Applied Numerical Methods - Computer Lab 2

Goyens Florentin & Weicker David

1st October 2015

Part A. Accuracy of a Runge-Kutta method

bla bla bla

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 :

$$\mathbf{x}' = \begin{pmatrix} -k_1 x_1 + k_2 x_2 x_3 \\ k_1 x_1 - k_2 x_2 x_3 - k_3 x_2^2 \\ k_3 x_2^2 \end{pmatrix}$$

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

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 1.

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

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

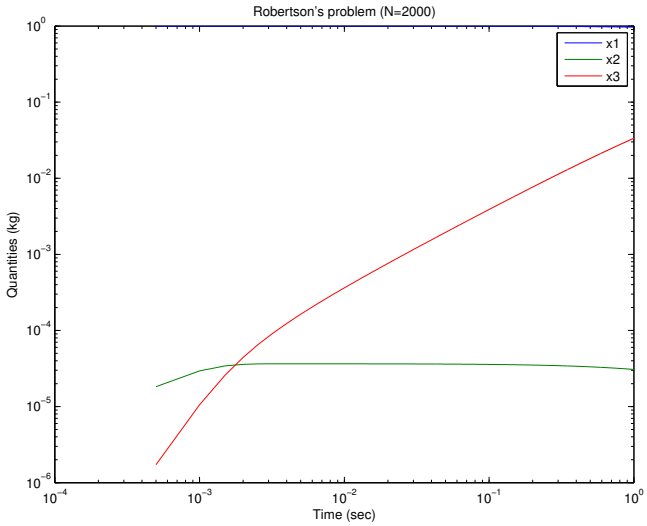

Figure 1: Robertson's problem with $n = 2000$

```
    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.

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

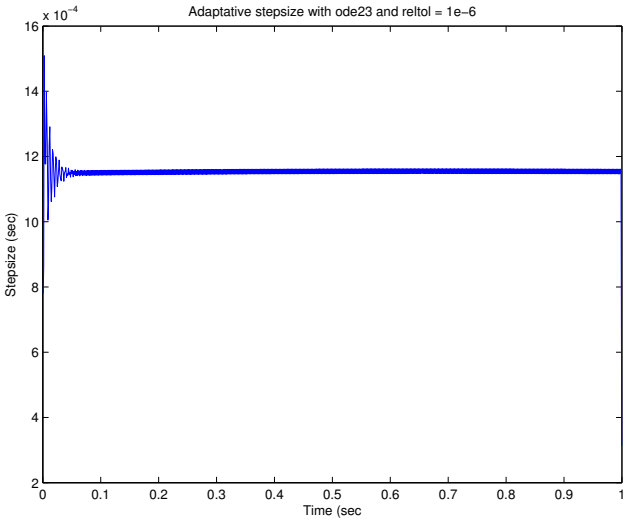

Figure 2: Adaptative stepsize with *ode*23 for $RelTol = 10^{-6}$

constraint increases. The plot of the stepsize as a function of time for the $RelTol = 10^{-6}$ can be seen in figure 2.

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 *ode*23*s*, 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 *ode*23*s* 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 *ode*23*s*. We choose the same relative tolerance that the one used before, that is $RelTol = 10^{-6}$ This is shown in figure 3.

We can see that the value of the stepsize is way larger than for *ode*23. This is an example of why a stiff solver is better when confronting a stiff problem.

```
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
```

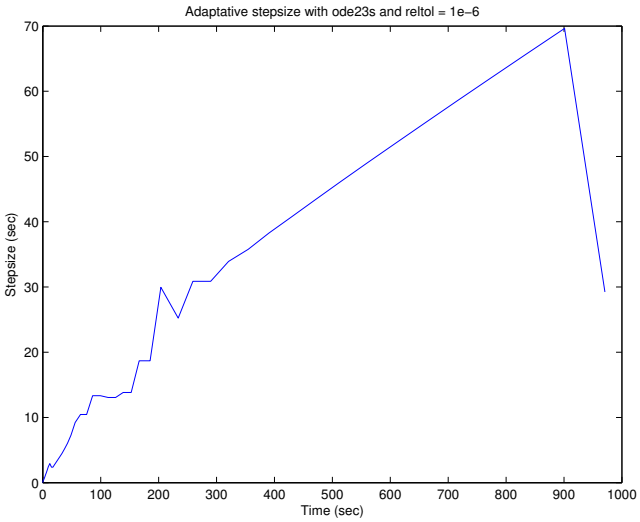

Figure 3: Adaptative stepsize with $ode23s$ for $RelTol = 10^{-6}$

```
    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

blo blo blo