Applied Numerical Methods - Computer Lab 2

GOYENS Florentin & WEICKER David

1st October 2015

Part A. Accuracy of a Runge-Kutta method

We want to determine the order of the following Runge-Kutta method using a numerical experiment.

$$u_k = u_{k-1} + \frac{h}{6}(k_1 + k_2 + 4k_3),\ t_k = t_{k-1} + h,\ k = 1, 2, \dots, N.$$

$$\begin{aligned} k_1 &= f(t_{k-1}, u_{k-1}) \\ k_2 &= f(t_{k-1} + h, u_{k-1} + hk_1) \\ k_3 &= f(t_{k-1} + h/2, u_{k-1} + hk_1/4 + hk_2/4). \end{aligned}$$

We solve the Van der Pol's equation for an increasing number of time steps on $t \in [0, 1]$. The number of steps follows a geometric progression, $N = 10, 20, 40, 80, 160, 320$. The error is estimated at $t = 1$. Define $e_N = y_N - y(1)$, $y(1)$ is estimated with our best approximation. That is y_{Nmax}, the most precise numerical value. In our case $Nmax = 320$.

The observed error can be seen on figure 1. It is clear from the plot that we deal with a third order method. For a step size multiplied by 10, the error is multiplied by 1000.

j'ai fait en fonction de h comme ils demandaient, t'avais fait en N. Nettoyer code

./a1-eps-converted-to.pdf

Figure 1: Loglog plot for empirical error of Runge-Kutta method on Van der Pol's equation

We recognized the classical third order Runge-Kutta method and the order can be obtained analytically. This experiment confirms very well the results on a practical example. Here below is the Matlab code we used to estimate the order of the method.

```
function [] = LAB2A()
 % Function for LAB2 question A.
% Authors: David Weicker and Florentin Goyens
close all;

% N = [10 20 40 80 160 320]; for N
N = [320 160 80 40 20 10];
value = zeros(1,length(N));
init = [1 0]'; %initial conditions

% Solve equation
for j = 1:length(N)
    n = N(j);
    u = [init zeros(2,n)];
    h = 1/n;
    for i=1:n
```

```
        k1 = vanderpol(i*h,u(:,i));
        k2 = vanderpol((i+1)*h,u(:,i)+h*k1);
        k3 = vanderpol((i+0.5)*h,u(:,i)+h*k1./4+h*k2./4);
        u(:,i+1) = u(:,i)+h*(k1+k2+4*k3)./6;
    end
    value(j) = u(1,end);
end

% Plot error to estimate the order of the method
% error = abs(value-value(end)); for N
error = abs(value-value(1));% for h
% loglog(N,error,'b-',N,error,'r.','MarkerSize',15);% for N
loglog(1./N,error,'b-',1./N,error,'r.','MarkerSize',15);% for h
title('Evolution of error with step size');
xlabel('Step size h');
ylabel('Estimated error at t=1');
end

function dudt = vanderpol(t,u)
%System of ODE's for Van Der Pol's equation
epsilon = 1;
dudt = u;
dudt(1) = u(2);
dudt(2) = -u(1) - epsilon*(u(1)*u(1)-1)*u(2);
end
```

Part B. Stability investigation of a Runge-Kutta method

In this section, we will investigate the stabilty of a Runge-Kutta method. We will study a stiff problem, the Robertson's equations. As a remainder, here is the problem :

$$\mathbf{x}' = \begin{pmatrix} -k_1x_1 + k_2x_2x_3 \\ k_1x_1 - k_2x_2x_3 - k_3x_2^2 \\ k_3x_2^2 \end{pmatrix}$$

$$\mathbf{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

The problem is stiff because of the practical values of the parameters. In this example, we have $k_1 = 0.04$, $k_2 = 10^4$ and $k_3 = 3 \cdot 10^7$.

B1. Constant stepsize experiment

We are going to solve the Robertson's problem first with a constant stepsize. The Matlab code can be found at the end of this section. We had to try and see if the method was stable for this problem using different numbers of steps, namely $N = [125, 250, 500, 1000, 2000]$. The answer given by our code was that it was not stable for $N = [125, 250, 500]$. Thus, the smallest number of steps (from the 5 given) needed to obtain a stable solution is $n = 1000$. The solution for the smallest step size ($n = 2000$) is given in figure 2.

```
function [] = LAB2B1()
%LAB2B
close all;

N = [125 250 500 1000 2000];
init = [1 0 0]'; %initial conditions
for j = 1:length(N)
```

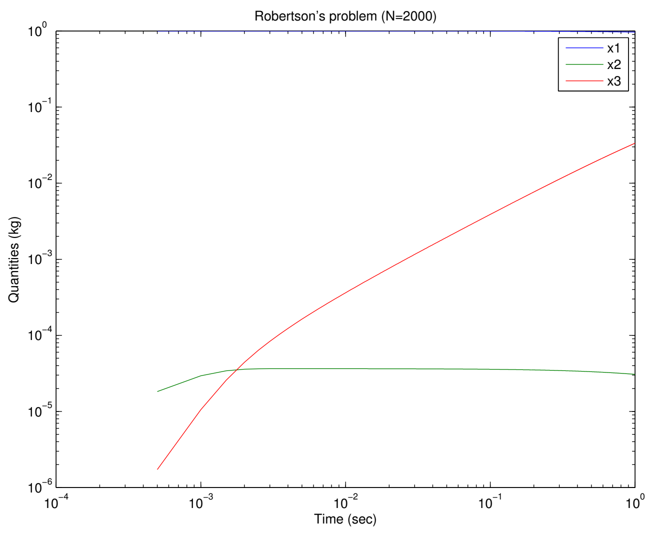

Figure 2: Robertson's problem with $n = 2000$

```
    n = N(j);
    u = [init zeros(3,n)];
    h = 1/n;
    for i=1:n %Runge-Kutta
        k1 = robertson(i*h,u(:,i));
        k2 = robertson((i+1)*h,u(:,i)+h*k1);
        k3 = robertson((i+0.5)*h,u(:,i)+h*k1./4+h*k2./4);
        u(:,i+1) = u(:,i)+h*(k1+k2+4*k3)./6;
    end
    str = sprintf('The last value for n = %d is x = %f,%f,%f',n,u(1,n),u(2,n),u(3,n));↩
        %stable or not?
    display(str);
end
figure;
loglog(0:h:1,u);title('Robertson''s problem (N=2000)');xlabel('Time (sec)');ylabel('↩
    Quantities (kg)');legend('x1','x2','x3');
end


function dxdt = robertson(t,x)
k1 = 0.04;
k2 = 1e4;
k3 = 3e7;
dxdt = x;
dxdt(1) = -k1*x(1) + k2*x(2)*x(3);
dxdt(2) = k1*x(1) -k2*x(2)*x(3) -k3*x(2)*x(2);
dxdt(3) = k3*x(2)*x(2);
end
```

B2. Adaptative stepsize experiment using Matlab functions

In this section, we are going to use the built-in IVP-solvers. As usual, the code can be found at the end of the section.

There are many different solvers available. Our first task was to use the non-stiff solver *ode*23 for different relative tolerances (using *odeset*) and for a time span of one second. Then, we were to plot a graph of the step size as a function of time for one of the tolerance. The following table gives the number of steps for each relative tolerance.

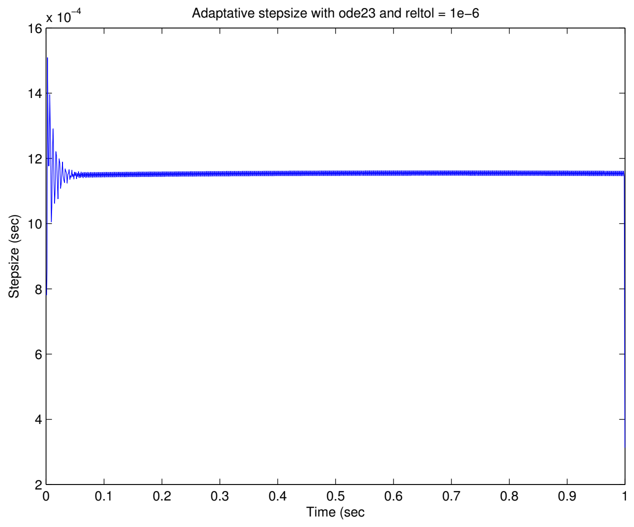

Figure 3: Adaptative stepsize with $ode23$ for $RelTol = 10^{-6}$

RelTol	**Number of steps**
10^{-3}	866
10^{-4}	867
10^{-5}	868
10^{-6}	868

We can see that the number of step increases but not significantly as the relative tolerance constraint increases. The plot of the stepsize as a function of time for the $RelTol = 10^{-6}$ can be seen in figure 3.

Even if we can see some oscillations in the beginning, the stepsize appear to stay around the same value. That is hardly adaptative.

Our second task was to use a stiff solver, namely $ode23s$, and do the experiment again but for a larger time span, from 0 to 1000 seconds. We obtained the following numbers of step for the given tolerances.

RelTol	**Number of steps**
10^{-3}	30
10^{-4}	37
10^{-5}	48
10^{-6}	62

First, we can noticed that even though the t-interval is a thousand time longer, the number of steps taken by $ode23s$ is lesser. Second, now the number of steps increases as the relative tolerance constraint increases. That is a good thing because it means that the stepsize is governed by the relative tolerance and not some other factor. We were also asked to plot the stepsize as a function of time for $ode23s$. We choose the same relative tolerance that the one used before, that is $RelTol = 10^{-6}$ This is shown in figure 4.

We can see that the value of the stepsize is way larger than for $ode23$. This is an example of why a stiff solver is better when confronting a stiff problem.

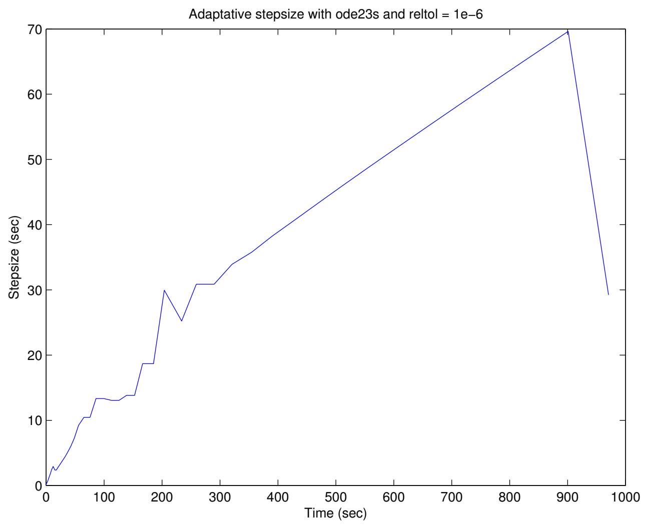

Figure 4: Adaptative stepsize with $ode23s$ for $RelTol = 10^{-6}$

```
function [] = LAB2B2()
%LAB2B2
close all;

tol = [1e-3 1e-4 1e-5 1e-6];
numbStep = zeros(1,4);
numbStiff = zeros(1,4);
for i=1:4
    reltol = tol(i);
    options = odeset('RelTol',reltol);%set relative tolerance
    [t,~] = ode23(@robertson,[0 1],[1 0 0]',options);
    [tstiff,~] = ode23s(@robertson,[0 1000],[1 0 0]',options);
    numbStep(i) = length(t)-1;
    numbStiff(i) = length(tstiff)-1;
end
display(numbStep);
display(numbStiff);

h = diff(t);
hstiff = diff(tstiff);

figure;
plot(t(1:end-1),h);title('Adaptative stepsize with ode23 and reltol = 1e-6');xlabel('↩
    Time (sec'); ylabel('Stepsize (sec)');
figure;
plot(tstiff(1:end-1),hstiff);title('Adaptative stepsize with ode23s and reltol = 1e-6'↩
    );xlabel('Time (sec)'); ylabel('Stepsize (sec)');
end

function dxdt = robertson(t,x)
k1 = 0.04;
k2 = 1e4;
k3 = 3e7;
dxdt = x;
dxdt(1) = -k1*x(1) + k2*x(2)*x(3);
dxdt(2) = k1*x(1) -k2*x(2)*x(3) -k3*x(2)*x(2);
dxdt(3) = k3*x(2)*x(2);
end
```

Part C. Parameter study of solutions of an ODE-system

For the first problem, we look at the position of a flow particle with a cylinder as obstacle. The results are displayed in the following way. On figure 5 one can see the trajectory for different initial positions. The particle never touches the cylinder but they get ever closer as the initial y-position gets close to zero.

`./c1-eps-converted-to.pdf`

Figure 5: Trajectories of the flow in the plane

The second problems is the motion of a particle thrown in the plane. There are two parameters that come into play. The launch angle α and k the air resistance coefficient. We considered $\alpha = 30, 45, 60$ degrees and $k = 0.002, 0.065$. For this problem we have one figure for each value of k. Each figure shows the motion of the particle for the different launch angles. We see that the launch angle that gives the largest horizontal travelled distance depends on the coefficient k. In the options of the Matlab solver `ode45`, we specified that the solution y can not become negative (as the ground prevents it).

y positif comme on fait c'est ça qu'ils veulent ? voir remarque dans enonce.

./c2-eps-converted-to.pdf

Figure 6: Trajectories of the particle in the plane

Below the reader can find our codes for the two problems.

```
function [] = LAB2C1()
% Function for LAB2 question C1.
% Authors: David Weicker and Florentin Goyens
close all;

% Resolution of the ode
[~,x1] = ode45(@flow,[0 10],[-4 0.2]');
[~,x2] = ode45(@flow,[0 10],[-4 0.6]');
[~,x3] = ode45(@flow,[0 10],[-4 1]');
[~,x4] = ode45(@flow,[0 10],[-4 1.6]');

% Plot of the trajectories and the cylinder
plot(x1(:,1),x1(:,2),x2(:,1),x2(:,2),x3(:,1),x3(:,2),x4(:,1),x4(:,2),2*sin(0:0.1:2*pi)↩
    ,2*cos(0:0.1:2*pi),'k--');axis equal;
title('Flow of four particles');
legend('y(0)=0.2','y(0)=0.6','y(0)=1.0','y(0)=1.6');
xlabel('Direction x');
ylabel('Direction y');
```

```
end

function dxdt = flow(t,x)
R = 2;
dxdt = x;
square = (x(1)*x(1)+x(2)*x(2))*(x(1)*x(1)+x(2)*x(2));
dxdt(1) = 1 - R*R*(x(1)*x(1)-x(2)*x(2))/square;
dxdt(2) = -2*R*R*x(1)*x(2)/square;
end
```

```
function [] = LAB2C2()
% Function for LAB2 question C2.
% Authors: David Weicker and Florentin Goyens
close all;
a = [pi/6 pi/4 pi/3];
k = [0.02 0.065];
options = odeset('NonNegative',3); % ensures that y stays non negative
result = cell(3,2);

for j=1:2
    for i=1:3
    init = [0 20*cos(a(i)) 1.5 20*sin(a(i))]';
    [~,y] = ode45(@(t,x) throw(t,x,k(j)),[0 3],init,options);% quid du @(tx,) ?
    result{i,1} = y(:,1);
    result{i,2} = y(:,3);
    end
    subplot(2,1,j);
    plot(result{1,1},result{1,2},result{2,1},result{2,2},result{3,1},result{3,2});
    string = sprintf('Motion for k = %f',k(j));
    title(string);
    legend('alpha = 30','alpha = 45','alpha = 60');
    xlabel('Distance (m)');
    ylabel('Height (m)');
end
end

function dxdt = throw(t,x,k)
square = sqrt(x(2)*x(2)+x(4)*x(4));
dxdt = x;
dxdt(1) = x(2);
dxdt(2) = -k*x(2)*square;
dxdt(3) = x(4);
dxdt(4) = -9.81-k*abs(x(4))*square;
end
```