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

August 7, 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.

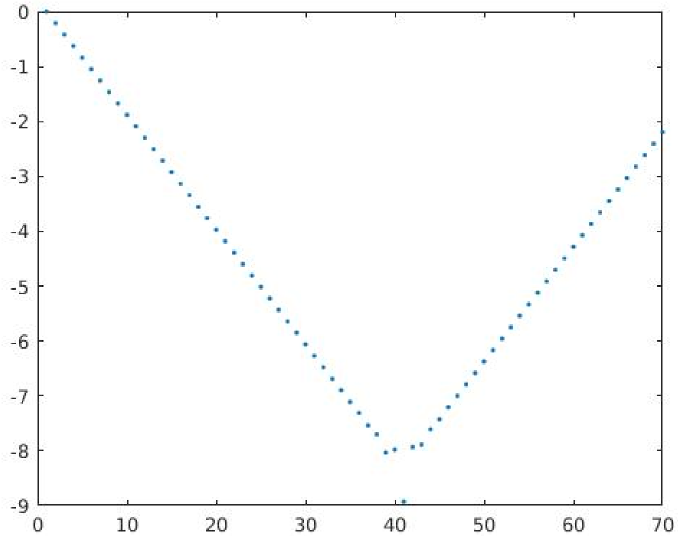


Figure 2: Effects of rounding in 64 bit precision

We can verify the divergence in 32 bit precision, modyfing the *fatalbit* function in such manner:

```
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:

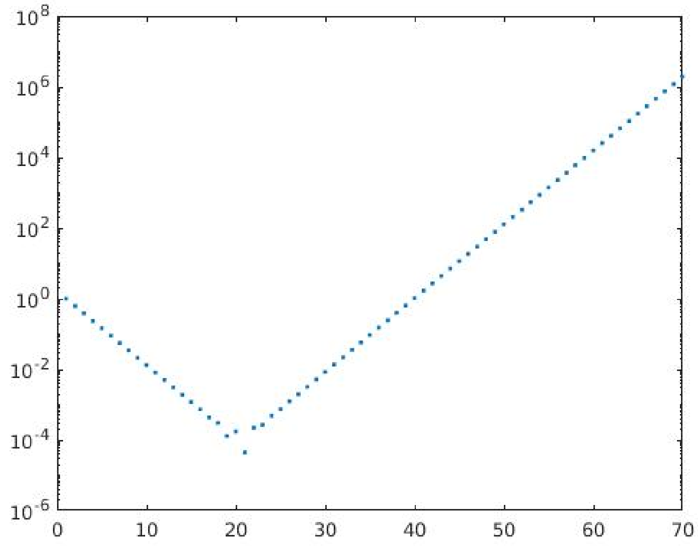


Figure 3: Effects of rounding in 32 bit precision

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:

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

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:

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

Now we must 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

Hence we can execute the following script:

```
C = 10000;
interest=0.05;
N=100;
hold all;
for i=1:5
ratei=loan(C,460+(i-1)*20,interest,N);
plot(ratei,'DisplayName',['installment amount = ' num2str(460+(i-1)*20)]);
end
legend(gca,'show','Location','northwest');
```

Listing 7: Loan repayment plot script with different installment amount

Obtaining the plot in figure 4.
We note that if interest is the 5% and installment amount is too low, the debt growth exponentially.

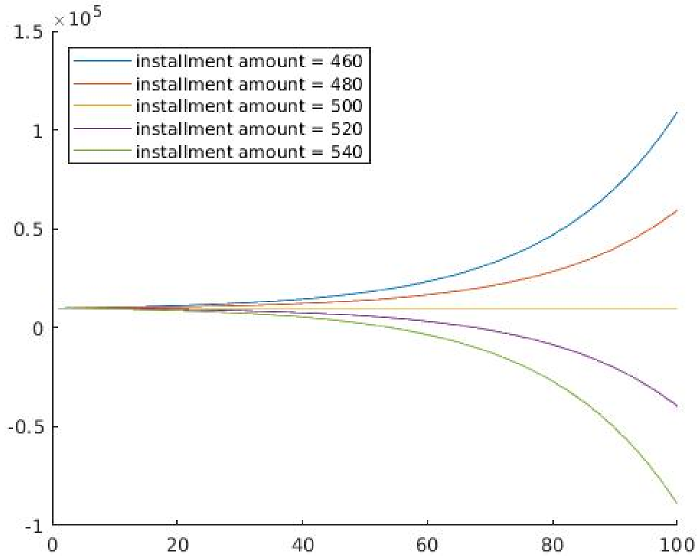


Figure 4: Residual debt with different installments amounts

The debt remains constant if the installment covers only interests (in this case, 500), and it reaches zero if and only if the installment amount is greater to 500.
This behaviour is compliant to the (2), in fact, if $\frac{r}{i} < C$ the succession diverges.
Suppose we want to contract a morgage for a capital of 10000, with an interest of 5%, single installment import 1406.9, repayable in 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

[resDebt,intQuote,capQuote] = loantable(10000,0.05,1406.9,10);
N = 11;
table = zeros(N,3);
table(1,1)=10000;
for j=2:N
  table(j,1)=resDebt(j-1);
end
for i=1:N
  table(i,2)=intQuote(i);
  table(i,3)=capQuote(i);
end
```

Listing 8: Mortgage repayment plan table

Executing this script, we obtain:

```
table =

   1.0e+04 *

    1.0000         0         0
    1.0000    0.0500    0.0907
    0.9093    0.0455    0.0952
    0.8141    0.0407    0.1000
    0.7141    0.0357    0.1050
    0.6091    0.0305    0.1102
    0.4989    0.0249    0.1157
    0.3831    0.0192    0.1215
    0.2616    0.0131    0.1276
    0.1340    0.0067    0.1340
    0.0000    0.0000    0.1407
```

Figure 5: Loan table obtained executing listing 7

Which represents the following dataset:

Installment	Residual Debt	Interest Quote	Capital Quote
0	10000	-	-
1	10000	500	907
2	9093	455	952
3	8141	407	1000
4	7141	357	1050
5	6091	305	1102
6	4989	249	1157
7	3831	192	1215
8	2616	131	1276
9	1340	67	1340
10	0	0	1407

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:

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

For sake of simplicity we assume g as follows:

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

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:

$$S_n = D_n \forall n$$

$$\Downarrow$$

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

$$\Downarrow$$

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

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

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

(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 9: 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 behaviour 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 10: 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 11: 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 12: 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.

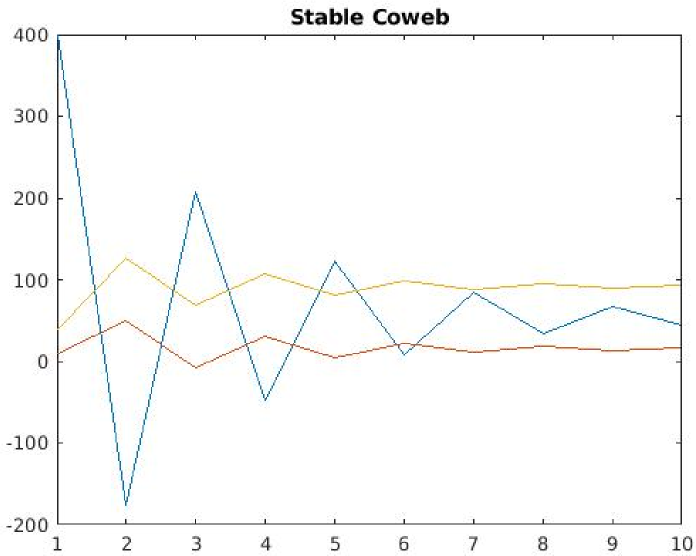


Figure 6: Plot of Stable Coweb successions

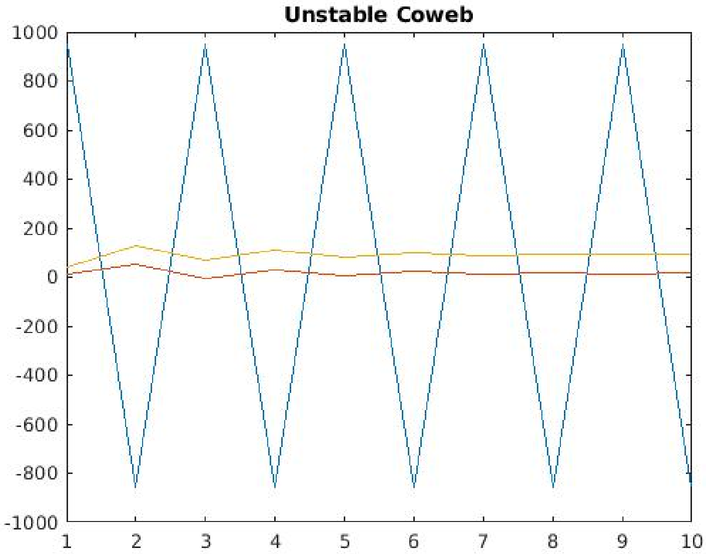


Figure 7: Plot of Unstable Coweb successions

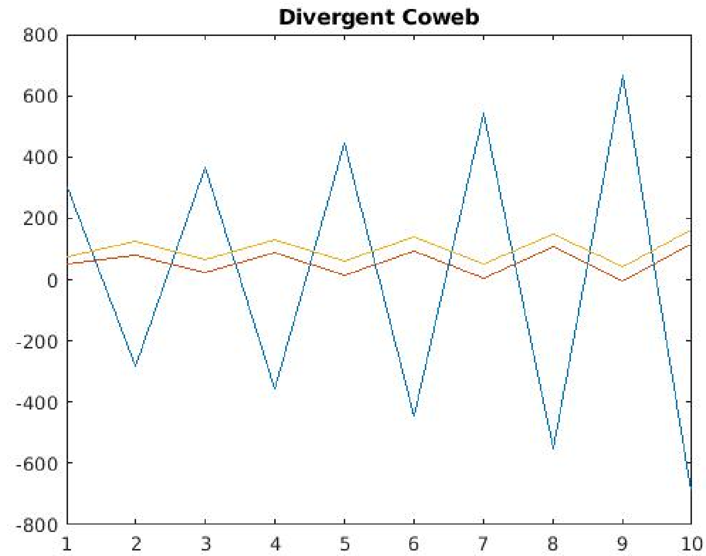


Figure 8: Plot of Divergent Coweb successions

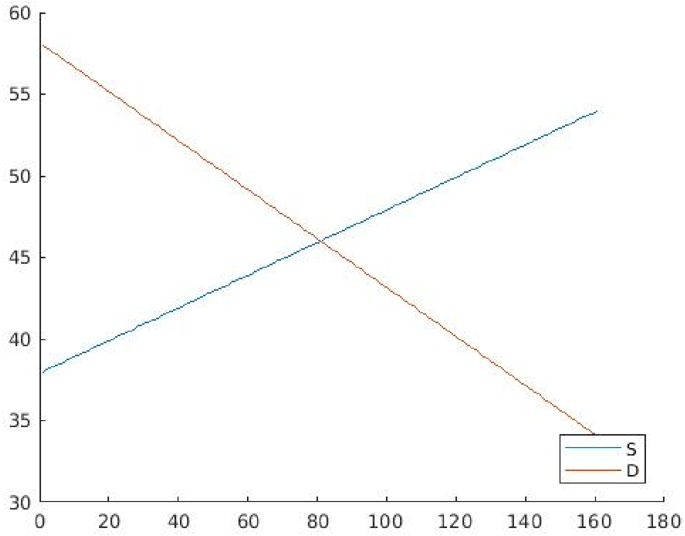


Figure 9: Supply and Demand functions plot

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 13: 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 14: GDP first case plot

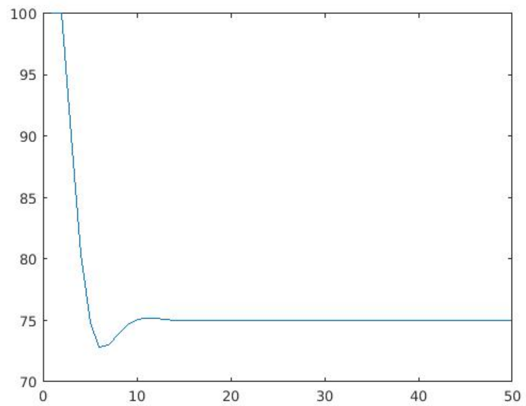


Figure 10: GDP values in case 1

2. $\alpha > 1$:

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

Listing 15: GDP second case plot

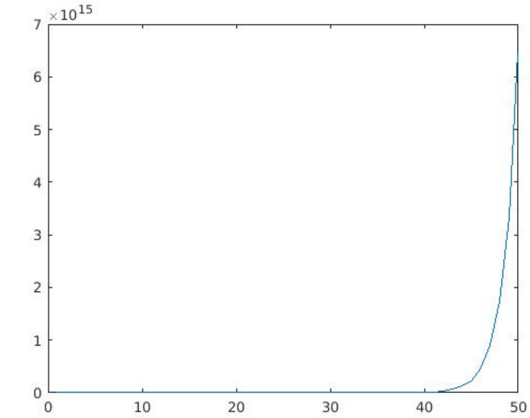


Figure 11: GDP values in case 2

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

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

Listing 16: GDP third case plot

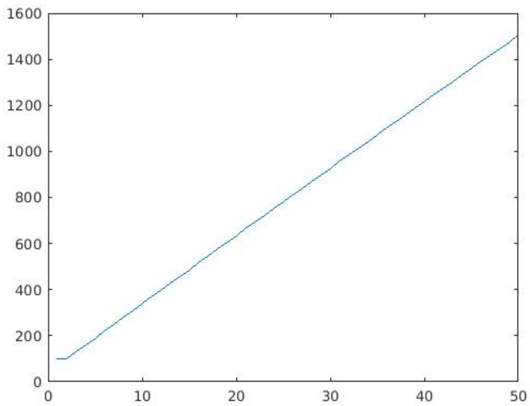


Figure 12: GDP values in case 3

6 Linear Multistep Methods

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:

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

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:

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

Consider the following $k = 1$-step methods:

$$y_{i+1} - y_i = hf_i \quad (*)$$

$$y_{i+1} - y_i = hf_{i+1} \quad (**)$$

$(*)$ is called the *explicit euler method* or *forward euler* and it approximates the unknown function y in the following way:

$$y_{i+1} = y_i + hf_i = y_i + hf(t_i, y_i)$$

This method is easy to implement:

```
function [approx_values] = euler_expl(abscisse,x0,step_size,n_steps,fun)
  approx_values = zeros(1,n_steps);
  approx_values(1) = x0;
  for i = 0:n_steps-2
    h = step_size;
    ti = abscisse(i+1);
    yi = approx_values(i+1);
    fi = fun(ti,yi);
    approx_values(i+2) = yi + h * fi;
  end
end
```

Listing 17: Explicit euler MATLAB implementation

More complex is to handle method $(**)$.
This method is called *implicit euler method* or *backward euler*. It is more stable than the *explicit* or *forward* one, but it requires a little more attention.
In fact, the method states:

$$y_{i+1} - y_i = hf_{i+1}$$

Hence:

$$y_{i+1} = y_i + hf(t_{i+1}, y_{i+1})$$

Term y_{i+1} appears both in left and right members of the equation.
A possible solution could be use a *fixed point iteration* to approximate y_{i+1}, or perform a *forward step* using *forward euler* method in order to get an initial estimation of y_{i+1}.

```
function [approx_values] = euler_impl(abscisse,x0,step_size,n_steps,fun)
  approx_values = zeros(1,n_steps);
  approx_values(1) = x0;
  for i = 0:n_steps-2
    h = step_size;
    ti = abscisse(i+2);
    yi = approx_values(i+1);
    init_e = one_step_fwd(ti,yi,step_size,fun);
    fi = fun(ti,init_e);
    approx_values(i+2) = yi + h * fi;
  end
end

function [y1] = one_step_fwd(t, y0, h, f)
  y1 = y0 + h*f(t,y0);
end
```

Listing 18: Implicit euler MATLAB implementation

Let us assume we have the following problem:

$$\begin{cases} \dot{y} = f(t,y) = cos(t) + y \\ y(0) = 1 \end{cases}$$

By solving it analitycally, we get:

$$\begin{cases} y(t) = c_1 e^t + \frac{sin(t)}{2} - \frac{cos(t)}{2} \\ y(0) = 1 \end{cases}$$

$$\Downarrow$$

$$c_1 e^0 + \frac{sin(0)}{2} - \frac{cos(0)}{2} = 1$$

$$c_1 - \frac{1}{2} = 1$$

$$c_1 = \frac{3}{2}$$

$$\Downarrow$$

$$y(t) = \frac{3}{2}e^t + \frac{sin(t)}{2} - \frac{cos(t)}{2}$$

We can execute the following script in order to see how much accurated are the implemented methods shown above.

```
FUN = @(t) (3/2)*(exp(t))+sin(t)/2-cos(t)/2;
fun = @(t,y) (cos(t)+y);
a = input('Insert left extreme of the interval\n');
b = input('Insert right extreme of the interval\n');
n = input('Insert step numbers\n');
if a>b
  temp = a;
  a = b;
  b = temp;
end
x0 = FUN(a);
real_abs = linspace(a,b,10000);
abscisse = linspace(a,b,n);
h = (b-a)/n;
expl = euler_expl(abscisse,x0,h,n,fun);
impl = euler_impl(abscisse,x0,h,n,fun);
exact = FUN(real_abs);
plot(abscisse,expl,'DisplayName','Explicit Euler');
hold all;
strm = num2str(n);
title(['n steps = ' strm]);
plot(abscisse,impl,'DisplayName','Implicit Euler');
plot(real_abs,exact,'DisplayName','Exact Function');
legend(gca,'show','Location','northwest');
```

Listing 19: Implicit and explicit euler plot

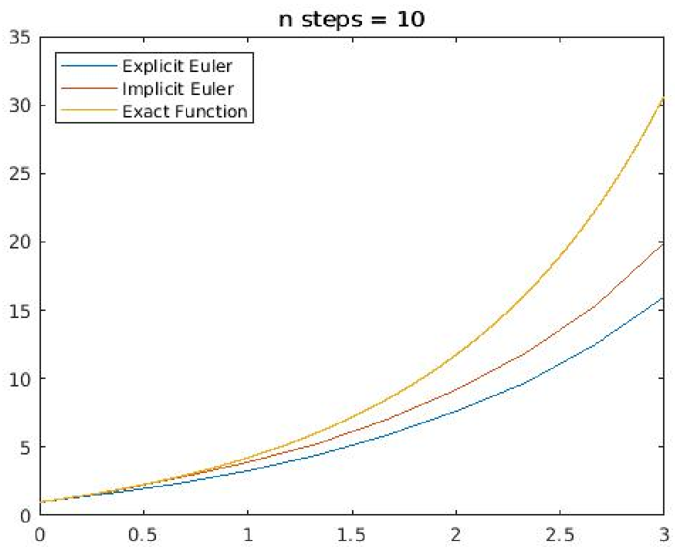


Figure 13: Approximation with mesh integration step $h = \frac{3}{10}$

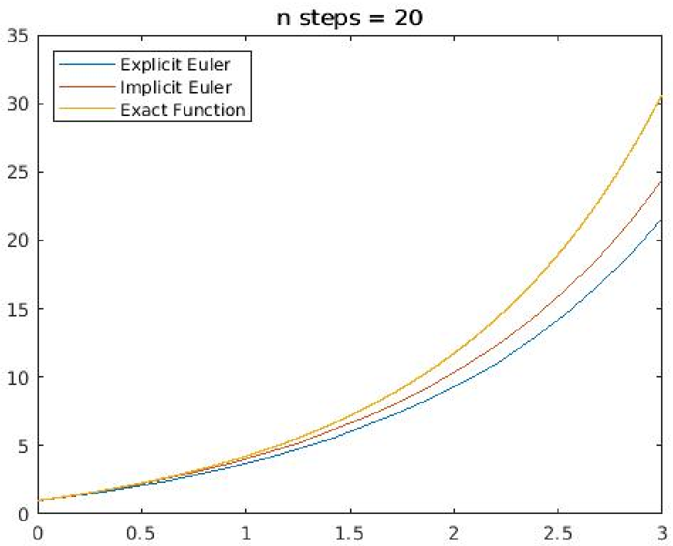


Figure 14: Approximation with mesh integration step $h = \frac{3}{20}$

For a more clear example of the explicit's possible errors, let us assume we are in case:

$$\begin{cases} \dot{y} = f(t,y) = cos(t) - \frac{1}{2\sqrt{t}} \\ y(0.001) \approx 0.9694 \end{cases}$$

$$y(t) = sin(t) + (1 - \sqrt{t})$$

Hence we modify the first 2 lines of the previous seen script, in the way it follows:

```
fun = @(t,y) cos(t)-1/(2*sqrt(t))+0*y;
FUN = @(t) sin(t)+(1-sqrt(t));
```

Listing 20: Implict and explicit euler plot in another case

Executing the script with following values:

$$a = 1$$

$$b = 25$$

$$n = 50\ steps$$

We obtain the plot shown in Figure 15.
It is immediate to notice that $f(t,y)$ has a vertical asymptote in $t = 0$. If we execute the script and set a left extreme value near to 0^+, for example $a = 0.001$, we obtain the plot shown in Figure 16.

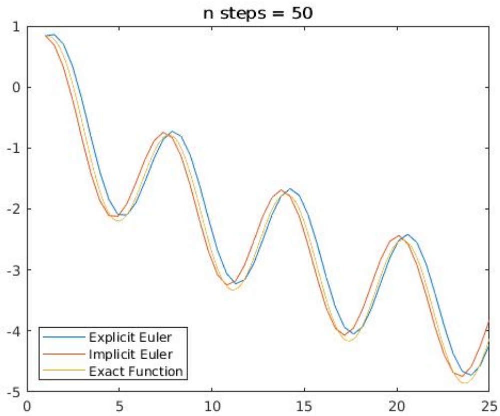


Figure 15: Implicit and explicit euler's methods correct behaviour

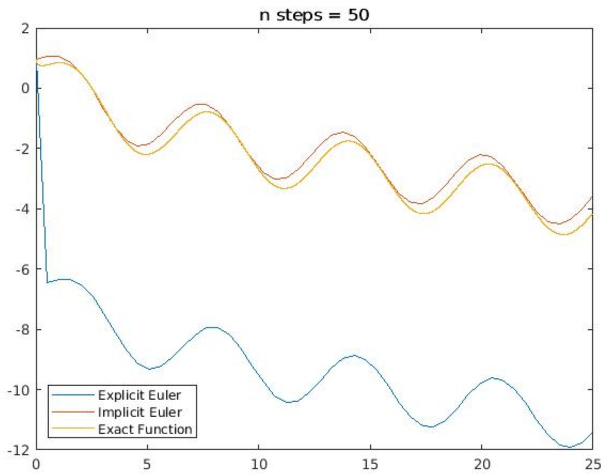


Figure 16: Explicit euler's divergence as a reaches 0^+

6.1 A more general implementation

The subject of the discussion up to this point was the implementation of a particular case of linear multistep method. In fact, Euler's method (whether implicit or explicit) is a constant k = 1 step method, characterized by the following characteristics:

	α_0	α_1	β_0	β_1	$\rho(z)$	$\sigma(z)$
Implicit	-1	1	0	1	$z-1$	z
Explicit	-1	1	1	0	$z-1$	1

More complex but also more useful is the implementation of a generic method that, received as parameters the α_i, β_i, k initial conditions, the integration interval and the known function $f(t, y)$ of which we want to approximate the primitive, returns an approximation of the function $y(t)$ on the desired mesh. Consider the following MATLAB function:

```
function [yn] = multistep(ai,bi,init_conds,intval,f)
  lbi = length(bi);
  if bi(lbi) ~= 0
    disp('last beta_i must be equal to 0');
  return;
  end
  N = length(intval);
  k = length(init_conds);
  lai = length(ai);
  if lai ~= lbi
    disp('alpha_i and beta_i cannot have different sizes');
  return;
  end
  if lai ~= k+1
    disp('alpha_i and beta_i must have dimension k+1');
  return;
  end
  if N < k
    disp('mesh size is too small for the chosen number of steps');
  return;
  end
  intval = sort(intval);
  h = intval(2)-intval(1);
  yn = zeros(1,N);
  yn(1:k)=init_conds(1:k);
  n = k;
  while n<N
    known_yn  = yn(n:-1:n-k+1);
    comb_lin1 = comblin(known_yn,ai(2:length(ai)));
    known_yn1 = yn(n+1-k:n+1);
    tn = intval(n+1-k:n+1);
    fn = zeros(1,length(tn));
    for i=1:length(fn)
      fn(i)=f(tn(i),known_yn1(i));
    end
    comb_lin2 = comblin(fn,bi);
    yn(n+1)  = (comb_lin1 + h * comb_lin2)/-ai(1);
    n = n+1;
  end
end
```

```
function [cl] = comblin(v,alpha)
  n = length(v);
  if n ~= length(alpha)
    disp('vector and coefficients must have same sizes');
  return;
  end
  cl = 0;
  for i = 1:n
    cl = cl+v(i)*alpha(i);
  end
end
```

Listing 21: Generic explicit LMF solver

This function is able to perform this task for an explicit LMF method. Its behavior derives directly from the definition of LMF method seen in the previous paragraph.
Consider the following problem:

$$\begin{cases} \dot{y} = y \\ y(0) = 1 \end{cases}$$

The exact solution of this problem is evidently:

$$y(t) = e^t$$

If we execute the following script, we can see a comparison between the solution of such problem performed by LMF methods:

- Euler explicit
- Adams Bashforth
- Midpoint

```
fun = @(t,y) y;
FUN = @(t) exp(t);
intval = linspace(0,5,200);
real_values =zeros(1,length(intval));
for i=1:length(intval)
  real_values(i)=FUN(intval(i));
end
y0 = 1;
midpoint_ai = [-1, 0, 1];
midpoint_bi = [0, 2,  0];
adams_bf_ai = [-1, 1, 0];
adams_bf_bi = [-1/2,3/2,0];
euler_ai = [-1,1];
euler_bi = [1,0];
euler=multistep(euler_ai,euler_bi,y0,intval,fun);
midpoint=multistep(midpoint_ai,midpoint_bi,[y0,y0],intval,fun);
adams_bf=multistep(adams_bf_ai,adams_bf_bi,[y0,y0],intval,fun);
hold on;
plot(intval,midpoint,'DisplayName','Midpoint');
plot(intval,euler,'DisplayName','Euler');
plot(intval,adams_bf,'DisplayName','Adams - BF');
plot(intval,real_values,'DisplayName','Exact solution');
legend(gca,'show','Location','northwest');
```

Listing 22: Comparison between explicit LMF methods

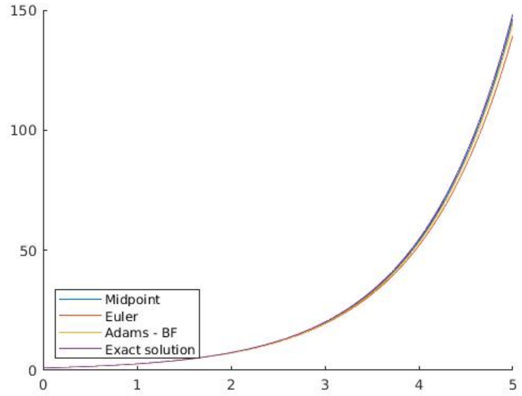


Figure 17: LMF methods behaviour in interval [0,5] with step h = 0.025

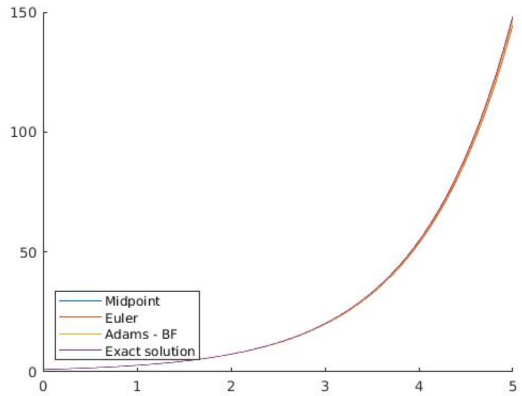


Figure 18: LMF methods behaviour in interval [0,5] with step h = 0.001

Consider now the following problem:

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

The exact solution will be

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

Notice that:

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

We could modify the function to approximate in the previous seen script, in order to test LMF methods in such case.
In the following figures 20 and 21, it is possible to see spurious oscillations performed by the ap-

proximated values with the midpoint method in the interval $[0, 2.5]$ with step size $h = 0.01$ and $h = 0.001$.

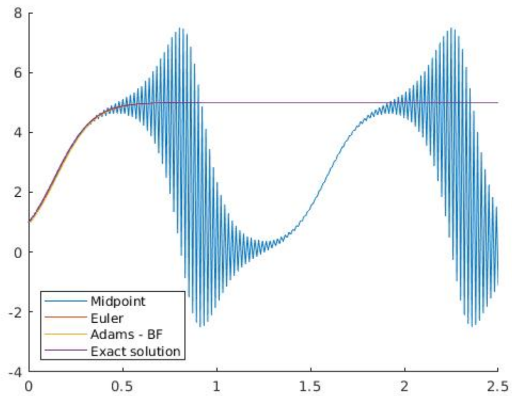


Figure 19: Midpoint formula problem with step h = 0.01

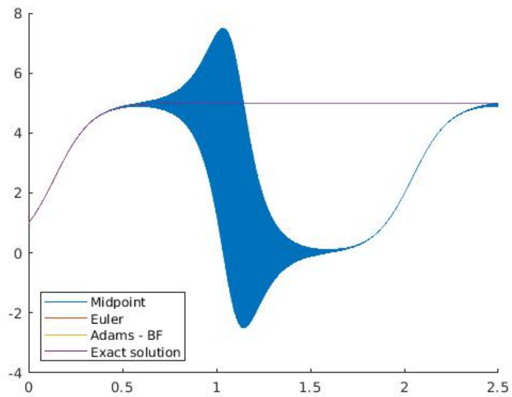


Figure 20: Midpoint formula problem with step h = 0.001

Last we examine the errors committed by these three LMF methods by executing the following script:

```
comparelmf;
close;
ea = abs(adams_bf-real_values);
em = abs(midpoint-real_values);
ee = abs(euler-real_values);
hold on;
left_in=1;
right_in=50; %change for others
plot(intval(left_in:right_in),em(left_in:right_in),'DisplayName','Midpoint Error');
plot(intval(left_in:right_in),ea(left_in:right_in),'DisplayName','Adams-BF Error');
plot(intval(left_in:right_in),ee(left_in:right_in),'DisplayName','Euler error');
legend(gca,'show','Location','northwest');
```

Listing 23: Error of LMF methods

Here it follows the resulting plots:

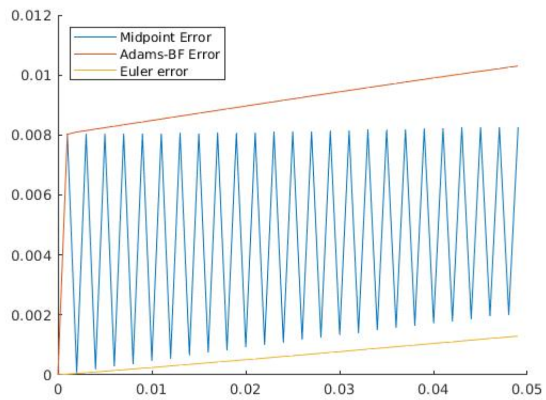


Figure 21: Error of LMF methods from 0 to 0.005

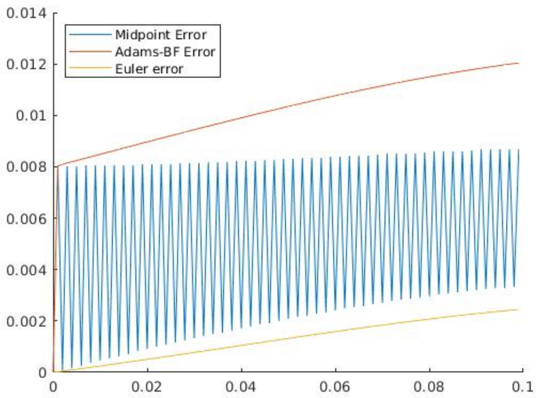


Figure 22: Error of LMF methods from 0 to 0.1

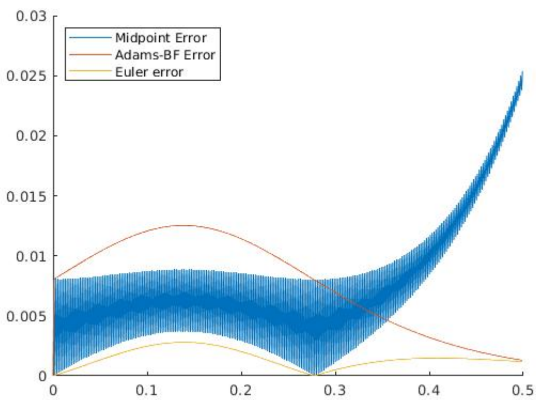


Figure 23: Error of LMF methods from 0 to 0.5

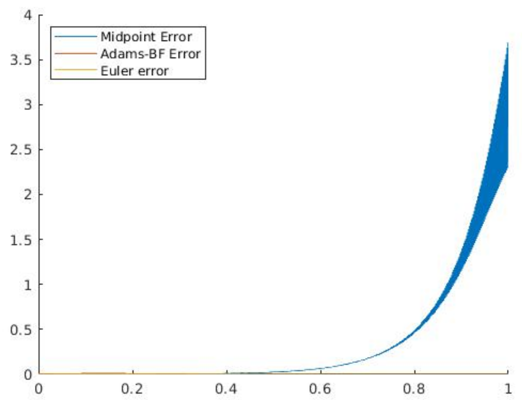


Figure 24: Error of LMF methods from 0 to 1

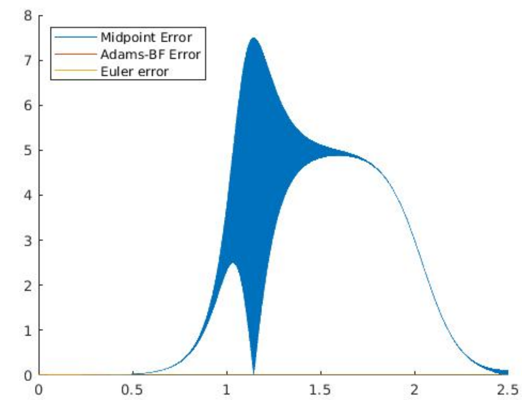
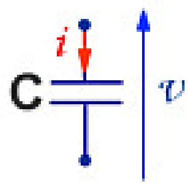


Figure 25: Error of LMF methods from 0 to 2.5

6.2 A pratical application of a LMF method

Consider a capacitor in an electric circuit.



Without any more assumption, we only know:

- Capacitor's capacity (C)
- Electric current i flowing in the capacitor:

$$i(t) = f(t, v(t)) = ln|sin(t)|\ cos(v)$$

Suppose also that we have made a direct measurement at time $t_0 = 10\ s$, and therefore we know that the voltage $v(t_0)$ is:

$$v(t_0) = v(10\ s) = 2\ V$$

Electric current and voltage are characterized by the following relationship:

$$i(t) = C\frac{d(v(t))}{dt} = C\dot{v}$$

Summarizing, we have:

$$\begin{cases} \dot{v} = \frac{i(t)}{C} = \frac{ln|sin(t)|\ cos(v)}{C} \\ v(10) = 2\ V \end{cases}$$

For sake of simplicity, we fix $C = 1F$. Obtaining:

$$\begin{cases} \dot{v} = ln|sin(t)|\ cos(v) \\ v(10) = 2\ V \end{cases}$$

If we want to know an approximation of the voltage value in the interval $[10, 50]s$ with an integration step $h = 0.1$ we can use the implicit euler method implemented before:

```
dotv = @(t,v) log(abs(sin(t)))*cos(v); % Electric current function
ti = linspace(10,50,400); % Interval of interest
x0 = 2; % v(10) = 2 V
yi = euler_impl(ti,x0,0.1,400,dotv);
plot(ti,yi);
```

Listing 24: Approximation of capacitor's voltage

Obtaining the result shown in Figure 17.
Without solving any (complex) differential equation, we see that for $t \approx 20\ s$ the system stabilizes on a capacitor's head voltage $v_c \approx 4.7\ V$.

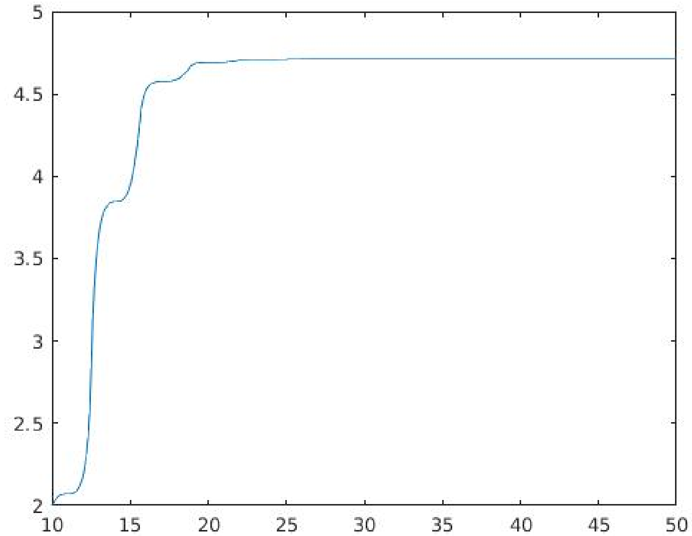
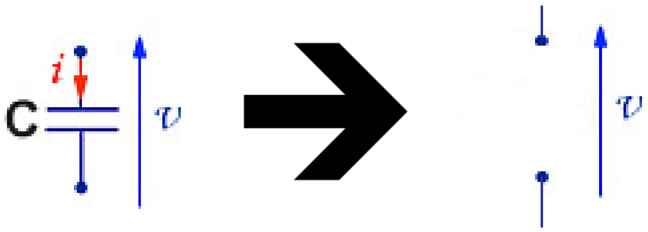


Figure 26: Plot of approximated capacitor's head voltage

This is a very interesting result because it indicates that the capacitor behaves like an "open switch" when it is fully operational. In fact, if the voltage at the heads of a capacitor is constant, then the current is null.



This behavior, although we explicitly know the current's equation, was however difficult to forecast. Even if have such a complex expression for the current might seem to be an implausible scenario, in reality, in a design phase of a certain electrical circuit, there could be an ideal current generator on whose behavior we place some constraints, and we want to size other components accordingly. In this way the circuit, once realized, behaves as if it were equipped with such an ideal current generator at a certain level of abstraction on the system.

6.3 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 25: 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 26: 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 27: Plot Boundary Locus Script

By setting 5 maximum steps and 6 as theta initial value, we obtain the results shown in figures 27,28 and 29.

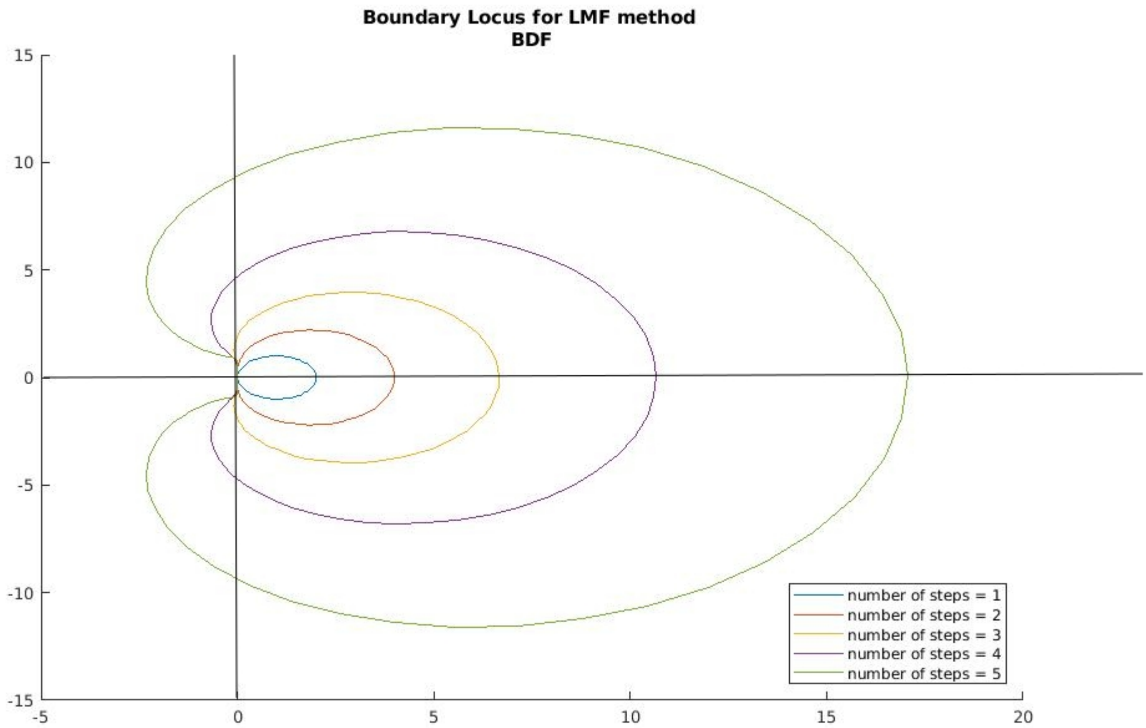


Figure 27: BDF Boundary Locus

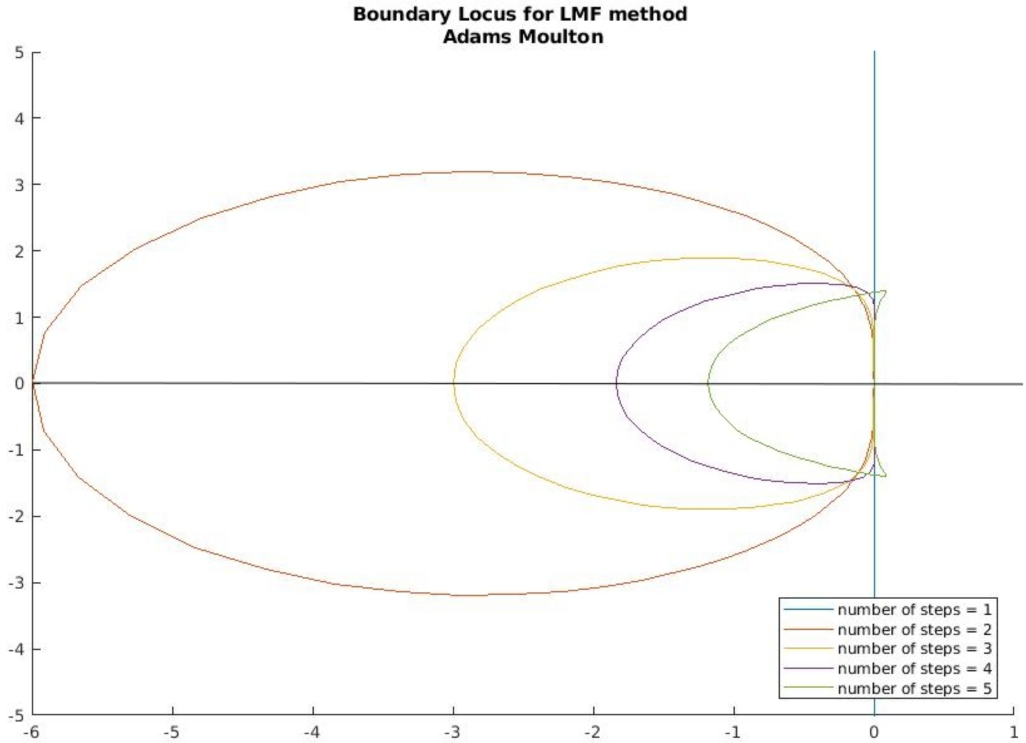


Figure 28: Adams Moulton Boundary Locus

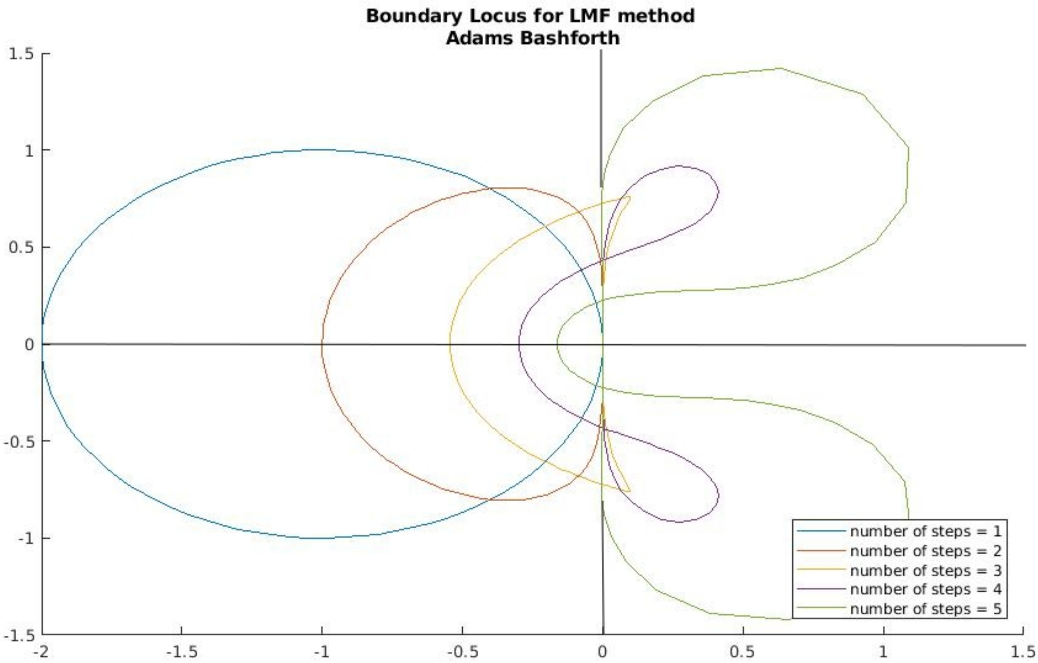


Figure 29: Adams Bashforth Boundary Locus

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

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

The number of people aged between

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

Where

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

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

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

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

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

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 = linsolve (M,actgen);
end
```

Listing 28: Leslie model function

It is clear that it 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 29: Leslie simulation function for a fixed number of steps

Consider now to execute this script:

```
gens = population([0.1,2,3],[1,2],[5,5,5],3);
plot(gens(:,1));
hold on;
plot(gens(:,2));
plot(gens(:,3));
axis([1 3 0 7]);
```

Listing 30: Leslie script

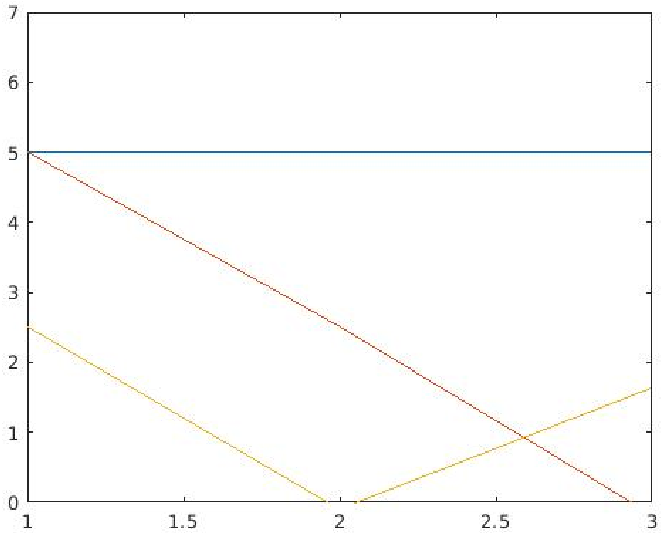


Figure 30: With these parameters, population dies out

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:

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

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

Thise 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.
We can write the following MATLAB function:

```
function [res] = arms(a,b,c,d,xi,eta,x0,y0,h,T)
  A = [-a,b;
       c,-d];
   res = [x0,y0];
   f = [xi,eta];
   iter = round(T/h);
   for i = 1:iter
     res = res+h*(A*transpose(res)+f);
   end
end
```

Listing 31: Arms race model

And the following script:

```
a=0.9;
b=2;
c=1;
d=2;
xi=1.6;
eta=1.4;
x0=5;
y0=0.5;
T=7;
h=0.01;
V=arms(a,b,c,d,xi,eta,x0,y0,h,T);
plot(V);
```

Listing 32: Arms race model plot script

We obtain:

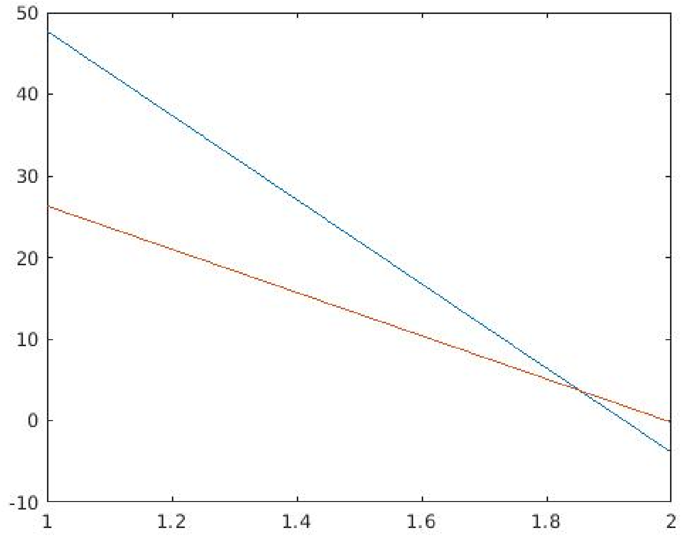


Figure 31: Arms race model with a=0.9,b=2,c=1,d=2 for which is not verified $ad > bc$

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:

$$g'(t) = -m_1 g - m_2 h + \gamma_g(g, h) \ \ (*)$$
$$h'(t) = m_3 g - m_4 h + \gamma_h(g, h)$$

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 33: 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:

```
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);
hold all;
plot(optimalg,'DisplayName','Normal glycemic level');
plot(badg,'DisplayName','Pathologic glycemic level');
legend(gca,'show','Location','northeast');
```

Listing 34: Diabetes Mellitus Script

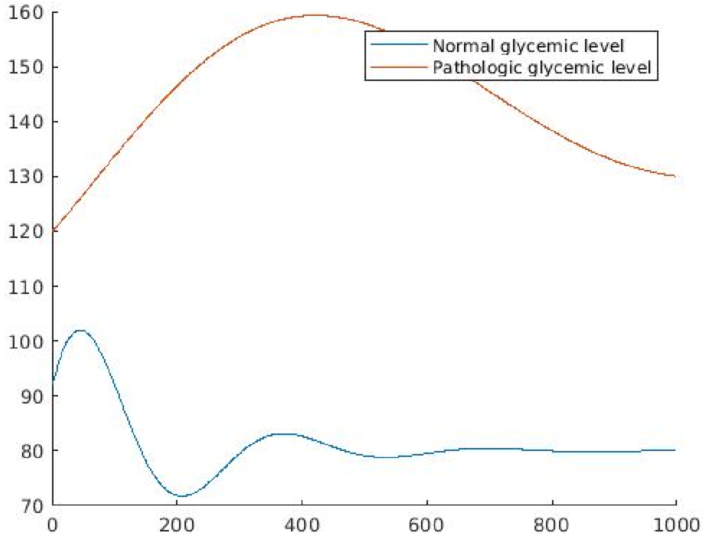


Figure 32: Diabetes mellitus plot for an healty and a non healty patient