



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 8, 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

1	Stirling Numbers	5
2	Fibonacci Numbers: the fatal bit problem	7
3	Mortgage repayment plan model	9
4	Coweb Model	14
5	Model of a national economy	18
6	Linear Multistep Methods	22
6.1	The problem	22
6.2	An implementation	23
6.3	A more general implementation	28
6.4	LMF vs Runge Kutta methods	34
6.5	A practical application	38
6.6	Linear Multistep Methods: The Boundary Locus	41
7	Leslie Model	47
8	Arms Race model	49
9	Diabetes mellitus model	51

Listings

1	Stirling Numbers MATLAB function	5
2	First n rows of Stirling Numbers	5
3	Fibonacci / Fatal Bit function in 64 bit precision	7
4	Plot of Fatal Bit	7
5	Fibonacci / Fatal Bit function in 32 bit precision	8
6	Mortgage repayment function	10
7	Loan repayment plot script with different installment amount	11
8	Mortgage repayment plan table	12
9	Matlab Coweb Model	15
10	Stable Coweb Model	15
11	Unstable Coweb Model	15
12	Divergent Coweb Model	15
13	Samuelson Model	19
14	GDP first case plot	20
15	GDP second case plot	21
16	GDP third case plot	21
17	Explicit euler MATLAB implementation	23
18	Implicit euler MATLAB implementation	24
19	Implicit and explicit euler plot	25
20	Implicit and explicit euler plot in another case	26
21	Generic explicit LMF solver	28
22	Comparison between LMF methods (comparelmf)	29
23	Error of LMF methods	31
24	Explicit RK implementation	35
25	Runge Kutta approximation plot (rk.m)	36
26	Plot of RK error	37
27	Approximation of capacitor's voltage	39
28	Comparison between euler and RK	40
29	LMF solver	42
30	Boundary Locus calculation	43
31	Plot Boundary Locus Script	43
32	Leslie model function	47
33	Leslie simulation function for a fixed number of steps	48
34	Leslie script	48
35	Arms race model	49
36	Arms race model plot script	50
37	Diabetes Mellitus Function	52
38	Diabetes Mellitus Script	52

List of Figures

1	First 7 rows of second specie Stirling Numbers	6
2	Effects of rounding in 64 bit precision	8
3	Effects of rounding in 32 bit precision	9
4	Residual debt with different installments amounts	11
5	Loan table obtained executing listing 7	12
6	Plot of Stable Coweb successions	16
7	Plot of Unstable Coweb successions	17
8	Plot of Divergent Coweb successions	17
9	Supply and Demand functions plot	18
10	GDP values in case 1	20
11	GDP values in case 2	21
12	GDP values in case 3	21
13	Approximation with mesh integration step $h = \frac{3}{10}$	25
14	Approximation with mesh integration step $h = \frac{3}{20}$	26
15	Implicit and explicit euler's methods correct behavior	27
16	Explicit euler's divergence as a reaches 0^+	27
17	Midpoint formula problem with step $h = 0.01$	30
18	Midpoint formula problem with step $h = 0.001$	31
19	Error of LMF methods from 0 to 0.01	32
20	Error of LMF methods from 0 to 0.05	32
21	Error of LMF methods from 0 to 0.75	33
22	Error of LMF methods from 0 to 1	33
23	Error of LMF methods from 0 to 2.5	34
24	The approximation performed by the shown RK method overlaps with the exact solution	36
25	Error of the implemented RK method	37
26	Plot of approximated capacitor's head voltage	39
27	Verification of the obtained voltage result	40
28	Comparison between approximated voltage with euler and RK	41
29	BDF Boundary Locus	44
30	Adams Moulton Boundary Locus	45
31	Adams Bashforth Boundary Locus	46
32	With these parameters, population dies out	48
33	Arms race model with $a=0.9, b=2, c=1, d=2$ for which is not verified $ad > bc$	50
34	Diabetes mellitus plot for an healthy and a non healthy patient	53

1 Stirling Numbers

Consider the following:

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

As the standard definition of numbers power in the continuous 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:

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

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

S_i^n are defined as:

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

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 .

```

1 function [s] = getStirling(n,k)
2     if (k > n)
3         s=0;
4     elseif (k == 1 || n == k)
5         s=1;
6     elseif (k == 0 && n >= 1)
7         s=0;
8     else
9         s = k * getStirling(n-1, k)
10            + getStirling(n-1, k-1);
11     end
12 end

```

Listing 1: Stirling Numbers MATLAB function

Consider now the following function:

```

1 function [S] = stirling(n)
2     S=zeros(n);
3     for i=1:n
4         for j=1:i
5             S(i,j) = getStirling(i,j);
6         end
7     end
8 end

```

Listing 2: First n rows of Stirling Numbers

Given n , the function `stirling(n)` returns a matrix of size $n \times n$ 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:

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

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

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

Which is known as the Fibonacci succession of numbers.

In order to get a closed form which explicitly define the n -th element of the succession, we must solve the following difference equation:

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

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 explicitly 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 solving $(**)$, but we may face the *fatal bit problem*.

Consider the following MATLAB functions which calculate the terms of the succession defined by $(*)$, given y_1 and y_2 as the initial conditions.

```
1 function [yn] = fatalbit (len)
2   yn = zeros(1, len);
3   yn(1)=1;
4   yn(2)=(1-sqrt(5))/2;
5   for i=3:len
6       yn(i)=yn(i-1)+yn(i-2);
7   end
8 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:

```
1 yn = fatalbit(70);
2 semilogy(abs(yn), 'r');
```

Listing 4: Plot of Fatal Bit

It is possible to see how succession values start 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, modifying the *fatalbit* function in such manner:

```

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

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

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:

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

Listing 6: Mortgage repayment function

Hence we can execute the following script:

```

1 C = 10000;
2 interest=0.05;
3 N=100;
4 hold all;
5 for i=1:5
6 ratei=loan(C,460+(i-1)*20,interest,N);
7 plot(ratei,'DisplayName',['installment amount = ' num2str(460+(i-1)*20)]);
8 end
9 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.

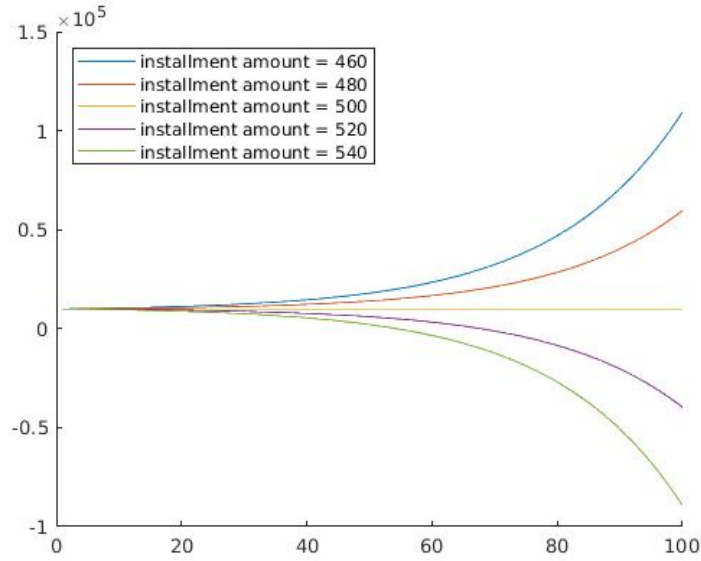


Figure 4: Residual debt with different installments amounts

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 behavior is compliant to the (2), in fact, if $\frac{r}{i} < C$ the succession diverges.

Suppose we want to contract a mortgage 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

```

1 function [resDebt,intQuote,capQuote] = loantable(C,interest ,singleimport ,n)
2     resDebt = loan(C,singleimport ,interest ,n);
3     intQuote = zeros(1,n+1);
4     intQuote(1) = 0;
5     intQuote(2) = C*interest ;
6     for i = 3:n+1
7         intQuote(i)=resDebt(i-1)*interest ;
8     end
9     capQuote = zeros(1,n+1);
10    capQuote(1) = 0;
11    for i = 2:n+1
12        capQuote(i) = singleimport-intQuote(i);
13    end
14 end
15
16 [resDebt,intQuote,capQuote] = loantable(10000,0.05,1406.9,10);
17 N = 11;
18 table = zeros(N,3);
19 table(1,1)=10000;
20 for j=2:N
21     table(j,1)=resDebt(j-1);
22 end
23 for i=1:N
24     table(i,2)=intQuote(i);
25     table(i,3)=capQuote(i);
26 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:

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

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:

$$d_0 \gg s_0 > 0$$

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:

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

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

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

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

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

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:

```

1 function [pn,sn,dn,sfun,dfun] = coweb(d0,a,s0,b,p0,nmax)
2   pn = zeros(1,nmax);
3   sn = zeros(1,nmax);
4   dn = zeros(1,nmax);
5   sfun = @(x) b*x+s0;
6   dfun = @(x) -a*x+d0;
7   diff = (d0-s0);
8   pn(1)= p0;
9   sn(1)= s0;
10  dn(1)= -a*p0+d0;
11  for i = 2:nmax
12    pn(i) = diff -b*pn(i-1)/a;
13    sn(i) = sfun(pn(i-1));
14    dn(i) = dfun(pn(i));
15  end
16 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 behavior is sensible to parameter variations:

```

1 [pnStable,snStable,dnStable,sfun,dfun] = coweb(100,0.15,10,0.10,400,10);
2 plot(pnStable);
3 hold on;
4 title('Stable Coweb');
5 plot(snStable);
6 plot(dnStable);

```

Listing 10: Stable Coweb Model

```

1 [pnUnstable,snUnstable,dnUnstable,sfun,dfun] = coweb(100,0.05,10,0.05, 950,10);
2 plot(pnUnstable);
3 hold on;
4 title('Unstable Coweb');
5 plot(snStable);
6 plot(dnStable);

```

Listing 11: Unstable Coweb Model

```

1 [pnDivergent,snDivergent,dnDivergent,sfun,dfun] = coweb(100,0.09,50,0.1, 300,10);
2 plot(pnDivergent);
3 hold on;
4 title('Divergent Coweb');
5 plot(snDivergent);
6 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.

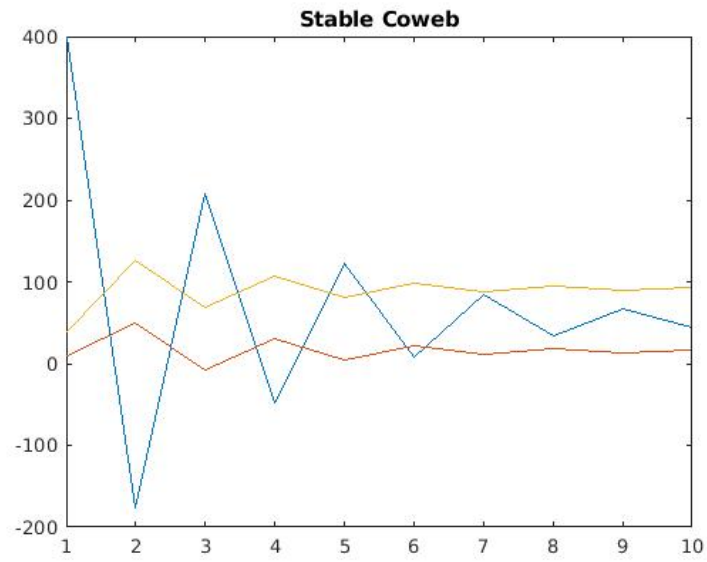


Figure 6: Plot of Stable Coweb successions

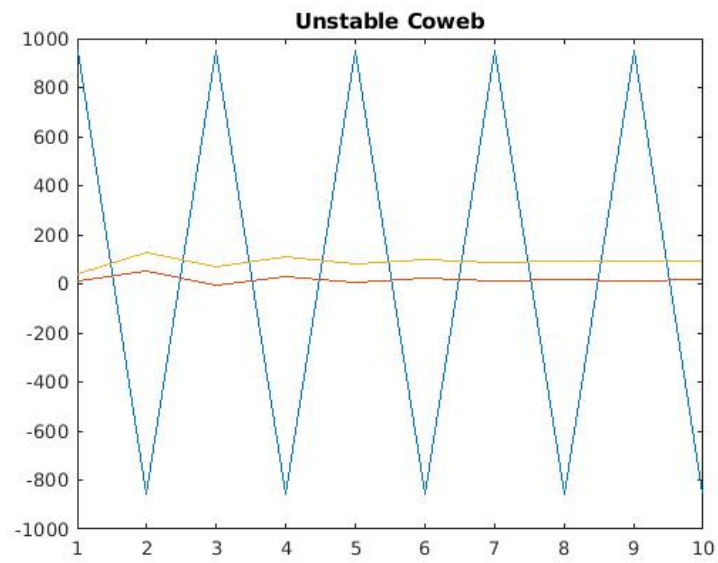


Figure 7: Plot of Unstable Coweb successions

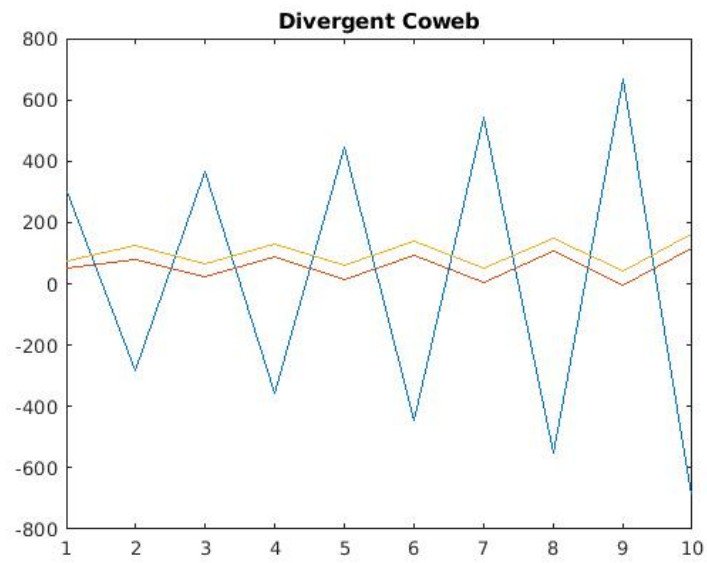


Figure 8: Plot of Divergent Coweb successions

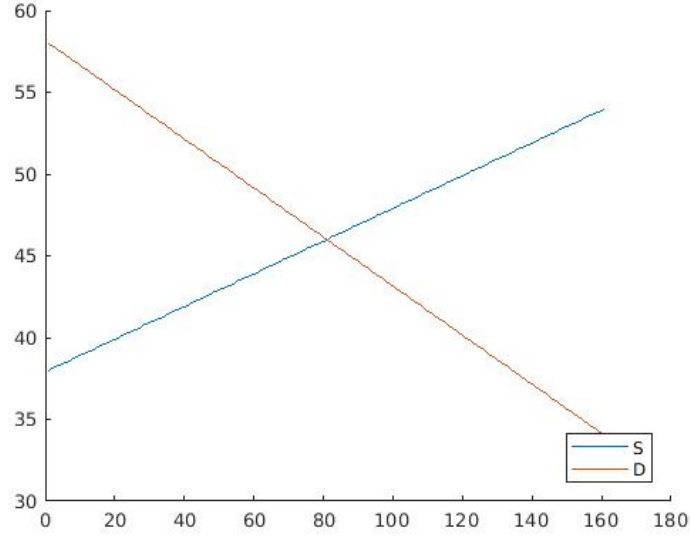


Figure 9: Supply and Demand functions plot

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:

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

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.

```

1 function [Y,C,I] = samuelson(alpha, rho, g, y0, y1, nmax)
2   Y=zeros(1, nmax);
3   Y(1)=y0;
4   Y(2)=y1;
5   C=zeros(1, nmax);
6   C(1)=alpha*y0;
7   C(2)=C(1);
8   I = zeros(1, nmax);
9   I(1)=0;
10  I(2)=rho*(C(2)-C(1));
11  for i=3:nmax
12      C(i)=alpha*Y(i-1);
13      I(i)=rho*(C(i)-C(i-1));
14      Y(i)=C(i)+I(i)+g;
15  end
16 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:

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

Listing 14: GDP first case plot

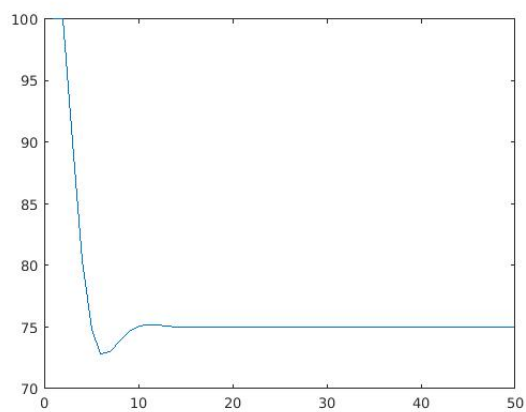


Figure 10: GDP values in case 1

2. $\alpha > 1$:

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

Listing 15: GDP second case plot

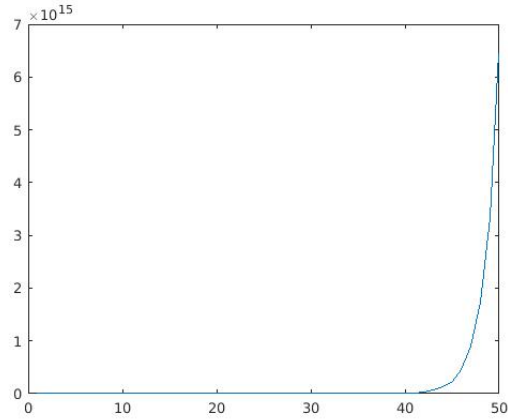


Figure 11: GDP values in case 2

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

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

Listing 16: GDP third case plot

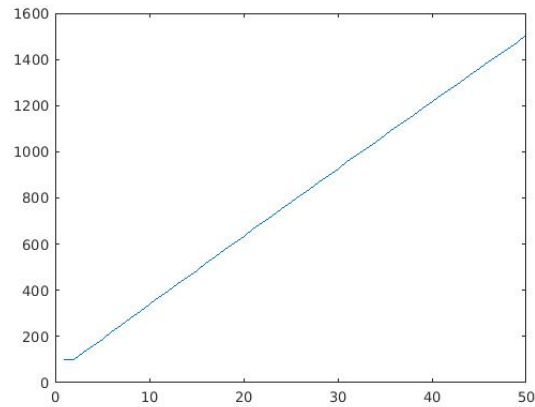


Figure 12: GDP values in case 3

6 Linear Multistep Methods

6.1 The problem

Consider the following problem:

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

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 continuous 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:

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

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

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:

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

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 accurate 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 polynomials, 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:

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

6.2 An implementation

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:

```

1 function [approx_values] = euler_expl( abscisse, x0, step_size, n_steps, fun)
2     approx_values = zeros(1, n_steps);
3     approx_values(1) = x0;
4     for i = 0:n_steps-2
5         h = step_size;
6         ti = abscisse(i+1);
7         yi = approx_values(i+1);
8         fi = fun(ti, yi);
9         approx_values(i+2) = yi + h * fi;
10    end
11 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} .

```

1 function [approx_values] = euler_impl( abscisse ,x0 ,step_size , n_steps , fun)
2     approx_values = zeros(1,n_steps);
3     approx_values(1) = x0;
4     for i = 0:n_steps-2
5         h = step_size;
6         ti = abscisse(i+2);
7         yi = approx_values(i+1);
8         init_e = one_step_fwd(ti ,yi ,step_size , fun);
9         fi = fun(ti ,init_e);
10        approx_values(i+2) = yi + h * fi;
11    end
12 end
13
14 function [y1] = one_step_fwd(t , y0 , h , f)
15     y1 = y0 + h*f(t ,y0);
16 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 analitically, we get:

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

↓

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

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

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

↓

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

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


```

1 FUN = @(t) (3/2)*(exp(t))+sin(t)/2-cos(t)/2;
2 fun = @(t,y) (cos(t)+y);
3 a = input('Insert left extreme of the interval\n');
4 b = input('Insert right extreme of the interval\n');
5 n = input('Insert step numbers\n');
6 if a>b
7     temp = a;
8     a = b;
9     b = temp;
10 end
11 x0 = FUN(a);
12 real_abs = linspace(a,b,10000);
13 abscisse = linspace(a,b,n);
14 h = (b-a)/n;
15 expl = euler_expl(abscisse,x0,h,n,fun);
16 impl = euler_impl(abscisse,x0,h,n,fun);
17 exact = FUN(real_abs);
18 plot(abscisse,expl,'DisplayName','Explicit Euler');
19 hold all;
20 strn = num2str(n);
21 title(['n steps = ', strn]);
22 plot(abscisse,impl,'DisplayName','Implicit Euler');
23 plot(real_abs,exact,'DisplayName','Exact Function');
24 legend(gca,'show','Location','northwest');

```

Listing 19: Implicit and explicit euler plot

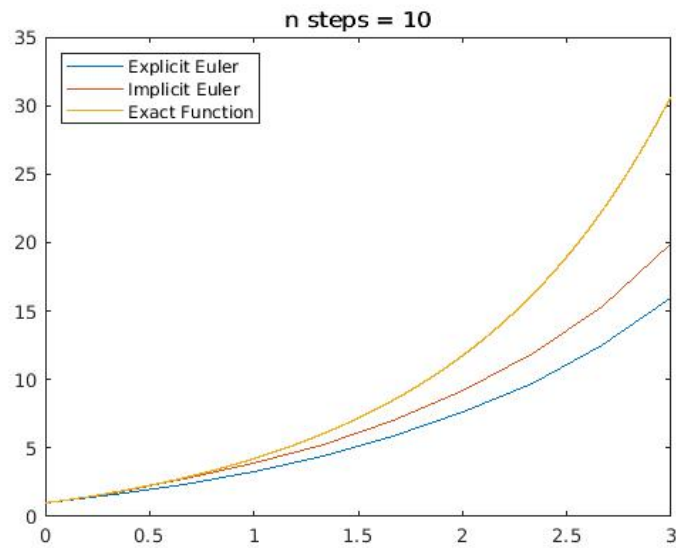


Figure 13: Approximation with mesh integration step $h = \frac{3}{10}$

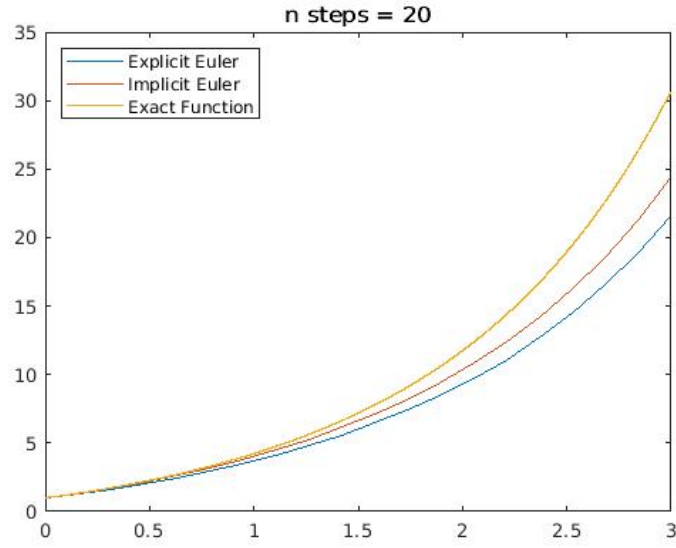


Figure 14: Approximation with mesh integration step $h = \frac{3}{20}$

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:

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

Listing 20: Implicit and explicit euler plot in another case

Executing the script with following values:

$$a = 1$$

$$b = 25$$

$$n = 50 \text{ steps}$$

We obtain the plot shown in figure 15.

It is immediate to note 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.

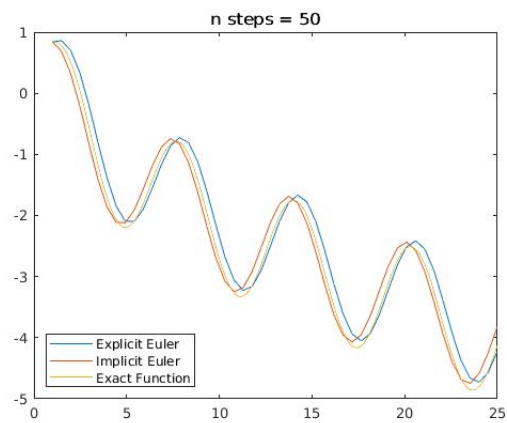


Figure 15: Implicit and explicit euler's methods correct behavior

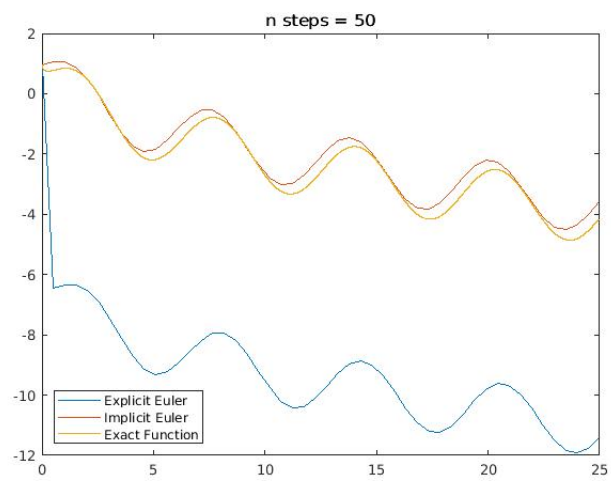


Figure 16: Explicit euler's divergence as a reaches 0^+

6.3 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:

```

1 function [yn] = multistep(ai, bi, init_conds, intval, f)
2     lbi = length(bi);
3     impl = false;
4     if bi(lbi) ~= 0
5         impl = true;
6     end
7     N = length(intval);
8     k = length(init_conds);
9     lai = length(ai);
10    if lai ~= lbi
11        disp('alpha_i and beta_i cannot have different sizes');
12        return;
13    end
14    if lai ~= k+1
15        disp('wrong ai and bi size');
16        return;
17    end
18    if N < k
19        disp('mesh size is too small for the chosen number of steps');
20        return;
21    end
22    intval = sort(intval);
23    h = intval(2)-intval(1);
24    yn = zeros(1, N);
25    yn(1:k) = init_conds(1:k);
26    n = k;
27    while n < N
28        known_yn = yn(n:-1:n-k+1);
29        comb_lin1 = comblin(known_yn, ai(2:length(ai)));
30        known_yn1 = yn(n+1-k:n+1);
31        tn = intval(n+1-k:n+1);
32        fn = zeros(1, k+1);
33        for i = 1:k+1
34            fn(i) = f(tn(i), known_yn1(i));
35        end
36        if impl
37            fn(k+1) = one_step_fwd(tn(k+1), yn(n), h, f);
38        end
39        comb_lin2 = comblin(fn, bi);
40        yn(n+1) = (comb_lin1 + h * comb_lin2) / -ai(1);
41        n = n+1;
42    end
43 end

```

```

44 function [cl] = comblin(v,alpha)
45     n = length(v);
46     if n ~= length(alpha)
47         disp('vector and coefficients must have same sizes');
48     return;
49     end
50     cl = 0;
51     for i = 1:n
52         cl = cl+v(i)*alpha(i);
53     end
54 end

```

Listing 21: Generic explicit LMF solver

This function is able to perform this task for a generic LMF method. Its behavior derives directly from the definition of LMF method seen in the previous paragraph.

If:

$$\beta_i(k) \neq 0$$

Then it means we are facing with an implicit method, so the strategy is the same as the one we've seen before: perform a one-step forward to get an initial estimation of the next value to compute. 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:

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

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

- Midpoint (explicit)
- Euler (explicit)
- Adams Bashforth (explicit)
- Adams Moulton (implicit)
- Trapezes (implicit)
- Simpson (implicit)

```

1 %comparelmf.m:
2 fun = @(t,y) 10*y-2*y^2;
3 FUN = @(t) 5/(1+4*exp(-10*t));
4 intval = linspace(0,2.5,250); %change 250 to 2500 for step 0.001
5 real_values = zeros(1,length(intval));
6 for i=1:length(intval)
7     real_values(i)=FUN(intval(i));
8 end
9 y0 = 1;
10 midpoint_ai = [-1, 0, 1];

```

```

11 midpoint_bi = [0, 2, 0];
12 adams_bf_ai = [-1,1,0];
13 adams_bf_bi = [-1/2,3/2,0];
14 adams_m_ai = [-1,1,0];
15 adams_m_bi = [-1/12,8/12,5/12];
16 simpson_ai = [-1,1,0];
17 simpson_bi = [1/3,4/3,1/3];
18 euler_ai = [-1,1];
19 euler_bi = [1,0];
20 trap_ai = [-1,1];
21 trap_bi = [1/2,1/2];
22 euler=multistep(euler_ai,euler_bi,y0,intval,fun);
23 trap=multistep(trap_ai,trap_bi,y0,intval,fun);
24 midpoint=multistep(midpoint_ai,midpoint_bi,[y0,y0],intval,fun);
25 adams_bf=multistep(adams_bf_ai,adams_bf_bi,[y0,y0],intval,fun);
26 adams_m=multistep(adams_m_ai,adams_m_bi,[y0,y0],intval,fun);
27 simpson=multistep(simpson_ai,simpson_bi,[y0,y0],intval,fun);
28 hold on;
29 plot(intval,midpoint,'DisplayName','Midpoint');
30 plot(intval,euler,'DisplayName','Euler');
31 plot(intval,trap,'DisplayName','Trapezes');
32 plot(intval,adams_m,'DisplayName','Adams - Moulton');
33 plot(intval,adams_bf,'DisplayName','Adams - BF');
34 plot(intval,simpson,'DisplayName','Simpson');
35 plot(intval,real_values,'DisplayName','Exact solution');
36 legend(gca,'show','Location','southwest');

```

Listing 22: Comparison between LMF methods (comparelmf)

In the following figures 20 and 21, it is possible to see spurious oscillations performed by the approximated values with the midpoint method in the interval $[0, 2.5]$ with step size $h = 0.01$ (figure 17) and $h = 0.001$ (figure 18).

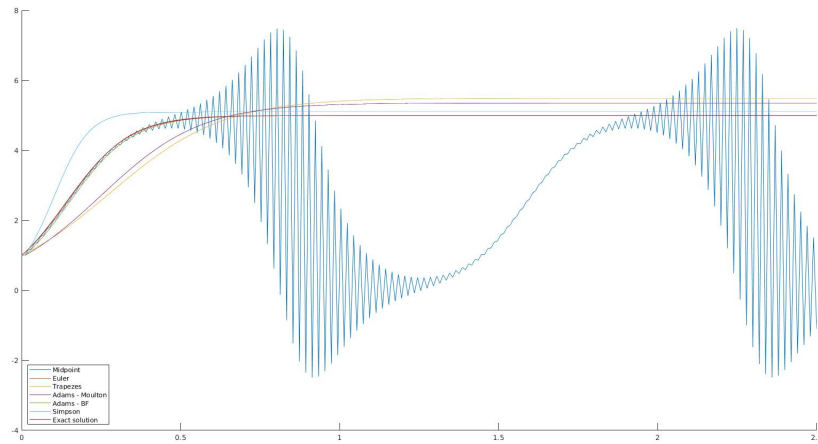


Figure 17: Midpoint formula problem with step $h = 0.01$

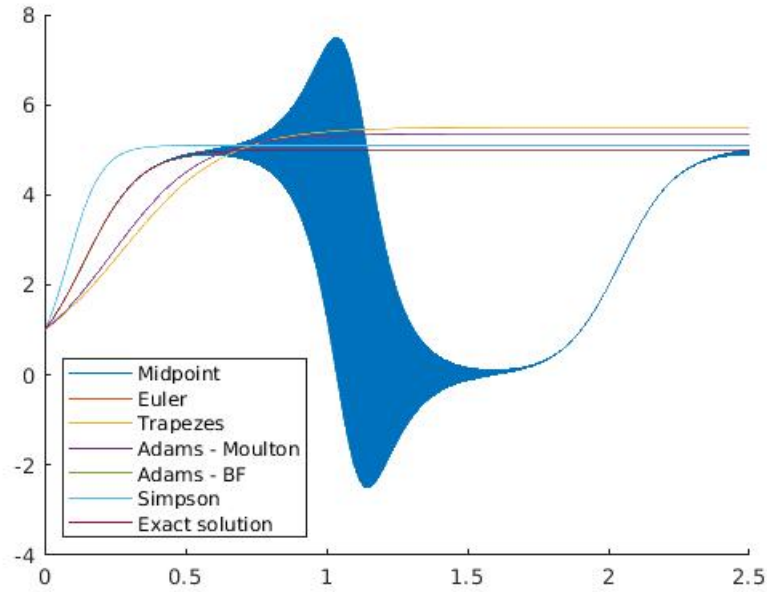


Figure 18: Midpoint formula problem with step $h = 0.001$

Last we examine the errors committed by these six LMF methods by executing the following script with step $h = 0.001$:

```

1 comparelmf;
2 close;
3 ea = abs(adams_bf-real_values);
4 em = abs(midpoint-real_values);
5 ee = abs(euler-real_values);
6 em = abs(adams_m-real_values);
7 et = abs(trap-real_values);
8 es = abs(simpson-real_values);
9 hold on;
10 left_in=1;
11 right_in=10; %change for others
12 plot(intval(left_in:right_in),em(left_in:right_in),'DisplayName','Midpoint Error');
13 plot(intval(left_in:right_in),ea(left_in:right_in),'DisplayName','Adams-BF Error');
14 plot(intval(left_in:right_in),ee(left_in:right_in),'DisplayName','Euler error');
15 plot(intval(left_in:right_in),et(left_in:right_in),'DisplayName','Trapezes Error');
16 plot(intval(left_in:right_in),em(left_in:right_in),'DisplayName','Adams-M Error');
17 plot(intval(left_in:right_in),es(left_in:right_in),'DisplayName','Simpson Error');
18 legend(gca,'show','Location','northwest');

```

Listing 23: Error of LMF methods

Here it follows the resulting plots:

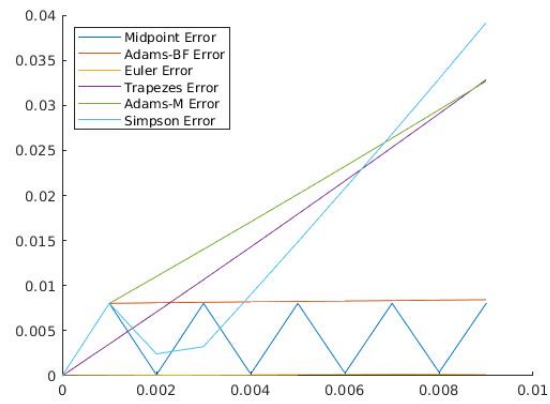


Figure 19: Error of LMF methods from 0 to 0.01

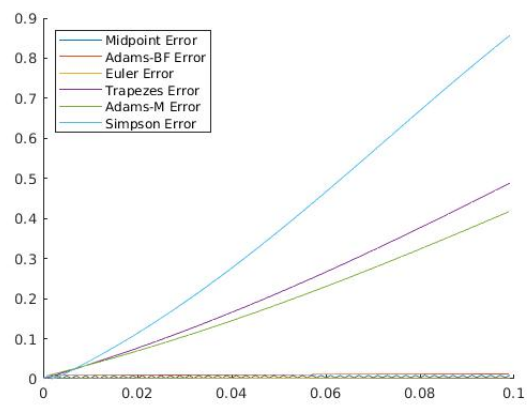


Figure 20: Error of LMF methods from 0 to 0.05

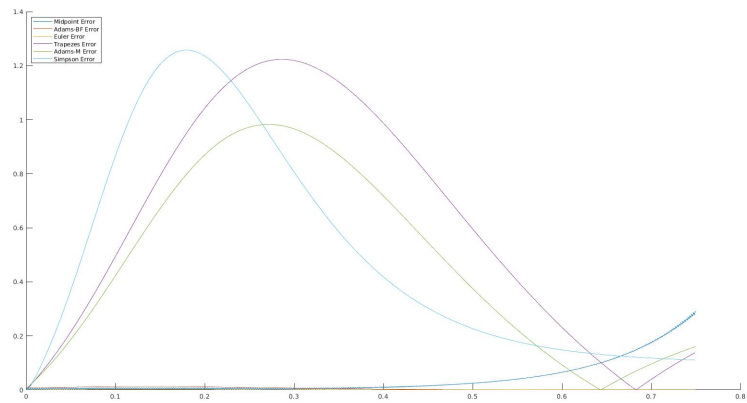


Figure 21: Error of LMF methods from 0 to 0.75

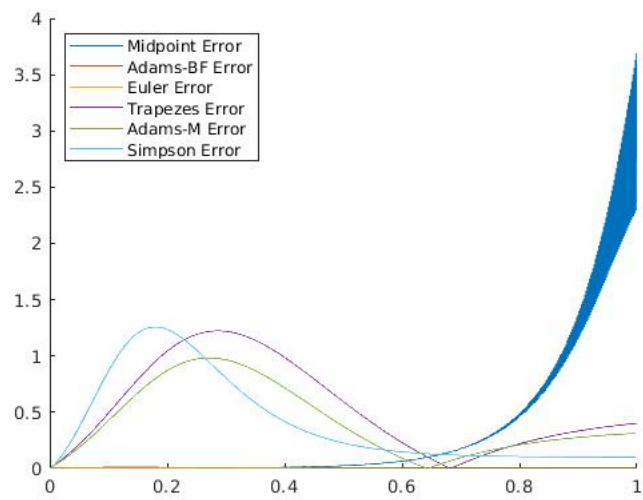


Figure 22: Error of LMF methods from 0 to 1

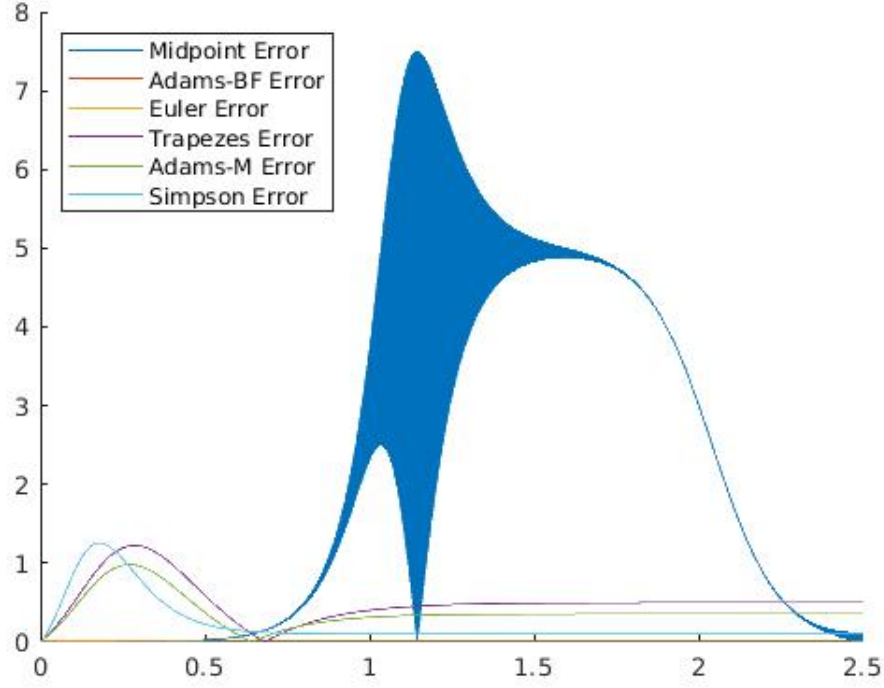


Figure 23: Error of LMF methods from 0 to 2.5

6.4 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

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

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

```

1 function [yn] = explrungekutta(y0, intval, f)
2   N = length(intval);
3   h = intval(2)-intval(1);
4   yn = zeros(1,N);
5   yn(1) = y0;
6   for i=1:N-1
7       Y1 = f(intval(i), yn(i));
8       Y2 = f(h/2+intval(i), yn(i)+h/2*Y1);
9       Y3 = f(h/2+intval(i), yn(i)+h/2*Y2);
10      Y4 = f(intval(i)+h, yn(i)+h*Y3);
11      yn(i+1) = yn(i) + (h/6)*(Y1+2*Y2+2*Y3+Y4);
12   end
13 end

```

Listing 24: Explicit RK implementation

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

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

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

Recap that the mesh is:

$$\begin{aligned} \{t_i\} &\in [0, 2.5] \\ h &= 0.001 \end{aligned}$$

Executing the following script:

```
1 %rk.m:
2 comparelbf; %inherit y0, fun, interval and real_values
3 close;
4 rk = explrungekutta(y0,intval,fun);
5 hold on;
6 plot(intval,real_values,'DisplayName','Exact Solution');
7 plot(intval,rk,'DisplayName','Runge Kutta');
8 legend(gca,'show','Location','southeast');
```

Listing 25: Runge Kutta approximation plot (rk.m)

We obtain the plot shown in figure 24.

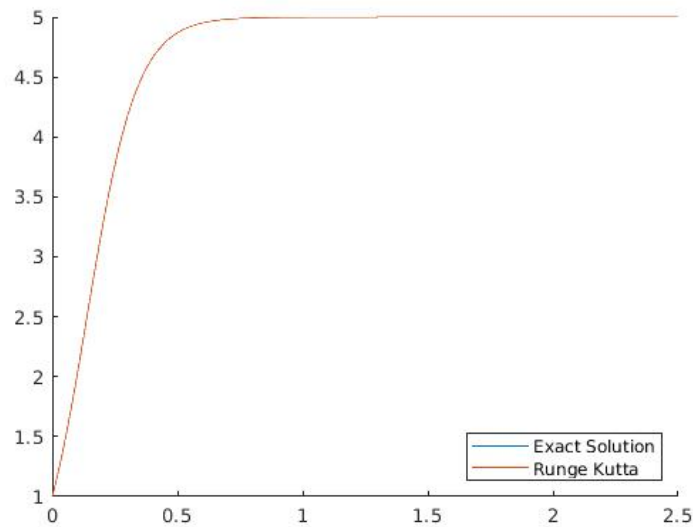


Figure 24: The approximation performed by the shown RK method overlaps with the exact solution

The error committed by the implemented RK method, however, is not always 0. Consider the following script:

```

1 rk;
2 close;
3 er = abs(rk-real_values);
4 hold on;
5 l_in=1;
6 r_in=2500;
7 plot(intval(l_in:r_in),er(l_in:r_in),'DisplayName','Runge Kutta Error');
8 legend(gca,'show','Location','northwest');

```

Listing 26: Plot of RK error

Executing it, we obtain the result shown in figure 25.

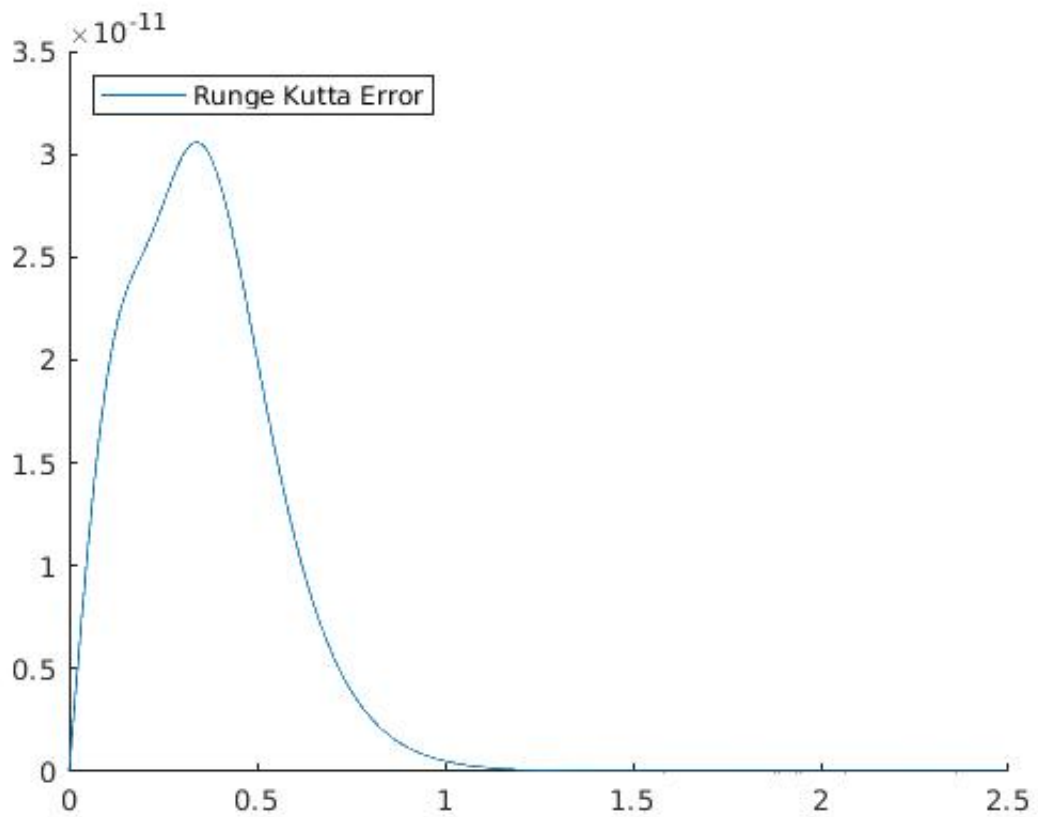


Figure 25: Error of the implemented RK method

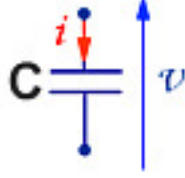
We note that the implemented RK method commits its maximum error approximately at $t_i \approx 0.45$, and this error results $\approx 3 * 10^{-11}$. However, after a settling period in which the error is however substantially negligible, we note that for $t_i > \approx 1$ the error becomes null.

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.

6.5 A practical application

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:

$$v(t_0) = v(10 \text{ s}) = 2 \text{ V}$$

Electric current and voltage are characterized by the following relationship:

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

Summarizing, we have:

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

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

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

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

```

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

```

Listing 27: Approximation of capacitor's voltage

Obtaining the result shown in figure 26.

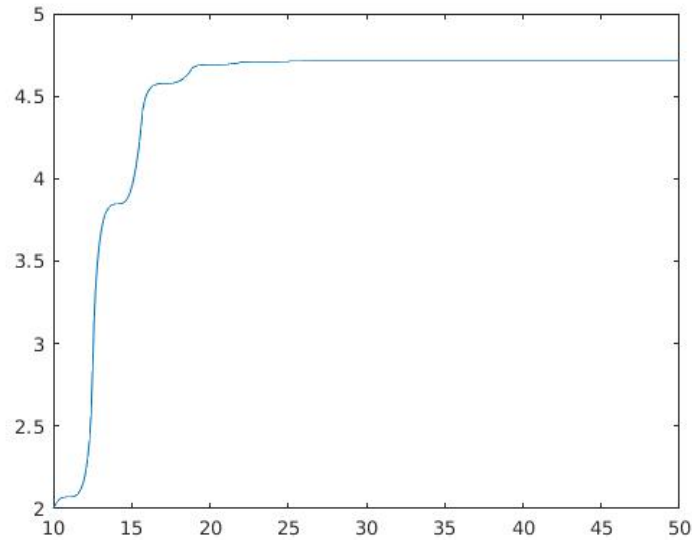
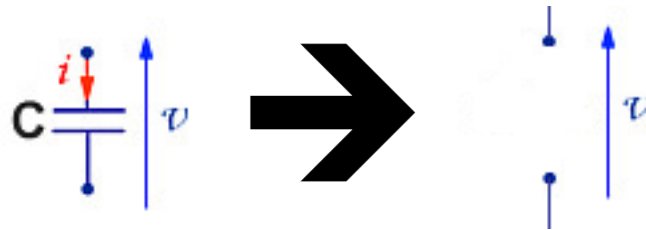


Figure 26: Plot of approximated capacitor's head voltage

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. 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, considering only the known current's differential equation, was difficult to forecast. Even if we 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.

It follows a brief experimental proof of our results using the mathematical engine *Wolfram Alpha*, in figure 27.

Last, in figure 28, we compare the implicit Euler and RK approximations of the capacitor's head voltage, executing:

```
1 dotv = @(t,v) log(abs(sin(t)))*cos(v); % Electric current function
2 ti = linspace(10,50,400); % Interval of interest
3 x0 = 2; % v(10) = 2 V
4 y1 = euler_impl(ti,x0,0.1,400,dotv);
5 y2 = explrungekutta(x0,ti,dotv);
6 plot(ti,y1,'DisplayName','Euler');
7 hold on;
8 plot(ti,y2,'DisplayName','Runge Kutta');
9 legend(gca,'show','location','southeast');
```

Listing 28: Comparison between euler and RK

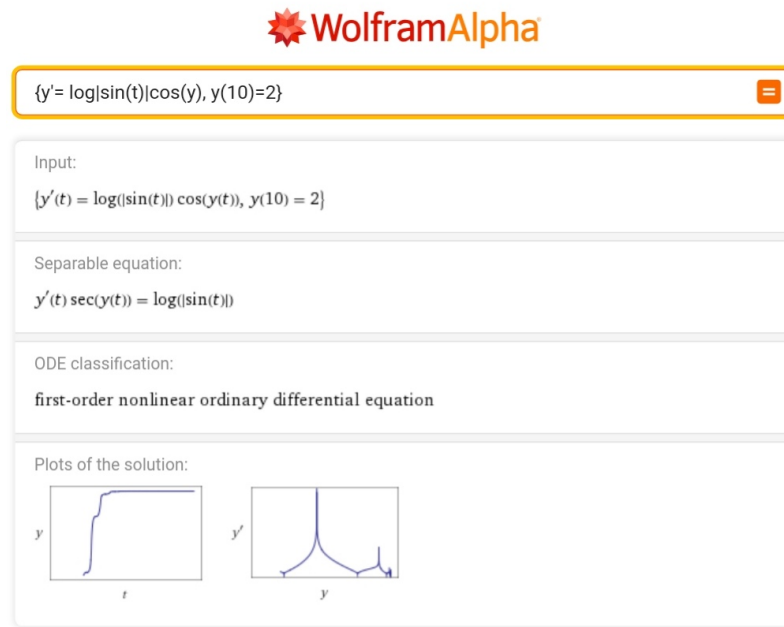


Figure 27: Verification of the obtained voltage result

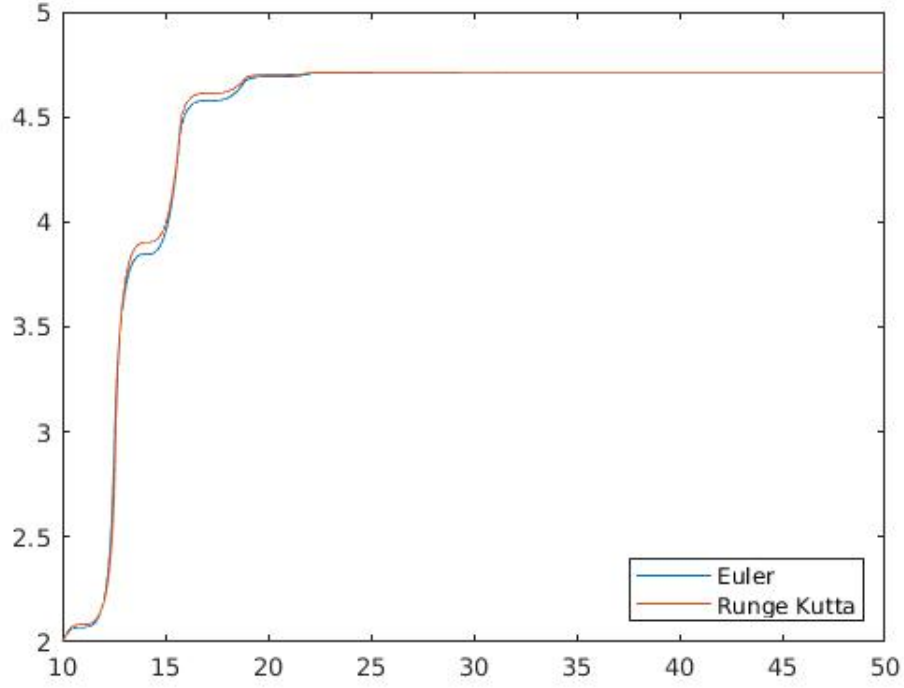


Figure 28: Comparison between approximated voltage with euler and RK

6.6 Linear Multistep Methods: The Boundary Locus

A particular Linear Multistep Method is identified by its ρ, σ polynomials.

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 polynomials, and

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

is the stability polynomial 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 calculate (ρ, σ) polynomials of LMF methods:

- BDF
- Adams-Moulton
- Adams-Bashfort

```

1 function [ro,sigma] = lmf( tipo , k )
2 %
3 % [ro,sigma] = lmf( tipo , k )      Calcola i polinomi ro e sigma del metodo
4 %                                  lmf a k passi specificato dal tipo:
5 %
6 %                                  0 : BDF
7 %                                  1 : Adams–Moulton
8 %                                  2 : Adams–Bashforth
9 %
10 if tipo==0 % BDF
11     sigma = [1 zeros(1,k)];
12     b     = [0:k].*[k.^[0:k]]/k;
13     ro     = vsolve(k:-1:0,b);
14 elseif tipo==1 % Adams–Moulton
15     ro     = [1 -1 zeros(1,k-1)];
16     j      = [1:k+1];
17     b      = (k.^j-(k-1).^j)./j;
18     sigma  = vsolve( k:-1:0, b );
19 elseif tipo==2 % Adams–Bashforth
20     ro     = [1 -1 zeros(1,k-1)];
21     j      = [1:k];
22     b      = (k.^j-(k-1).^j)./j;
23     sigma  = [0 vsolve( k-1:-1:0, b )];
24 else
25     disp(' tipo invalido!'), disp(' '),
26     help lmf, ro=[]; sigma=[];
27 end
28 return
29
30 function f = vsolve( x, b )
31 %
32 %     f = vsolve( x, b )      Risolve il sistema lineare  $W(x) f = b$ ,
33 %                             dove  $W(x)$  e' la Vandermonde definita
34 %                             dagli elementi del vettore x.
35 %
36 f = b;
37 n = length( x )-1;
38 for k = 1:n
39     for i = n+1:-1:k+1
40         f(i) = f(i) - x(k)*f(i-1);
41     end
42 end
43 for k = n:-1:1
44     for i = k+1:n+1
45         f(i) = f(i)/( x(i) - x(i-k) );
46     end
47 for i = k:n
48     f(i) = f(i) - f(i+1);
49 end
50 end
51 return

```

Listing 29: LMF solver

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

```

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

```

Listing 30: Boundary Locus calculation

Consider now to execute the following script:

```

1 method=input('insert method to use\n');
2 if method<0 || method > 2
3     disp('invalid method');
4     return;
5 end
6 steps=input('insert max number of steps\n');
7 theta0=input('insert theta initial value\n');
8 hold all;
9 for j=1:steps
10    q=boundaryLocus(theta0,j,method);
11    plot(q,'DisplayName',[ 'number of steps = ' num2str(j)]);
12    if method==0—
13        strm="BDF";
14    elseif method==1
15        strm="Adams Moulton";
16        axis([-6 1 -5 5]);
17    else
18        strm="Adams Bashforth";
19    end
20    title(['Boundary Locus for LMF method ' strm]);
21 end
22 legend(gca,'show','Location','southeast');

```

Listing 31: Plot Boundary Locus Script

By setting 5 maximum steps and 6 as theta initial value, we obtain the results shown in figures 29,30 and 31.

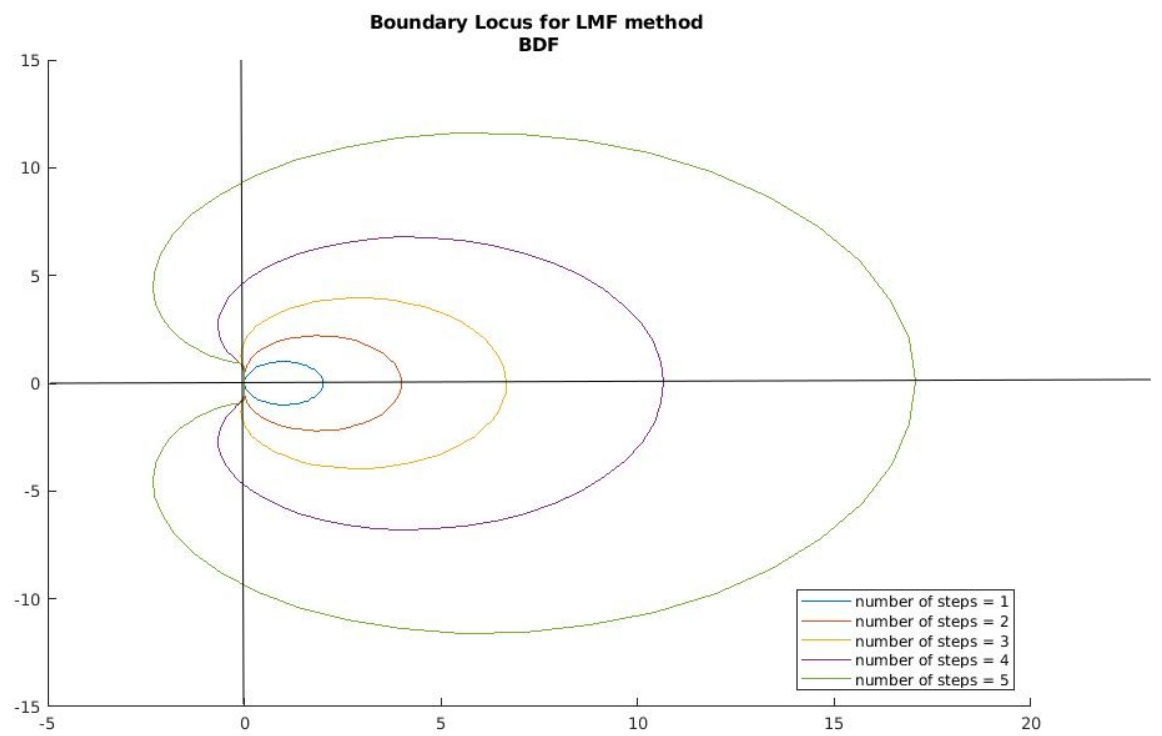


Figure 29: BDF Boundary Locus

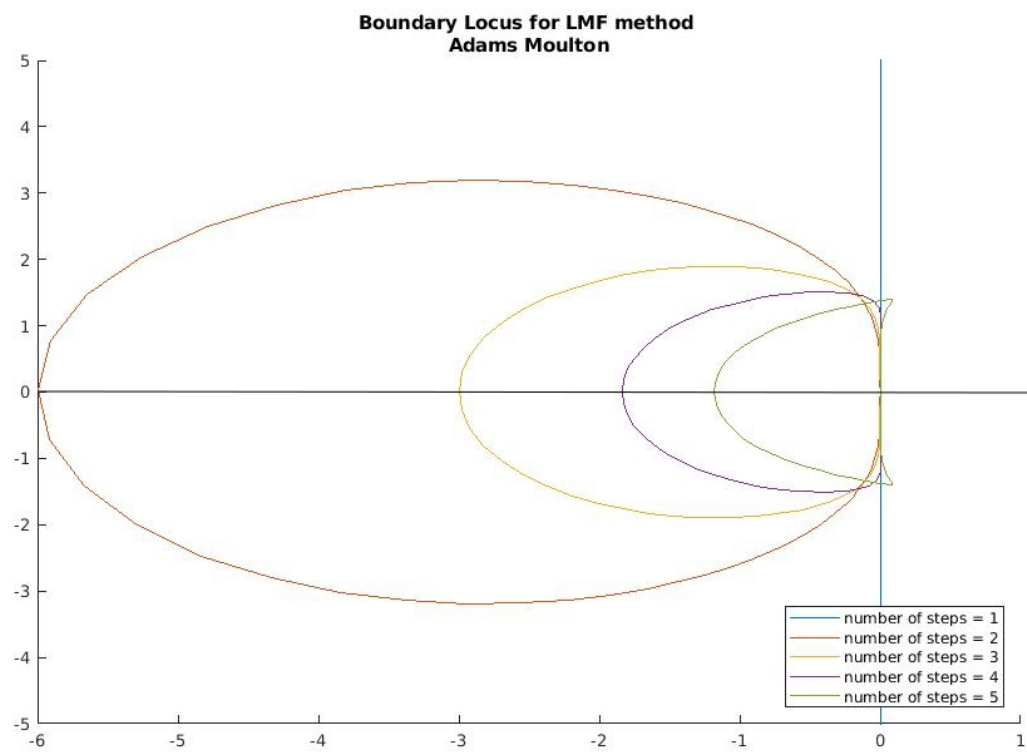


Figure 30: Adams Moulton Boundary Locus

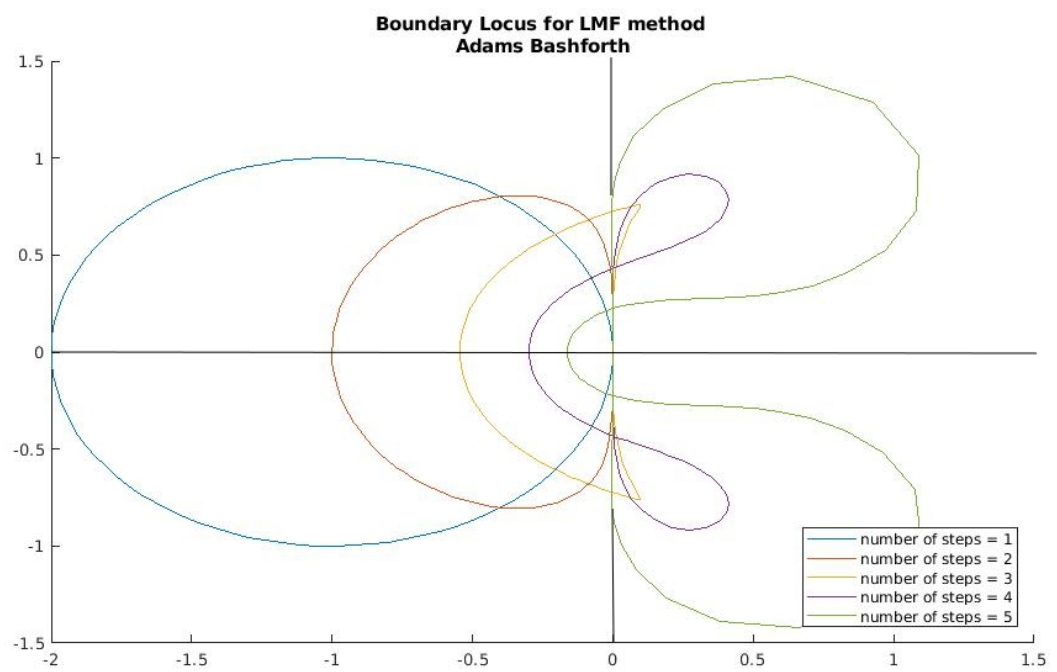


Figure 31: Adams Bashforth Boundary Locus

7 Leslie Model

This model describes the demographic structure of a homogeneous population, structured in age classes.

Let L be the maximum reachable 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

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

Is the survival coefficient 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 respective age classes.

We obtain the following discrete dynamic system:

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

Consider the following MATLAB function:

```

1 function [nextgen] = leslie(alphai,betai,actgen)
2     m = length(alphai);
3     if m ~= length(betai)+1
4         error('beta i must be length as alpha i -1');
5     end
6     M = zeros(m,m);
7     for i=1:m
8         M(1,i)=alphai(i);
9     end
10    for i=2:m
11        for j=1:m
12            if(i-1 == j)
13                M(i,j) = betai(j);
14            end
15        end
16    end
17    nextgen = linsolve (M,actgen);
18 end

```

Listing 32: Leslie model function

It is clear that it calculates next generation for each 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:

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

Listing 33: Leslie simulation function for a fixed number of steps

Consider now to execute this script:

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

Listing 34: Leslie script

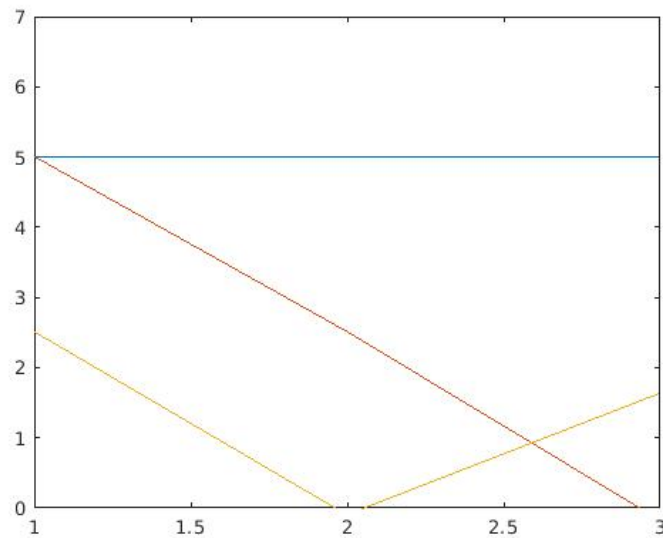


Figure 32: With these parameters, population dies out

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:

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

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

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.

We can write the following MATLAB function:

```

1 function [res] = arms(a,b,c,d,xi,eta,x0,y0,h,T)
2     A=[-a,b;
3         c,-d];
4     res = [x0,y0];
5     f = [xi,eta];
6     iter = round(T/h);
7     for i = 1:iter
8         res = res+h*(A*transpose(res)+f);
9     end
10    app = res(1,:);
11    res(1,:)=res(2,:);
12    res(2,:)=app;
13 end

```

Listing 35: Arms race model

And the following script:

```
1 a=0.9;  
2 b=2;  
3 c=1;  
4 d=2;  
5 xi=1.6;  
6 eta=1.4;  
7 x0=5;  
8 y0=0.5;  
9 T=7;  
10 h=0.01;  
11 V=arms(a,b,c,d,xi,eta,x0,y0,h,T);  
12 plot(V);
```

Listing 36: Arms race model plot script

We obtain:

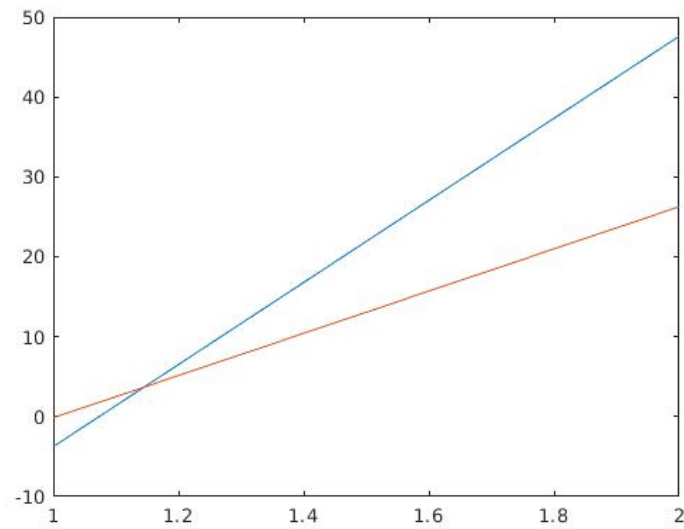


Figure 33: Arms race model with $a=0.9, b=2, c=1, d=2$ for which is not verified $ad > bc$

9 Diabetes mellitus model

Diabetes 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 \bar{G}, \bar{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:

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

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

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

$$g(t) = G(t) - \bar{G}$$

$$h(t) = H(t) - \bar{H}$$

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:

$$g'(t) = -m_1g - m_2h + \gamma_g(g, h) \quad (*)$$

$$h'(t) = m_3g - m_4h + \gamma_h(g, h)$$

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.

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

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

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 , $i = 1, \dots, m$, we can obtain the remaining parameters with the least squares method.

In other words, we look for the minimum of the following function:

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

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

```

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

```

Listing 37: 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 $[0, \text{maxt}]$.

Executing the following script, we obtain:

```

1 optimalg = diab(80,30,0.1,1.2,3.3,1.1,-20,10,1000);
2 badg = diab(140,30,0.1,2,0.125,0.1,-40,10,1000);
3 hold all;
4 plot(optimalg,'DisplayName','Normal glycemic level');
5 plot(badg,'DisplayName','Pathologic glycemic level');
6 legend(gca,'show','Location','northeast');

```

Listing 38: Diabetes Mellitus Script

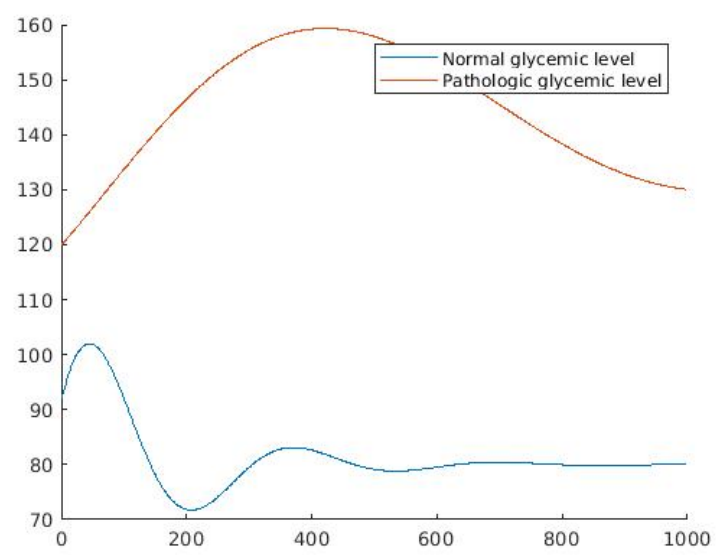


Figure 34: Diabetes mellitus plot for an healthy and a non healthy patient