Project 1

NUMN12 Numerical Methods for Solving Differential Equations

Felicia Segui and Ebba Rickard

November 2019

2 Explicit Adaptive Runge-Kutta Methods

2.1 Task 2.1

An ODE solver was created in Matlab using the Runge-kutta method of the fourth order. The method was used to approximate solutions to initial value problems on the form described in equation 1. Sampled stage derivatives given by equations 2 put in equation 3 gives the next approximated function value y_{n+1} [1].

$$y' = f(t, y) \tag{1}$$

$$\begin{aligned} Y_1' &= f(t_n, y_n) \\ Y_2' &= f(t_n + \frac{h}{2}, y_n + h\frac{Y_2'}{2}) \\ Y_3' &= f(t_n + \frac{h}{2}, y_n + h\frac{Y_2'}{2}) \\ Y_4' &= f(t_n + h, y_n + hY_3') \end{aligned} \tag{2}$$

$$y_{n+1} = y_n + \frac{h}{6}(Y_1' + 2Y_2' + 2Y_3' + Y_4') \tag{3}$$

The ODE solver function was called $RK4int$, which used the help function $RK4step$. $RK4int$ returns an approximated solution to a specified ODE like in equation 1, while $RK4step$ calculates the actual next function value using equation 3. The function scripts are presented in figure 11 and 12 in Appendix. The solver was tested by solving a simple linear testfunction described in equation 4, setting λ equal to -1. Figure 1 shows the result as well as the exact solution.

$$f(t, y) = \lambda y \tag{4}$$

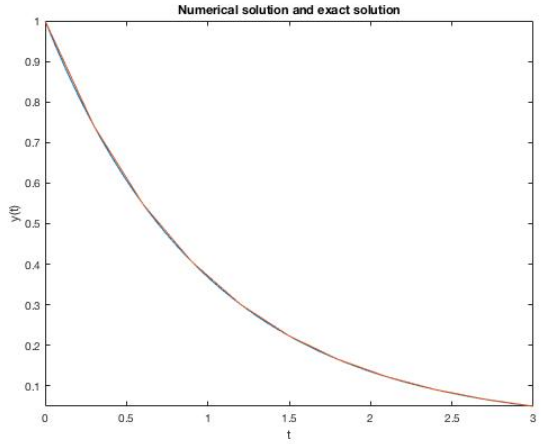


Figure 1: The numerical approximation in blue and the exact solution in red.

Figure 2 depicts how the global error grows bigger for a bigger step size. This is uniform with expectations, since a bigger step size means less approximated points on the interval and thus lower accuracy. The global error should theoretically be proportional to $O(h^4)$ which in a logarithmic scale translates to a slope of four. $O(h^4) + C$ is plotted next to the calculated global error and it can be observed that they indeed have the same slope. It is also observed that the error stops behaving in line with theory when the step size exceeds 1. On the time interval 0 to 10 that the error was evaluated this means less than 10 sample points which is not enough to approximate the function correctly.

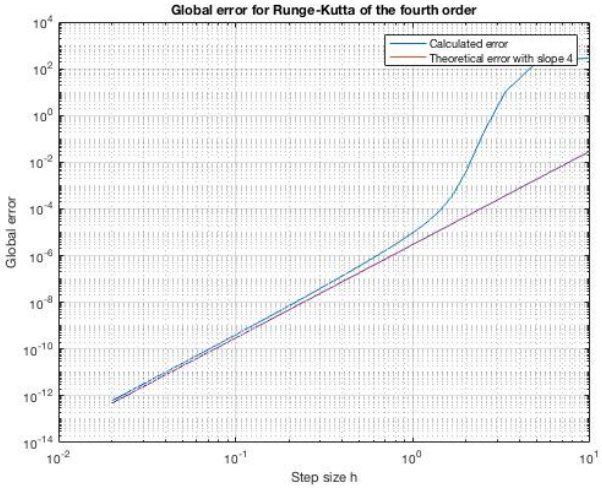


Figure 2: The global error logarithmically plotted as a function of step size.

2.2 Task 2.2-2.4

The embedded third and fourth order Runge-Kutta method, described by quation 5, 6 and 7, reuses the third order approximation when calculating the fourth order one, which means that only one extra stage derivative Z_3' needs to be calculated. It has the advantage that a local error can be estimated for each step with equation 8. The new function $RK34step$ was created in Matlab to take a step with RK34 as well as evaluate the local error. See figure 13 in appendix for the script.

$$\begin{aligned} Y_1' &= f(t_n, y_n) \\ Y_2' &= f(t_n + \frac{h}{2}, y_n + h\frac{Y_2'}{2}) \\ Y_3' &= f(t_n + \frac{h}{2}, y_n + h\frac{Y_2'}{2}) \\ Z_3' &= f(t_n + h, y_n - hY_1' + 2hY_2') \\ Y_4' &= f(t_n + h, y_n + hY_3') \end{aligned} \tag{5}$$

$$y_{n+1} = y_n + \frac{h}{6}(Y_1' + 2Y_2' + 2Y_3' + Y_4') \tag{6}$$

$$z_{n+1} = y_n + \frac{h}{6}(Y_1' + 4Y_2' + Z_3') \tag{7}$$

$$l_{n+1} = z_{n+1} - y_{n+1} \tag{8}$$

The local error can now be used to calculate a step size for each step that makes sure that the error does not exceed a certain tolerance. The formula used to calculate the new step sizes is in equation 9. This was done in Matlab by creating the function *newstep* which calculated the step size from a specified tolerance, the previous local errors and the previous step size. For $n = 1$ the error r_0 was put equal to the tolerance since it is the first error estimation. See figure 14 in appendix for the script.

$$h_n = \left(\frac{\text{TOL}}{r_n}\right)^{2/(3k)} \left(\frac{\text{TOL}}{r_{n-1}}\right)^{-1/(3k)} \cdot h_{n-1} \tag{9}$$

Using the functions $RK34step$ and *newstep* an adaptive ODE solver was created, called $adaptiveRK34$. See appendix for MatLab script. The solver was tested using the testfunction in equation 4 with $\lambda = -1$, see figure 3. Compared to figure 1, where the approximated and the exact solutions and the error can clearly be seen, it is in contrary hard to distinguish the two in figure 3. This is as expected, since the error was kept small through calculating a new step size in each iteration.

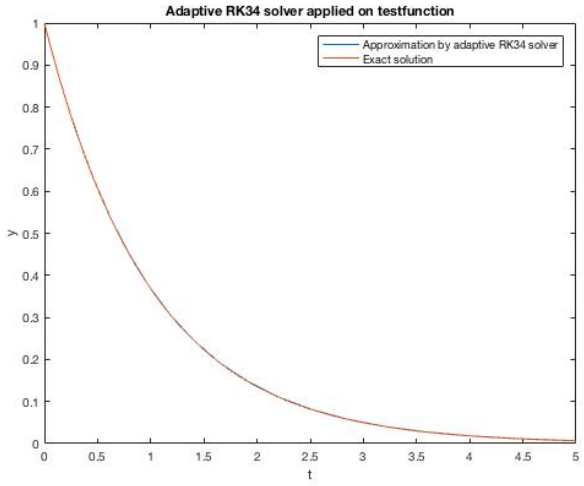


Figure 3: Approximation of the test function with the RK34 method compared to the exact solution.

3 A nonstiff problem

3.1 Task 3.1

After testing the *adaptiveRK*34 solver on simple ODE:s and making sure it performed well it was now used to solve other, more difficult ODE:s. The equation below is the Lotka Volterra equation which describes the interaction between prey and predator [1].

$$\begin{aligned} \dot{x} &= ax - bxy \\ \dot{y} &= cxy - dy \end{aligned} \tag{10}$$

This equation was written in a script in Matlab, called *lotka*, see figure 16 in Appendix. The equation was solved with a script called *lotkatest*1, see figure 17 in Appendix.

The solution to the Lotka Volterra equation is presented in figure 4. The periodicity for both predator and prey is shown in the plot, with a period time of 1 s for both $x(t)$ and $y(t)$. The periodicity is also confirmed by the closed loop in the phase portrait $y(x)$, see figure 5a. The script for the latter figure is presented in Appendix in figure 18. These two plots, figure 4 and figure 5a are solutions to the Lotka Volterra equation with initial value $u = [1 \;\; 1]$. Other initial values either gives the solution a different period, or makes the solution not periodic at all. With initial value $u = [1 \;\; 15]$, the phase portrait is not longer a closed loop and therefore not periodic, see figure 5b.

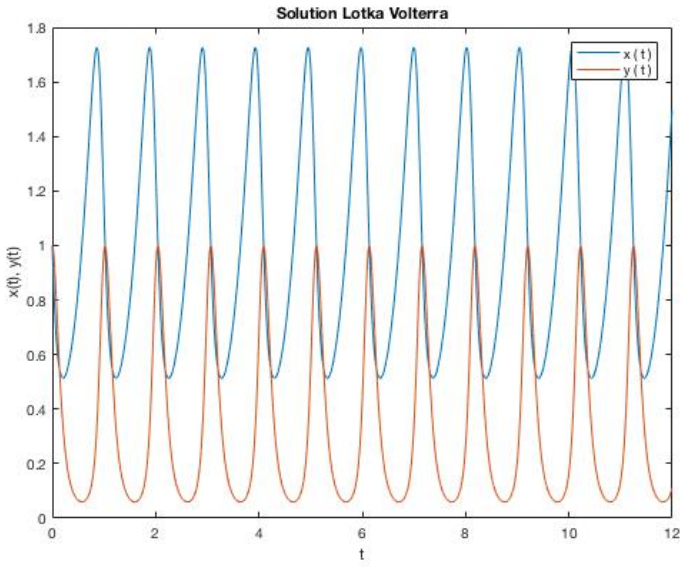
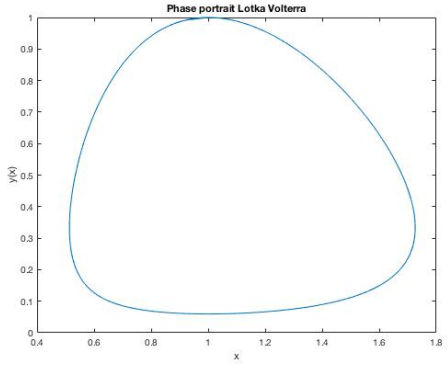
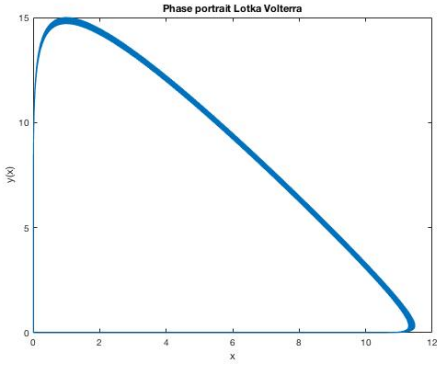


Figure 4: Prey (blue) and predator (red) ratio as a function of time.

With analytic calculations it is shown that the equation 11 should be constant for the Lotka Volterra equation, which as well as the phase portrait confirms the periodic solution[1].



(a) The phase portrait $y(x)$ for the Lotka Volterra equation with initial value $u = [1 \;\; 1]$.



(b) The phase portrait $y(x)$ for the Lotka Volterra equation with initial value $u = [1 \;\; 15]$.

$$H(x, y) = cx + by - dlog(x) - alog(y) \tag{11}$$

In figure 6, expression 12 is plotted as a function of time. With the big integration time it is shown that the value will be close to a constant, which confirms the conclusion from analytical calculations described earlier. This could indicate that our function *adaptive*34 is more accurate for a big integration time. The Matlab script is presented in Appendix, see figure 19.

$$\left| \frac{H(x, y)}{H(x(0), y(0))} - 1 \right| \tag{12}$$

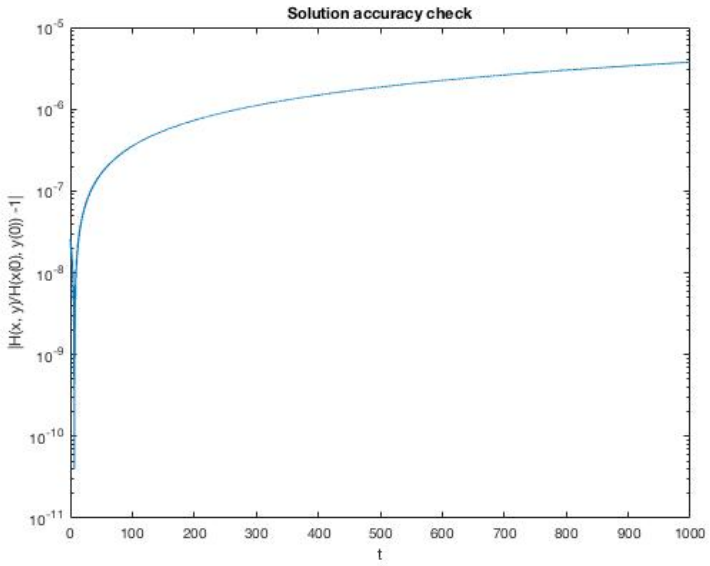


Figure 6: The expression 12 plotted as a function of time.

The expression 12 was plotted with the Matlab plot function *semilogy*, as it made it easy to see the convergence.

4 Nonstiff and stiff problems

Equation 13 is called the van der Pol equation and describes an electric oscillator circuit, which has a 2μ periodic solution. By changing μ, the solution can either be stiff or nonstiff [1].

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu \cdot (1 - y_1^2) \cdot y_2 - y_1 \end{aligned} \tag{13}$$

The van der Pol equation was written as a function in Matlab, called *vanderpol*, see figure 20 in Appendix, which is used in tasks 4.1-4.3.

4.1 Task 4.1 - 4.3

The solution to y_2 with initial conditions presented in equation 14 is shown in figure 7. In this task μ was set equal to 100, the tolerance was set to 10^{-8} and the ODE was evaluated on the interval [0, 2μ]. The differential equation was once again solved using the *adaptiveRK*34 function, see script in Appendix, figure 21.

$$y(0) = (2 \;\; 0)^T \tag{14}$$

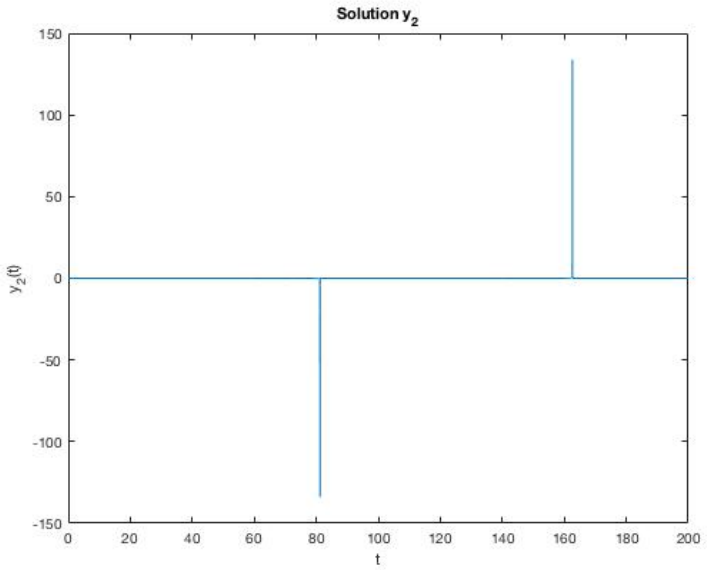
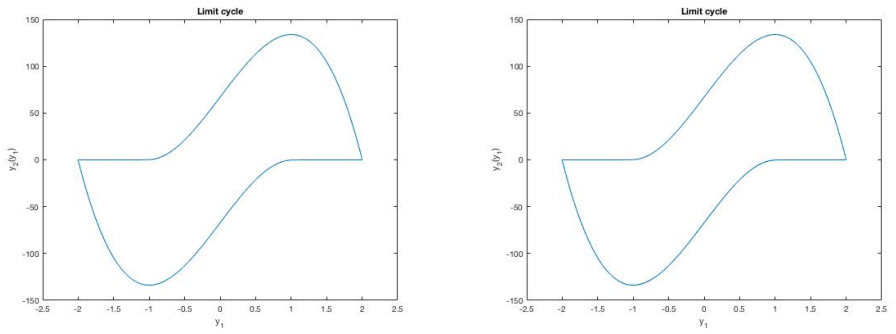


Figure 7: The solution y_2 as a function of time t.

The phase portrait is obtained by plotting y_2 as a function of y_1, and can also be called the limit cycle. In figure 8a and figure 8b the limit cycle for the initial conditions (14) and (15) are shown. Both figures contain a closed loop which confirms the periodicity. The script for these plots is presented in Appendix, figure 21.

$$y(0) = (1 \;\; 0)^T \tag{15}$$



(a) Initial condition (14). (b) Initial condition (15).

Figure 8: The limit cycles for the van der Pol equation with different initial conditions.

To investigate how the stiffness depends on μ one needs to know how big the step size can be without obtaining instability. This can be done by calculating the amount of steps N needed to approximate a solution with $adaptiveRK34$ for difference values of μ. In figure 9 the number of steps N as a function of μ is presented. The initial value (14) was used together with the time interval $[0,\ 0{,}7\mu]$, the tolerance 10^{-8} and the μ values presented in expression (16). This was calculated by creating function *vanderpoltest*, see figure 22 in Appendix, with a for loop which uses the van der Pol function, the $adaptiveRK34$ function and stores the number of steps in a vector N.

$$\mu = [10\ \ 15\ \ 22\ \ 33\ \ 47\ \ 68\ \ 100\ \ 150\ \ 220\ \ 330\ \ 470\ \ 680\ \ 1000] \tag{16}$$

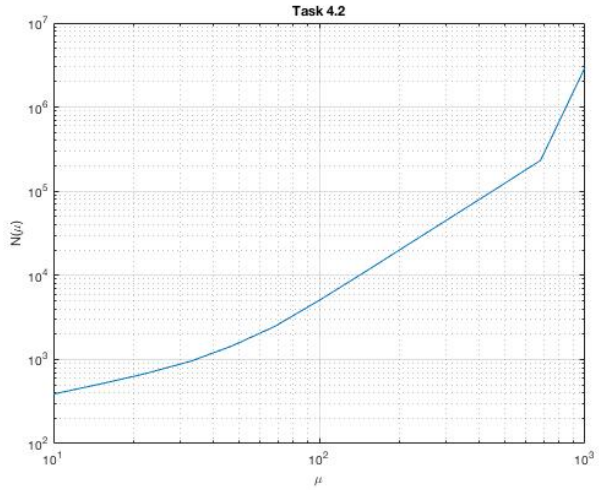
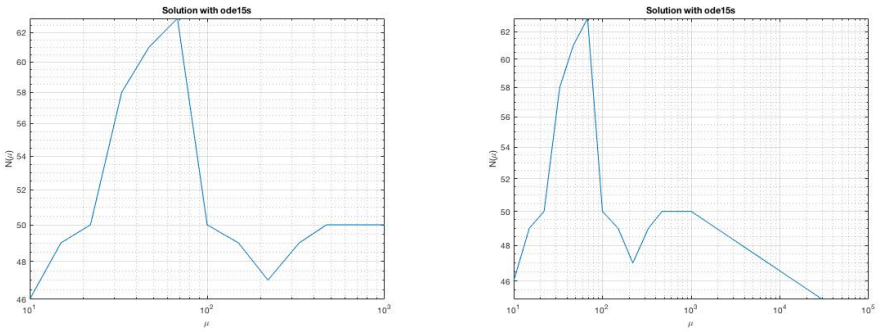


Figure 9: N as a function of μ.

In the plot 9 the slope has an approximate value of 2 in the middle part, which means that $N \propto C\mu^2$.

For small μ, the value of the slope is lower and for big μ the value of the slope increases. This means that the stiffness for this problem increases with a bigger μ, as a result of that the increase of N needed is proportional to the stiffness[1].

Ode15s is a Matlab solver for stiff differential equations with a tolerance value 10^{-6} [2]. A function with name $ode15stest$ was created, see figure 23 in Appendix, and the van der Pol equation was solved with $ode15s$. The number of steps N as a function of μ is presented in figure 10a and in figure 10b, with $\mu_{max} = 1000$ respectively $\mu_{max} = 30000$.



(a) N as a function of μ solved with ode15s, with $\mu_{max} = 1000$.

(b) N as a function of μ solved with ode15s, with $\mu_{max} = 30000$.

Figure 10

The value of N varies very little in plot 10a and 10b in comparison to the N value in plot 9. The low tolerance and N value makes the $ode15s$ solver much more preferable to use than the $adaptiveRK34$ solver, especially for similar problems with big μs. Because of the big N in plot 9 it takes long time to compile the script, even with $\mu_{max} = 1000$, which is not a problem for $ode15s$ since it has small and non increasing N values for big μs. As mentioned earlier, the ode15s is a solver for stiff problems which works good for the van der Pol differential equation, because of the "step" looking solution, see figure 7.

5 Acknowledgements and division of work

5.1 Acknowledgements

We want to acknowledge our instructor Peter Meisrimel for explaining and helping with the Matlab scripts.

5.2 Division of work

We both wrote our own code but helped eachother when we ran in to problems. We also compared all plots and scripts to ensure we obtained the same results and similar scripts.

References

[1] T Sillfjord, G Söderlind, Lund University, [Internet]; *Project 1 in FMNN10 and NUMN12*, 2019 [Citerad: 2019-11-13]. URL: https://canvas.education.lu.se/courses/902/files/118613/download?verifier=xGOM5Yw6Uru6pod70MESc3ZIhwxlp

[2] MathWorks, [Internet]; *ode15s*, 2019 [Citerad: 2019-11-15]. URL: https://se.mathworks.com/help/matlab/ref/ode15s.html

6 Appendix

6.1 RK4step.m

```
function unew = RK4Step(f,uold,told,h)
Yprime1 = f(told, uold);
Yprime2 = f(told + h/2, uold + h*Yprime1/2);
Yprime3 = f(told+h/2, uold + h*Yprime2/2);
Yprime4 = f(told+h, uold + h*Yprime3);

unew = uold + (Yprime1 + 2*Yprime2 + 2*Yprime3 + Yprime4)*h/6;

end
```

Figure 11: Function script for RK4step.

6.2 RK4int.m

```
function [approxvalues,timevect] = RK4int(f, y0, t0, t1, N)

h=(t1-t0)/N;

timevect = t0;
approxvalues = y0;
uold = y0;

for n = 1:N
uold=RK4Step(f,uold,t0,h);
approxvalues(n+1)=uold;
timevect(n+1) = h*n;
end
end
```

Figure 12: Function script for RK4int.

6.3 RK34step.m

```
function [unew,err] = RK34step(f,uold,told,h)
Yprime1 = f(told, uold);
Yprime2 = f(told + h/2, uold + h.*Yprime1./2);
Yprime3 = f(told + h/2, uold + h.*Yprime2./2);
Zprime3 = f(told + h, uold - h.*Yprime1 + 2.*h.*Yprime2);
Yprime4 = f(told + h, uold + h.*Yprime3);

unew = uold + (Yprime1 + 2.*Yprime2 + 2.*Yprime3 + Yprime4).*h./6;
err = (2.*Yprime2 + Zprime3 -2.*Yprime3-Yprime4).*h./6;
err = norm(err);

end
```

Figure 13: Function script for RK34step.

6.4 newstep.m

```
function hnew = newstep(tol,err,errold,hold,k)
hnew = (tol/err)^(2/(3*k))*(tol/errold)^(-1/(3*k))*hold;
end
```

Figure 14: Function script for newstep.

6.5 adaptiveRK34.m

```
function [t,y] = adaptiveRK34(f,y0,t0,tf,tol)
y = y0;
t = t0;
err = tol;
h = (abs(tf-t0)*tol^(1/4))/(100*(1+norm(f(0,y0))));
x = h;
k=4;

[ynew,errnew] = RK34step(f,y(:,end),t(end),h); % One step with this h value
y = [y ynew];
err = [err errnew];
x = t(end) + h;
t = [t x];
cond =1;
while cond>0
    h = newstep(tol,err(end),err(end-1),h,k);
    x = t(end) + h;
    if x>tf
        cond = -1;
    else
        [ynew,errnew] = RK34step(f,y(:,end),t(end),h);
        y = [y ynew];
        err = [err errnew];
        t = [t x];
    end
end
% How much is left to the tf = h
x = x-h; % Back one step
h = tf-x;
x = t(end) + h;

[ynew,errnew] = RK34step(f,y(:,end),t(end),h);
y(:,end) = ynew;
err(:,end)=errnew;
t(:,end)=x;
end
```

Figure 15: Function script for adaptiveRK34.

6.6 lotka.m

```
function dudt = lotka(t,u)
a=3;
b=9;
c=15;
d=15;
dudt = [a.*u(1)-b.*u(1).*u(2); c.*u(1).*u(2)-d.*u(2)];
end
```

Figure 16: Function script for the Lotka Volterra equation.

6.7 lotkatest1.m

```
clc;
clear all
f = @lotka;
y0 = [1 ; 1];
t0 = 0;
tf = 12;
tol = 10^(-8);
adaptiveRK34(f, y0, t0, tf, tol);
```

Figure 17: Script for solving Lotka Volterra equation.

6.8 lotkatest2.m

```
clear all
f = @lotka;
y0 = [1 ; 1];
t0 = 0;
tf = 1000;
tol = 10^(-8);
[t, y] = adaptiveRK34(f, y0, t0, tf, tol);
plot(y(1,:),y(2,:));
```

Figure 18: Script for solving Lotka Volterra phase portrait.

6.9 lotkaaccuracy.m

```
f = @lotka;
y0 = [1 ; 1];
t0 = 0;
tf = 1000;
tol = 10^(-8);
[t, y] = adaptiveRK34(f, y0, t0, tf, tol);
a=3;
b=9;
c=15;
d=15;
semilogy(t, abs((c.*y(1,:)+b.*y(2,:)-d.*log(y(1,:))-a.*log(y(2,:))./(c+b)-1));
```

Figure 19: Script for solving |H(x,y)/H(x(0),y(0)) -1| for the Lotka Volterra equation.

6.10 vanderpol.m

```
function v = vanderpol(t, u)
mu =100;
v = [u(2); mu.*(1-u(1).^2).*u(2)-u(1)];
end
```

Figure 20: Function script for the van der Pol differential equation.

6.11 vanderpol41.m

```
clc;

f = @vanderpol;
y0 = [2 ; 0]; % start value
t0 = 0;
mu =100;
tf = 2*mu;
tol = 10^(-8);

[t, y] = adaptiveRK34(f, y0, t0, tf, tol);

%plot(t,y(2,:)); %y2(t)
%title('Solution y_2'); %y2(t)
%xlabel('t'); %y2(t)
%ylabel('y_2(t)'); %y2(t)
plot(y(1,:),y(2,:)); %Limit cycle
title('Limit cycle'); %Limit cycle
xlabel('y_1'); %Limit cycle
ylabel('y_2(y_1)'); %Limit cycle
```

Figure 21: Script for solving van der Pol equation and its phase portrait.

6.12 vanderpoltest.m

```
clc;
y0 = [2 ; 0];
t0 = 0;
tol = 10^(-8);
N = 0;
mu =[10, 15, 22, 33, 47, 68, 100, 150, 220, 330, 470, 680, 1000];
tf = 0.7.*mu;

for i = 1:length(mu)
f = @(t,u) [u(2); mu(i).*(1-u(1).^2).*u(2)-u(1)]; % van der Pol
[t, y] = adaptiveRK34(f, y0, t0, tf(i), tol);
N(i) = length(t);
end

loglog(mu, N);
grid on
title ('Task 4.2');
xlabel('\mu');
ylabel('N(\mu)');
```

Figure 22: Function script for plotting number of steps as a function of μ for the van der Pol equation.

6.13 ode15stest.m

```
clc;
y0 = [2 ; 0];
t0 = 0;
N = 0;
mu =[10, 15, 22, 33, 47, 68, 100, 150, 220, 330, 470, 680, 1000, 30000];
tf = 0.7.*mu;

for i = 1:length(mu)
f = @(t,u) [u(2); mu(i).*(1-u(1).^2).*u(2)-u(1)]; % van der Pol
[t, y] = ode15s(f,[t0, tf(i)],y0);
N(i) = length(t);
end

loglog(mu, N);
```

Figure 23: Function script for solving van der Pol with ode15s.