UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di scienze matematiche fisiche e naturali
CdL Magistrale in Informatica
Curriculum resilient and secure cyber-physical systems

Approximation Methods 2017-18: implemented models and algorithms

Alex Foglia
6336805
alex.foglia@stud.unifi.it

October 19, 2018

This relation describes the implemented algorithms, in MATLAB language, which solve some of the problems shown during lectures.
All the implemented routines are available at the following remote repository:
https://github.com/alexfoglia1/approximations_methods.git

Contents

Listings

List of Figures

1 Stirling Numbers

Consider the following:

$$x^n = \prod_{i=1}^{n} x$$

As the standard definition of numbers power in the continuos domain ($x \in \mathbb{R}$).
In the discrete domain, it is possible to define a discrete counterpart of number powers: pseudo powers.
Let $x, n \in \mathbb{N}$, we define x *pseudo-power* n as:

$$x^{(n)} = x(x-1)(x-2)\dots(x-n+1) = \frac{x!}{(x-n)!}$$

We can define number powers in terms of pseudo powers, in the way it follows:

$$x^n = \sum_{i=1}^{n} S_i^n x^{(n)}$$

Where the S_i^n are called *Stirling Numbers* (of second specie).
S_i^n are defined as:

$$S_n^n = S_1^n = 0$$
$$S_i^{n+1} = S_{i-1}^n + iS_i^n \quad i = 2, \dots, n$$

This recursive definition of Stirling Numbers makes it easy to implement a MATLAB function which receives n, k as input parameters and returns S_k^n.

```
function [s] = getStirling(n,k)
  if (k > n)
    s=0;
  elseif (k == 1 || n == k)
    s=1;
  elseif (k == 0 && n >= 1)
    s=0;
  else
    s = k * getStirling(n - 1, k)
      + getStirling(n - 1, k - 1);
  end
end
```

Listing 1: Stirling Numbers MATLAB function

Consider now the following function:

```
function [S] = stirling(n)
  S=zeros(n);
  for i=1:n
    for j=1:i
      S(i,j) = getStirling(i,j);
    end
  end
end
```

Listing 2: First n rows of Stirling Numbers

Given n, the function stirling(n) returns a matrix of size n x n containing the first n rows of Stirling Numbers.

When executing it, we obtain:

```
>> A = stirling(7)

A =

     1     0     0     0     0     0     0
     1     1     0     0     0     0     0
     1     3     1     0     0     0     0
     1     7     6     1     0     0     0
     1    15    25    10     1     0     0
     1    31    90    65    15     1     0
     1    63   301   350   140    21     1
```

Figure 1: First 7 rows of second specie Stirling Numbers

2 Fibonacci Numbers: the fatal bit problem

Consider the following finite difference equation:

$$y_{n+2} = y_{n+1} + y_n \qquad n \geq 0 \;\; (*)$$

Choosing $y_0 = y_1 = 1$ as the initial conditions, the expression evaluates to:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Which is known as the Fibonacci succession of numbers.
In order to get a closed form which explictly define the n-th element of the succession, we must solve the following difference equation:

$$\sum_{i=0}^{2} p_i y_{n+i} = 0 \;\; (**)$$

Where $p_0 = 1$, $p_1 = -1$ and $p_2 = -1$. Due to the fact that the $(**)$ is an *homogeneous* difference equation with constant coefficients, it is possible to explicit calculate an enclosed form for the n-th term of the succession.
In a numerical analysis environment, such as MATLAB, it is possible to avoid solve (**), but we may face to the *fatal bit problem*.
Consider the following MATLAB functions which calculates the terms of the succession defined by $(*)$, given y_1 and y_2 as the initial conditions.

```
function [yn] = fatalbit (len)
  yn = zeros(1,len);
  yn(1)=1;
  yn(2)=(1-sqrt(5))/2;
  for i=3:len
    yn(i)=yn(i-1)+yn(i-2);
  end
end
```

Listing 3: Fibonacci / Fatal Bit function in 64 bit precision

Consider to set the following initial conditions:

$$y_1 = 1$$

$$y_2 = \frac{1 - \sqrt{5}}{2}$$

Executing the following instructions:

```
yn = fatalbit(70);
semilogy(abs(yn),'.');
```

Listing 4: Plot of Fatal Bit

It is possibile to see how succession values starts to diverge for $n = 38$. This is due to the 64 bit double precision, in fact, in 32 bit precision the divergence starts for n = 17.

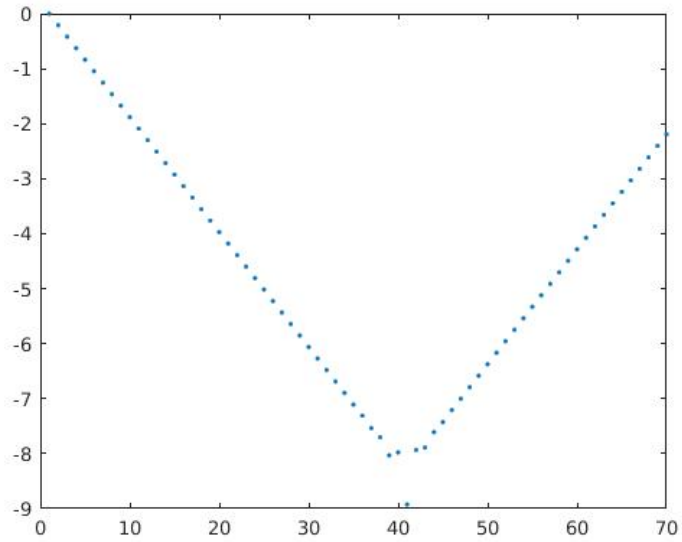

Figure 2: Effects of rounding in 64 bit precision

We can verify the divergence in 32 bit precision, modyfing the *fatalbit* function in this way:

```
function [yn] = fatalbit (len)
  yn = zeros(1,len);
  yn(1)=1;
  yn(2)=single((1-sqrt(5))/2);
  for i=3:len
    yn(i)=yn(i-1)+yn(i-2);
  end
end
```

Listing 5: Fibonacci / Fatal Bit function in 32 bit precision

Re-executing the previous seen script, we obtain the plot shown in figure 3.

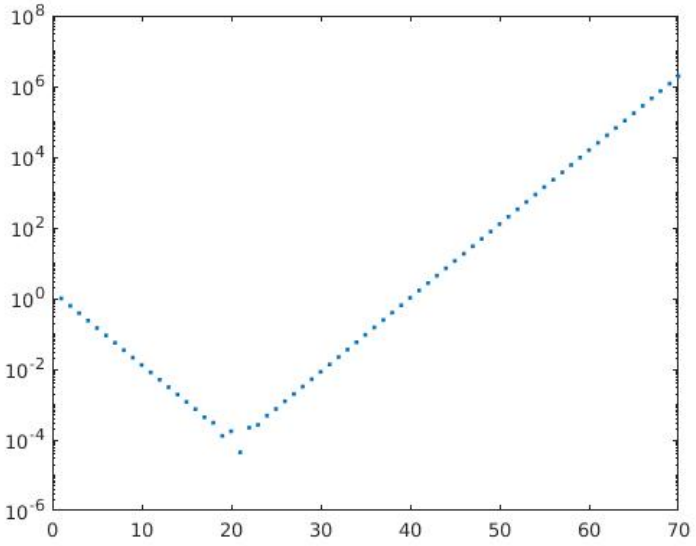

Figure 3: Effects of rounding in 32 bit precision

3 Mortgage repayment plan model

Suppose we want to contract a mortgage for a capital C, to repay in N installments at a rate i, which we assume it is constant.
We want to estimate the amount of a single installment r:
Let y_n be the residual debt after the $n-th$ payment. We have:

$$y_n = (1+i)y_{n-1} - r, \;\; y_0 = C, y_N = 0 \;\; (1)$$

In other words, the residual debt at time n is the residual debt at time $n-1$ plus accrued interests minus the installment amount.
homogeneous equation associated to (1) is:

$$y_n - (1+i)y_{n-1} = 0$$

Which characteristc polynome is:

$$z^2 - (i+1)z$$

And it has roots:

$$z_1 = 0, z_2 = (1+i)$$

Hence the general solution of the homogenous equation is:

$$y_n = \alpha(1+i)^n$$

Let us look for a particular solution for the non-homogeneous problem. In this case, we can find a constant solution for the (1):

$$\overline{y} = \frac{r}{i}$$

Hence the general solution for the (1) is:

$$y_n = \alpha(i+1)^n + \frac{r}{i}$$

Where α is obtained by imposing initial condition that $y_0 = C$. The final solution for the (1) is the following:

$$y_n = (C - \frac{r}{i})(i+1)^n + \frac{r}{i} \quad (2)$$

We can write a MATLAB function which calculates the residual debt at time N as follows:

```
function [yn] = loan(C,r,i,N)
  yn=zeros(1,N);
  yn(1)=C;
  for j=2:N
    yn(j)=(1+i)*yn(j-1) - r;
  end
end
```

Listing 6: Mortgage repayment function

Suppose we want to contract a morgage for a capital of $C = 140000$, with an interest of $i = 0.05 = 5\%$, single installment import defined as $r = C\frac{i}{1-(1+i)^{-N}}$, repayable in $N = 10$ installments. It is possible to use the following script in order to obtain a table where:

- First column is the residual debt at installments i
- Second column is the interest quote at installments i
- Third column is the capital quote at installments i

```
function [resDebt,intQuote,capQuote] = loantable(C,interest,singleimport,n)
  resDebt = loan(C,singleimport,interest,n);
  intQuote = zeros(1,n+1);
  intQuote(1) = 0;
  intQuote(2) = C*interest;
  for i = 3:n+1
    intQuote(i)=resDebt(i-1)*interest;
  end
  capQuote = zeros(1,n+1);
  capQuote(1) = 0;
  for i = 2:n+1
    capQuote(i) = singleimport-intQuote(i);
  end
end

Capital = 140000; %vary for other examples
Interest = 0.05;
N = 10;
Import = Capital*(Interest)/(1-(1+Interest)^-N);
[resDebt,intQuote,capQuote] = loantable(Capital,Interest,Import,N);
table = zeros(N+1,3);
for j=1:N+1
if j<=N
  table(j,1)=resDebt(j);
end
  table(j,2)=intQuote(j);
  table(j,3)=capQuote(j);
end
```

Listing 7: Mortgage repayment plan table

Executing this script, we obtain the following matrix: Which represents the following dataset:

```
table =

   1.0e+05 *

    1.4000         0         0
    1.2887    0.0700    0.1113
    1.1718    0.0644    0.1169
    1.0491    0.0586    0.1227
    0.9203    0.0525    0.1289
    0.7850    0.0460    0.1353
    0.6429    0.0392    0.1421
    0.4937    0.0321    0.1492
    0.3371    0.0247    0.1566
    0.1727    0.0169    0.1645
         0    0.0086    0.1727
```

Figure 4: Loan table obtained executing listing 7

Installment	Residual Debt	Interest Quote	Capital Quote
0	140000	-	-
1	128870	7000	11130
2	117180	6444	11690
3	104910	5860	12270
4	92030	5250	12890
5	78500	4600	13530
6	64290	3920	14210
7	49370	3210	14920
8	33710	2470	15660
9	17270	1690	16450
10	0	860	17270

4 Coweb Model

The coweb model describes the evolution of the market price related to an asset.
Let:

- p_n be the unitary price of an asset at time n;
- p_0 be the initial price;
- S_n be the asset units supplied at time n;
- D_n be the asset units demanded at time n.

We state that demand and supply are functions of the price. In fact:

$$S_n = g(p_{n-1}) \quad (1)$$

For sake of simplicity we assume g as follows:

$$g = b(p_{n-1}) + s_0 \quad (2)$$

Were $b, s_0 > 0$ represent respectively the response in productivity in base to the latest price, and the supply with a null price.
This function describes the tendence to increase productivity when price grows up, and decrease otherwise. We can substitute the (2) in the (1) obtaining:

$$S_n = b(p_{n-1}) + s_0$$

As concerns demand, we assume the following model:

$$D_n = -a(p_n) + d_0$$

Reasonably, it will be:

$$d_0 >> s_0 > 0$$

We obtain price dinamicity by imposing:

$$S_n = D_n \forall n$$

$$\Downarrow$$

$$b(p_{n-1}) + s_0 = -a(p_n) + d_0$$

$$\Downarrow$$

$$a(p_n) + b(p_{n-1}) = d_0 - s_0 \quad (3)$$

Homogeneous equation associated to the (3) has the following characteristic polynome:

$$az^2 + bz \quad (4)$$

(4) has the following roots:

$$az^2 + bz = 0 \Rightarrow z(az + b) = 0 \Rightarrow z_1 = 0,\ z_2 = -\frac{b}{a}$$

We note that exists a price $\overline{p}$, greater to 0 for hypotesis.

$$\overline{p} = \frac{d_0 - s_0}{a + b} > 0$$

$\overline{p}$ is a particular solution for (3) so we can write the general solution:

$$p_n = \overline{p} + \left(-\frac{b}{a}\right)^n (p_0 - \overline{p})$$

We can implement the following MATLAB function that explicitly calculates terms of the p_n S_n D_n successions, given a, b, p_0, s_0, d_0 as parameters:

```
function [pn,sn,dn,sfun,dfun] = coweb(d0,a,s0,b,p0,nmax)
  pn = zeros(1,nmax);
  sn = zeros(1,nmax);
  dn = zeros(1,nmax);
  sfun = @(x) b*x+s0;
  dfun = @(x) -a*x+d0;
  diff = (d0-s0);
  pn(1)= p0;
  sn(1)= s0;
  dn(1)= -a*p0+d0;
  for i = 2:nmax
    pn(i) = diff -b*pn(i-1)/a;
    sn(i) = sfun(pn(i-1));
    dn(i) = dfun(pn(i));
  end
end
```

Listing 8: Matlab Coweb Model

Note that *nmax* is the maximum time steps we want to calculate.
We can now execute these three scripts and observe how the model behavior is sensible to parameter variations:

```
[pnStable,snStable,dnStable,sfun,dfun] = coweb(100,0.15,10,0.10,400,10);
plot(pnStable);
hold on;
title('Stable Coweb');
plot(snStable);
plot(dnStable);
```

Listing 9: Stable Coweb Model

```
[pnUnstable,snUnstable,dnUnstable,sfun,dfun] = coweb(100,0.05,10,0.05, 950,10);
plot(pnUnstable);
hold on;
title('Unstable Coweb');
plot(snStable);
plot(dnStable);
```

Listing 10: Unstable Coweb Model

```
[pnDivergent,snDivergent,dnDivergent,sfun,dfun] = coweb(100,0.09,50,0.1, 300,10);
plot(pnDivergent);
hold on;
title('Divergent Coweb');
plot(snDivergent);
plot(dnDivergent);
```

Listing 11: Divergent Coweb Model

In the following figures, blue lines represent the p_n succession, yellow line represent the d_n succession, and brown lines represent the s_n succession.

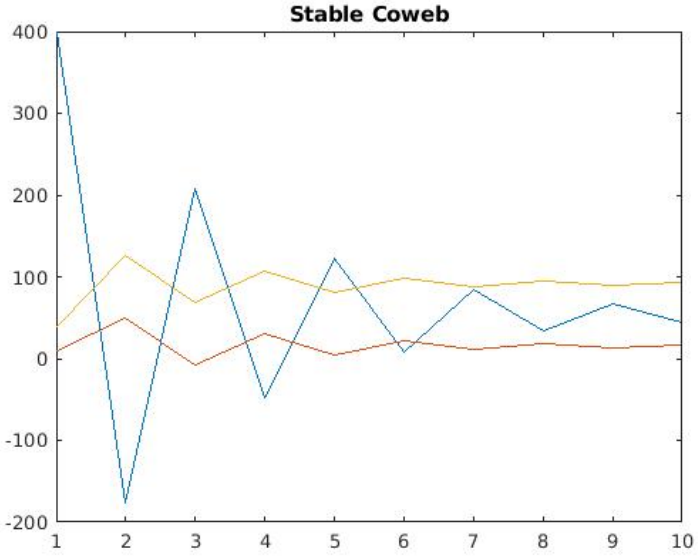

Figure 5: Plot of Stable Coweb successions

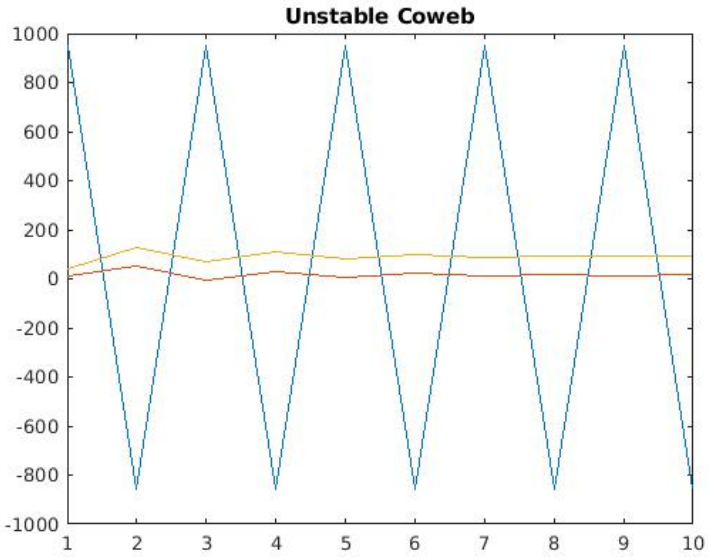

Figure 6: Plot of Unstable Coweb successions

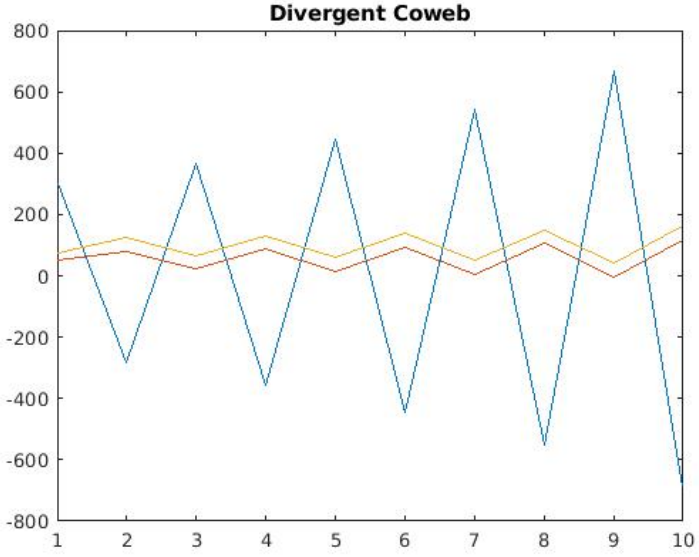

Figure 7: Plot of Divergent Coweb successions

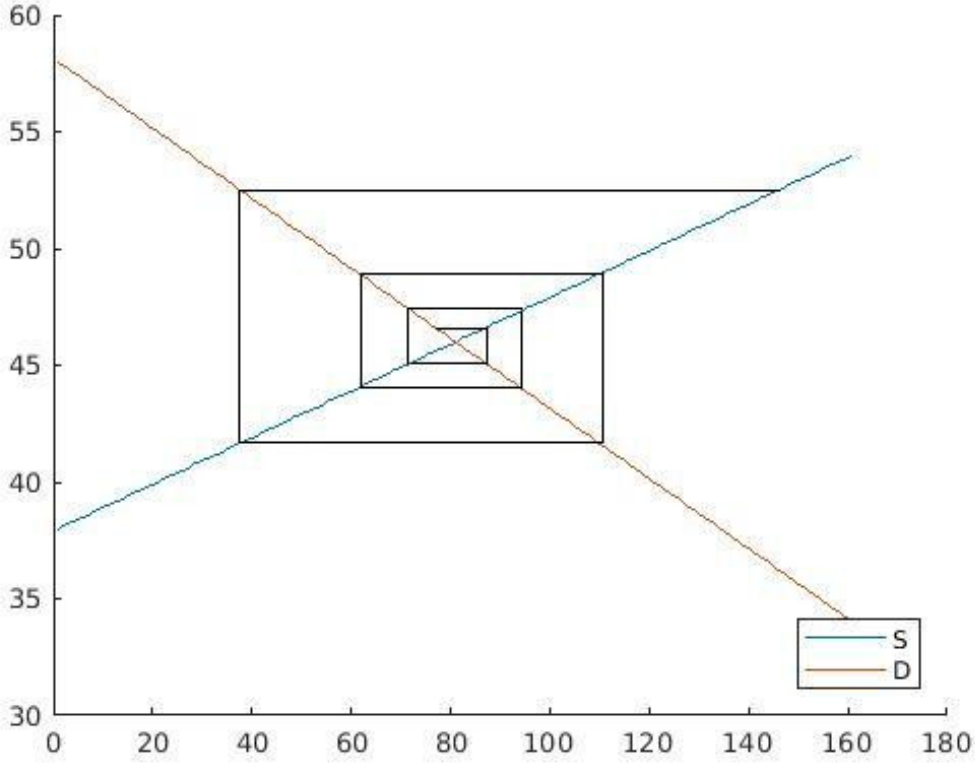

Figure 8: Supply and Demand in the stable case

5 Model of a national economy

This model describes dinamicity of the Gross Domestic Product of a nation. Let:

- Y_n be the GDP
- I_n be the private investments
- G_n be the governative expense
- C_n be the final consume

According to J.M.-Keynes theory, we have:

$$Y_n = I_n + G_n + C_n \quad (1)$$

Suppose it is:

$$C_n = \alpha Y_{n-1} \quad (2)$$

With $0 < \alpha < 1$. Suppose also that investments are proportional to consume increasings:

$$I_n = \rho(C_n - C_{n-1}), \quad \rho > 0 \quad (3)$$

By substituting (3) and (2) in the (1) we obtain:

$$G_n = Y_n - \alpha(\rho + 1)Y_{n-1} + \alpha\rho Y_{n-2} \quad (4)$$

The constant solution is:

$$\overline{Y} = \frac{G}{1 - \alpha}$$

And the characteristic polynome associated to the (4) is:

$$p(z) = z^2 - \alpha(\rho + 1)z + \alpha\rho$$

If we want that solution is asymptotically stable, $p(z)$ must be a Schur's polynome. For this purpose, it is enough that $\rho < \alpha^{-1}$.

1. If α, ρ are both close to 0 there is no economic growth, but the system is stable.
2. If $\alpha > 1$ the system is unstable.
3. If $\alpha \approx 1, \rho < \alpha^{-1}$ the system is stable and the economy stabilizes quickly to an high equilibrium point. This seems to be an optimal condition.
4. If $\alpha \approx 1, \rho\alpha \approx 1$ the system is in a stability limit and we have wide oscillations.
5. if $\alpha\rho > 1$ the system loses its stability.

We can implement a MATLAB functions which iteratively calculates the terms of Y_n, I_n, C_n because their definitions comes explicitly from the model.

```
function [Y,C,I] = samuelson(alpha,rho,g,y0,y1,nmax)
  Y=zeros(1,nmax);
  Y(1)=y0;
  Y(2)=y1;
  C=zeros(1,nmax);
  C(1)=alpha*y0;
  C(2)=C(1);
  I = zeros(1,nmax);
  I(1)=0;
  I(2)=rho*(C(2)-C(1));
  for i=3:nmax
    C(i)=alpha*Y(i-1);
    I(i)=rho*(C(i)-C(i-1));
    Y(i)=C(i)+I(i)+g;
  end
end
```

Listing 12: Samuelson Model

Observe now how Y (GDP) varies when varying α, ρ in the first 3 cases

1. α, ρ close to 0:

```
[Y,C,I]=samuelson(0.6,0.6,30,100,100,50);
plot(Y);
```

Listing 13: GDP first case plot

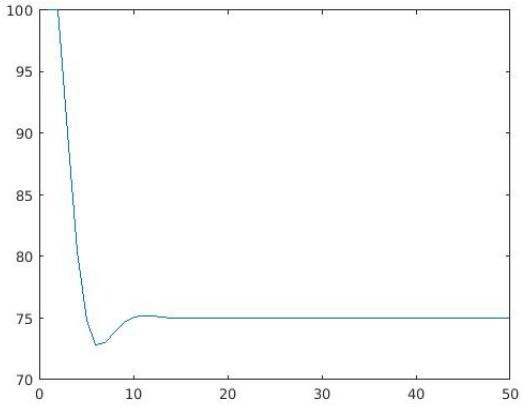

Figure 9: GDP values in case 1

2. $\alpha > 1$:

```
[Y,C,I]=samuelson(1.5,0.6,30,100,100,50);
plot(Y);
```

Listing 14: GDP second case plot

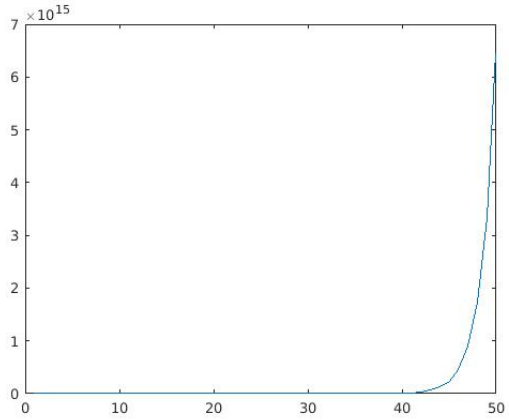

Figure 10: GDP values in case 2

3. $\alpha \approx 1, \rho < \alpha^{-1}$:

```
[Y,C,I]=samuelson(0.999,0.001,30,100,100,50);
plot(Y);
```

Listing 15: GDP third case plot

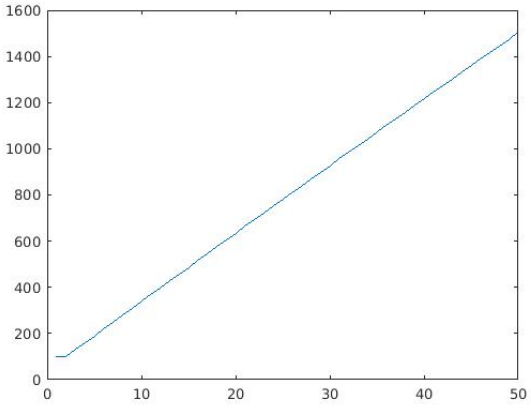

Figure 11: GDP values in case 3

6 Linear Multistep Methods

6.1 The problem

Consider the following problem:

$$\begin{cases} \dot{y} = f(t, y) \\ y(t_0) = y_0 \end{cases} \qquad (*)$$

This problem is known as the *cauchy problem* and it may be solved both in analytic way and numerical.
Since we are interested in numerical methods, our aim is to approximate function $y(t)$ knowing only its derivative $\dot{y} = f(t, y)$ and an initial condition $y(t_0) = y_0$.
Linear Multistep Methods are numerical methods which, fixed an interval $[a, b] \in D(y(t))$ and a number N of times we intend to divide such interval, solve the continuos problem shown in $(*)$ by defining a discrete domain t_i and approximating y values in that discrete domain.
t_i is defined as follows:

$$t_i = a + ih$$

Where:

$$i = 0, \ldots, N-1$$

$$h = \frac{b-a}{N}$$

For example:

$$a = 1,\ b = 2,\ N = 5$$

$$h = \frac{2-1}{5} = \frac{1}{5} = 0.25$$

$$t_0 = a + 0h = a = 1.00$$

$$t_1 = a + 1h = 1 + 0.25 = 1.25$$

$$t_2 = a + 2h = 1 + 2 * 0.25 = 1.50$$

$$t_3 = a + 3h = 1 + 3 * 0.25 = 1.75$$

$$t_4 = a + 4h = 1 + 4 * 0.25 = 2.00 = b$$

Hence:

$$\{t_i\} = \{1.00, 1.25, 1.50, 1.75, 2.00\}$$

We call such t_i a *uniform mesh* with *integration step* $h = 0.25$.
Let y_i be the approximation of the unknown $y(i)$ and remembering that $\dot{y} = f(t, y)$, we define:

$$f_i \equiv f(t_i, y_i)$$

Linear Multistep Methods, more formally, solve discrete problems in the form:

$$\sum_{i=0}^{k} \alpha_i y_{n+1} = h \sum_{i=0}^{k} \beta_i f_{n+1} \qquad n = 0, \ldots, N-k \ \ (*)$$

Where $\{\alpha_i\}$,$\{\beta_i\}$ and k define the particular *k-step linear multistep formula (LMF).*
It is important to distinguish between k and N, in fact, the first is an intrinsic characteristic of a particular LMF method, and the latter, indeed, is a characteristic of the problem we need to

solve. N is the number of sub-intervals we intend to divide the interval $[a, b]$, and it defines its relative *integration step* $h = \frac{b-a}{N}$. The greater is N, the smaller is h and the more accurated is the approximation with respect to the real function $y(t)$. Integration step and inherent method's step are two completely different concepts.
In conclusion, we define that a particular LMF method is completely identified by its ρ, σ polynomes, where:

$$\rho(z) = \sum_{i=0}^{k} \alpha_i z^i$$

$$\sigma(z) = \sum_{i=0}^{k} \beta_i z^i$$

With these definitions, we can write the $(*)$ problem in a more compact notation:

$$\rho(E)y_n - h\sigma(E)f_n = 0 \qquad n = 0, \ldots, N-k$$

6.2 An implementation

Linear Multistep Methods are distinguished in implicit and explicit. Indeed, if $\beta_k = 0$ we lose the dependence between y_{k+1} and f_{k+1}, which explicitly contains it. We call this case an explicit method, otherwhise if it were $b_k \neq 0$ then we will not lose this dependence and we would talk about an implicit method.
Here it follows an implementation of a general multistep method that, received as parameters the α_i and β_i, t_0, y_0 as the initial condition, the integration step h, the right extreme of the mesh t_f, and the known function $f(t, y)$ of which we want to approximate the primitive, returns an approximation of the function $y(t)$ and the desired mesh.

```
function [t,y] = implexpl(f,t0,y0,h,tf,ai,bi)
  if tf<t0(1)
    error('Last abscissa is less than the first');
  elseif length(t0)~=length(y0)
    error('Length of t0 is different from length of y0');
  elseif length(ai)~=length(bi)
    error('Length of ai is different from length of bi');
  elseif length(ai)~=length(t0)+1
    error('Length of ai and bi is different from length of y0 and t0 +1');
  elseif sum(ai)>0.00001
    warning('Sum of ai is not equal to 0.'+ ...
        'this will lead to an unconsistant result');
  elseif ai(end)==0
    error('last ai cannot be zero');
  end
  implicit = 0;
  if bi(end) ~= 0
    implicit = 1;
  end
  n = ceil((tf-t0(1))/h);
  t = NaN*ones(1,n+1);
  y = zeros(1,n+1);
  s1 = length(ai) -1;
  dim = length(t0);
  y(1:s1) = y0;
  t(1:s1) = t0;
  last_t = s1;
```

```
  k = 1;
  while t(last_t)<tf
    act_t = t(k:last_t);
    act_y = y(k:last_t);
    if implicit == 1
      [next_t,next_y] = impl(f,act_t,act_y,h,ai,bi,s1,dim);
    else
      [next_t,next_y] = expl(f,act_t,act_y,h,ai,bi,s1);
    end
  y(last_t+1) = next_y;
  t(last_t+1) = next_t;
  last_t = last_t + 1;
  k = k + 1;
  end
end
function [t,y] = impl(f,act_t,act_y,h,ai,bi,steps,dim)
  cl1 = 0;
  for j = 1:steps
    cl1 = cl1 + bi(j)*f(act_t(j),act_y(j));
  end
  cl2 = 0;
  for j = 1:steps
    cl2 = cl2 + ai(j) * act_y(j);
  end
  cl = ((h*cl1)-cl2)/ai(end);
  w = act_y(end);
  J = numjacobian(f,act_t,act_y);
  lastbi = bi(end);
  lastai = ai(end);
  new_abscissa = act_t(end) + h;
  A =  inv(eye(dim) -  h*( (lastbi/lastai)* J ) );
  B = w - h* ( (lastbi)*f( new_abscissa,w ) ) - cl;
  C = -A*B;
  tol = 10^-10;
  itmax = 1000;
  it = 0;
  while norm(C) > tol && it < itmax
    B = w - h* ( (lastbi)*f( new_abscissa,w ) ) - cl;
    C = -A*B;
    w = w + C;
    it = it+1;
  end
  if it == itmax
    warning("Reached max iteration");
  end
  y = w(1,1);
  t = new_abscissa;
end
function [t,y] = expl(f,act_t,act_y,h,ai,bi,steps)
  cl1 = 0;
  for j = 1:steps
    cl1 = cl1 + bi(j)*f(act_t(j),act_y(j));
  end
  cl2 = 0;
  for j = 1:steps
    cl2 = cl2 + ai(j) * act_y(j);
  end
  y = ((h*cl1)-cl2) / ai(end);
  t = act_t(end) + h;
end
```

```
function [f0] = numjacobian( fun, t0, y0 )
%
%  NUMERICAL JACOBIAN  (slightly adapted from the code BiM/BiMD).
%  2015.02.12
%
  m  = length(y0);
  Jf = zeros(m);
  f0 = reshape( feval( fun, t0, y0 ), m, 1 );
  dd = eps^(1/3);
  for i = 1:m
    ysafe = y0(i);
    delt  = sqrt( eps*max(dd,abs(ysafe)) );
    y0(i) = ysafe +delt;
    f1    = feval( fun, t0, y0 );
    Jf(:,i) = ( f1(:) -f0 )/delt;
    y0(i) = ysafe;
  end
  if nargout>1, f0 = f0.'; end
  return
end
```

Listing 16: General LMF solver

We note that this function behaves differently depending on the value of β_k where k is the length of the vector β_i. In fact, if this value is zero then it calculates the approximation using an explicit function, otherwise it relies on another function, which calculates the approximation implicitly by a fixed point iteration.
With the sole purpose of making the code more readable and modular, we define a class that we call LMMCoefficients in which we declare as constants the α_i, β_i of the main LMF methods. We have:

```
classdef LMMCoefficients
  properties(Constant)
    AB1A = ([-1.0,1.0]);
    AB1B = ([1.0,0.0]);
    AB2A = ([0.0, -1.0 , 1.0]);
    AB2B = ([-0.5 , 1.5, 0.0]);
    MIDA = ([-1.0, 0.0,  1.0]);
    MIDB = ([0.0,  2.0,  0.0]);
    AB3A = ([0.0, 0.0, -1.0 , 1.0]);
    AB3B = ([5.0 / 12.0, - 4.0 / 3.0, 23.0 / 12.0, 0.0]);
    AB4A = ([0.0, 0.0, 0.0,-1.0, 1.0]);
    AB4B = ([- 3.0 / 8.0, 37.0 / 24.0, -59.0 / 24.0, 55.0 / 24, 0.0]);
    AB5A = ([0.0, 0.0, 0.0, 0.0, -1.0, 1.0]);
    AB5B = ([251.0 / 720.0, -637.0 / 360.0, 109.0 / 30.0, ...
    -1387.0 / 360.0, 1901.0 / 720.0, 0.0]);
    AM1AA = ([-1.0, 1.0]);
    AM1AB = ([0.0, 1.0]);
    AM1BA = ([-1.0, 1.0]);
    AM1BB = ([0.5, 0.5]);
    AM2A = ([0.0, -1.0, 1.0]);
    AM2B = ([-1.0 / 12.0, 2.0 / 3.0, 5.0 / 12.0]);
    AM3A = ([0.0, 0.0, -1.0, 1.0]);
    AM3B = ([1.0 / 24.0, -5.0 / 24.0, 19.0 / 24.0, 3.0 / 8.0]);
    AM4A = ([0.0, 0.0, 0.0, -1.0, 1.0]);
    AM4B = ([-19.0 / 720.0, 106.0 / 720, -264.0 / 720.0, 646.0 / 720.0, ...
    251.0 / 720]);
    BDF1A = ([-1.0, 1.0]);
    BDF1B = ([0.0, 1.0]);
    BDF2A = ([1.0 / 3.0, -4.0 / 3.0, 1.0]);
```

```
    BDF2B = ([0.0, 0.0, 2.0 / 3.0]);
    BDF3A = ([-2.0 / 11.0, 9.0 / 11.0, -18.0 / 11.0, 1.0]);
    BDF3B = ([0.0, 0.0, 0.0, 6.0 / 11.0]);
    BDF4A = ([3.0 / 25.0, -16.0 / 25.0, 36.0 / 25.0, -48.0 / 25.0, 1.0]);
    BDF4B = ([0.0, 0.0, 0.0, 0.0, 12.0 / 25.0]);
    BDF5A = ([-12.0 / 137.0, 75.0 / 137.0, -200.0 / 137.0, 300 / 137.0, ...
     -300.0 / 137.0, 1.0]);
    BDF5B = ([0.0, 0.0, 0.0, 0.0, 0.0, 60.0 / 137.0]);
    BDF6A = ([10.0 / 147.0, -72.0 / 147.0, 225.0 / 147.0, -400.0 / 147.0, ...
     450.0 / 147.0, -360.0 / 147.0, 1.0]);
    BDF6B = ([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 60.0 / 147.0]);
  end
end
```

Listing 17: LMM Coefficients

Consider the following problem:

$$\begin{cases} \dot{y} = 10y - 2y^2 \\ y(0) = 1 \end{cases}$$

The exact solution will be

$$y(t) = \frac{5}{1 + 4e^{-10t}}$$

Note that:

$$\lim_{t \to +\infty} y(t) = 5$$

Now that we have the class LMMCoefficients and the function implexpl, we can test several different LMF methods in the above case with the following script:

```
fun = @(t,y) 10*y-2*y.^2;
FUN = @(t) 5/(1+4*exp(-10*t));
tf = 3;
h = 0.01;
ai = LMMCoefficients.AB1A; %vary for other plots
bi = LMMCoefficients.AB1B; %vary for other plots
leny0 = length(ai)-1;
t0 = zeros(1,leny0);
y0 = zeros(1,leny0);
t0(1)=0;
y0(1)=1;
for i=2:leny0
  t0(i)=t0(i-1)+h;
  y0(i)= FUN(t0(i));
end
[ti,appr]=implexpl(fun,t0,y0,h,tf,ai,bi);
n = length(ti);
real_values = zeros(1,n);
for i=1:n
  real_values(i)=FUN(ti(i));
end
hold on;
plot(ti,appr,'DisplayName','Approximation');
grid on;
plot(ti,real_values,'DisplayName','Exact solution');
legend(gca,'show','Location','best');
```

Listing 18: Test LMF

Executing the following script, we can see the error committed by the methods.

```
comparelmf;
close all;
error = abs(real_values-appr);
plot(error,'DisplayName','Error');
hold on;
grid on;
legend(gca,'show','Location','best');
```

Listing 19: LMF Error

We note that the midpoint formula makes spurious oscillations starting from $t \approx 1.4$. Decreasing the step size to $h = 0.001$ does not solve the problem, the oscillations are only translated at $t \approx 2$.

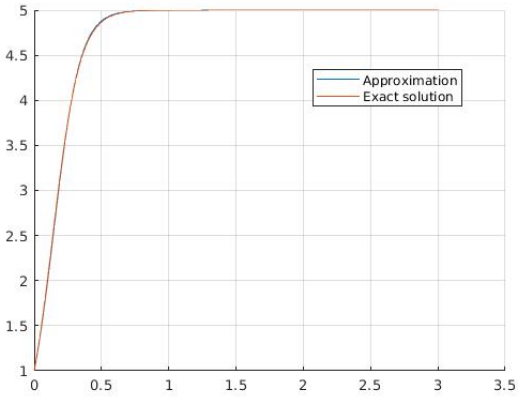

Figure 12: Order 1 Adams Bashforth with step size h = 0.01

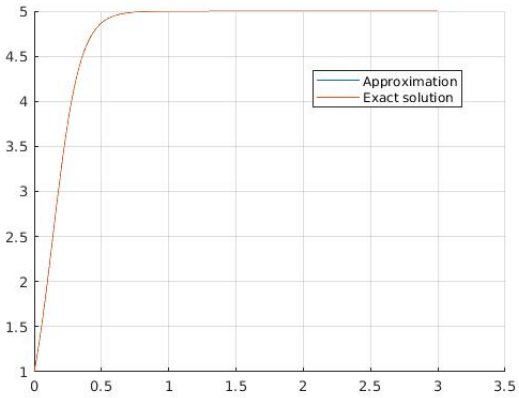

Figure 13: Order 1 Adams Bashforth with step size h = 0.001

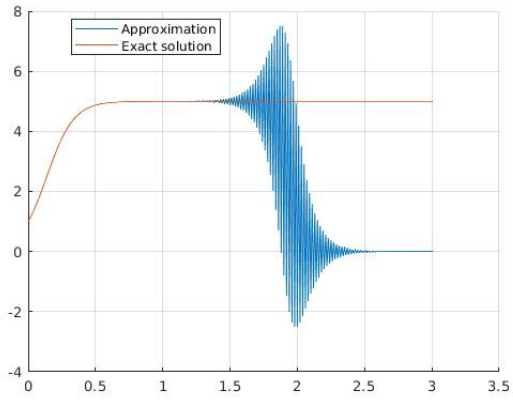

Figure 14: Order 2 Midpoint formula with step size h = 0.01

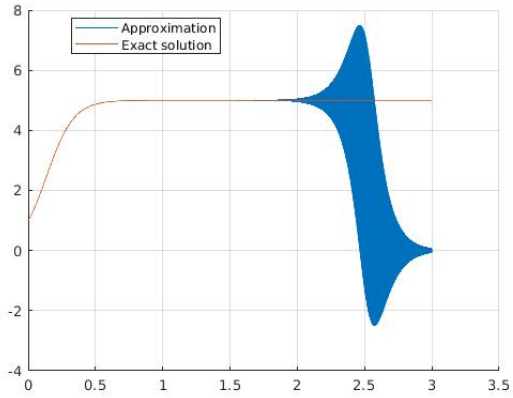

Figure 15: Order 2 Midpoint formula with step size h = 0.001

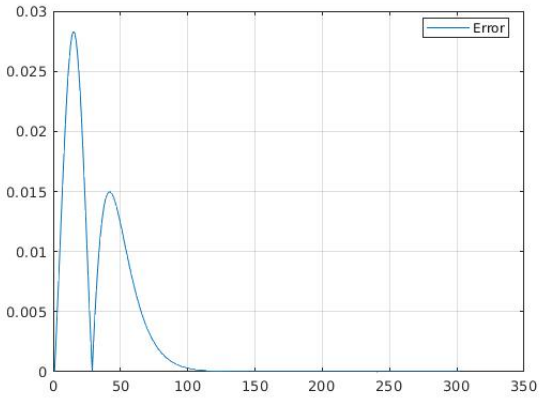

Figure 16: Order 1 Adams Bashforth error with step size h = 0.01

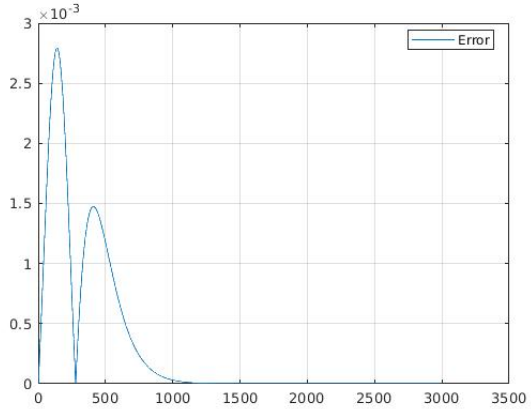

Figure 17: Order 1 Adams Bashforth error with step size h = 0.001

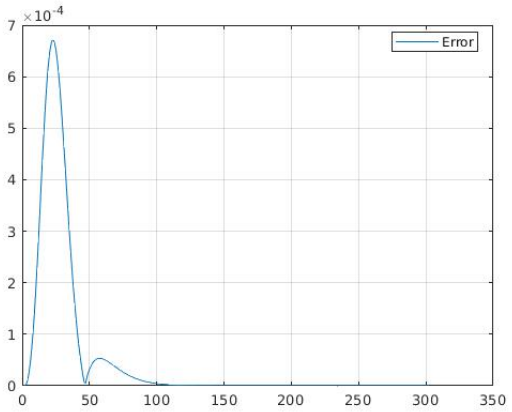

Figure 18: Order 1 Adams Moluton error with step size h = 0.01

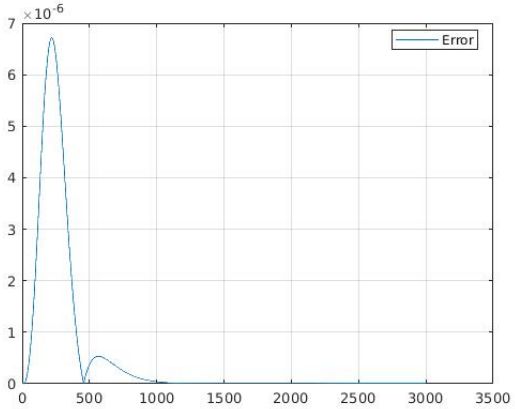

Figure 19: Order 1 Adams Moulton error with step size h = 0.001

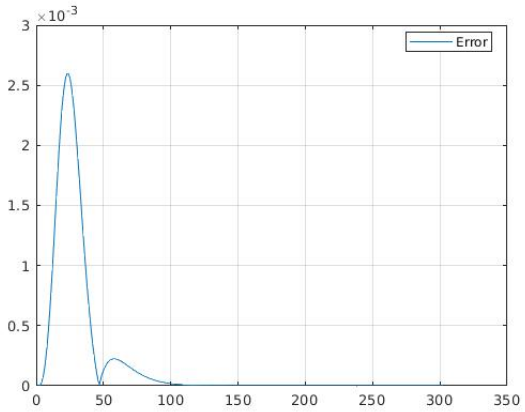

Figure 20: Order 2 Backward Differentiae Formula error with step size h = 0.01

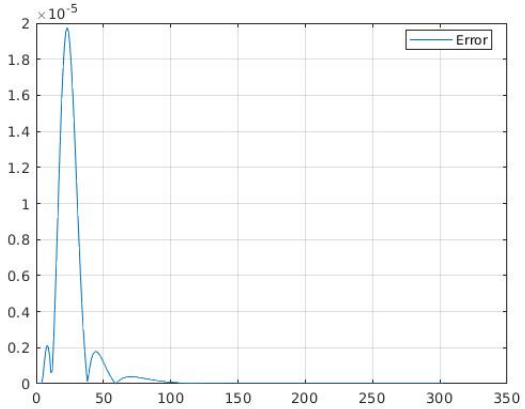

Figure 21: Order 4 Backward Differentiae Formula error with step size h = 0.01

6.3 LMF vs Runge Kutta methods

First we define the *local truncation error* τ_{n+k} as:

$$\tau_{n+k} = \sum_{i=0}^{k} \alpha_i y(t_{n+i}) - h \sum_{i=0}^{k} \beta_i \hat{f}_{n+i} \quad n = 0, \dots, N-k$$

Where, once assigned $y(t_n)$ as the exact solution of $y(t)$ evaluated on mesh nodes $\{y(t_n)\}$:

$$\hat{f}_n = f(t_n, y(t_n))$$

We state that a LMF method has *order* p if and only if:

$$\tau_{n+k} = O(h^{p+1})$$

The order of a LMF method gives useful informations about the error affecting the approximation of the integrand function. The bigger is the order, the smaller is the error.
LMF methods ((ρ, σ) methods) are characterized by an important limit on their order: the *Dahlquist barriers*. *Runge Kutta* methods (from here on, RK methods) are iterative methods designed to overcome the *Dahlquist barriers*.
RK methods are the ones used by MATLAB in its built-in functions `ode45` and `ode23`.
RK methods require more than one function evaluation of the integrand function $f(t, y)$ in each interval $[t_n, t_{n+1}] \in \{t_i\}$ (remember that $\{t_i\}$ is our mesh of integration).
In its most general form, a RK method can be written as:

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i K_i$$

Where:

$$K_i = f(t_n + c_i h, y_n + h \sum_{j=1}^{s} a_{ij} K_j), \quad i = 1, \dots, s$$

and s is the number of *stages* of the RK method, and coefficients $\{a_{ij}\}, \{b_i\}, \{c_i\}$ completely characterize a RK method, likewise α_i and β_i in LMF methods, and they are stored in a particular matrix called *Butcher's array*:

$\{c_i\}$	$\{a_{ij}\}$
	$\{b_i\}$

One of the most known RK method is the following:

$$y_{n+1} = y_n + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4) \quad (*)$$

Where

$$K_1 = f_n$$
$$K_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} K_1)$$
$$K_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} K_2)$$
$$K_4 = f(t_{n+1}, y_n + hK_3)$$

This method is derived by appling Simpson's quadrature formula to integrate f between mesh nodes.
Consider the following MATLAB function, which perform approximation using the RK method $(*)$:

```
function [yn] = explrungekutta(y0,intval,f)
  N = length(intval);
  h = intval(2)-intval(1);
  yn = zeros(1,N);
  yn(1) = y0;
  for i=1:N-1
    Y1 = f(intval(i),yn(i));
    Y2 = f(h/2+intval(i),yn(i)+h/2*Y1);
    Y3 = f(h/2+intval(i),yn(i)+h/2*Y2);
    Y4 = f(intval(i)+h,yn(i)+h*Y3);
    yn(i+1) = yn(i) + (h/6)*(Y1+2*Y2+2*Y3+Y4);
  end
end
```

Listing 20: Explicit RK implementation

Let us examine the behavior of this RK method in the case shown before:

$$\begin{cases} \dot{y} = 10y - 2y^2 \\ y(0) = 1 \end{cases}$$

$$y(t) = \frac{5}{1 + 4e^{-10t}} \qquad \lim_{t \to +\infty} y(t) = 5$$

Recap that the mesh is:

$$\{t_i\} \in [0, 2.5]$$

$$h = 0.001$$

Executing the following script:

```
comparelmf; %inherit y0, fun, ti and real_values
close all;
RK = explrungekutta(y0(1),ti,fun);
hold on;
plot(ti,real_values,'DisplayName','Exact Solution');
plot(ti,RK,'DisplayName','Runge Kutta');
legend(gca,'show','Location','southeast');
```

Listing 21: Runge Kutta approximation plot

We obtain the plot shown in figure 22.
The error committed by the implemented RK method, however, is not always 0. Consider the following script:

```
comparelmf;
close all;
errorlmf = abs(real_values-appr);
RK = explrungekutta(y0(1),ti,fun);
plot(ti,errorlmf,'DisplayName','LMF Error');
errorrk = abs(real_values-RK);
hold on;
grid on;
plot(ti,errorrk,'DisplayName','RK Error');
legend(gca,'show','Location','best');
```

Listing 22: Plot of RK error

Executing it, we obtain the result shown in figure 23.

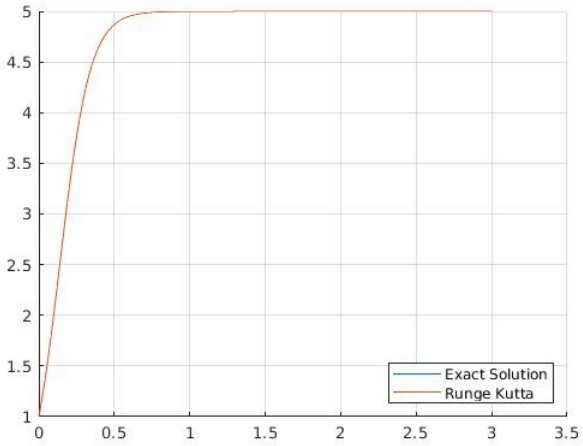

Figure 22: The approximation performed by the shown RK method overlaps with the exact solution

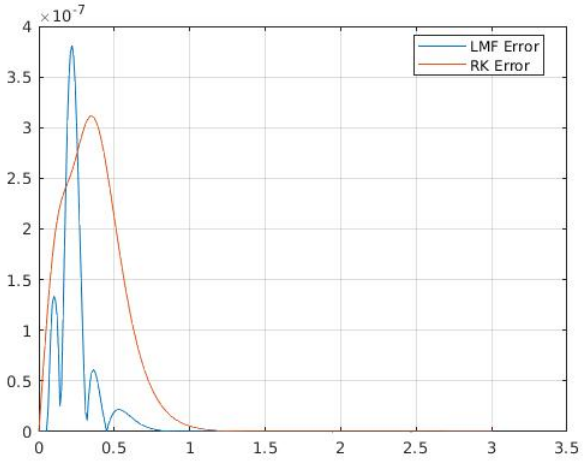

Figure 23: Error of the implemented RK method compared with the BDF Formula of order 6

We note that the implemented RK method commits its maximum error approximately at $t_i \approx 0.45$, and this error results $\approx 3 * 10^{-11}$. However, after a settling period in which the error is however substantially negligible, we note that for $t_i >\approx 1$ the error becomes null.
In conclusion, we state that RK methods are characterized by a greater computational complexity with respect to LMF methods since they perform sN function evaluations, where s is the method's number of stages and N is the number of mesh nodes, but they are:

1. More precise, since they overcome the first Dahlquist barrier;
2. More stable, since they overcome the second Dahlquist barrier.

6.4 Linear Multistep Methods: The Boundary Locus

A particular Linear Multistep Method is identified by its ρ, σ polynomes.
Considering a (ρ, σ) method, we define $D = \{q \in \mathbb{C} : \pi(z, q) \in S\}$ where S is the set of Schur's polynomes, and

$$\pi(z, q) = \rho(z) - q\sigma(z)$$

is the stability polynome associated to the test equation, as the absolute stability region of the considered (ρ, σ) method.
In general, determining the stability region for a LMF method using Schur's criteria could result difficult, so we can study the border of this region: the boundary locus.
The boundary locus is defined as follows:

$$\Gamma = \left\{ q(\theta) = \frac{\rho(e^{i\theta})}{\sigma(e^{i\theta})} : 0 \leq \theta \leq 2\pi \right\}$$

Given the following MATLAB functions, which calulcates (ρ, σ) polynomes of LMF methods:

- BDF
- Adams-Moulton
- Adams-Bashfort

```
function [ro,sigma] = lmf( tipo, k )
%
%  [ro,sigma] = lmf( tipo, k )      Calcola i polinomi ro e sigma del metodo
%                                   lmf a k passi specificato dal tipo:
%
%                                    0 : BDF
%                                    1 : Adams-Moulton
%                                    2 : Adams-Bashforth
%
  if tipo==0     % BDF
    sigma = [1 zeros(1,k)];
    b     = [0:k].*[k.^[0:k]]/k;
    ro    = vsolve(k:-1:0,b);
  elseif tipo==1 % Adams-Moulton
    ro    = [1 -1 zeros(1,k-1)];
    j     = [1:k+1];
    b     = (k.^j-(k-1).^j)./j;
    sigma = vsolve( k:-1:0, b );
  elseif tipo==2 % Adams-Bashforth
    ro    = [1 -1 zeros(1,k-1)];
    j     = [1:k];
    b     = (k.^j-(k-1).^j)./j;
    sigma = [0 vsolve( k-1:-1:0, b )];
  else
    disp('   tipo invalido!'), disp(' '),
  help lmf, ro=[]; sigma=[];
  end
return

function f = vsolve( x, b )
%
%     f = vsolve( x, b )      Risolve il sistema lineare W(x) f = b,
%                             dove W(x)  e' la Vandermonde definita
%                             dagli elementi del vettore x.
%
  f = b;
  n = length( x )-1;
  for k = 1:n
    for i = n+1:-1:k+1
      f(i) = f(i) - x(k)*f(i-1);
    end
  end
  for k = n:-1:1
    for i = k+1:n+1
      f(i) = f(i)/( x(i) - x(i-k) );
    end
  for i = k:n
    f(i) = f(i) - f(i+1);
  end
  end
return
```

Listing 23: LMF solver

We can write the following MATLAB function which explicitly calculates boundary locus values for a given LMF method:

```
function [q] = boundaryLocus(theta0,steps,method)
  if method<0 || method > 2
    disp('metodo non valido\n');
    return;
  end
  theta=linspace(theta0,2*pi+theta0,100);
  [ro,sigma]=lmf(method,steps);
  q=zeros(1,100);
  for j=1:100
    q(j)=polyval(ro,exp(1i*theta(j)))/polyval(sigma,exp(1i*theta(j)));
  end
end
```

Listing 24: Boundary Locus calculation

Consider now to execute the following script:

```
method=input('insert method to use\n');
if method<0 || method > 2
  disp ('unvalid method');
  return;
end
steps=input('insert max number of steps\n');
theta0=input('insert theta initial value\n');
hold all;
for j=1:steps
  q=boundaryLocus(theta0,j,method);
  plot(q,'DisplayName',['number of steps = ' num2str(j)]);
  if method==0--
    strm="BDF";
  elseif method==1
    strm="Adams Moulton";
    axis([-6 1 -5 5]);
  else
    strm="Adams Bashforth";
  end
  title(['Boundary Locus for LMF method ' strm]);
end
legend(gca,'show','Location','southeast');
```

Listing 25: Plot Boundary Locus Script

By setting 5 maximum steps and 6 as theta initial value, we obtain the results shown in figures 24,25 and 26.

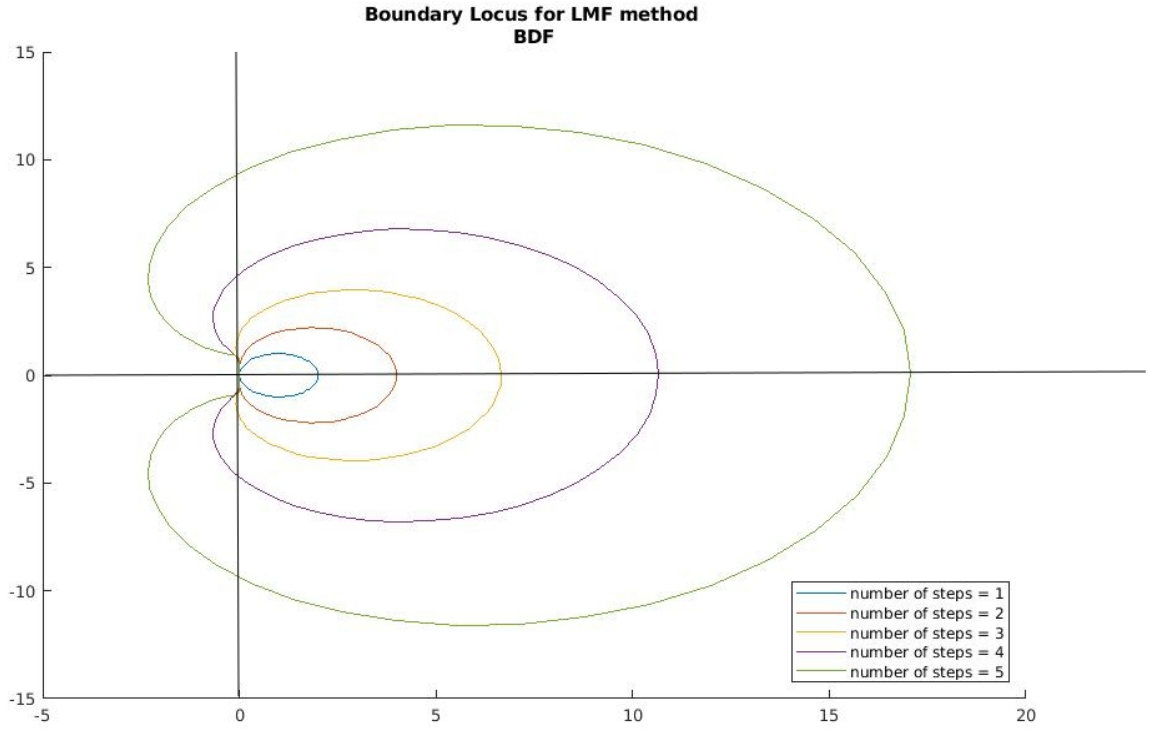

Figure 24: BDF Boundary Locus

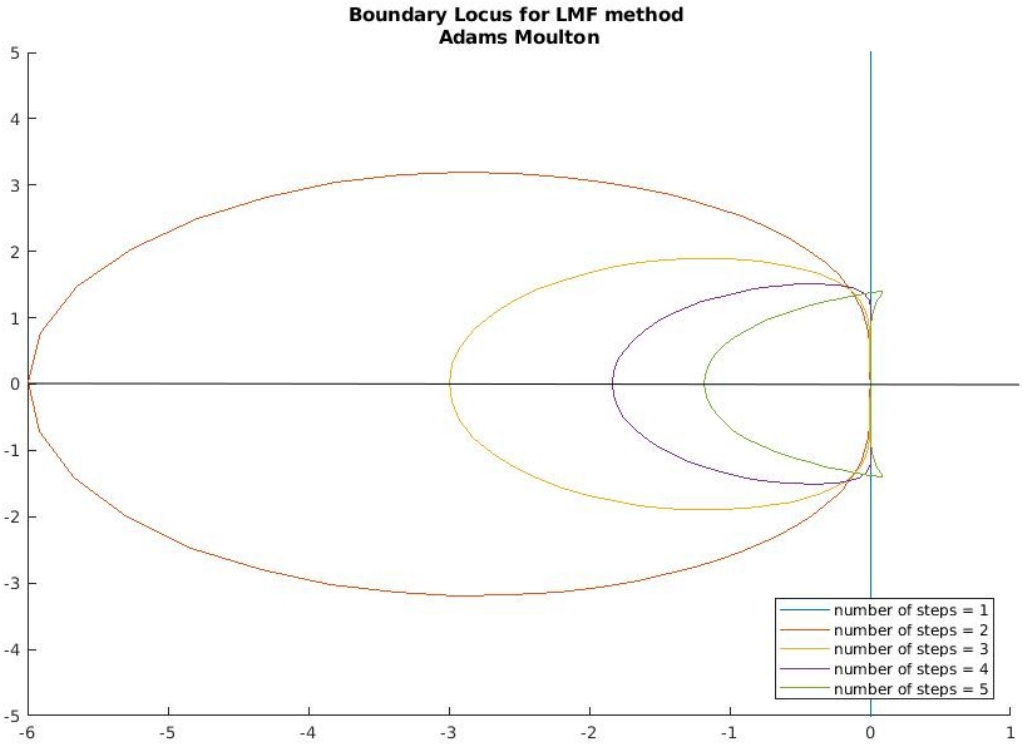

Figure 25: Adams Moulton Boundary Locus

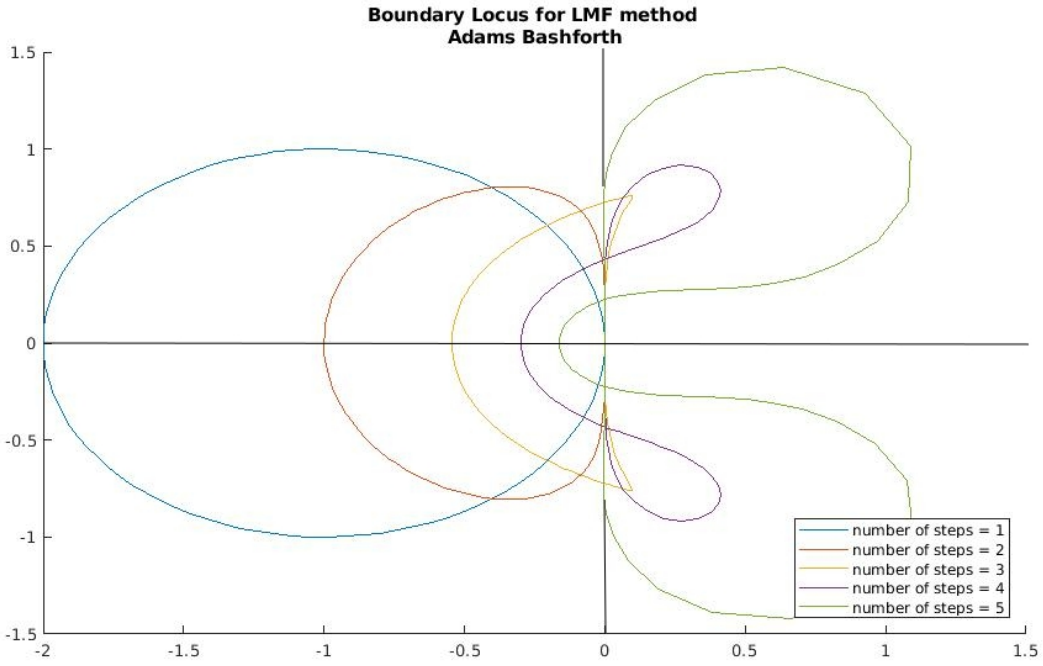

Figure 26: Adams Bashforth Boundary Locus

7 Leslie Model

This model describes the demographic structure of a homogeneous population, structured in age classes.
Let L be the maximum reacheable age in a population. We divide this temporal arc in intervals of size:

$$\tau = \frac{L}{m}$$

We denote by

$$x_i(k), \quad i = 1, \dots, m$$

The number of people aged between

$$[(i-1)\tau, i\tau], \quad i = 1, \dots, m-1$$

Where

$$0 < \beta_i \leq 1 \quad i = 1, \dots, m-1$$

Is the survival coefficent of class i at time τ.
We have:

$$\begin{cases} x_{i+1}(k+1) = \beta_i x_i(k) & i = 1, \dots, m-1 \\ x_1(k+1) = \sum_{i=1}^{m} \alpha_i x_i(k) & \text{are the new born} \end{cases}$$

Where $\alpha_i > 0$ are birth rate of respectives age classes.
We obtain the following discrete dynamic system:

$$\mathbf{x}(k+1) \equiv \begin{bmatrix} x_1(k+1) \\ x_2(k+1) \\ \cdot \\ \cdot \\ \cdot \\ x_m(k+2) \end{bmatrix} = \begin{bmatrix} \alpha_1 & \dots & \alpha_{m-1} & \alpha_m \\ \beta_1 & & & \\ & \ddots & & \\ & & \beta_{m-1} & \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \\ \cdot \\ \cdot \\ \cdot \\ x_m(k) \end{bmatrix}$$

Consider the following MATLAB function:

```
function [nextgen] = leslie(alphai,betai,actgen)
  m = length(alphai);
  if m ~= length(betai)+1
    error('beta i must be length as alpha i -1');
  end
  M = zeros(m,m);
  for i=1:m
    M(1,i)=alphai(i);
  end
  for i=2:m
    for j=1:m
      if(i-1 == j)
        M(i,j) = betai(j);
      end
    end
  end
  nextgen = M*actgen;
end
```

Listing 26: Leslie model function

This function calculates next generation foreach age class. We can call it iteratively for a fixed number of steps N to obtain a simulation model which describes the evolution of a population. For this purpose, we consider to use the following MATLAB function:

```
function [gens] = population(alphai,betai,startgen,steps)
  gens=zeros(length(startgen),steps);
  gens(:,1)=startgen;
  for i=2:steps
    gens(:,i)=leslie(alphai,betai,gens(:,i-1));
  end
end
```

Listing 27: Leslie simulation function for a fixed number of steps

Consider now to execute this script:

```
%age classes      [Children]   [Young]     [Adult]     [Olds]
%birth rates      0.81844      0.070636    0.024407     0.003298
%survival         0.071936     0.189763    0.115040     0
%start gen        100          250         300          150
birth_rates=[0.81844,0.070636,0.024407,0.003298];
survival = [0.071936,0.189763,0.115040];
start_gen= [100,250,300,150];
n_steps = 5;
gens = population(birth_rates,survival,start_gen,n_steps);
hold on;
population_amounts = zeros(1,n_steps);
for i=1:n_steps
  population_amounts(i)= (sum(gens(:,i)));
  disp(['generation ' num2str(i) ':']);
  disp(['[Children]          ------------> ' num2str(gens(1,i)) ]);
  disp(['[Youngs]            ------------> ' num2str(gens(2,i)) ]);
  disp(['[Adults]            ------------> ' num2str(gens(3,i)) ]);
  disp(['[Olds]              ------------> ' num2str(gens(4,i)) ]);
  disp(['[Total population] ------------> ' num2str(population_amounts(i)) ]);
end
plot(population_amounts,'DisplayName','Amount of people');
legend(gca,'show','location','northeast');
```

Listing 28: Leslie script

We obtain the plot shown in figure 27. The console output produced by this script leads to the construction of the following dataset:

Classes	**Generation 1**	**Generation 2**	**Generation 3**	**Generation 4**	**Generation 5**
Children	100	107	90	74	61
Young people	250	7	8	6	5
Adults	300	47	1	2	1
Old men	150	34	5	0	0
Total	800	195	104	82	67

If we increase the number of generation as it reaches infinity, we obtain the plot shown in figure 28.

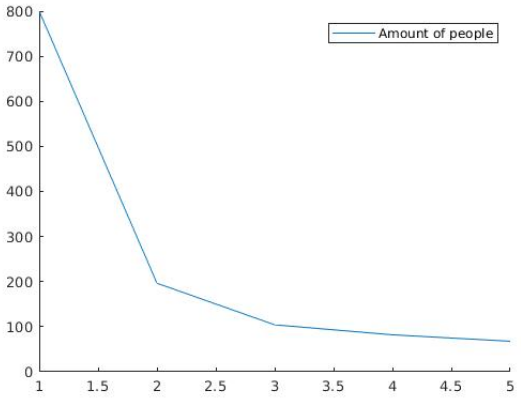

Figure 27: With these parameters, population approaches to extinction

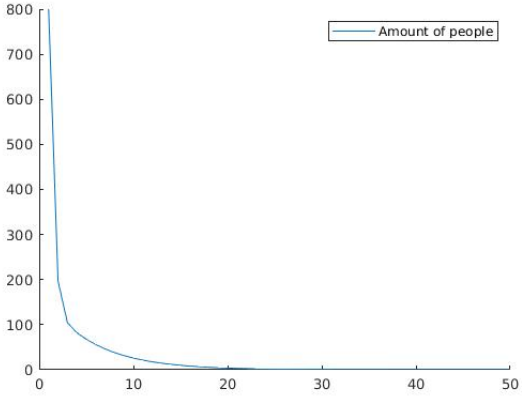

Figure 28: Extinction of the population

8 Arms Race model

We formulate a continuous time model which describes the level of two nation's arms. Let $x(t), y(t)$ be the nation's arms levels, we have:

$$x'(t) = -ax(t) + by(t) + \xi_0$$

$$y'(t) = cx(t) - dy(t) + \eta_0$$

Where all coefficients are positive:

- a, d are called fatigue coefficients
- b, c are called competition coefficients
- ξ_0, η_0 are called "base" levels, and they depend solely on the cultural connotations of the two nations in competition

In vectorial terms, we obtain the following positive dynamic system:

$$\begin{bmatrix} x \\ y \end{bmatrix}' = \begin{bmatrix} -a & b \\ c & -d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \xi_0 \\ \eta_0 \end{bmatrix}$$

By applying Cramer rule, the equilibrium point is given by:

$$\overline{x} = \frac{d\xi_0 + b\eta_0}{ad - bc}$$

$$\overline{y} = \frac{c\xi_0 + a\eta_0}{ad - bc}$$

This point is asimptotically stable if and only if it has positive components, so if and only if

$$ad > bc$$

In other words, if the product of fatigue coefficients are greater than competition ones.
Consider the following MATLAB function:

```
function [xsol,ysol,tsol] = arms(a,b,c,d,xi,eta,x0,y0,t0,tf,h)
  xt = @(t,x,yt) -a*x+ + b*yt + xi;
  yt = @(t,y,xt) c*xt - d*y + eta;
  [xsol,ysol,tsol] = expliciteuler(xt,yt,h,x0,y0,t0,tf);
end

function [xn,yn,tn] = expliciteuler(f,g,h,x0,y0,t0,tf)
  N = 1+((tf-t0)/h);
  tn = linspace(t0,tf,N);
  xn = zeros(1,N);
  yn = zeros(1,N);
  xn(1) = x0;
  yn(1) = y0;
  for i=2:N
    xn(i) = xn(i-1) + h*f(tn(i-1),xn(i-1),yn(i-1));
    yn(i) = yn(i-1) + h*g(tn(i-1),yn(i-1),xn(i-1));
  end
end
```

Listing 29: Arms race model

We notice that the solution is approximated using an explicit euler method, slightly modified in a way such that functions x, y (the known derivatives) can support to being evaluated considering each other, indeed they depend on each other since we are solving an ODEs system. Another parameter is added to the anonymous functions x, y:

$$x = \lambda t, x, y_t.\ -ax + by_t + \xi$$

$$y = \lambda t, y, x_t.\ cx_t - dy + \eta$$

Parameters y_t,x_t represent the approximation of each other solution at the time step t, and ξ, η are free variables given from the context Γ. Since we are evaluating the solution of the system using an explicit method, we calculate the next solution using explicitly the last we obtained, thus at each iteration we know x_t and y_t. Consider the following script:

```
a=0.9; b=2; c=1; d=2;
xi=1.6; eta = 1.4;
x0 = 5; y0 = 2.5;
```

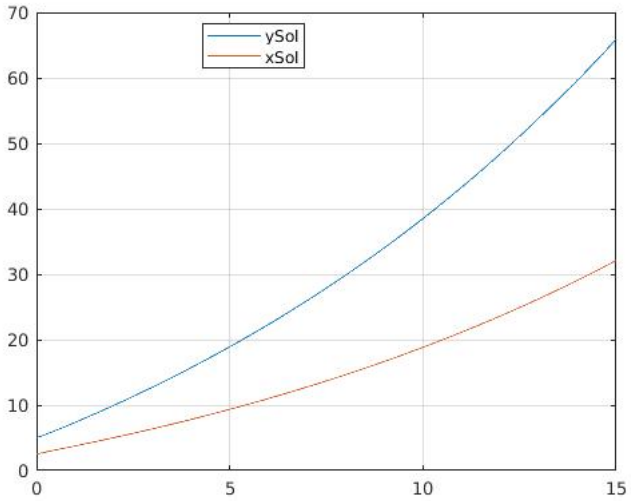

Figure 29: Arms race model with a=0.9,b=2,c=1,d=2 for which is not verified $ad > bc$

```
[ySol,xSol,tsol]=arms(a,b,c,d,xi,eta,x0,y0,0,15,0.01);
plot(tsol,ySol);
hold on;
plot(tsol,xSol);
grid on;
legend('ySol','xSol','Location','best')
```

Listing 30: Arms race model plot script

Executing it we obtain the results shown in figures 29 and 30.

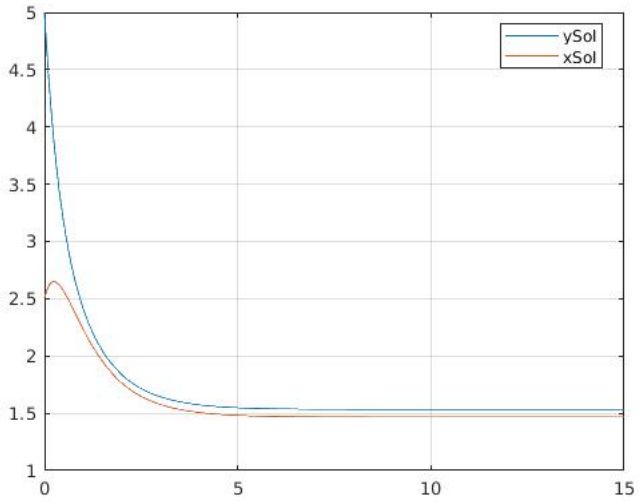

Figure 30: Arms race model with a=2,b=1,c=1,d=2 for which is verified $ad > bc$

9 Diabetes mellitus model

Diabates mellitus is a pathology that manifests through a high concentration of glucose in the blood and urine.
When the regulatory mechanisms in a healthy person works properly, the insulin produced by the pancreas counteracts the level of glucose in the blood, but in a non-healthy person who has diabetes, these mechanisms does not work properly and the glucose remains too high; and this is a threat for the individual health.
Let $G(t), H(t)$ be respectively the concentrations of glucose and insuline, and call $\overline{G}, \overline{H}$ the optimal concentrations.
In general we can hypotize a relation between the variation of these two concentrations and their actual concentration, in fact:

$$G'(t) = F_g(G(t), H(t))$$
$$H'(t) = F_h(G(t), H(t))$$

When concentrations are at their optimal level, the regulation mechanism has no need to intervene, so the system is in equilibrium. We have:

$$F_g(\overline{G}, \overline{H}) = 0$$

$$F_h(\overline{G}, \overline{H}) = 0$$

For sake of simplicity, we trranslate $G(t), H(t)$ in order to have the optimal levels equals to 0. So we'll consider:

$$g(t) = G(t) - \overline{G}$$
$$h(t) = H(t) - \overline{H}$$

Adding a constant does not change derivatives. Supposing that F_h and F_g are Taylor-developable at second order, we obtain the following system:

$$g'(t) = -m_1 g - m_2 h + \gamma_g(g,h) \quad (*)$$

$$h'(t) = m_3 g - m_4 h + \gamma_h(g,h)$$

Where γ_g, γ_h are functions which represents higher order terms of the development.
We assume all $m_i > 0$.
We define:

$$M = \begin{bmatrix} -m_1 & -m_2 \\ m_3 & m_4 \end{bmatrix}$$

The eigenvalues of matrix M have negative real parts, so due to the Perron theorem, all the system admits origin as a solution asimptotically stable.
We can solve $(*)$ to obtain an explicit form of $G(t)$ function, which describes the blood's glucose concentration with respect to the time.

$$g(t) = Ae^{-\alpha t}\cos(\omega t + \varphi)$$

$$G(t) = \overline{G} + Ae^{-\alpha t}\cos(\omega t + \varphi)$$

For diagnostic purposes, the quantity ω is the most useful parameter. In fact, if

$$T = \frac{2\pi}{\omega} > 3.5h$$

the patient is in a pathological state.
If the first measure of the glycemic level is performed after an enough long period of fasting, we can assume that this has reached the optimal equilibrium state.
Then if we give to the patient a quantity of glucose proportional to the body weight and making m measures at prefixed instants t_i, $i = 1, \ldots, m$, we can obtain the remaining parameters with the least squares method.
In other words, we look for the minimum of the following function:

$$F(\alpha, \omega, A, \varphi) = \sum_{i=1}^{m} (G_i - \overline{G} - Ae^{-\alpha t}\cos(\omega t + \varphi))^2 \quad (**)$$

If we have approximated parameters, we can use the following MATLAB function:

```
function [G] = diab(opt,A,m1,m2,m3,m4,phi,maxt,measures)
  abscissa = linspace(0,maxt,measures);
  N=length(abscissa);
  beta = sqrt(m1*m4+m2*m3);
  alpha = (m1+m4)/2;
  omega = sqrt(beta^2-alpha^2);
  G=zeros(1,N);
  for i=1:N
    G(i)=opt+A*exp(-alpha*abscissa(i))*cos(omega*abscissa(i)+phi);
  end
end
```

Listing 31: Diabetes Mellitus Function

Where opt is the optimal glycemic blood value, A,m1,m2,m3,m4 and phi are the model parameters, which must be approximated by solving (**), maxt is the maximum time measured, and measures is the number of measurements done in the interval $[0, maxt]$.
Executing the following script, we obtain the result shown in figure 31.

```
optimalg = diab(80,30,0.1,1.2,3.3,1.1,-20,10,1000);
badg = diab(140,30,0.1,2,0.125,0.1,-40,10,1000);
plot(optimalg,'DisplayName','Normal glycemic level');
hold on;
grid on;
plot(badg,'DisplayName','Pathologic glycemic level');
legend(gca,'show','Location','best');
```

Listing 32: Diabetes Mellitus Script

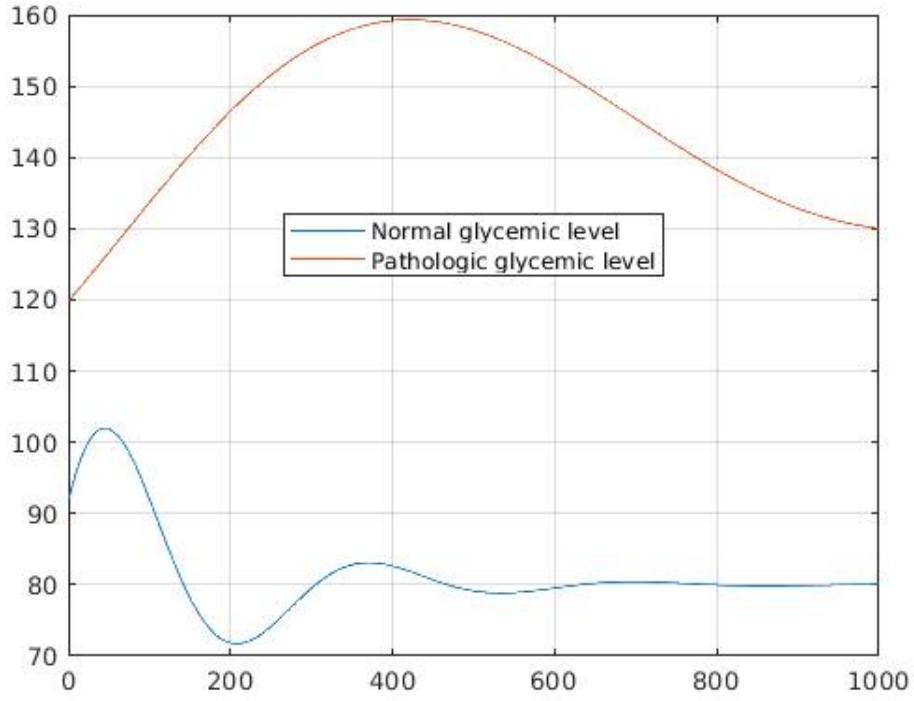

Figure 31: Diabetes mellitus plot for a healty and a non healty patient