Multibody Dynamics B - Assignment 6

July 4, 2019

Short problem statement

We are determining the motion of a double pendulum by numerical integration, expressed in terms of independent generalized coordinates α and β. We have the same double pendulum as in Homework assignment 4 where our initial conditions are both bars vertical up with zero velocity and there acts a gravitational field of strength $g = 9.81$ in the horizontal direction. We want to determine the angle in radians of both bars, with respect to horizontal, after 3 seconds with a maximum absolute error of 10^{-6}. To find those angles we first have to determine the maximum step size with regard to the maximum absolute error. We will first use 4 numerical integration methods to determine the angles and step sizes and those are

- **Euler**
- **Heun**
- **Runge-Kutta** 3^{rd} **order**
- **Classical Runge-Kutta** 4^{th} **order**

Furthermore, we will also use three ODE solvers, ode23, ode45, ode113, to compare with the other 4 methods. Now we have a set of n second-order differential equations with 2n initial values

$$\bar{M}\ddot{Q} = \bar{Q}(t, q, \dot{q}), \quad q(t_0) = q_0, \quad \dot{q}(t_0) = \dot{q}_0$$

However our integral numerical methods need to be of the form of ODEs(ordinary differential equations). This can be done by formulating the differential equations into the standard first-order form by making

$$\dot{q} = u \quad q(t_0) = q_0$$

$$\dot{u} = \bar{M}^{-1}\bar{Q}(t, q, u), \quad u(t_0) = u_0 = \dot{q}_0$$

where

$$y = (q^T, u^T)^T \quad \dot{y} = f(t, y), \quad y(t_0) = y_0$$

Numerical integration methods

- **Euler**

The Euler step method is the simplest and can be derived from the definition of a derivative where

$$\lim_{h \to 0} \frac{q(t+h) - q(t)}{h} = \dot{q(t)}$$

which gives

$$q(t+h) = q(t) + h * \dot{q(t)}$$

- **Heun**

A more accurate and refined method is the Heun method, where we first predict the end value $y*_{n+1}$ using an Euler step, to take the average between the derivative at the mentioned point and y_n, this results in

$$y*_{n+1} = y_n + hf(t_n, y_n)$$

$$y_{n+1} = y_n + h/2 * (f(t_n, y_n) + f(t_{n+1}, y*_{n+1}))$$

- **Runge-Kutta**

This is a popular method that is a generalization of the Euler's method by allowing a number of evaluations of the derivative within a step.

- For the 3rd order method we have

$$k_1 = f(t_n, y_n)$$
$$k_2 = f(t_n + h/2, y_n + h/2 * k_1)$$
$$k_3 = f(t_n + h, y_n - hk_1 + 2hk_2)$$
$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 4k_2 + k_3)$$

- For the 4th order method we have

$$k_1 = f(t_n, y_n)$$
$$k_2 = f(t_n + h/2, y_n + h/2 * k_1)$$
$$k_3 = ff(t_n + h/2, y_n + h/2 * k_2)$$
$$k_4 = ff(t_n + h, y_n + h * k_3)$$
$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

- **Error estimates**

We have both truncation errors and round off errors, due to Matlab only being able to work with finite amount of words to represent the number. The truncation errors are a accumulation of local truncation errors, errors after one numerical step, due to our result not being accurate enough, since we lack higher order terms, like in a Taylor expansion. Now the numerical solution has the form of

$$y = \bar{y} + C_1 h^p + C_2 \frac{1}{h}$$

where we can estimate the truncation error as

$$E_{truncation} = C_1 * |y_n - y_{n-1}|$$

where $C_1 = \frac{1}{2^p - 1}$ and $p = 1$ for Euler, $p = 2$ for Heun, $P = 3$ for Runge-Kutta of 3rd order and $p = 4$ for Runge-Kutta of 4th order, and the round of error can be estimates as

$$E_{round} = \frac{C_2}{h}$$

So the total error is

$$E_{total} = \frac{1}{2^p - 1}|y_n - y_{n-1}| + \frac{C_2}{h}$$

where for Matlab we have $C_2 = 10^{-16}$.

Results

We first find the converging angles for different methods, we will look at a few steps, every 4th step. We have the step size defined as $h = T/(2^n)$, and we look at $n = 8, 12, 16, 20$

- Euler

For Euler we get

h	alpha	beta	error alpha	error beta
0.011718750000000	-1.106664175256109	4.720515621861588	1.450087991696280	25.088438099732603
0.000732421875000	-1.212487956291334	-2.924235522261515	0.039983562885109	0.373036498022980
0.000045776367188	-1.248703027689693	-2.598181533360753	0.002421297560526	0.019878381357347
0.000002861022949	-1.250963051801287	-2.579828265556614	0.000150319365153	0.001214936771111

- Heun

For Heun we get

h	alpha	beta	error alpha	error beta
0.011718750000000	-1.291486473308608	-2.416734764969545	0.037147780987974	0.125078898161654
0.000732421875000	-1.251281243139352	-2.577892212091306	0.000166219189531	0.000712224833550
0.000045776367188	-1.251113978574084	-2.578611716133587	0.000000660628441	0.000002848119354
0.000002861022949	-1.251113320054372	-2.578614555741426	0.000000002600903	0.000000011159080

- Runge-Kutta 3rd order

For Runge-Kutta we get

h	alpha	beta	error alpha	error beta
0.011718750000000	-1.253361845839678	-2.566628507154399	0.001806414945560	0.011404133781318
0.000732421875000	-1.251113812481844	-2.578611846395433	0.000000498355326	0.000002734803134
0.000045776367188	-1.251113317590621	-2.578614566223569	0.000000000121330	0.000000000662585
0.000002861022949	-1.251113317470084	-2.578614566886015	0.000000000017524	0.000000000017676

- Runge-Kutta 4th order

For Runge-Kutta we get

h	alpha	beta	error alpha	error beta
0.011718750000000	-1.251039583149531	-2.578724193741765	0.000182575334540	0.000355072853966
0.000732421875000	-1.251113317102798	-2.578614568206353	0.000000000422588	0.000000001334802
0.000045776367188	-1.251113317470385	-2.578614566885106	0.000000000001102	0.000000000001131
0.000002861022949	-1.251113317470079	-2.578614566886129	0.000000000017481	0.000000000017478

- **Step size estimation**

Now we estimate at what maximum step size we get an error equal to 10^{-6} for all the different methods. Matlab gives us

Step size	Euler	Heun	Runge-Kutta 3rd	Runge-Kutta 4th
h	NaN	$5.362805796531916 10^{-05}$	$8.361618940209792 10^{-04}$	0.003453733565808

Table 1: Step size estimation for when the error in the computation of alpha is equal to 10^{-6}

Step size	Euler	Heun	Runge-Kutta 3rd	Runge-Kutta 4th
h	NaN	$2.596842327934829\ 10^{-05}$	$4.671184590907266\ 10^{-04}$	0.003210869197420

Table 2: Step size estimation for when the error in the computation of beta is equal to 10^{-6}

Now we can see that the Euler method is not able to get an error of less or equal then 10^{-6}. The step size becomes so small that it is not practical to reach the end of the interval. Now the other methods all have a maximum step size for when the error is equal to 10^{-6} and we can compute the angles for when this holds up, for the Euler method we just choose the smallest step size we computed, $h = 0.000002861022949$. We also choose the smaller step size between alpha and beta to fulfill that both angles have an global error less than 10^{-6}.

Now for the Heun method we find that the first step size that gives an error smaller than 10^{-6} is when $n = 17$ which results in step size $h = 2.288818359375000 10^{-05}$, for the Runge-Kutta 3rd order method we get that the error is smaller thant 10^{-6} when $n = 13$ which results in step size of $h = 3.662109375000000 10^{-04}$ and for the Runge-Kutta

4th order method the error is smaller than 10^{-6} when $n = 10$ which results in step size of $h = 0.002929687500000$. So table 3 shows the corresponding angles for these step sizes.

Angles	Euler	Heun	Runge-Kutta 3rd	Runge-Kutta 4th
alpha	-1.250963051801287	-1.251113482791075	-1.251113379168993	-1.251113183607986
beta	-2.579828265556614	-2.578613853950130	-2.578614227596832	-2.578614916492579

Table 3: Angle position at the end of the time interval when the error is less than 10^{-6}

We see that the angles are the same up to the 6th character, except for Euler since it does not reach an error less than 10^{-6}.

Now we can calculate the number of functions called since we know that

$$N = p * T/h$$

So we have

- Euler; $N = T/h_{euler}$
- Heun; $N = 2 * T/h_{heun}$
- Runge-Kutta 3rd; $N = 3 * T/h_{rk3}$
- Runge-Kutta 4th; $N = 4 * T/h_{rk4}$

So we get

Function calls	Euler	Heun	Runge-Kutta 3rd	Runge-Kutta 4th
N	$1.048576000080173\ 10^6$	262144	24576	4096

Table 4: Number of function calls for the 4 different methods

We see that even though the functions are called more often in the more accurate methods the function call still decreases, leading to less computational power needed, because the step size is that much bigger, and thus not as many iterations are needed to converge to the final interval.

- **Plots**

In figure 1 and 2 we can see how the error converges as a function of the step size for the four different methods.

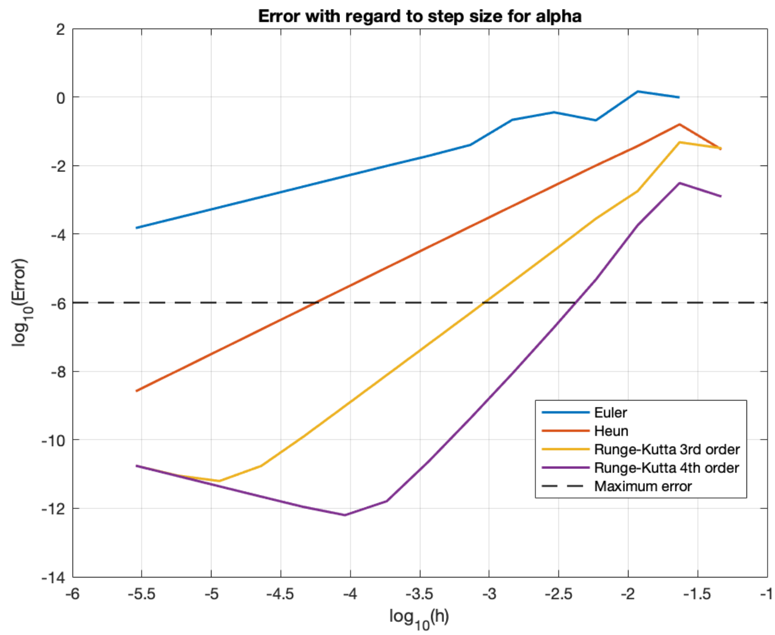

Figure 1: Error convergence as a function a step size for alpha

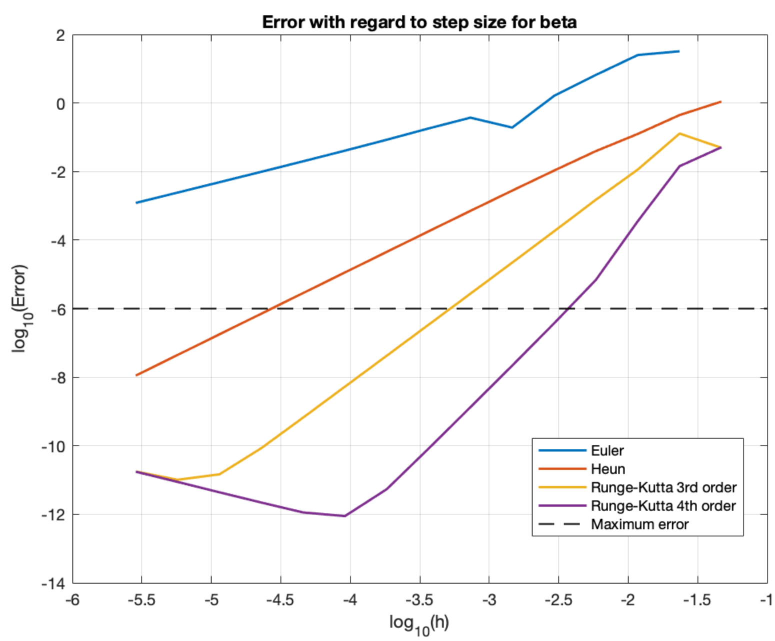

Figure 2: Error convergence as a function a step size for bet

Now we compute the average slope for the 4 different methods, and compared it to method specific exponent p. We only took the slope where the truncation error was dominant, since the truncation error is dependent on p, so the nonlinear parts to the far left, the are where round off error is dominant, i.e. to the left of h optimal, were excluded and the unstable parts to the far right were also excluded. By using Matlab we got

- Euler; average slope = 0.973714295753074 $\approx 1 = p_{euler}$
- Heun; average slope = 1.989695551679098 $\approx 2 = p_{heun}$
- Runge-Kutta 3rd order; average slope = 2.984915291915134 $\approx 3 = p_{rk3}$
- Runge-Kutta 4th order; average slope = 4.023670942564379 $\approx 4 = p_{rk4}$

- **ODE solvers**

We used three ode solvers provided by Matlab to determine the motion of the system, these solvers were `ode23`, `ode45` and `ode113`.

Now the relative tolerance that must be set is related to the number of significant digits m, $R_{tol} = 10^{-(m+1)}$ and since matlab has double precision of approximately 10^{-16} we put $R_{tol} = 10^{-16}$. Now the maximum error allowed is 10^{-6} so we put that as our absolute tolerance, this results in

$$R_{tol} = 10^{-16}$$

$$A_{tol} = 10^{-6}$$

Now, the angles that the solvers compute can be seen in table 5 below.

Angles	ode23	ode45	ode113
alpha	-1.251121751912791	-1.251117079132932	-1.251116301457757
beta	-2.578573016146374	-2.578597241031283	-2.578604457046552

Table 5: Angles at the end of the time interval for the ode solvers.

and the average step sizes are

Average step size	ode23	ode45	ode113
h	0.001564129301356	0.004048582995951	0.007025761124122

Table 6: Average step sizes, h, for ode solvers

and finally the total number of function calls are

Function calls	ode23	ode45	ode113
N	5794	1201	871

Table 7: Total number of function calls, N, for ode solvers

If we first examine the angles, we see that this corresponds rather nicely with what we had computed with the other 4 methods, resulting in the same results up till 4 significant characters, except for Euler of course. Now looking at the step sizes we notice that these step sizes are significantly larger, which is again reflected in the number of callbacks, where the solvers are of more complexity than the other 4 methods but still result in fewer callbacks, due to the larger step size. Thus it follows that with increasing complexity the number of callbacks actually decreases resulting in a lower computational effort, and more accuracy if we use the same step size.

Appendix A

Matlab code

```matlab

%% Set up EoM
syms alpha beta
syms alphadot betadot

q=[alpha; beta];
qd=[alphadot; betadot];

rho=670;
l=0.6;
V=l*0.05*0.003;
m=rho*V;
I=1/12*m*l^2;
g=9.81;
Qa=0;
Qb=0;

M=[I+5/4*m*l^2, 1/2*m*l^2*cos(alpha-beta);
    1/2*m*l^2*cos(alpha-beta), I+1/4*m*l^2];
F=[Qa-3/2*m*g*l*sin(alpha)-1/2*m*l^2*betadot^2*sin(alpha-beta);
    Qb-1/2*m*g*l*sin(beta)+1/2*m*l^2*alphadot^2*sin(alpha-beta)];


qdd=M\F;

matlabFunction(qdd,'File','qddhw6')

%% Setup
T=3;
nn=5:1:20;
N=2.^nn;
h=T./N;

format('Long')
%% Euler method

% Integration of all time steps

for i=1:length(nn)

    y0=[pi/2;pi/2;0;0];
    alpha=y0(1);
    beta=y0(2);
    alphadot=y0(3);
    betadot=y0(4);

    for j = 1:N(i)

        p=[alpha; beta];
        v=[alphadot; betadot];

        pn=p+h(i)*v;
        vn=v+h(i)*qddhw6(alpha,alphadot,betadot,beta);

        alpha = pn(1);
        beta = pn(2);
        alphadot = vn(1);
        betadot = vn(2);

    end
    q_end_h(i,:) = [alpha, beta];
end

% Global error
ph=1;
C2=10^(-16);
```

```

for i=1:length(nn)-1
    Da_eul(i) = q_end_h(i+1,1)-q_end_h(i,1);
    Db_eul(i) = q_end_h(i+1,2)-q_end_h(i,2);

    Eta_euler(i)=abs(1/(2^ph-1)*Da_eul(i));
    Etb_euler(i)=abs(1/(2^ph-1)*Db_eul(i));

    Eround(i)=abs(C2/h(i));

    E_tota_euler(i)=Eta_euler(i)+Eround(i);
    E_totb_euler(i)=Etb_euler(i)+Eround(i);

end

%% Heun method

% Integration of all time steps
for i=1:length(nn)

    alpha=pi/2;
    beta=pi/2;
    alphadot=0;
    betadot=0;

    for j = 1:N(i)

        y0=[alpha;beta;alphadot;betadot];

        ystar= y0+h(i)*qa(y0);
        qn=y0+h(i)/2*(qa(y0)+qa(ystar));

        alpha = qn(1);
        beta = qn(2);
        alphadot = qn(3);
        betadot = qn(4);

    end
    q_end_h(i,:) = [alpha, beta];
end

% Global error
ph=2;
C2=10^(-16);

for i=1:length(nn)-1
    Da_heun(i) = q_end_h(i+1,1)-q_end_h(i,1);
    Db_heun(i) = q_end_h(i+1,2)-q_end_h(i,2);

    Eta_heun(i)=abs(1/(2^ph-1)*Da_heun(i));
    Etb_heun(i)=abs(1/(2^ph-1)*Db_heun(i));

    Eround(i)=abs(C2/h(i));

    E_tota_heun(i)=Eta_heun(i)+Eround(i);
    E_totb_heun(i)=Etb_heun(i)+Eround(i);

end


%% Runge-Kutta 3rd order

% Intergration of all time steps
for i=1:length(nn)

    alpha=pi/2;
    beta=pi/2;
    alphadot=0;
    betadot=0;

    for j=1:N(i)
```

```
        y0=[alpha;beta;alphadot;betadot];
        k1=qa(y0);
        k2=qa(y0 + h(i)/2*k1);
        k3=qa(y0 + h(i)*(2*k2-k1));

        qn= y0+1/6*h(i)*(k1+4*k2+k3);

        alpha=qn(1);
        beta=qn(2);
        alphadot=qn(3);
        betadot=qn(4);
    end
    q_end_rk3(i,:)=[alpha,beta];
end

% Global error
p_rk3=3;
C2=10^(-16);

for i=1:length(nn)-1
    Da_rk3(i) = q_end_rk3(i+1,1)-q_end_rk3(i,1);
    Db_rk3(i) = q_end_rk3(i+1,2)-q_end_rk3(i,2);

    Eta_rk3(i)=abs(1/(2^p_rk3-1)*Da_rk3(i));
    Etb_rk3(i)=abs(1/(2^p_rk3-1)*Db_rk3(i));

    Eround(i)=abs(C2/h(i));

    E_tota_rk3(i)=Eta_rk3(i)+Eround(i);
    E_totb_rk3(i)=Etb_rk3(i)+Eround(i);

end

%% Runge-Kutta 4th order
% Intergration of all time steps
for i=1:length(nn)

    alpha=pi/2;
    beta=pi/2;
    alphadot=0;
    betadot=0;

    for j=1:N(i)
        y0=[alpha;beta;alphadot;betadot];
        k1=qa(y0);
        k2=qa(y0 + h(i)/2*k1);
        k3=qa(y0 + h(i)/2*k2);
        k4=qa(y0 + h(i)*k3);

        qn= y0+1/6*h(i)*(k1+2*k2+2*k3+k4);

        alpha=qn(1);
        beta=qn(2);
        alphadot=qn(3);
        betadot=qn(4);
    end
    q_end_rk4(i,:)=[alpha,beta];
end

% Global error
p_rk4=4;
C2=10^(-16);

for i=1:length(nn)-1
    Da_rk4(i) = q_end_rk4(i+1,1)-q_end_rk4(i,1);
    Db_rk4(i) = q_end_rk4(i+1,2)-q_end_rk4(i,2);

    Eta_rk4(i)=abs(1/(2^p_rk4-1)*Da_rk4(i));
    Etb_rk4(i)=abs(1/(2^p_rk4-1)*Db_rk4(i));

    Eround(i)=abs(C2/h(i));
```

```matlab

    E_tota_rk4(i)=Eta_rk4(i)+Eround(i);
    E_totb_rk4(i)=Etb_rk4(i)+Eround(i);

end

%% Plot the error vs step size for the 4 different methods

figure(1)
plot(log10(h(2:end)),log10(E_tota_euler),log10(h(2:end)),log10(E_tota_heun),log10(h(2:end)),log10(E_tota_rk3),log
    'linewidth', 1.5)
title('Error with regard to step size for alpha')
xlabel('log_{10}(h)')
ylabel('log_{10}(Error)')
grid on
hold on
plot([log10(10^-6),log10(10^-1)],[log10(10^-6),log10(10^-6)],'k--', 'linewidth', 1);
legend('Euler', 'Heun', 'Runge-Kutta 3rd order', 'Runge-Kutta 4th order', 'Maximum error')



figure(2)
plot(log10(h(2:end)),log10(E_totb_euler), log10(h(2:end)),log10(E_totb_heun), ...
    log10(h(2:end)),log10(E_totb_rk3), log10(h(2:end)),log10(E_totb_rk4), 'linewidth', 1.5)
title('Error with regard to step size for beta')
xlabel('log_{10}(h)')
ylabel('log_{10}(Error)')
grid on
hold on
plot([log10(10^-6),log10(10^-1)],[log10(10^-6),log10(10^-6)], 'k--', 'linewidth', 1);
legend('Euler', 'Heun', 'Runge-Kutta 3rd order', 'Runge-Kutta 4th order', 'Maximum error')



%% H max
% maximum allowed error
maxerror=10^(-6);

%Find where the maximum error intercepts with the error function, and
%return the step size at that location
h_alpha_eul=interp1(E_tota_euler(2:end),h(3:end),maxerror);
h_beta_eul=interp1(E_totb_euler(2:end),h(3:end),maxerror);
h_alpha_heun=interp1(E_tota_heun,h(2:end),maxerror);
h_beta_heun=interp1(E_totb_heun,h(2:end),maxerror);
h_alpha_rk3=interp1(E_tota_rk3,h(2:end),maxerror);
h_beta_rk3=interp1(E_totb_rk3,h(2:end),maxerror);
h_alpha_rk4=interp1(E_tota_rk4,h(2:end),maxerror);
h_beta_rk4=interp1(E_totb_rk4,h(2:end),maxerror);

%% Find P for the 4 different methods

% Euler
Y_eul=diff(log10(E_tota_euler));
X_eul=diff(log10(h(2:end)));
avg_slope_euler=nanmean(Y_eul./X_eul);

% Heun
Y_heun=diff(log10(E_tota_heun));
X_heun=diff(log10(h(2:end)));
I_heun=find(Y_heun>0);
Y_heun(I_heun) = [];
X_heun(I_heun) = [];
avg_slope_heun=nanmean(Y_heun/X_heun);

% Runge-Kutta 3rd order
Y_rk3=diff(log10(E_tota_rk3));
X_rk3=diff(log10(h(2:end)));
I_rk3=find(Y_rk3>0);
Y_rk3(I_rk3) = [];
X_rk3(I_rk3) = [];
avg_slope_rk3=nanmean(Y_rk3/X_rk3);
```

```
% Runge-Kutta 4th order
Y_rk4=diff(log10(E_tota_rk4));
X_rk4=diff(log10(h(2:end)));
I_rk4=find(Y_rk4>0);
Y_rk4(I_rk4) = [];
X_rk4(I_rk4) = [];
avg_slope_rk4=nanmean(Y_rk4/X_rk4);


%% ODE
tspan = [0 T];
y0 = [pi/2; pi/2; 0; 0];
options = odeset('RelTol', 1e-16, 'AbsTol', 1e-6, 'Stats', 'on');
format long;
odefun = @qa2;
% ode23
[t23,y23] = ode23(odefun,tspan,y0,options);
alpha23=y23(end,1:2);
h23=T/length(t23);

% ode45
[t45,y45] = ode45(odefun,tspan,y0,options);
alpha45=y45(end,1:2);
h45=T/length(t45);

% ode113
[t113,y113] = ode113(odefun,tspan,y0,options);
alpha113=y113(end,1:2);
h113=T/length(t113);

%% Standard First-Order Form
function acc = qa(y)
alpha=y(1);
beta=y(2);
alphadot=y(3);
betadot=y(4);

acc=qddhw6(alpha,alphadot,betadot,beta);

acc = [y(3); y(4); acc];

end
% Function for ODE solver
function acc2 = qa2(t,y)
alpha=y(1);
beta=y(2);
alphadot=y(3);
betadot=y(4);

acc2=qddhw6(alpha,alphadot,betadot,beta);

acc2 = [y(3); y(4); acc2];

end
```