McGill University

Numerical Solutions to Ordinary Differential Equations

MECH 309 Projects in Numerical Methods

Bao, Weijie 260360432
Harizaj, Dritan 260426327
April 23, 2012

Table of Contents

1. One-dimensional Initial Value Problem

1.1 Autonomous Ordinary Differential Equation

Consider the initial value problem:

$$\begin{cases} \dfrac{dy}{dt} = \cos y \quad defined\ on\ D \\ y(0) = -1 \end{cases} \tag{1}$$

where $\{(t, y) | 0 \le t \le 10; -\infty < y < \infty\}$.

1.1.1 Prove well-posed problem

According to Theorem 5.6 in the textbook:

" Suppose $D = \{(t, y) | a \le t \le b \text{ and } -\infty < y < \infty\}$. If f is continuous and satisfies a Lipschitz condition in the variable y on the set D, then the initial-value problem

$$\frac{dy}{dt} = f(t, y)\ , a \le t \le b, y(a) = \alpha$$

is well-posed."

Prove the existence and uniqueness of a solution y(t) on Domain D:

Holding t constant and apply the Mean Value Theorem to the function

$$f(y) = \frac{dy}{dt} = \cos y\ ,$$

We find that when $y_1 < y_2$, a number ξ in (y_1, y_2) exists with

$$\frac{f(y_2) - f(y_1)}{(y_2) - (y_1)} = \frac{\partial}{\partial y} f(\xi) = -\sin \xi.$$

Thus,

$$|f(y_2) - f(y_1)| = |y_2 - y_1||\sin(\xi)| \le 1|y_2 - y_1|,$$

And f satisfies a Lipschitz condition in the variable y with Lipschtz constant L=1. Also, f(y) is continuous when $0 \le t \le 10; -\infty < y < \infty$. Theorem 5.6 in the textbook implies that this is a well-posed problem.

1.1.2 MATLAB Algorithm to solve the ODE

In the section, the ODE is solved numerically by Euler's method and Runge-Kutta method of order 4 using MATLAB algorithm. Here's the algorithm to solve the ODE with h=1.

```
clc;
clear;
%f(t,y)=cosy;
a=0;%start time
b=10;%end time
j=0;
h=10^j;%time steps
N=(b-a)/h;% number of steps

%initial conditions
t=a;
y0=-1;
%initial condition for Euler's method
w=y0;
%initial condition for Runge-Kutta method
v=y0;

A=zeros(N,2);
A(1,1)=a;
A(1,2)=y0;

B=zeros(N,2);
B(1,1)=a;
B(1,2)=y0;
for i=1:1:N;
    w=w+h*cos(w);%euler

    K1=h*cos(v);%Runge-Kutta
    K2=h*cos(v+K1/2);
    K3=h*cos(v+K2/2);
    K4=h*cos(v+K3);
    v=v+(K1+2*K2+2*K3+K4)/6;

    t=a+i*h;

    %output of euler
    A(i+1,1)=t;
    A(i+1,2)=w;

    %output of runge-kutta
    B(i+1,1)=t;
    B(i+1,2)=v;

end
%plot
%plot vector field
[T,Y]=meshgrid(0:.4:10,-3:.24:3);
dY=cos(Y);
dT=ones(size(dY));
dYu=dY./sqrt(dT.^2+dY.^2);
```

```
dTu=dT./sqrt(dT.^2+dY.^2);
quiver(T,Y,dTu,dYu,'k');
hold on
plot(A(:,1),A(:,2),'-ob'); %plot euler method
plot(B(:,1),B(:,2),'-*r'); %plot runge-kutta
hold off
xlabel('t');
ylabel('y');
```

Plots generated by running the algorithm above:

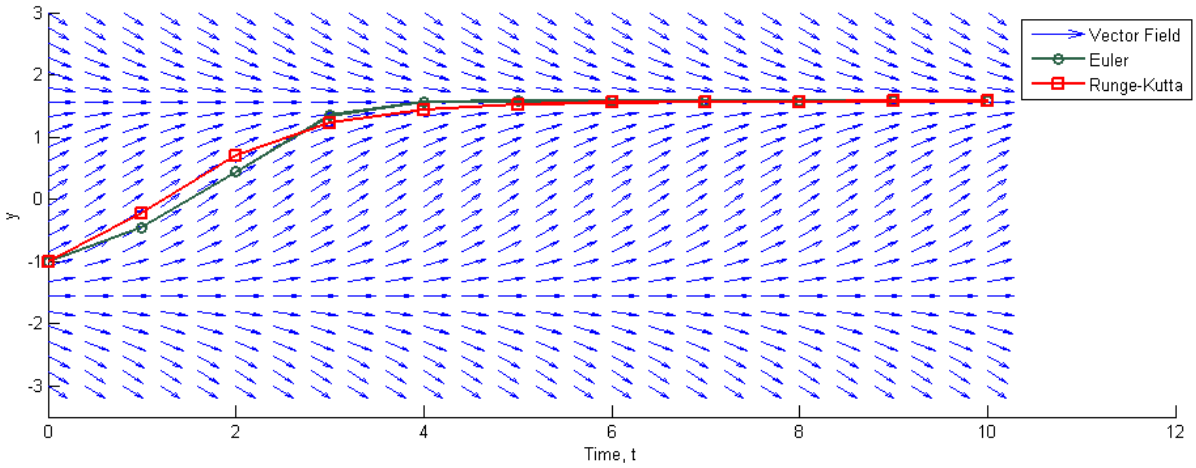


Figure 1. Solution of the ODE by Euler's Method and Runge-Kutta Method in the vector field

Provide an upper bound of the global truncation error.

According to Theorem 5.9 in the textbook, global truncation error is

$$|y(t_i) - w_i| \leq \frac{hM}{2L}\left[e^{L(t_i-a)} - 1\right]; \qquad where\ M \geq |y''(t)|, for\ all\ t\ \epsilon\ [a,b]$$

In order to find the second derivative of y(t), usually we need to figure out the closed solution of y is. This condition often prohibits us from obtaining a realistic error bound, but the chain rule for partial differentiation also works to find out the second derivative of the solution:

$$y''(t) = \frac{dy'}{dt}(t) = f'(t,y) = \frac{\partial f}{\partial t}(t,y(t)) + \frac{\partial f}{\partial y}(t,y(t)) \cdot y'(t)$$

In this case f=y'=cosy, so

$$y''(t) = \frac{\partial}{\partial t}(\cos y) + \frac{\partial}{\partial y}(\cos y) \cdot \cos y = -\sin y \cdot \cos y = -0.5\sin(2y)$$

Maximum absolute value of y''(t) would be ½ , so M=1/2. This is the upper-bound error using approximation of y''(t).

1.1.3 Local and global truncation errors with different time steps

Find the closed-form solution to this ODE.

$$\frac{dy}{dt} = \cos y$$

$$\int \sec y \; dy = \int dt$$

$$\ln(\tan y + \sec y) = t + C$$

Solving for y:

$$\mathrm{y(t)} = 2\tan^{-1}(\tanh(\frac{1}{2}(t + C)))$$

Apply the initial condition, y(0)=-1,

$$\mathrm{y(t)} = 2\tan^{-1}\left\{\tanh\left[\frac{1}{2}\left(\mathrm{t} - 2\tanh^{-1}\left(\tan\frac{1}{2}\right)\right)\right]\right\} \tag{2}$$

In the MATLAB code below, we are asked to calculate Euler's and Runge-Kutta method with decreasing time-steps. Also the local and global truncation errors are shown in log-log scale plot. Local truncation error measures how well the continuous equation has been approximated by the discrete difference equation, and the global discretization measures how well the true solution has been approximated.

Global truncation error:

$$|\mathrm{y}_{(\mathrm{ti})} - w_i|$$

where y(ti) is the exact solution from Equation (2) above, and w_i is the approximated solution by either Euler's or Runge-Kutta's method.

Local Truncation error:

As shown in Definition 5.11 and chapter 5.5 in the textbook:

$$\tau_{i+1}(h) = \frac{y(t_{i+1}) - y(t_i)}{h} - \mathrm{f}(t_i, y(t_i), h) = \frac{y(t_{i+1}) - w(t_i)}{h} - \mathrm{f}(t_i, w(t_i), h)$$

$$= \frac{y(t_{i+1}) - [w(t_i) + h\mathrm{f}(t_i, y(t_i), h)]}{h}$$

$$= \frac{y(t_{i+1}) - w(t_{i+1})}{h}$$

```
clc;
clear;
a=0;%start time
b=10;%end time
format long;
max_error=zeros(6,5);
f=@(t,y)cos(y);
y_exact=@(t)2*atan(tanh(0.5*(t-2*atanh(tan(0.5))))); %The exact solution
%
for j=0:1:4
    h=10^-j;
    N=(b-a)/h;
    y0=-1;
    t=zeros(N,1);
    %initial values
    t(1)=a;
    w=y0;
    v=y0;

    A=zeros(N,8);
    A(1,1)=a;
    A(1,2)=w;
    A(1,3)=y0;

    A(1,4)=abs(y0-w);
    A(1,5)=((y0-w)/h);%local truncation error of euler method

    A(1,6)=v;
    A(1,7)=abs(y0-v);%global truncation error of euler method
    A(1,8)=((y0-v)/h);%local truncation error of euler method

    p_w=w;
    p_v=v;

    for i=2:1:N
        %numerical solution after each steps
        w=p_w+h*f(t(i-1),p_w);

        %global truncation error of euler method
        A(i,4)=abs(y_exact(t(i-1))-p_w);

        A(i,5)=(y_exact(t(i-1))-p_w)/h;
        %A(i+1,5)=((y-p_w)/h-cos(p_w));%local truncation error of euler
method


        K1=h*f(t(i-1),p_v);%Runge-Kutta
        K2=h*f((t(i-1)+h/2),(p_v+K1/2));
        K3=h*f((t(i-1)+h/2),(p_v+K2/2));
        K4=h*f((t(i-1)+h),(p_v+K3));
        phi=(K1+2*K2+2*K3+K4)/(6*h);
        v=p_v+h*phi;


        t(i)=a+i*h;
```

```
        A(i,1)=t(i);
        A(i,2)=p_w;
        A(i,3)=y_exact(t(i-1)); %exact value

        %column 4,5,7,8 of matrix A are local and trucation errors of
two
        %different method
        %Global is defined as the difference between approximation
solution
        %and exact solution


        A(i,6)=p_v;
        A(i,7)=abs(y_exact(t(i-1))-p_v);%global truncation error of
runge-kutta method


        %A(i+1,8)=((y-p_v)/h-phi);%local truncation error of runge-kutta
method
        A(i,8)=(y_exact(t(i-1))-p_v)/h;
        %p_y=y;
        p_w=w;
        p_v=v;

    end

    max_error(j+1,1)= h;
    max_error(j+1,2)= max(A(:,4)); %global euler error
    max_error(j+1,3)= max(A(:,5)); %local truncation error--euler
    max_error(j+1,4)= max(A(:,7)); %global runge-kutta error
    max_error(j+1,5)= max(A(:,8)); %local truncation error--runge-kutta

end
%A
%max_error
loglog(max_error(:,1),max_error(:,2),max_error(:,1),max_error(:,3),max_e
rror(:,1),max_error(:,4),max_error(:,1),max_error(:,5));
hleg1=legend('Euler-Global Error','Euler-Local Error','Runge-Kutta-
Global Error','Runge-Kutta-Local Error');
set(hleg1,'Location','NorthEast')
set(hleg1,'Interpreter','none')
xlabel('time-steps h');
ylabel('Maximum Truncation Errors from t=0 to t=10');
set(gca,'XDir','reverse');
```

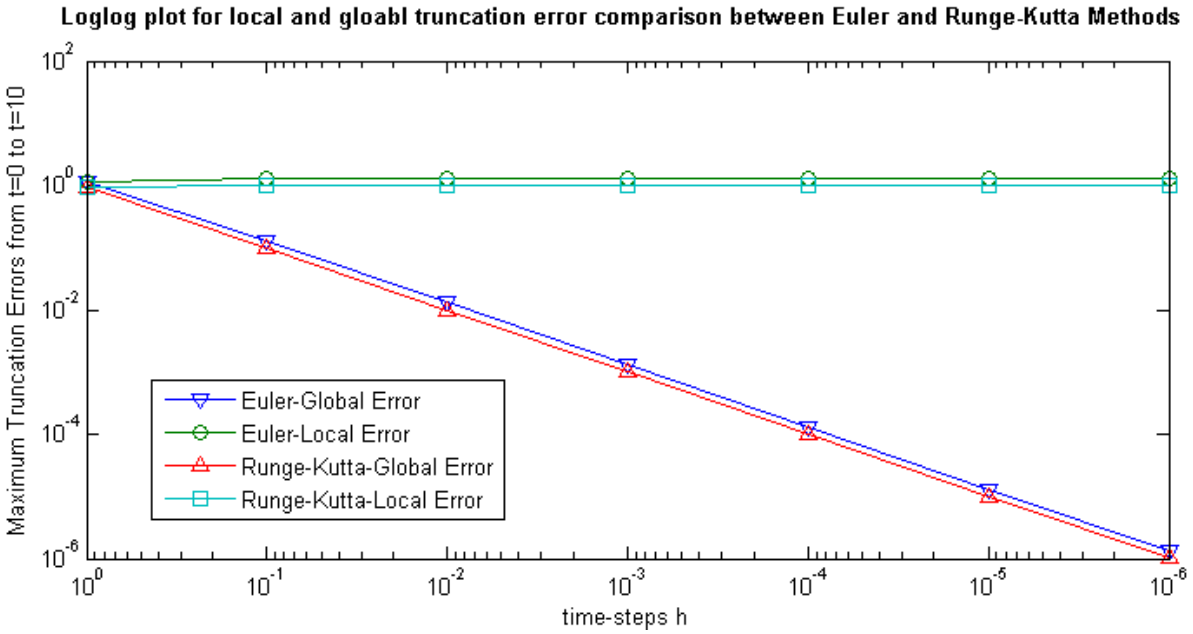


Figure 2. loglog plot of truncation error comparison between Euler's Method and Runge-Kutta Method

As expectation, the local truncation errors remain constant as the step-size change and global truncation errors decreases. Since local truncation error are due to the approximation made at each step of the method, the error causes by each step with different time-steps are the same because this ODE is an autonomous one which does not depends on time. And the global truncation error is the cumulative effect of all the local truncation errors, so they are decrease as the time-step, h is approaching to zero.

1.1.4 Upper-bound of the global truncation error based on the exact solution

The exact solution is shown in 1.1.3 above, in order to find the upper-bound of the global truncation error M, we take the second derivative of Equation (2).

$$2\tanh^{-1}\left(\tan\frac{1}{2}\right) \approx 1.226;$$

$$y(t) = 2\tan^{-1}(\tanh(\frac{1}{2}(t - 1.226)))$$

$$y''(t) = \frac{\tanh(0.613 - 0.5t)\,\operatorname{sech}^2(0.613 - 0.5t)\,(\tanh^2(0.613 - 0.5t) + \operatorname{sech}^2(0.613 - 0.5t) + 1)}{(\tanh^2(0.613 - 0.5t) + 1)^2}$$

Plot the graph of y''(t), it is easier to see the maximum of y''(t) from t=0 to t=10 is 0.5 around t=0.345.

$$M = |y''(t)| \leq 0.5$$

The upper bound M from exact solution is equal to 0.5 that is exactly the same from approximation value in 1.1.2.

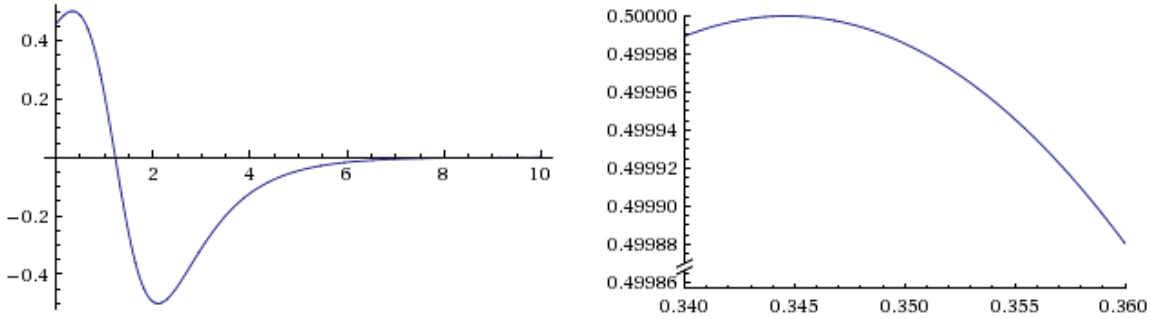


Figure 3 and 4. Plot of second derivative of y from t=0 to t=10 and the zoomed plot

1.2 Time-dependent Ordinary Differential Equation

Consider the initial value problem:

$$\begin{cases} \frac{dy}{dt} = t\ \cos y \quad \text{defined on D} \\ y(0) = -1 \end{cases} \tag{3}$$

where $\{(t, y) | 0 \le t \le 10; -\infty < y < \infty\}$.

1.2.1 Prove well-posed problem

Same as shown in 1.1.1,

We find that when $y_1<y_2$, a number ξ in (y_1,y_2) exists with

$$\frac{f(y_2) - f(y_1)}{(y_2) - (y_1)} = \frac{\partial}{\partial y} f(\xi) = -t \sin \xi.$$

Thus,

$$|f(y_2) - f(y_1)| = |y_2 - y_1||t\sin(\xi)| \le 10|y_2 - y_1|,$$

And f satisfies a Lipschitz condition in the variable y with Lipschtz constant L=10. Also, f(y) is continuous when $0 \le t \le 10; -\infty < y < \infty$. Theorem 5.6 in the textbook implies that this is a well-posed problem.

1.2.2 MATLAB Algorithm to solve the ODE

In the section, the ODE is solved numerically by Euler's method and Runge-Kutta method of order 4 using MATLAB algorithm. Here's the algorithm to solve the ODE with h=0.1. This code is similar to code in 1.1.2 with different f(t,y) function.

```
clc;
clear;%f(t,y)=cosy;
a=0;%start time
b=10;%end time
j=-1;
h=10^j;
N=(b-a)/h;
t=a;
y0=-1;
w=y0;
v=y0;

A=zeros(N,2);
A(1,1)=a;
A(1,2)=y0;

B=zeros(N,2);
B(1,1)=a;
B(1,2)=y0;
for i=1:1:N;
    w=w+h*t*cos(w);%euler

    K1=h*t*cos(v);%Runge-Kutta
    K2=h*(t+h/2)*cos(v+K1/2);
    K3=h*(t+h/2)*cos(v+K2/2);
    K4=h*(t+h)*cos(v+K3);
    v=v+(K1+2*K2+2*K3+K4)/6;

    t=a+i*h;

    A(i+1,1)=t;
    A(i+1,2)=w;

    B(i+1,1)=t;
    B(i+1,2)=v;

end

%plot
[T,Y]=meshgrid(0:.4:10,-3:.24:3);
dY=T.*cos(Y);
dT=ones(size(dY));
dYu=dY./sqrt(dT.^2+dY.^2);
dTu=dT./sqrt(dT.^2+dY.^2);
quiver(T,Y,dTu,dYu,'k');
hold on
plot(A(:,1),A(:,2),'-ob'); %plot euler method
plot(B(:,1),B(:,2),'-*r'); %plot runge-kutta
hold off
```

```
xlabel('t');
ylabel('y');
```

Plots generated by running the algorithm above:

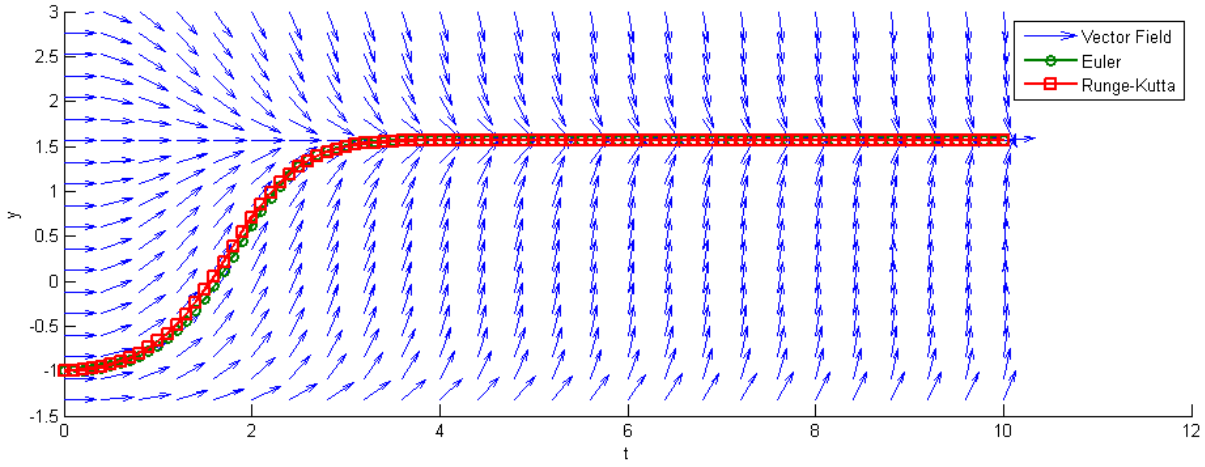


Figure 5. Solution of the ODE by Euler's Method and Runge-Kutta Method in the vector field

Provide an upper bound of the global truncation error is the same as shown in 1.1.2:

$$y''(t) = \frac{dy'}{dt}(t) = f'(t,y) = \frac{\partial f}{\partial t}(t,y(t)) + \frac{\partial f}{\partial y}(t,y(t)) \cdot y'(t)$$

In this case f=y'=t*cosy, so

$$y''(t) = \frac{\partial}{\partial t}(t\cos y) + \frac{\partial}{\partial y}(t\cos y) \cdot t\cos y = \cos y - t^2 \sin y \cdot \cos y = \cos\ y(1 - t^2 sin y)$$

max y''(t) at t=10 is ± 50.71:

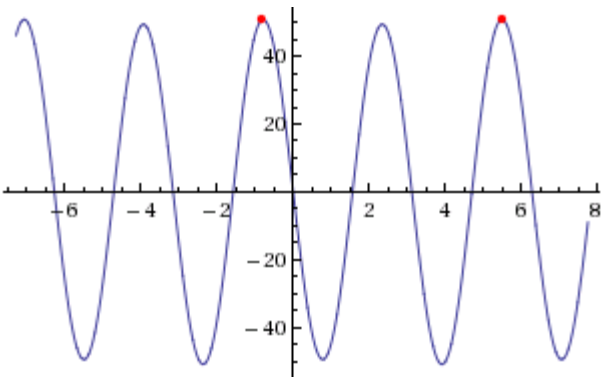


Figure 6. Plot of maximum y''(t) using approximation

$$M = |y''(t)| \leq 50.71$$

1.2.3 Local and global truncation errors with different time steps

Find the closed-form solution to this ODE.

$$\frac{dy}{dt} = t\cos y$$

$$\int \sec y \; dy = \int dt$$

$$\ln(\tan y + \sec y) = \frac{t^2}{2} + C$$

Solving for y:

$$y(t) = 2\tan^{-1}(\tanh(\frac{1}{4}(t^2 + C)))$$

Apply the initial condition, y(0)=-1,

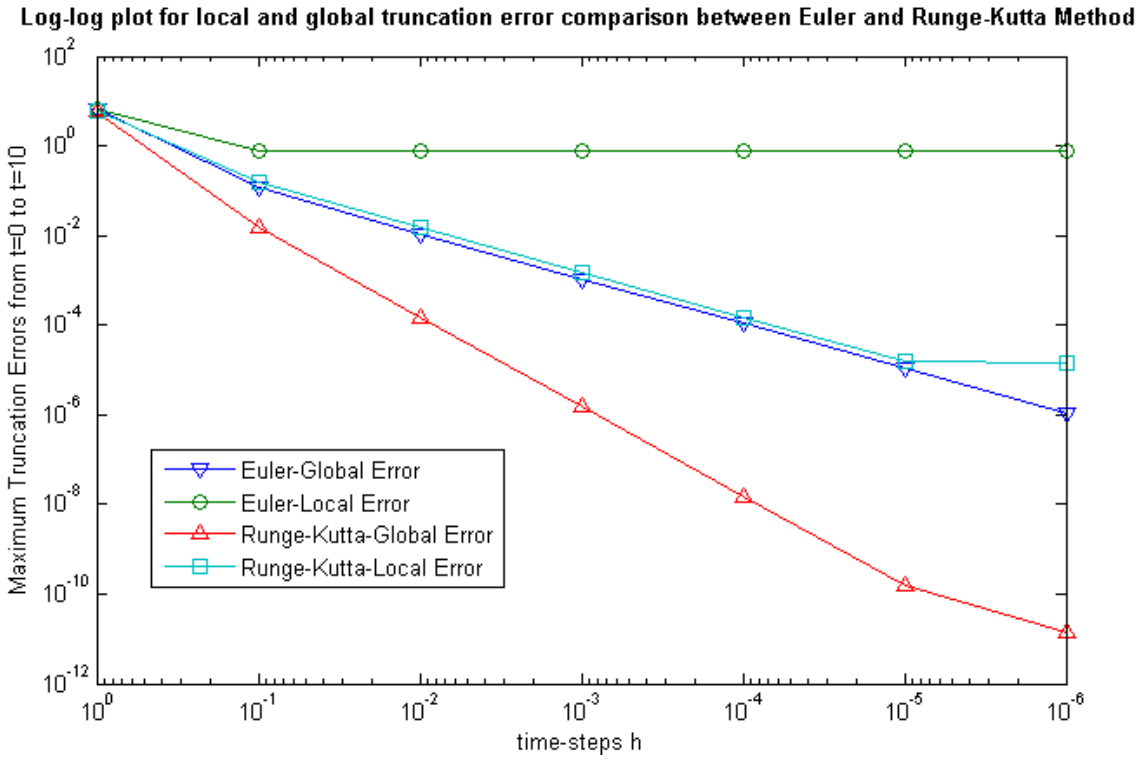
$$y(t) = 2\tan^{-1}\left\{\tanh\left[\frac{1}{4}\left(t^2 - 4\tanh^{-1}\left(\tan\frac{1}{2}\right)\right)\right]\right\} \quad (4)$$

Similar algorithm is implied for the log-log plot comparison. The only difference is the ODE and the corresponding exact solution as shown in Equation (4).

```
clc;
clear;
a=0;%start time
b=10;%end time
format long;
max_error=zeros(6,5);
y_exact=@(t)2*atan(tanh(0.25*(t.^2-4*atanh(tan(0.5)))));
f=@(t,y)t*cos(y);
%
for j=0:1:6
    h=10^-j;
    N=(b-a)/h;
    y0=-1;
    t=zeros(N,1);
    %initial values
    t(1)=a;
    w=y0;
    v=y0;

    A=zeros(N,8);
    A(1,1)=a;
    A(1,2)=w;
    A(1,3)=y0;

    A(1,4)=abs(y0-w);
    A(1,5)=((y0-w)/h);%local truncation error of euler method

    A(1,6)=v;
```

```
    A(1,7)=abs(y0-v);%global truncation error of runge-kutta method
    A(1,8)=((y0-v)/h);%local truncation error of runge-kutta method


    %p_y=y0;
    p_w=y0;
    p_v=y0;

    for i=2:1:N
        %numerical solution after each steps
        w=p_w+h*f(t(i-1),p_w);

        %global truncation error of euler method
        A(i,4)=abs(y_exact(t(i-1))-p_w);

        A(i,5)=abs(y_exact(t(i-1))-w)/h;
        %A(i+1,5)=((y-p_w)/h-cos(p_w));%local error of euler method


        K1=h*f(t(i-1),p_v);%Runge-Kutta
        K2=h*f((t(i-1)+h/2),(p_v+K1/2));
        K3=h*f((t(i-1)+h/2),(p_v+K2/2));
        K4=h*f((t(i-1)+h),(p_v+K3));
        phi=(K1+2*K2+2*K3+K4)/(6*h);
        v=p_v+h*phi;


        t(i)=a+i*h;

        A(i,1)=t(i);
        A(i,2)=p_w;
        A(i,3)=y_exact(t(i-1)); %exact value

        %column 4,5,7,8 of matrix A are local and global errors of two
        %different methods
        A(i,6)=p_v;
        A(i,7)=abs(y_exact(t(i-1))-p_v);
        %global truncation error of runge-kutta method
        %A(i+1,8)=((y-p_v)/h-phi);
        %local truncation error of runge-kutta method
        A(i,8)=abs(y_exact(t(i-1))-p_v)/h;
        %p_y=y;
        p_w=w;
        p_v=v;

    end
    max_error(j+1,1)= h;
    max_error(j+1,2)= max(A(:,4)); %global euler error
    max_error(j+1,3)= max(A(:,5)); %local truncation error--euler
    max_error(j+1,4)= max(A(:,7)); %global runge-kutta error
    max_error(j+1,5)= max(A(:,8)); %local truncation error--runge-kutta

end
%A
```

```
%max_error
loglog(max_error(:,1),max_error(:,2),max_error(:,1),max_error(:,3),max_err
or(:,1),max_error(:,4),max_error(:,1),max_error(:,5));
hleg1=legend('Euler-Global Error','Euler-Local Error','Runge-Kutta-Global
Error','Runge-Kutta-Local Error');
set(hleg1,'Location','NorthEast')
set(hleg1,'Interpreter','none')
xlabel('time-steps h');
ylabel('Maximum Truncation Errors from t=0 to t=10');
set(gca,'XDir','reverse');
```



Both local and global truncation errors decrease as the step-size h approaches to zero, that means we are getting more accurate with smaller errors as the step-size h decreases as expectation. Local truncation error is not as constant as the previous case, that's due to the time-dependent property of this ODE. And one other thing to discuss is that global error of runge-Kutta method decreases more rapidly, that means Runge-Kutta method is a better and more accurate way for approximation. It is also less time-consuming than Euler's method.

1.2.4 Upper-bound of the global truncation error based on the exact solution

The exact solution is shown in 1.1.3 above, in order to find the upper-bound of the global truncation error M, we take the second derivative of Equation (4).

$$4\tanh^{-1}\left(\tan\frac{1}{2}\right) \approx 2.452;$$

$$y(t) = 2\tan^{-1}\left\{\tanh\left[\frac{1}{4}(t^2 - 2.452)\right]\right\}$$

$y''(t) =$

$$\{-(\operatorname{sech}^2(0.613 - 0.25t^2)\,(-\tanh^2(0.613 - 0.25t^2) + (t^2\tanh(0.25(t^2 - 2.452))(\operatorname{sech}^2(0.613 - 0.25t^2) + 1) + (t^2\tanh^3(0.25(t^2 - 2.452)) - 1)\} * \frac{1}{(\tanh^2(0.613 - 0.25t^2) + 1)^2}$$

Plot the graph of y''(t), it is easier to see the maximum of y''(t) from t=0 to t=10 is 1.5 around t=1.3.

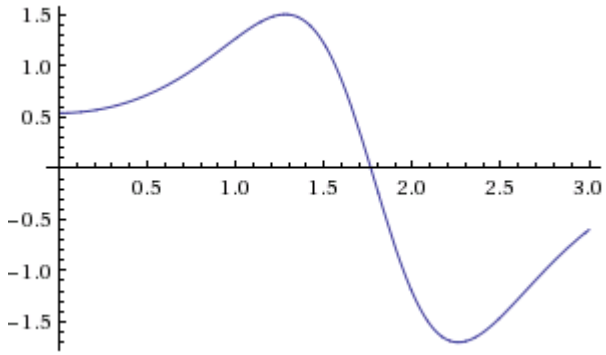


Figure 8. Plot of second derivative of y from t=0 to t=3

$$M = |y''(t)| \leq 1.5$$

Comparing with the approximate upper-bound of the global truncation error 50.71, the one from exact solution is way different from that. Time-dependent ODE causes the upper bound of the truncation error depends a lot on t, that causes the big difference between approximated upper bound and the one from the exact solution.,

1.2.5 Solve the ODE with different t range

We are asked to change the domain of time t from $0 \leq t \leq 10$ to $0 \leq t \leq 100$. Here's the modified MATLAB code to solve the ODE with Runge-Kutta and h=0.01 and h =0.001.

```
clear;
a=0;%start time
b=100;%end time changed from 10 to 100
j=-3; %change to -2 for h=0.01
h=10^j;%time step
N=(b-a)/h;% numbers of the steps
```

```
%initial condition
t=a;
y0=-1;
v=y0;

B=zeros(N,2);
B(1,1)=a;
B(1,2)=y0;
for i=1:1:N;

    %Runge-Kutta
    K1=h*t*cos(v);
    K2=h*(t+h/2)*cos(v+K1/2);
    K3=h*(t+h/2)*cos(v+K2/2);
    K4=h*(t+h)*cos(v+K3);
    v=v+(K1+2*K2+2*K3+K4)/6;

    t=a+i*h;

    B(i+1,1)=t;
    B(i+1,2)=v;
end
%plot
plot(B(:,1),B(:,2),'-*r'); %plot runge-kutta
xlabel('t');
ylabel('y');
```

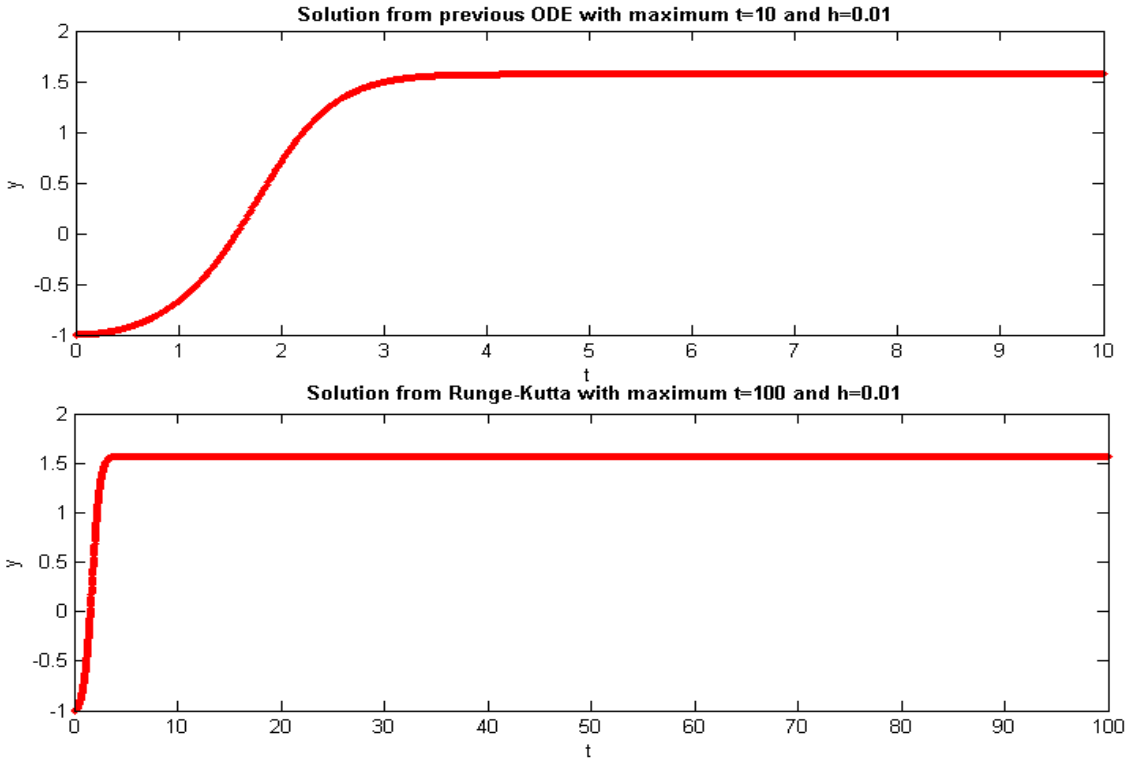


Figure 9. Solution solved by Runge-Kutta method with t_{max}=10 (and 100) and h=0.01

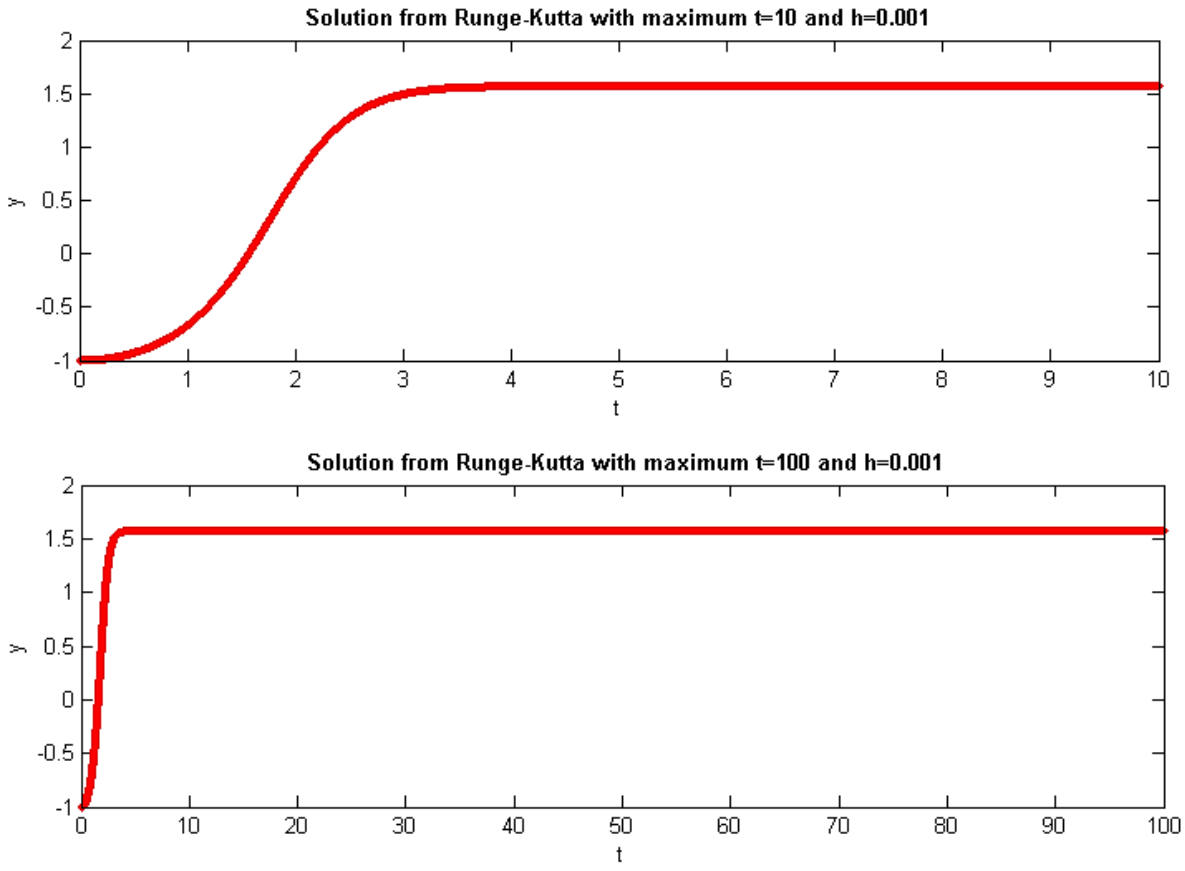


Figure 9. Solution solved by Runge-Kutta method with t_{max}= 10(and 100) and h=0.001

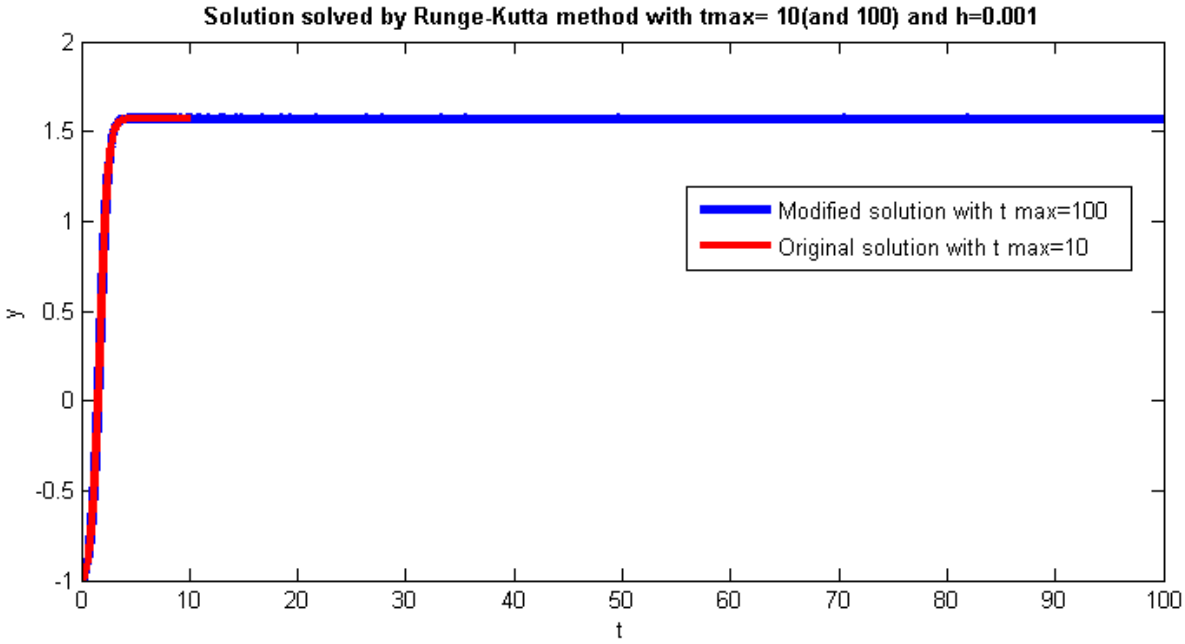


Figure 10. Solution solved by Runge-Kutta method with t_{max}= 10(and 100) and h=0.001

Figure 9 and 10 shows the comparison of solution of $0 \le t \le 10$ (previous) and $0 \le t \le 100$. Though it seems the original y(t) and modified f(t) are different in figure 9, that is due to the scale of the t-axes. So in Figure 10, plotting on the same graph we can conclude that change the range of t doesn't affect the solution solved by Runge-Kutta. They are overlapping on each other in Figure 10.

1.2.6 Solve the ODE with command ode45

Use command ode45 to solve the ODE. Here's the matlab algorithm.

```
clc;
clear;

f=inline('t*cos(y)','t','y');
[t,y]=ode45(f,[0,10],-1);

plot(t,y);
xlabel('t');
ylabel('y');
```

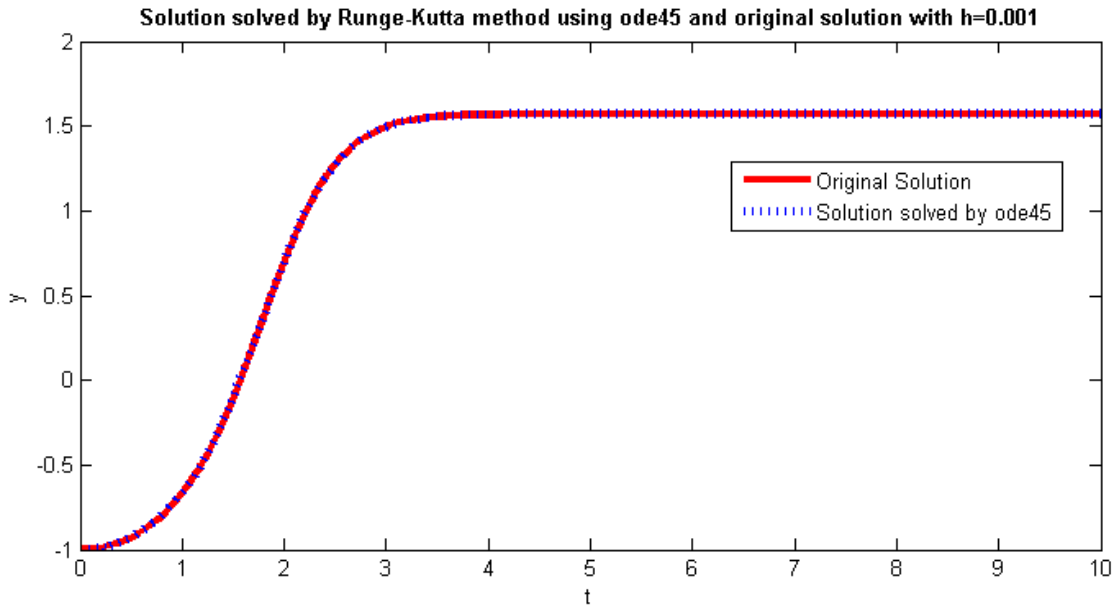
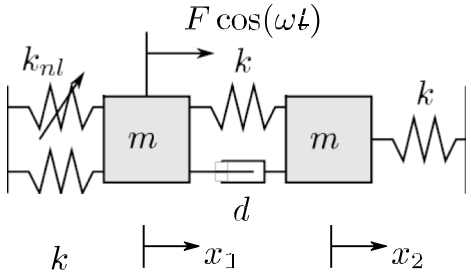


Figure 11. Solution solved by Runge-Kutta method with t_{max}= 10(and 100) and h=0.001

Ode45 is a MATLAB standard solver for ODE. This function implements a Runge-Kutta method with a variable time step for efficient computation. The red curve is the solution we get in previous section. Since the one plotted by ode45 and our code matches perfectly, we can conclude that our solution is correct. We can use this way to solve this ODE with very large t, but it's time-consuming as t goes to a very large number.

2. Harmonic Balance Method

2.1 Equations of Motion



2.1.1 Derivation

Apply Newton's 2nd Law of Motion to each block separately by considering the Free-Body Diagram for each block. For Block 1 obtain:

$$\sum F = ma_1 = m\ddot{x}_1$$

$$Fcos(\omega t) - F_{nl} - F_{k1} - F_{k2} - F_D = m\ddot{x}_1$$

$$Fcos(\omega t) - \epsilon x_1^3 - kx_1 - k(x_1 - x_2) - d(\dot{x}_1 - \dot{x}_2) = m\ddot{x}_1$$

$$m\ddot{x}_1 + d(\dot{x}_1 - \dot{x}_2) + 2kx_1 - x_2 = Fcos(\omega t) - \epsilon x_1^3 \quad (1)$$

2.4 Similarly, for Block 2 obtain:

$$m\ddot{x}_2 + d(\dot{x}_2 - \dot{x}_1) + 2kx_2 - x_1 = 0 \quad (2)$$

These 2 equations can be expressed in matrix form as:

$$\begin{bmatrix} m & 0 \\ 0 & m \end{bmatrix}\begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix} + \begin{bmatrix} d & -d \\ -d & d \end{bmatrix}\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} + \begin{bmatrix} 2k & -k \\ -k & 2k \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} Fcos(\omega t) - \epsilon x_1^3 \\ 0 \end{bmatrix}$$

2.1.2 Substitution

$$\frac{dz_1}{dt} = f_1(t, z_1, z_2, z_3, z_4) = z_2$$

$$\frac{dz_2}{dt} = f_2(t, z_1, z_2, z_3, z_4) = [Fcos(\omega t) - \epsilon z_1^3 - d(z_2 - z_4) - 2kz_1 + z_3]/m$$

$$\frac{dz_3}{dt} = f_3(t, z_1, z_2, z_3, z_4) = z_4$$

$$\frac{dz_4}{dt} = f_4(t, z_1, z_2, z_3, z_4) = [-d(z_4 - z_2) - 2kz_3 + kz_2]/m$$

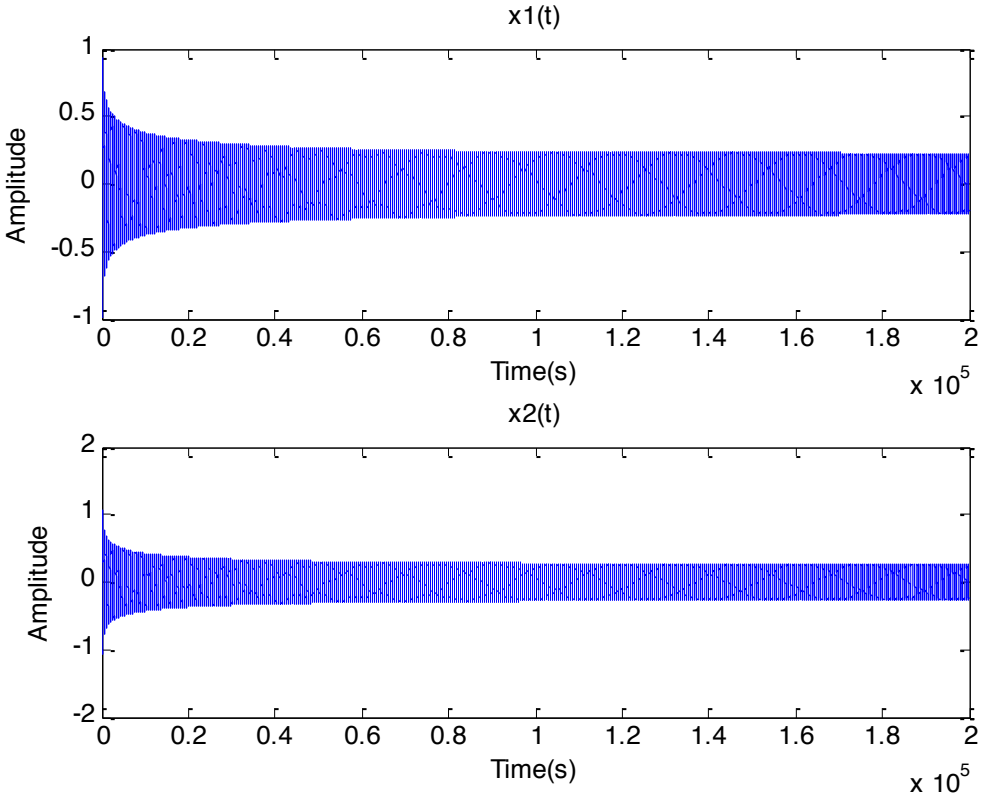
2.1.3 Implementation

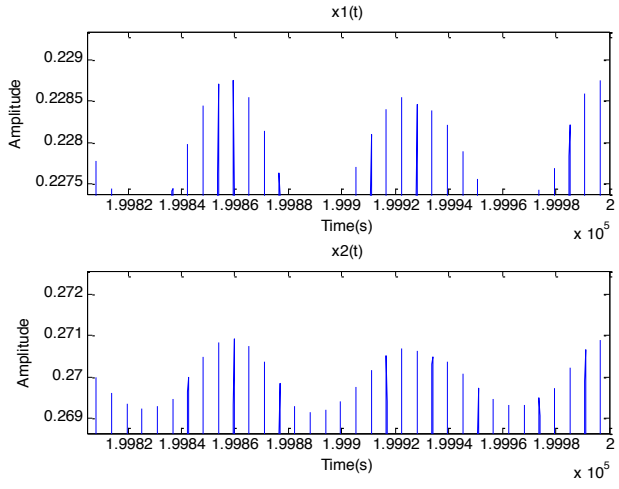
```
function [] = hbm21 ()
clc
% This code is an adaptation of the algorithm posted on the
% MECH 309 webCT page in the folder "week 11".
a = 0;                    % start time
b = 200;                  % end time
N = 2000;                 % # of steps
h= (b-a)/N;               % step size
%
t = a;
z(1) = 1;                   % initial value of x1 at t = a
z(2) = 0;                   % initial value of x1' at t = a
z(3) = 1;                   % initial value of x2 at t = a
z(4) = 0;                   % initial value of x2' at t = a
% Defines a matrix that will store each iteration for x1 and x2.
% It does not store the values of x1' or x2'
P=zeros(N,2);
% Runge Kutta Algorithm as discussed in class notes.
for i = 1:1:N
    K1 = h*ODE(t,z);
    K2 = h*ODE(t + h/2, z + K1/2);
    K3 = h*ODE(t + h/2, z + K2/2);
    K4 = h*ODE(t + h, z + K3);
%
    z = z + (K1 + 2*K2 + 2*K3 + K4)/6;
% Each iteration of x1 and x2 (i.e. z(1) and z(3)) are stored in P.
    P(i,1)=z(1);
    P(i,2)=z(3);
    t = a + i*h;
end
% Plots x1 and x2
t=linspace(a,b,N);
subplot(2,1,1)
plot(t,P(:,1))
xlabel('Time(s)');
ylabel('Amplitude');
title('x1(t)');
%
subplot(2,1,2)
plot(t,P(:,2))
xlabel('Time(s)');
ylabel('Amplitude');
title('x2(t)');
% Defines the system of two 2nd order ODEs as a system of
% four 1st order ODEs using the substitution z1=x1, z2=x1',
% z3=x2, z4=x2'
function z_prime = ODE(t, z)
m=1; k=1; d=0.5; e=0.5; w=1.1; F=0.1;
z_prime(1) = z(2);
z_prime(2) = (F*cos(w*t)-e*(z(1)^3)-d*z(2)+d*z(4)-2*k*z(1)+k*z(3))/m;
z_prime(3) = z(4);
z_prime(4) = (-d*z(4)+d*z(2)-2*k*z(3)+k*z(1))/m;
end
end
```

2.1.4 Application

Taking m=1, k=1, d=0.5, ϵ=0.5, ω=1.1, F=0.1, obtain the following solutions.

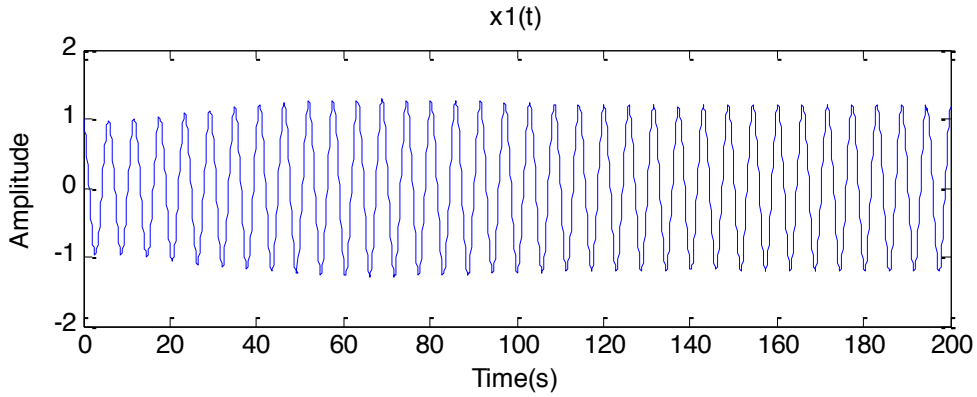
Set 1: $(x_1(0),x_2(0))=(-1,-1)$ and $(x_1'(0),x_2'(0))=(0,0)$

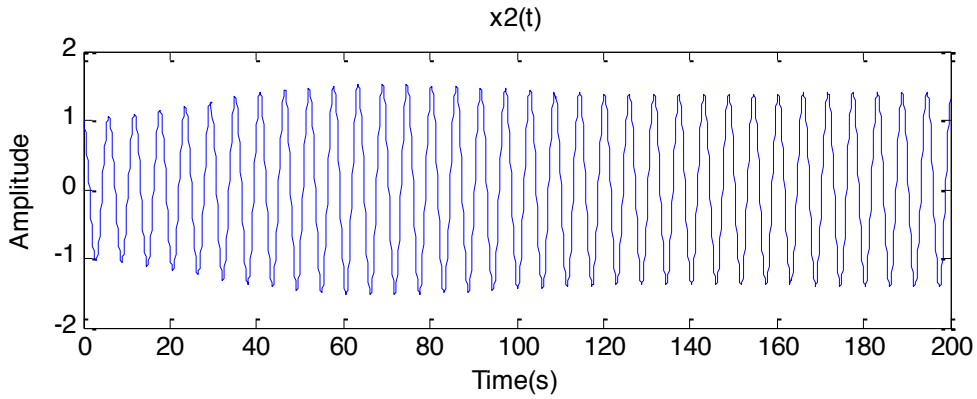


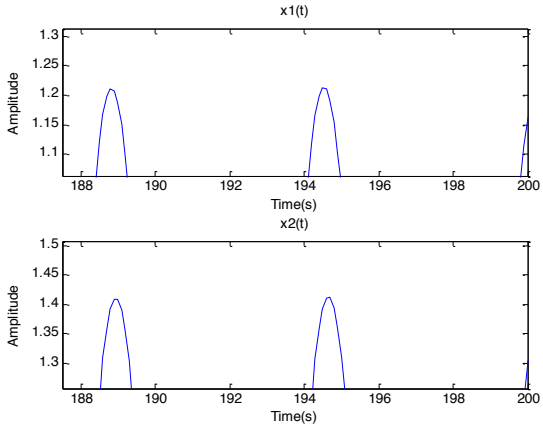


.

Set 2: $(x_1(0),x_2(0))=(1,1)$ and $(x_1'(0),x_2'(0))=(0,0)$







2.1.5 Discussion

For the initial point (1,1), the transient response vanishes very quickly, and the steady state is reached rapidly as well (after approximately 100 seconds). The amplitude of the steady state oscillation is approximately 1.2 for x1(t) and 1.4 for x2(t). The period of the oscillations is the same in both cases, at 5.5 seconds.

For the initial point (-1,-1), the convergence behavior is completely different. It takes a much longer period of time (around 10^5 seconds) to arrive close to the steady state. In addition, the steady state in this case is significantly different from the previous one, at amplitudes of 0.23 and 0.27 for x1 and x2 respectively. The periods of x1 and x2 are the same at approximately 6 seconds, and are very close to the periods of x1 and x2 for the previous initial condition (5.5 seconds for (1,1)).

Thus, the following steady states are identified:

$(1,1) \rightarrow x1 \cong 1.2 \cdot \sin(2\pi/5.5)$

$x2 \cong 1.4 \cdot \sin(2\pi/5.5)$

$(-1,-1) \rightarrow x1 \cong 0.23 \cdot \sin(2\pi/6)$

$x2 \cong 0.27 \cdot \sin(2\pi/6)$

The presence of multiple steady states and the sensitive convergence behavior may be attributed to the nonlinear spring present in the system.

2.2 Frequency-domain formulation

2.2.1 Implementation

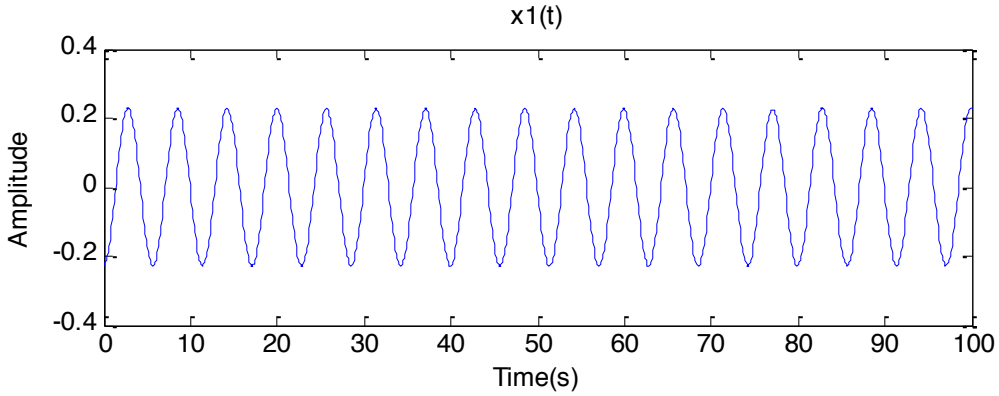
```
function [] = hbm22 () % HBM2.2 Solves the 8 equations
clc
syms x1 x2 a1 a2 a3 a4 b1 b2 b3 b4 t
w = 1.1;
% As instructed, x1(t) and x2(t) are expressed as a truncated
% Fourier series
x1=a1*cos(w*t)+a2*sin(w*t)+a3*cos(3*w*t)+a4*sin(3*w*t);
x2=b1*cos(w*t)+b2*sin(w*t)+b3*cos(3*w*t)+b4*sin(3*w*t);
% Define the first and second derivative for x1 and x2
x1_first=diff(x1,t);
x1_second=diff(x1,t,2);
x2_first=diff(x2,t);
x2_second=diff(x2,t,2);
% Equation 1 and 2
eq1=x1_second+0.5*(x1_first-x2_first)+2*x1-x2-0.1*cos(w*t)+0.5*(x1^3);
eq2=x2_second+0.5*(x2_first-x1_first)+2*x2-x1;
% Define the limits of integration
a=0;
b=(2*pi()/w);
% Integrate equation 1 and 2 as instructed to obtain 8 nonlinear
% algebraic equations in the 8 unknowns
fcn(1)=int(eq1*cos(w*t), t, a, b);
fcn(2)=int(eq1*sin(w*t), t, a, b);
fcn(3)=int(eq1*cos(3*w*t), t, a, b);
fcn(4)=int(eq1*sin(3*w*t), t, a, b);
%
fcn(5)=int(eq2*cos(w*t), t, a, b);
```

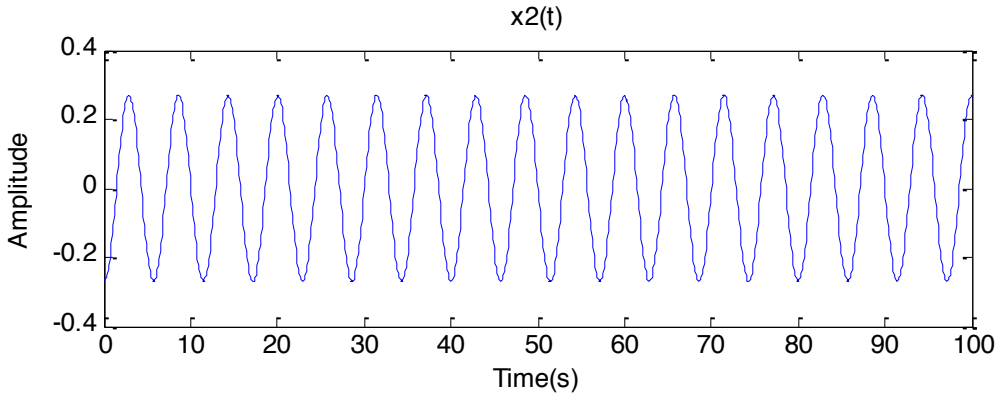
```
fcn(6)=int(eq2*sin(w*t), t, a, b);
fcn(7)=int(eq2*cos(3*w*t), t, a, b);
fcn(8)=int(eq2*sin(3*w*t), t, a, b);
% The Jacobian is calculated the easy way. 'fcn' contains the 8
% above equations, while 'v' contains the variables we will differentiate
% with respect to
v = [a1 a2 a3 a4 b1 b2 b3 b4];
B = jacobian(fcn,v);
% The tolerance is set, as well as the initial guess for the 8 unknowns.
tol = 10^(-8);
LeNorme=1;
m = 0.8;
a1 = m; a2 = m; a3 = m; a4 = m;
b1 = m; b2 = m; b3 = m; b4 = m;
% All the 'fcn' functions, are symbolic. To evaluate them, the 'eval'
% command is used. The values are then stored in an array.
    for i=1:8
        f(i,1)=eval(fcn(i));
    end
% The same thing is done for the Jacobian matrix B
    B=eval(B);
% The initial guess for the 8 unknowns is stored in the array x. This
% is also the array where the converged results will be stored
    x = [a1 a2 a3 a4 b1 b2 b3 b4]';
% Broyden's method with 50 max iterations and 10^(-8) tolerance
for k = 1 : 50
    s = B\(-f);
    x = x + s;
    a1 = x(1); a2 = x(2); a3 = x(3); a4 = x(4);
    b1 = x(5); b2 = x(6); b3 = x(7); b4 = x(8);
    for i = 1 : 8
        fnew(i,1) = eval(fcn(i));
    end
    y = fnew-f;
    LeNorme=norm(x);
    if LeNorme < tol
        break
    end
    f = fnew;
    B = B + ((y-B*s)*s')/(s'*s);
end
% Plots the results in the time domain.
t = linspace(0,100,1000);
x1 = a1*cos(w*t) + a2*sin(w * t) + a3*cos(3*w*t) + a4*sin(3*w*t);
x2 = b1*cos(w*t) + b2*sin(w*t)+ b3*cos(3*w*t) + b4*sin(3*w*t);
%
x
subplot(2,1,1)
plot(t,x1)
xlabel('Time(s)');
ylabel('Amplitude');
title('x1(t)');
%
subplot(2,1,2)
plot(t,x2)
xlabel('Time(s)');
ylabel('Amplitude');
```

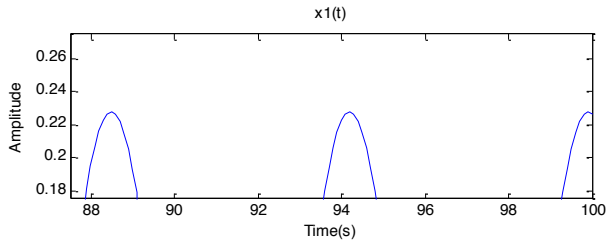
```
title('x2(t)');
end
```

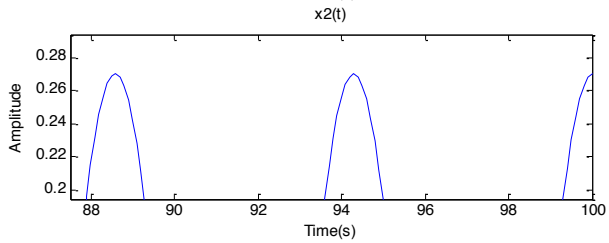
2.2.2 Application

Using the initial guess m=0.1 for a_i and b_i, i=1,4, we obtain the following steady state.

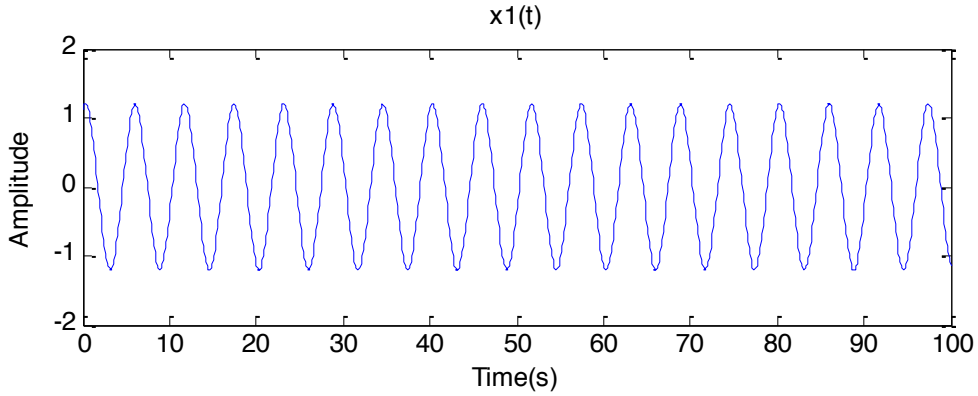


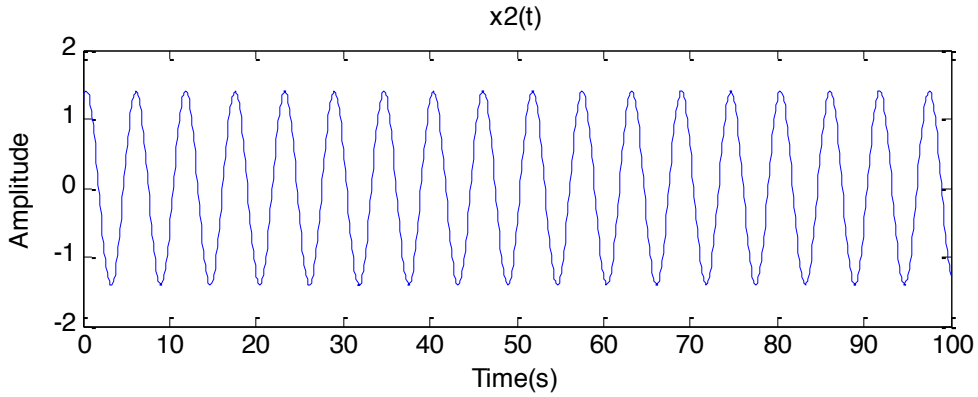


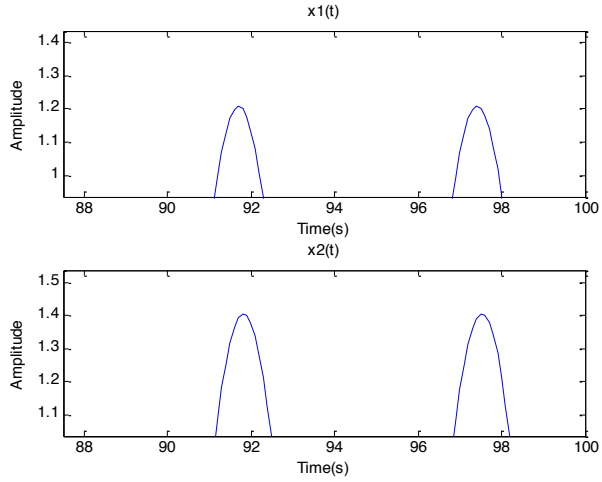




Using m=1.0, we obtain:







2.2.3 Discussion

Just like the method in 2.1, the Harmonic Balance Method is also sensitive to initial conditions. Depending on the initial guess for Newton's or Broyden's method, convergence to a different basin of attraction can occur. Using the conditions proposed in the project description, namely 0.1 and 0.5, we obtain the same solution. We therefore need to use different values for the initial guesses. Using 0.8 or 1.0 (as an example), we managed to obtain another steady state. Note that the time interval here does not matter because the steady state response is obtained right away; there is no time

variation due to the transient response. A large time interval is thus not necessary, so t=100s was used.

As can be deduced from the zoomed-in images above, the two steady states obtained are:

$m = 0.1 \rightarrow x1 \cong 0.23 \cdot \sin(2\pi/6)$

$x2 \cong 0.27 \cdot \sin(2\pi/6)$

$m = 1.0 \rightarrow x1 \cong 1.2 \cdot \sin(2\pi/5.5)$

$x2 \cong 1.4 \cdot \sin(2\pi/5.5)$

Comparing with the results obtained from part 2.1, it can be seen that the steady states obtained are identical. Both methods yield the same results, but differ in the time required to do so. The main challenge of method 2.1 is the unpredictable amount of time needed to convergence to the steady state solution. Using (1,1) as the initial guess leads to very rapid convergence, but (-1,-1) requires hundreds of thousands of iterations to obtain the steady state.

This problem is not present in method 2.2 (Harmonic Balance Method), where the transient response is filtered right away. Although this method leads to a significantly faster convergence, there are other disadvantages to it. First, obtaining the 8 equations via integration can be time consuming. However, this can be overcome by using a separate MATLAB file to compute the equations, then hardcoding the resulting equations into the main script. Another potential problem is the use of Newton's or Broyden's method to solve the nonlinear system. Neither of these two methods guarantees convergence to the solution.