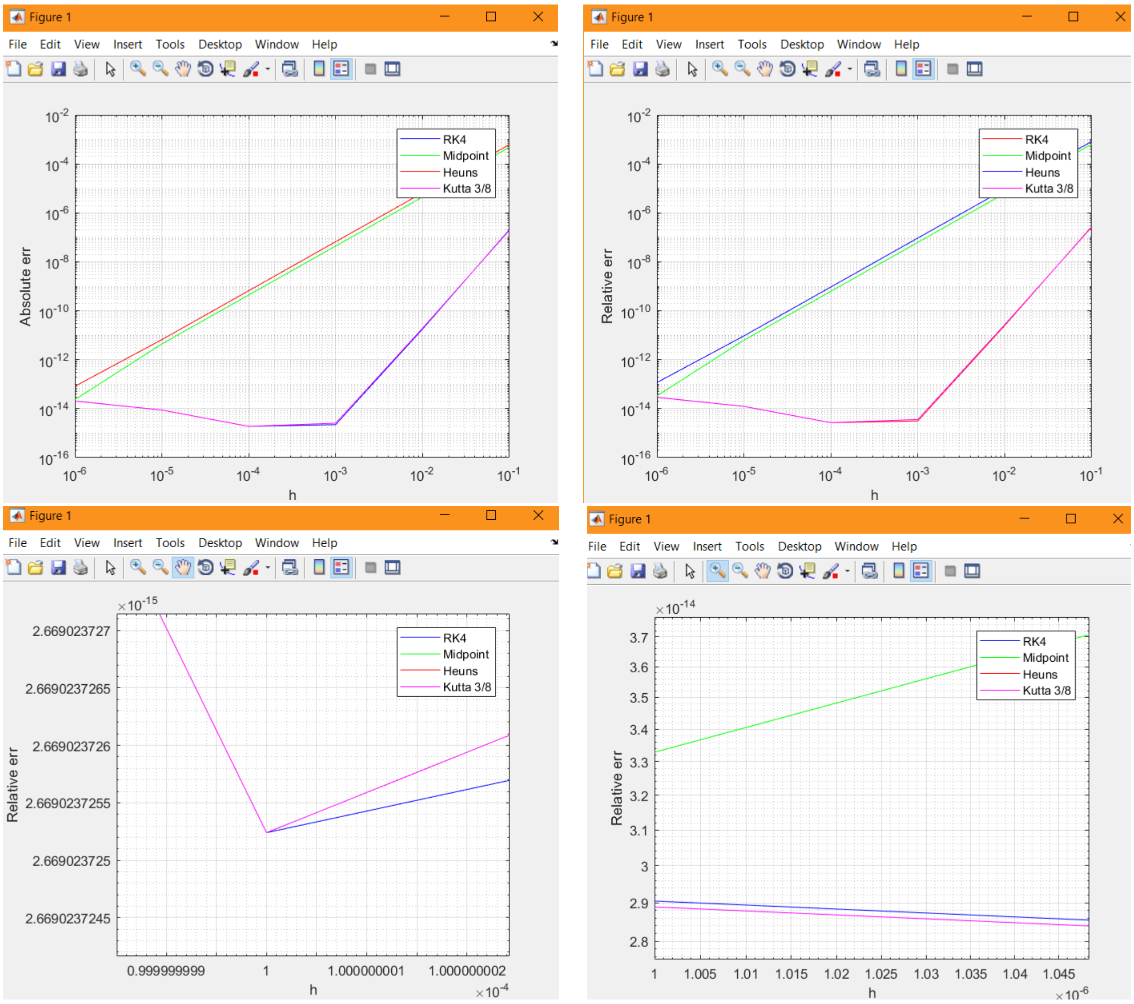

Figure 4.1: Absolute and relative error comparison of efficiency between the fourth method of Runge-Kutta

```
t(1)=a;
if(cm=='4')
 for k=1:N-1
    m1=f(y(:,k),t(k));
    m2=f(y(:,k)+(h/2)*m1,t(k)+(h/2));
    m3=f(y(:,k)+(h/2)*m2,t(k)+(h/2));
    m4=f(y(:,k)+h*m3,t(k)+h);
    y(:,k+1)=y(:,k)+(h/6)*(m1+2*m2+2*m3+m4);
    t(k+1)=a+h*k;
 end
end
if(cm=='m')
    for k=1:N-1
        K=f(y(:,k),t(k));
        m=f(y(:,k)+(h/2)*K,t(k)+(h/2));
        y(:,k+1)=y(:,k)+h*m;
        t(k+1)=a+h*k;
    end
end
```

```
if(cm=='H')
 for k=1:N-1
    K1=h*f(y(:,k),t(k));
    K2=h*f(y(:,k)+K1,t(k)+h);
    y(:,k+1)=y(:,k)+(1/2)*(K1+K2);
    t(k+1)=a+h*k;
 end
end
if(cm=='K')
 for k=1:N-1
    k1=h*f(y(:,k),t(k));
    k2=h*f(y(:,k)+(k1/3),t(k)+(h/3));
    k3=h*f(y(:,k)-k1/3+k2,t(k)+(2*h/3));
    k4=h*f(y(:,k)+k1-k2+k3,t(k)+h);
    y(:,k+1)=y(:,k)+(1/8)*(k1+3*k2+3*k3+k4);
    t(k+1)=a+h*k;
 end
end
```

Figure 4.1 second graph This code is not well optimized, we could have put yreal out of the loop.

```
close all; clear all; path(pathdef); clc;
p=-1;d=-6;% I can't succed to compute over 10^-6
h=logspace(p,d,6);
n=length(h);
y=@(x,t)-(x*t)/(1+t^2);
for i=1:n
    a=0;b=1;y0=1;
    N=(b-a)/h(i);
    [t,yk4]=rk(y,a,b,y0,h(i),'4');
    [t,ym]=rk(y,a,b,y0,h(i),'m');
    [t,yH]=rk(y,a,b,y0,h(i),'H');
    [t,yK]=rk(y,a,b,y0,h(i),'K');
    yreal=1./sqrt(t.^2+1);
    k4(i)=abs(yreal(end)-yk4(end));
    m(i)=abs(yreal(end)-ym(end));
    H(i)=abs(yreal(end)-yH(end));
    K(i)=abs(yreal(end)-yK(end));
end
loglog(h,k4,'b',h,m,'g',h,H,'r',h,K,'m')
xlabel('h')
ylabel('Absolute err')
legend('RK4','Midpoint','Heuns','Kutta 3/8')
grid on
figure
plot(t,yk4,'r',t,yreal,'k')
```

Figure 4.1 third graph and fourth

```
close all; clear all; path(pathdef); clc;
p=-1;d=-6;% I can't succed to compute over 10^-6
h=logspace(p,d,6);
n=length(h);
for i=1:n
    a=0;b=1;y0=1;
    N=(b-a)/h(i);
```

```
    [t,yk4]=rk(@(x,t)-(x*t)/(1+t^2),a,b,y0,h(i),'4');
    [t,ym]=rk(@(x,t)-(x*t)/(1+t^2),a,b,y0,h(i),'m');
    [t,yH]=rk(@(x,t)-(x*t)/(1+t^2),a,b,y0,h(i),'H');
    [t,yK]=rk(@(x,t)-(x*t)/(1+t^2),a,b,y0,h(i),'K');
    yreal=1./sqrt(t.^2+1);
    k4(i)=abs((yreal(end)-yk4(end))/yreal(end));
    m(i)=abs((yreal(end)-ym(end))/yreal(end));
    H(i)=abs((yreal(end)-yH(end))/yreal(end));
    K(i)=abs((yreal(end)-yK(end))/yreal(end));
end
loglog(h,k4,'b',h,m,'g',h,H,'r',h,K,'m')
xlabel('h')
ylabel('Relative err')
legend('RK4','Midpoint','Heuns','Kutta 3/8')
grid on
```

Chapter 5

Rock-scissors-paper

5.0.1 a

$$\frac{d(x+y+z)}{dt} = xy - xz + yz - yx + zx - zy = 0$$

5.0.2 b

$$\frac{d(xyz)}{dt} = yz\frac{d}{t} + xz\frac{dy}{t} + xy\frac{dz}{t}$$
$$\frac{d(xyz)}{dt} = yz(xy - xz) + xz(yz - yx) + xy(zx - zy)$$
$$\frac{d(xyz)}{dt} = xyz(x - z) + xyz(z - x) + xyz(x - y)$$
$$\frac{d(xyz)}{dt} = xyz[x - z + z - x + x - y]$$
$$\frac{d(xyz)}{dt} = 0$$

5.0.3 c

In order to see in detail each function, we will first plot for a small $t = 10$.

We can observe in the first graph, what we see mainly looks like the theory. First of all, no type is stronger than the other, on the graph we see that x, y and z oscillates with the same period and up to the same value and down to another same value.

But now, in order to see that x, y and z a continue to act like that indefinitely, we will plot this graph ones again but for another value of $t = 100$. Over the time, it seems that, the amplitude of x, y and z increase a bit abnormally. But the symmetry between x, y and z is conserved, they all increase of the same value.

Let plot for a much bigger $t = 1080$, in order to determine if this observation that goes against the theory is confirmed.

We see that, over the time, the amplitude of x, y and z definitively increase, then seems to stabilize before to diverge completely. The reason of the increase of the x, y and z amplitude is probably caused by the approximation error that add to itself overt time. Moreover, it makes perfect sens, seeing the definitions of the functions, that if one function diverge then all the derivatives do and indeed the functions also.

Code

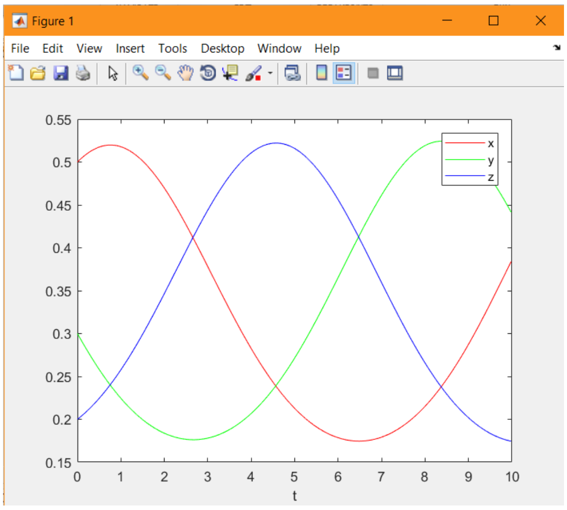

Figure 5.1: x, y and z against t for $h = 0.02$ and up to $t = 10$

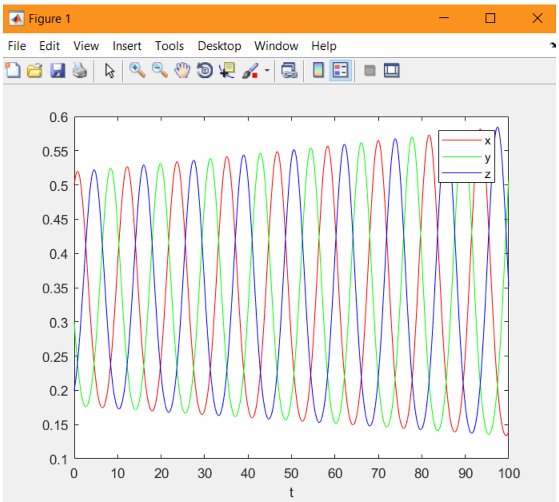

Figure 5.2: x, y and z against t for $h = 0.02$ and up to $t = 100$

```
a=0;b=1080;h1=0.02;y0=[0.5;0.3;0.2];h2=0.2;%with h1, it diverge for 1080, but this is because h1 is not enough precise because, when we use h2, it works perfectly
N=(b-a)/h1;
[t,yK]=rk(@(y,t)[y(1)*y(2)-y(1)*y(3),y(2)*y(3)-y(2)*y(1),y(3)*y(1)-y(3)*y(2)],a,b,y0,h1,'K');
plot(t,yK(1,:),'r',t,yK(2,:),'g',t,yK(3,:),'b')
legend('x','y','z')
xlabel('t')
```

5.0.4 d

We can observe, on the first graph, that the function $\frac{d(x+y+z)}{dt}$ is practically constant then start to diverge at $t = 1080$. So on the part that do not diverge the result is consistent with the mathematical theory proved in question (a). And the function start to diverge because it is the sum of 3 divergent functions where, two diverge to $-\infty$, indeed the sum diverges also to $-\infty$. But we can also observe, on the second graph, that the function $\frac{d(xyz)}{dt}$ follows a linear function.

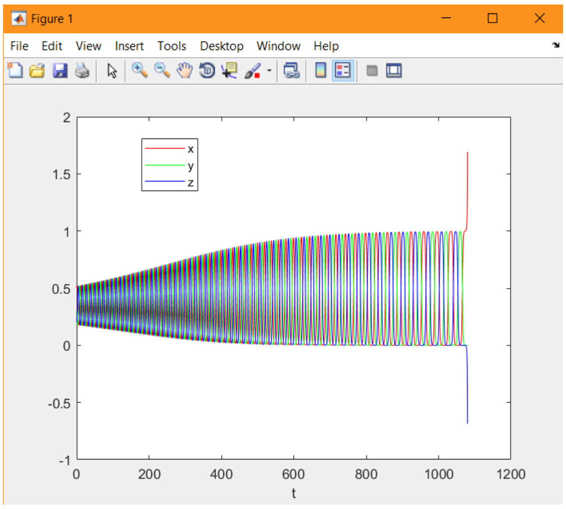

Figure 5.3: x, y and z against t for $h = 0.02$ and up to $t = 100$

And this is also confirmed by the third graph, which represents a linear least square curve-fitting method. This last observation is not consistent at all with the question (b). But this can be explained, as the amplitude of x, y and z increase over time, it is logical that the product decrease because we are talking about numbers under 1. This explanation is actually confirmed by the fact that when there is this stabilization in x, y, z, the product becomes almost a constant.
Indeed, why this increase of the amplitude do not affect that much the sum, because slight changes of values affect much more a product than a sum. And actually, if we look well, we can see that the sum increase slightly.

Code

We use here the functions programmed in the question 4.

```
a=0;b=1081;h1=0.02;y0=[0.5;0.3;0.2];h2=0.2;%with h1, it diverge for 80, but this is
    because h1 is not enough precise because, when we use h2, it works perfectly
N=(b-a)/h1;
[t,yK]=rk(@(y,t)[y(1)*y(2)-y(1)*y(3),y(2)*y(3)-y(2)*y(1),y(3)*y(1)-y(3)*y(2)],a,b,y0,h1,'K');
res=corr(t,yK(1,:).*yK(2,:).*yK(3,:),2);
plot(t,yK(1,:)+yK(2,:)+yK(3,:),'g')
xlabel('t')
ylabel('x+y+z')
figure
plot(t,yK(1,:).*yK(2,:).*yK(3,:),'g',t,res(1)+res(2)*t,'b')
xlabel('t')
ylabel('xyz')
```

5.0.5 e

With this new value of $h = 0.002$, we can observe that the diverges problems around $t = 1080$ (for x, y, z and $t = 80$ for $\frac{d(x+y+z)}{dt}$, $\frac{d(xyz)}{dt}$) was caused by the value of h not enough precise because this problems disappeared. Moreover, we now see that the predication, confirmed by the curve-fitting method, was good because now we see almost a linear decreasing function. And this do not get into contradiction with the explanation given in the previous question because there are no stabilization in the values of x, y, z.

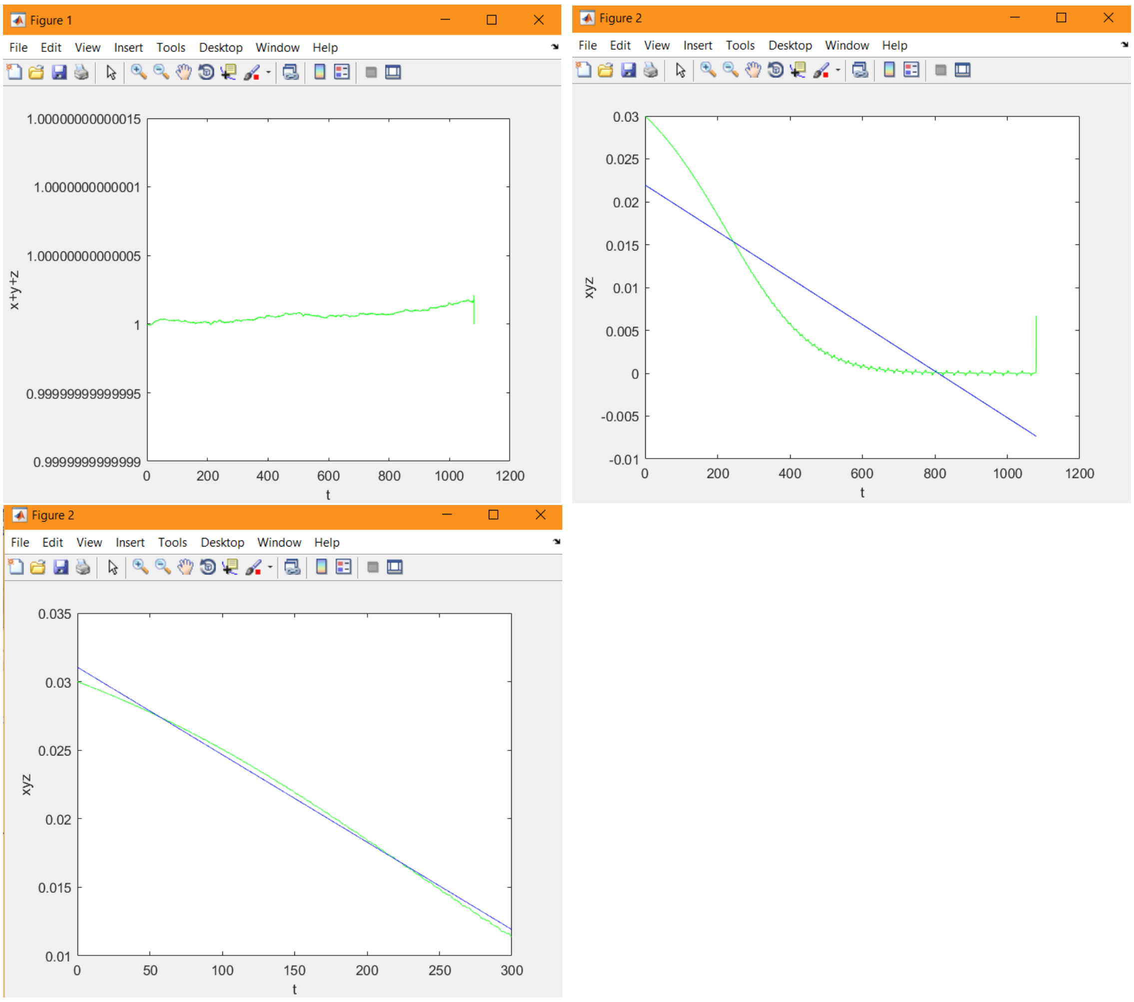

Figure 5.4: $\frac{d(x+y+z)}{dt}$ and $\frac{d(xyz)}{dt}$ against t for $h = 0.02$

Code

We use here the functions programmed in the question 4.

```
a=0;b=3000;h1=0.002;y0=[0.5;0.3;0.2];%with h1, it diverge for 1080, but this is because
    h1 is not enough precise because, when we use h2, it works perfectly
N=(b-a)/h1;
t=linspace(a,b,N);
[t,yK]=rk(@(y,t)[y(1)*y(2)-y(1)*y(3),y(2)*y(3)-y(2)*y(1),y(3)*y(1)-y(3)*y(2)],a,b,y0,h1,'K');
res=corr(t,yK(1,:).*yK(2,:).*yK(3,:),2);
plot(t,yK(1,:),'r',t,yK(2,:),'g',t,yK(3,:),'b')
legend('x','y','z')
xlabel('t')
figure
plot(t,yK(1,:)+yK(2,:)+yK(3,:),'g');
xlabel('t')
ylabel('x+y+z')
figure
plot(t,yK(1,:).*yK(2,:).*yK(3,:),'g',t,res(1)+res(2)*t,'b')
legend('data','curve-fiting method')
xlabel('t')
```

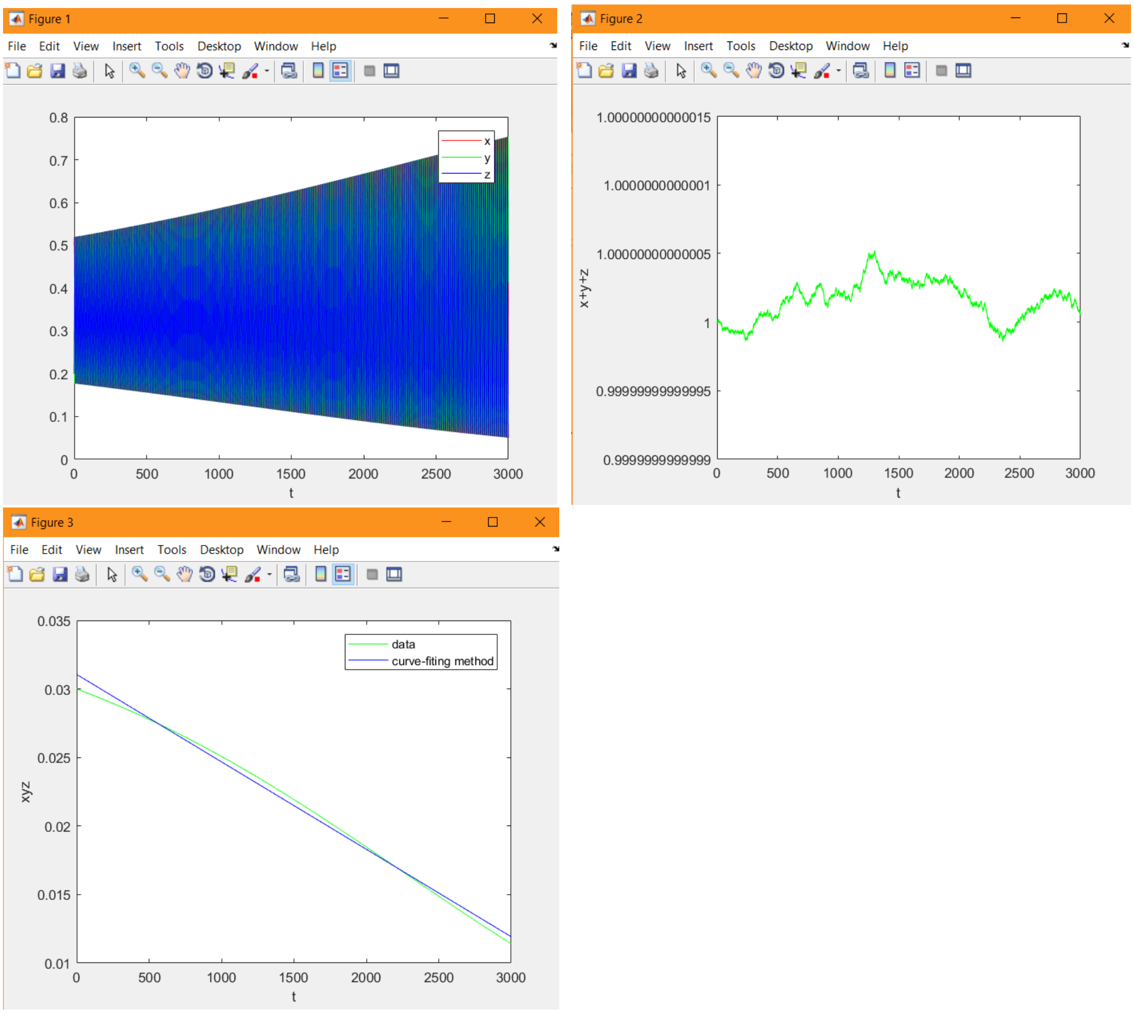

Figure 5.5: Graph of x, y, z, $x+y+z$, $\frac{d(x+y+z)}{dt}$ and $\frac{d(xyz)}{dt}$ against t for $h = 0.002$

```
ylabel('xyz')
```

5.0.6 f

If we increase again the time, up to $t = 5000$, we saw that, there are no divergence as when $h = 0.02$. So it seems confirmed that, this divergence was caused by the truncation and approximation error. Let now $t = 10^5$, we see that, there is a stabilization and that stay very long (more than $9 \cdot 10^4$). So is this because, the approximation with $ode45$ is very good, and there should be a divergence after a much longer time. I can't give answer to this question, but the most probable is that, the increase of amplitude is normal. And finally, we couldn't observe with Runge-Kutta method programmed in this report this final stabilization of x, y, z that correspond to the fact that no type is stronger than the others.

Code

```
y0=[0.5;0.3;0.2];
Tspan=1e5;
```

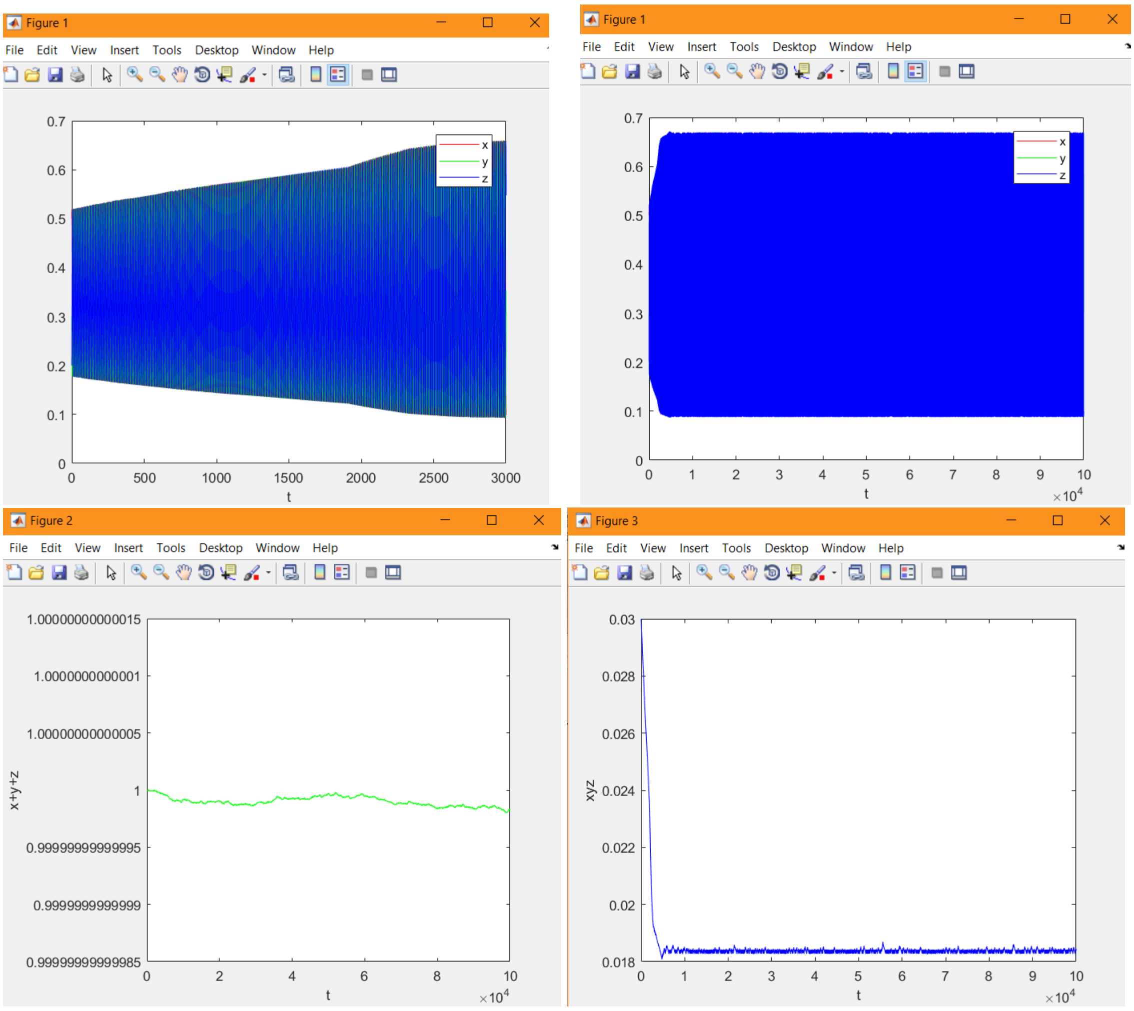

Figure 5.6: Graph of x, y, z, $x + y + z$, xyz against t comptued with $ode45$

```
[t,y]=ode45(@(t,y)[y(1)*y(2)-y(1)*y(3);y(2)*y(3)-y(2)*y(1);y(3)*y(1)-y(3)*y(2)],[0
    Tspan],y0);
%[t,y]=ode45(@(t,y)2*y,[0 2000],y0);
tspan = [0 5];
y0 = 0;
%[t,y] = ode45(@(t,y) 2*t, [-45,5], y0);
%plot(t,y)

plot(t,y(:,1),'r',t,y(:,2),'g',t,y(:,3),'b')
legend('x','y','z')
xlabel('t')
figure
plot(t,y(:,1)+y(:,2)+y(:,3),'g')
xlabel('t')
ylabel('x+y+z')
figure
plot(t,y(:,1).*y(:,2).*y(:,3),'b')
xlabel('t')
ylabel('xyz')
```

Part I

Annexe

Question 1 a

Code

```
function [nbsteps, value]=rootfinding2(f,fprime,a,b,n,err,cm)
%aerr: absolute error storage
%rerr : realtive error storage
%value : corresponds to the value of the nearest root function
%f: is the function
%fprime: derivative of the function
%a,b : is the bracket
%n: number maximum of steps used
%err: absolute error tolerances
%cm: correponds to the method we want to use

if(b-a<0)
   error("the values for a and be you've entered is not a bracket")
end

if(err<0)
   error("The error tolerance can't be negative")
end


x0=(a+b)/2;
if(cm=='nr')
x=x0;
c=0;
x2=x0+2*err;
while(c<n & abs(x2-x)>err)
   x2=x;
   t=x-f(x)./fprime(x);
   if(t>b |t<a)
      error('There is a convergence problem')
   end
   x=t;
   c=c+1;
end
value=x;
nbsteps=c;
end
if(cm=='se')
c=0;x=a;x2=b;
while(c<n & abs(x2-x)>err)
   t=x-((x-x2)/(f(x)-f(x2)))*f(x);
   x2=x;
   if(t>b |t<a)
      error('There is a convergence problem')
   end
   x=t;
   c=c+1;
end
value=x;
nbsteps=c;
end
if(cm=='rd')
   x1=a;x2=b;c=0;
   x4=(x1+x2)/2;
```

```
    xp=x4+2*err;
while(c<n & abs(xp-x4)>err)
    x3=(x1+x2)/2;
    xp=x4;
    x4=x3+(x3-x1)*(sign(f(x1)-f(x2))*f(x3))./sqrt(f(x3).^2-f(x1).*f(x2));
    if(x4>x2 |x4<x1)%this is impossible, we don't need it
       error('There is a convergence problem')
    end
    if(f(x1)*f(x4)<0)
        x2=x4;
    elseif(f(x2)*f(x4)<0)
        x1=x4;
    else
        break;
    end
    if(x1>x2)
        t=x1;
        x1=x2;
        x2=t;
    end
    c=c+1;
end
x=x4;
value=x;
nbsteps=c;
end
end
```

```
function [nbsteps, value]=ridder(f,a,b,n,err)
    x1=a;x2=b;c=0;
    x4=(x1+x2)/2;
    xp=x4+err;
while(c<n &(xp-x4)>10^(-err))
    x3=(x1+x2)/2;
    xp=x4;
    x4=x3+(x3-x1)*(sign(f(x1)-f(x2))*f(x3))./sqrt(f(x3).^2-f(x1).*f(x2));
    if(f(x1)*f(x4)<0)
        x2=x4;
    elseif(f(x2)*f(x4)<0)
        x1=x4;
    else
        break;
    end
    if(x1>x2)
        t=x1;
        x1=x2;
        x2=t;
    end
    c=c+1;
end
x=x4;
value=x;
nbsteps=c;
end
```