

## Multibody Dynamics B - Assignment 6

ME41055

Prof. Arend L. Schwab

Head TA: Simon vd. Helm

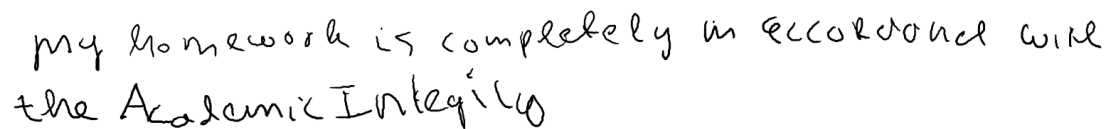
Rick Staa

#4511328

Lab Date: 26/04/2018

Due Date: 03/05/2018

## Statement of integrity



my homework is completely in accordance with  
the Academic Integrity

Figure 0.1: My handwritten statement of integrity

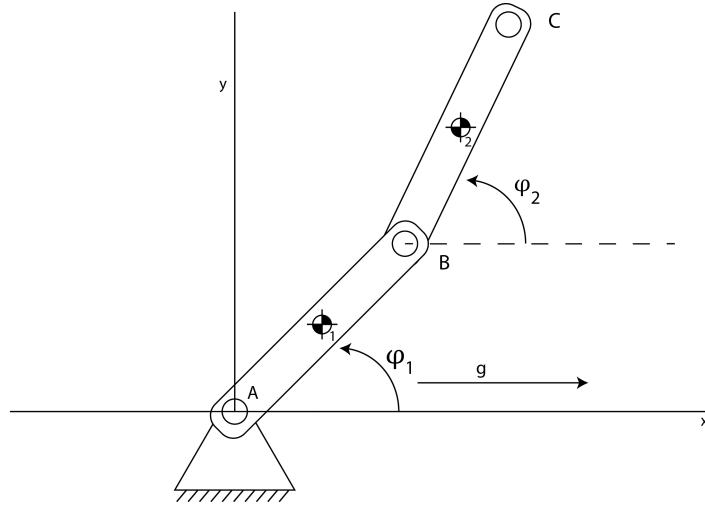
## Acknowledgements

I used [1] in making this assignment when finished I compared initial values with Prajish Kumar (4743873).

## Errata

Unfortunately the max error line in the figure is plotted wrong, as I noticed this too late I didn't have enough time to run the whole script again.

## Setup overview



**Figure 0.2:** Double pendulum used in this assignment. In this  $g$  depicts the gravity field and  $\varphi$  the angle of the bar with the horizontal.

## Problem Statement

In this assignment we were asked to determine the motion of the double pendulum from homework assignment 1 (see 0.2) by numerical integration of the equations of motion. While doing this we had to use the equations of motion as they were derived in homework assignment 4, meaning in terms independent generalized coordinates. For clarity the equations of motion from assignment 4 are depicted below:

### Equations of motion

In his method Lagrange makes use of the principle of energy to get the equations of motion (EOM). As explained in [1] these EOM can be derived out of the total derivative of the energy equation of the system. To do this we first need to define a potential energy function  $V$ :

$$\frac{\partial V}{\partial x} = -F \quad (0.1)$$

The energy equation of our system can be calculated by integrating the power over the time. The power of the system is equation to:

$$P = m\ddot{x}\dot{x} \quad (0.2)$$

So the energy of the system becomes:

$$\int F \dot{x} dt = \int m \ddot{x} \dot{x} dt \quad (0.3)$$

$$\int F dx = \int m \dot{x} d\dot{x} \quad (0.4)$$

Following we obtain the energy equation by evaluate the integrals. For the case that the forces are constant and conservative we get the following energy equation:

$$T + V = \text{constant} \quad (0.5)$$

From this we can see that the EOM can be derived by taking the total derivative of the energy equation. When extending this result for non-conservative forces we get the following total derivative:

$$\frac{d}{dt} \left( \frac{\partial T}{\partial \dot{x}_i} \right) - \frac{\partial V}{\partial x_i} = F_i \quad (0.6)$$

In this equation T contains the kinetic energy of the system while V contains the potential energy of the system and  $F_i$  depicts the energy of the non-conservative forces on the system. To make the resulting equations of motion more compact we can express this equation of motion in terms of our generalised coordinates q ( $\phi_1, \phi_2$ ). The new equation now becomes:

$$\frac{d}{dt} \left( \frac{\partial T}{\partial \dot{q}} \right) - \frac{\partial T}{\partial q_j} + \frac{\partial V}{\partial q_j} = Q_j \quad (0.7)$$

In this  $Q_j$  depicts the generalized forces. These generalised forces are the forces working on the body but now not acting on the COM but on the generalised coordinates. We now need to write this in a matrix vector product again to be able to solve for the unknown accelerations. We can do this by rewriting the first term with the multivariate chain rule:

$$\frac{d}{dt} \left( \frac{\partial T}{\partial \dot{q}} \right) = \frac{\partial}{\partial \dot{q}} \left( \frac{\partial T}{\partial \dot{q}} \right) \ddot{q} + \frac{\partial}{\partial q} \left( \frac{\partial T}{\partial \dot{q}} \right) \dot{q} \quad (0.8)$$

When we full this in in equation !!!Unresolved reference!!!(7) and rewrite the formula in a matrix vector product with the known terms at the right side we get:

$$\frac{\partial}{\partial \dot{q}} \left( \frac{\partial T}{\partial \dot{q}_i} \right) \ddot{q} = Q_i - \frac{\partial}{\partial q} \left( \frac{\partial T}{\partial \dot{q}_i} \right) \dot{q} - \frac{\partial V}{\partial q_i} + \frac{\partial T}{\partial q_i} \quad (0.9)$$

This results in the following matrix vector product:

$$M_{ij}\ddot{q}_j = F_i \quad (0.10)$$

Solving this matrix vector product gives us the accelerations in terms of the generalized coordinates. We still need to express the accelerations in terms of the COM coordinates. For this assignment the resulting generalized Equations of motion can be found in the accompanying MATLAB script (See appendix A). They unfortunately were too big to show them here.

## Goal

In this assignment the initial conditions of the bars were  $[\pi/2 \ \pi/2 \ 0 \ 0]$ , meaning both bars vertically up with zero speed. Further the gravitational field was said to work in the horizontal direction with a field strength of  $g = 9.81 \ [N/kg]$ . The end goal of this assignment was determining the angle, in radians, of both bars with respect to the horizontal axis after 3.0 seconds with a maximal absolute error of  $10^{-6}$  rad. While doing this we were asked to compare the truncation and ground-off error for 4 often used numerical methods:

- Euler's method

The Euler method uses a first order approximation to estimate the state on the next time step:

$$y_{n+1} = y_n + hf(t_n, y_n) \quad (0.11)$$

In this  $f(t_n, y_n)$  depicts the function for calculating the derivative.

- Heun's method

The Heun method can be thought of as an extension of the Euler method. Instead of using only the first order derivative at the start of the integration step now also the derivative at the end of the step is used. This method works by first performing one Euler step to get an estimate of the state at the end of the integration step  $y_{n+1}$ . Following the derivative at this end state  $y_{n+1}$  is calculated and is averaged with the derivative at the beginning of the step  $f(t_n, y_n)$ . The resulting derivative is then used to approximate the end state  $y_{n+1}$ . In formula form this becomes:

$$y_{n+1}^* = y_n + hf(t_n, y_n) \quad (0.12)$$

$$y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y_{n+1}^*)) \quad (0.13)$$

- Runge-Kutta method RK

Like the Heun method could be viewed as an extension of the Euler method the Runge-Kutta method can be viewed as a generalisation of Euler's method. The difference between the RK method and the euler heun method is that in the RK

method the integrant  $f(t_n, y_n)$  is evaluated multiple times per steps. In the RK3 and RK4 methods used here this is done 3 and 4 times respectively.

– 3rd order Runge-Kutta method (RK3)

$$k_1 = f(t_n, y_n) \quad (0.14)$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \quad (0.15)$$

$$k_3 = f(t_n + \frac{3h}{4}, y_n + \frac{3h}{4}k_2) \quad (0.16)$$

$$y_{n+1} = y_n + \frac{h}{9}(2k_1 + 3k_2 + k_3) \quad (0.17)$$

– 4th order Runge-Kutta method (RK4)

$$k_1 = f(t_n, y_n) \quad (0.18)$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \quad (0.19)$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \quad (0.20)$$

$$k_4 = f(t_n + h, y_n + hk_3) \quad (0.21)$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (0.22)$$

## Results

### Euler, Heun, RK3 and RK4

First let's examine the results of the "Euler", "Heun", "3rd order Runge-Kutta" and "4th order Runge-Kutta method". To do this the global error will be examined for different step sizes. The global error consists of both the method-inherent truncation error and the finite precision error [1] and can be approximated by the following formula:

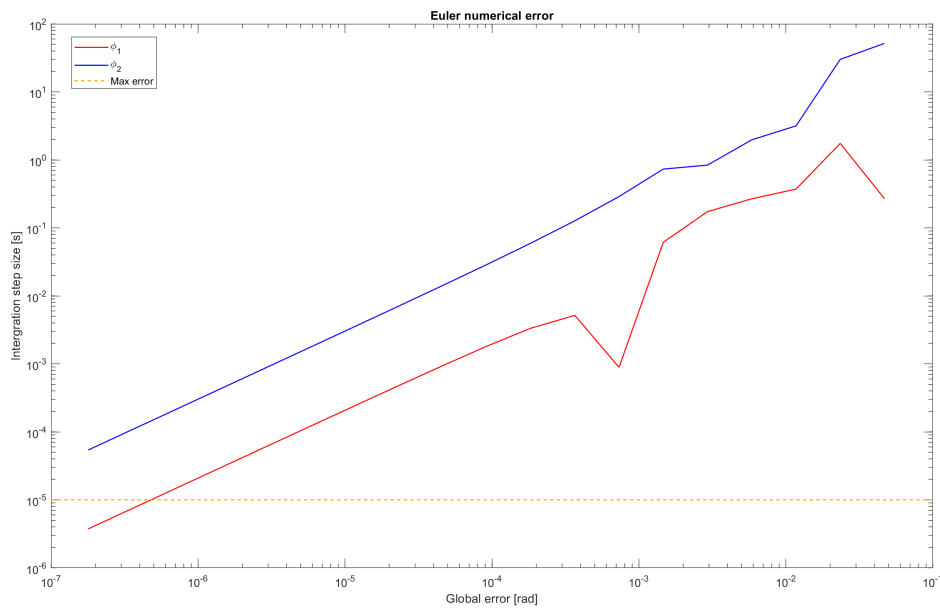
$$D_n = |y_n - y_{n+1}| \quad (0.23)$$

In this formula the values of the angles of the integration using the previous step size  $h$  are subtracted by the current integration values. While varying the stepsize  $h$  one should vary it according to the following formula [1]:

$$h = \frac{T}{2^n} \quad (0.24)$$

In which  $T$  is the full simulation time and the  $n$  is a value that is varied from 6:25. The result of this iteration of all the methods is shown below.

## Euler

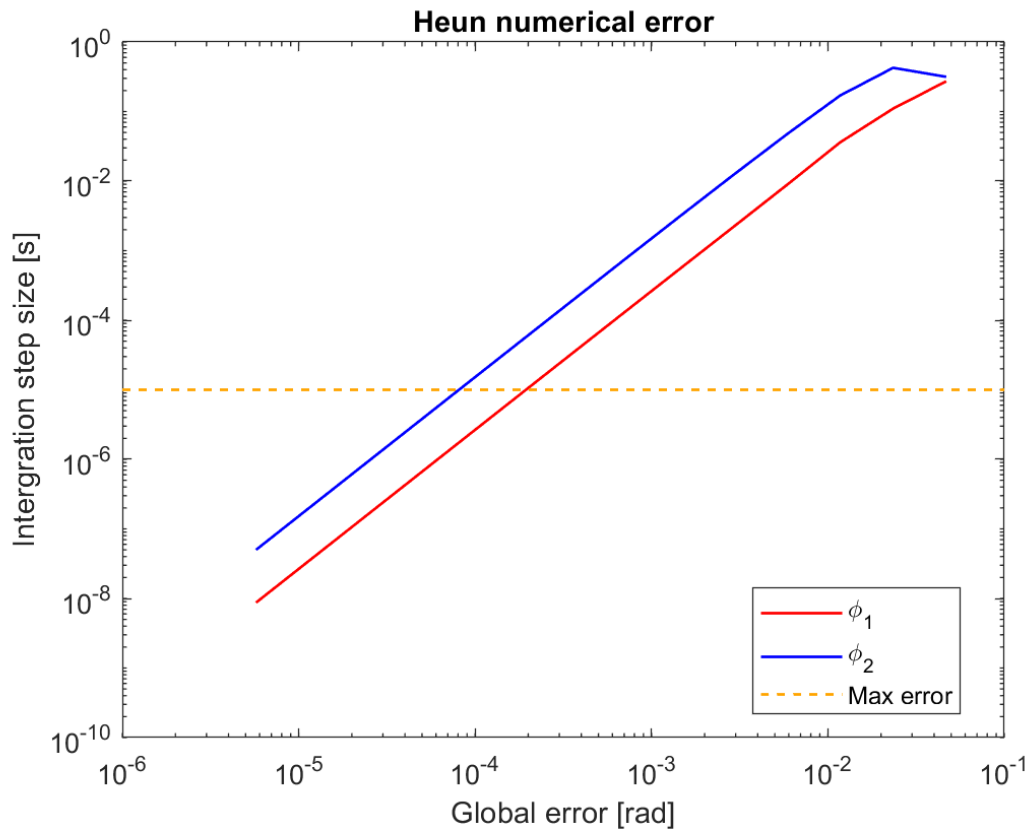


**Figure 0.3:** The global error [E] plotted on a log log scale vs the step size [h] for the Euler method.

From figure 0.3 we can see that with the Euler method we need a very small step size  $h \ll 9e * 10^{-8}s$  to obtain a max global error lower than  $10^{-6}$  for both the pendulum angles. Possibly this value can not even be obtained due to growing round-off errors. I did not examine smaller step sizes that  $h = 9e * 10^{-8}$  since it already took very long to run.

No units for h -0.5pt

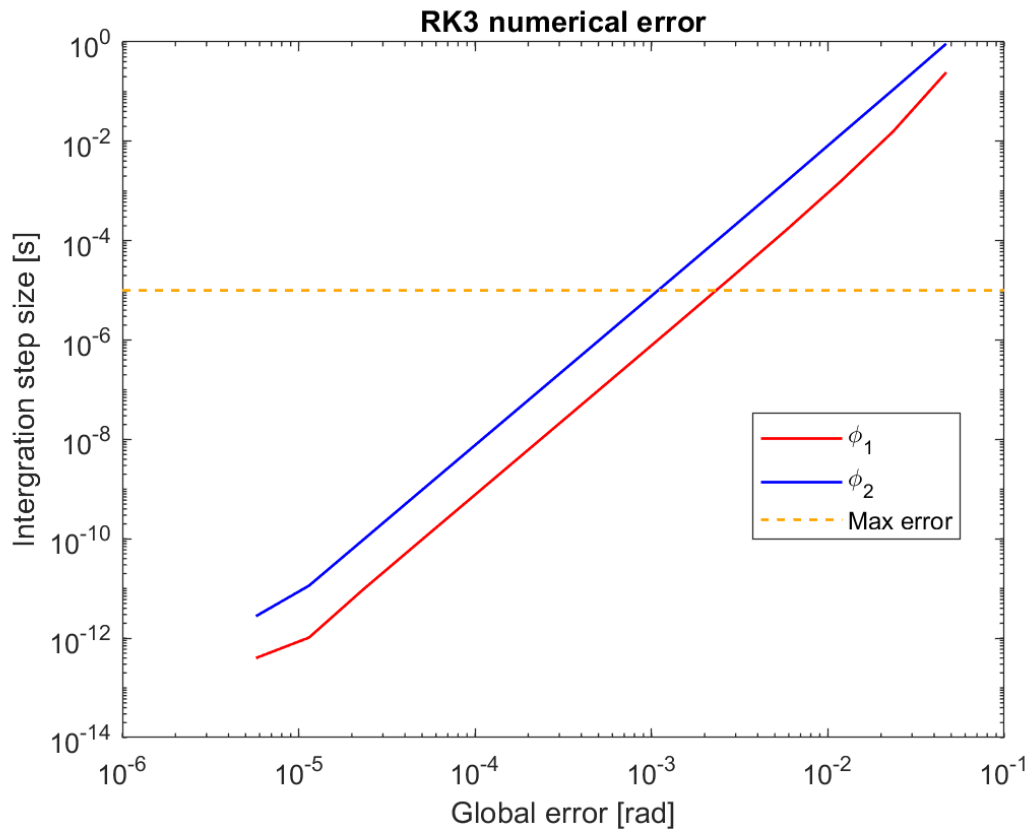
## Heun



**Figure 0.4:** The global error [E] plotted on a log log scale vs the step size [h] for the Heun method.

From figure 0.4 we can see that the Heun method obtains a maximal global error of  $10^{-6}$  with a bigger step size than the euler method. Due to the smaller needed step size less function evaluations are needed. This means that this method is preferred over the Euler method since it converges faster to an accurate end solution. Unfortunately I did not examine step sizes lower than  $h = 2.861022949218750 \cdot 10^{-6} s$  due to long run times. From the figure and the results we can see that a step size of  $h = 2.5 \cdot 10^{-5}$  already achieved the desired accuracy. We will use this step size in the comparison in the next question.

### 3rd order Runge-Kutta (RK3)



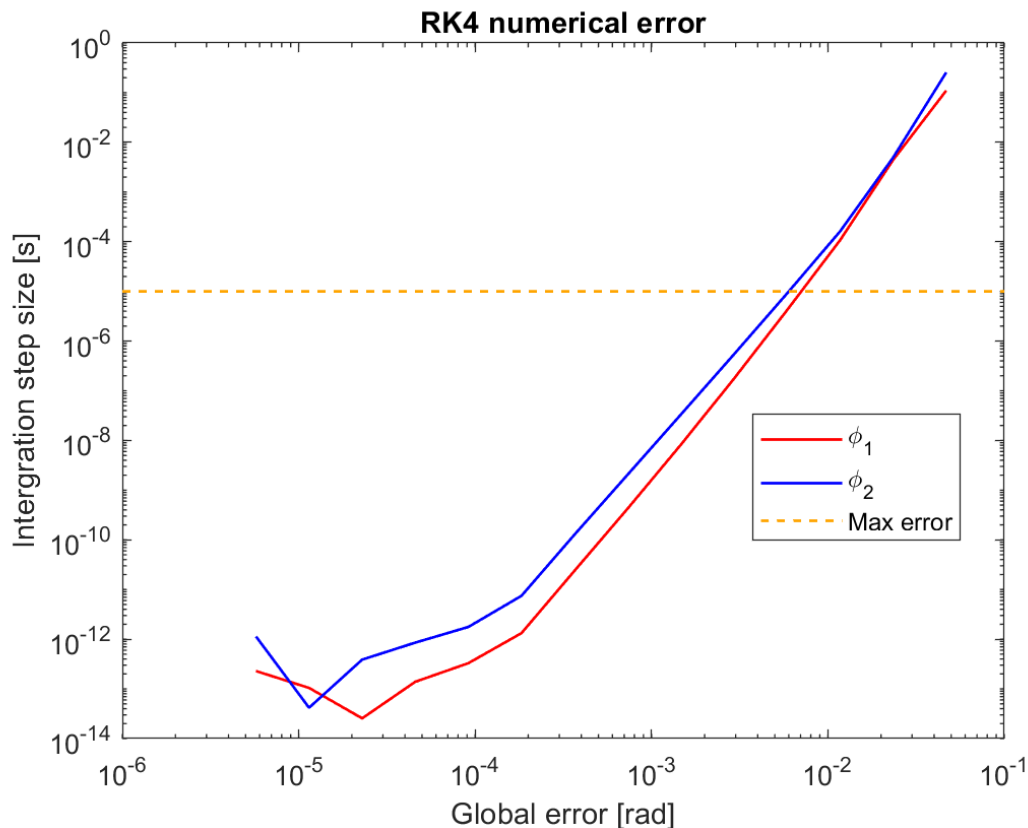
**Figure 0.5:** The global error [E] plotted on a log log scale vs the step size [h] for the 3rd order Runge-Kutta method.

From figure 0.5 we can see that the RK3 method obtains a maximal global error of  $10^{-6}$  with a bigger step size than both the Euler and Heun methods. Due to the smaller needed step size less function evaluations are needed. This means that this method is preferred over the earlier described methods since it converges faster to an accurate end solution. From the figure and the results we can see that a step size of  $h = 4 \cdot 10^{-4}$  already achieved the desired accuracy. We will use this step size in the comparison in the next question.

8



### 3rd order Runge-Kutta (RK4)



**Figure 0.6:** The global error [E] plotted on a log log scale vs the step size [h] for the 4th order Runge-Kutta method.

From figure ?? we can see that the RK4 method obtains a maximal global error of  $10^{-6}$  with a bigger step size than all the earlier described methods. Due to the smaller needed step size less function evaluations are needed. This means that this method is preferred over the earlier described methods since it converges faster to an accurate end solution. From the figure and the results we can see that a step size of  $h = 2 \cdot 10^{-3}$  already achieved the desired accuracy. We will use this step size in the comparison in the next question.

### MATLAB ODE solvers

In the last part of the assignment we were asked to examine some non-stiff ode solvers (ODE23, ODE45, ODE113). In which ODE 23 is the Runge-Kutta method with the lowest order and ODE114 the one with the highest order. I and see how they compare to the methods described above. The results are displayed in 1

8

	$\varphi_1$	$\varphi_2$	Function iterations	Mean step size [s]	Run time [s]
<b>Euler</b>	-1.211821 rad	-2.675981 rad	300001	1e-5	8.04 s
<b>Heun</b>	-1.212237 rad	-2.669923 rad	120001	2.5e-5	15.41 s
<b>RK3</b>	-1.212237 rad	-2.669924 rad	7501	4e-4	1.47 s
<b>RK4</b>	-1.212237 rad	-2.669925 rad	1501	2e-3	0.381
<b>ODE23</b>	-1.212252 rad	-2.669807 rad	4378	0.0021	0.28 s
<b>ODE45</b>	-1.212248 rad	-2.669863 rad	1033	0.0048	0.11 s
<b>ODE113</b>	-1.212231 rad	-2.669953 rad	798	0.0048	0.13s

Step size for ode113 should be smaller

**Table 1:** Overview of the results of all the numerical integration methods.

## Discussion

From the results above we can conclude the following things:

- The calculated angles at  $t = 3s$  are approximately equal for all the methods.
- The Euler method needs a very small step size to obtain a maximum global error lower than  $10^{-6}$ . For our example it is therefore not a good method to obtain fast convergence to a accurate solution.
- The Heun method needs more function evaluations than both the Runge-Kutta methods.
- Increasing the order of the Runge-Kutta method results in a lower number of needed function evaluations and a lower truncation error. This is caused since for the higher order methods as we see from [table ref] one should take into account the trade off between the number of steps taken and the overall execution time. In our example the ODE23 due to the large number of steps takes the longest time to run. However the runtime for the ODE113 is slightly higher than the ODE45. This could be caused by the fact that in the higher order solver more intermediate steps need to be taken which increases the computation time per function evaluation is increased. To really conclude this more examples need to be examined.
- The most obvious conclusion is that the by MATLAB supplied ODE solvers are faster and probably also more accurate than my own implementations. This mainly due to the fact that the ODE solvers in MATLAB use a variable step size and need less function evaluations while my implementations use a fixed step size and does need more function evaluations.

## Appendix A

### The main MATLAB script

```
1 %% MBD_B: Assignment 6 - Double pendulum numerical
  intergration
2 % Rick Staa (4511328)
3 % Last edit: 02/05/2018
4 % In this script I append the acceleration to the
  generalised state q so
5 % q = [phi1 phi2 phi1p phi2p phi1dp phi2dp]. This was done
  to save space.
6 clear all; close all; clc;
7 fprintf('--- A6 ---\n');
8
9 %% Set up needed symbolic parameters
10 % Create needed symbolic variables
11 syms phi1 phi2 phi1p phi2p
12
13 % Put in parms struct for easy function handling
14 parms.syms.phi1      = phi1;
15 parms.syms.phi2      = phi2;
16 parms.syms.phi1p     = phi1p;
17 parms.syms.phi2p     = phi2p;
18
19 %% Intergration parameters
20 time                 = 3;
21                      % Intergration time
22
23 parms.h              = 0.001;
24                      % Intergration step
25
26 size
27
28 %% Model Parameters
29 % Segment 1
30 parms.L              = 0.55;
31                      % [parms.m]
32
33 parms.w              = 0.05;
34                      % [parms.m]
35
36 parms.t              = 0.004;
37                      % [parms.m]
38
39 parms.p              = 1180;
40                      % [kg/parms.m^3]
```

```

29 | parms.m                                = parms.p * parms.w * parms.t *
    |     parms.L;          % [kg]
30 | parms.I                                = (1/12) * parms.m * parms.L^2;
    |         % [kg*parms.m^2]
31 |
32 | % World parameters
33 | parms.g                                = 9.81;
    |                                         % [parms.m/s^2]
34 |
35 | %% Initial state
36 | q0                                     = [0.5*pi 0.5*pi 0 0];
37 |
38 | %% Derive equation of motion
39 | [EOM_qdp] = EOM_calc(parms);
    |                                         % Calculate symbolic
    |     equations of motion and put in parms struct
40 |
41 | %% Calculate GLOBAL ERROR of the numerical intergration
    |     methods specified in the assignment
42 | % 1). Euler (Euler)
43 | % 2). Heun (Heun)
44 | % 3). Runge-Kutta 3th order (RK3)
45 | % 4). Runge-Kutta 4th order (RK4)
46 |
47 | tic
48 | %% Euler intergration
49 | % Calculate the error per step size for euler
50 |
51 | % Loop h and calculate global error
52 | n_range    = 6:1:25;
53 | h_range    = time./(2.^n_range);
54 | q_end_h_euler = zeros(length(h_range),6);
55 | for kk = 1:length(h_range)
56 |     parms.h                = h_range(kk);
57 |     [t,q]                  = ODE_custom(EOM_qdp,time,
    |         q0,'euler',parms);
58 |     % bar_animate(t,q,parms);
    |
    |     % Animate Bar
59 |     q_end_h_euler(kk,:)    = q(end,:);
60 | end
61 | glob_error                = abs(q_end_h_euler(2:end
    |         ,:)-q_end_h_euler(1:end-1,:)); % Calculate global error

```

```

62
63 % Create error plot
64 figure;
65 loglog(h_range(1:end-1),glob_error(:,1),'Color','red','
    LineWidth',1);hold on;
66 loglog(h_range(1:end-1),glob_error(:,2),'Color','blue','
    LineWidth',1);hold on;
67 line(xlim,[10e-6 10e-6],'Color',[1 0.6471 0],'LineStyle','--'
    , 'LineWidth',1);
68 legend('\phi_1','\phi_2','Max error','Location','Best');
69 title('Euler numerical error');
70 xlabel('Global error [rad]')
71 ylabel('Intergration step size [s]')
72
73 %% Heun intergration
74 % Calculate the error per step size for heun
75 % Calculate the error per step size for euler
76
77 % Loop h and calculate global error
78 n_range = 6:1:20;
79 h_range = time./(2.^n_range);
80 q_end_h_heun = zeros(length(h_range),6);
81 for kk = 1:length(h_range)
82     parms.h = h_range(kk);
83     [t,q] = ODE_custom(EOM_qdp,time,
        q0,'heun',parms);
84     % bar_animate(t,q,parms);
        % Animate Bar
85     q_end_h_heun(kk,:) = q(end,:);
86 end
87 glob_error_heun = abs(q_end_h_heun(2:end,:)-
    -q_end_h_heun(1:end-1,:)); % Calculate global error
88
89 % Create error plot
90 figure;
91 loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_heun(:,1))
    , 'Color','red','LineWidth',1);hold on;
92 loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_heun(:,2))
    , 'Color','blue','LineWidth',1);hold on;
93 line(xlim,[10e-6 10e-6],'Color',[1 0.6471 0],'LineStyle','--'
    , 'LineWidth',1);
94 legend('\phi_1','\phi_2','Max error','Location','Best');

```

```

95 title('Heun numerical error');
96 xlabel('Global error [rad]')
97 ylabel('Intergration step size [s]')
98
99 %% Runge-Kutta 3th order (RK3)
100 % Calculate the error per step size for RK3
101
102 % Loop h and calculate global error
103 n_range = 6:1:20;
104 h_range = time./(2.^n_range);
105 q_end_h_RK3 = zeros(length(h_range),6);
106 for kk = 1:length(h_range)
107     parms.h = h_range(kk);
108     [t,q] = ODE_custom(EOM_qdp,time,
109         q0,'RK3',parms);
110     % bar_animate(t,q,parms);
111
112     % Animate Bar
113     q_end_h_RK3(kk,:) = q(end,:);
114 end
115 glob_error_RK3 = abs(q_end_h_RK3(2:end,:)-
116     q_end_h_RK3(1:end-1,:)); % Calculate global error
117
118 % Create error plot
119 figure;
120 loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_RK3(:,1)'),
121     'Color','red','LineWidth',1);hold on;
122 loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_RK3(:,2)'),
123     'Color','blue','LineWidth',1);hold on;
124 line(xlim,[10e-6 10e-6],'Color',[1 0.6471 0],'LineStyle','--',
125     'LineWidth',1);
126 legend('\phi_1','\phi_2','Max error','Location','Best');
127 title('RK3 numerical error');
128 xlabel('Global error [rad]')
129 ylabel('Intergration step size [s]')
130
131 %% Runge-Kutta 4th order (RK4)
132 % Calclate the error per step size for RK4
133
134 % Loop h and calculate global error
135 n_range = 6:1:20;
136 h_range = time./(2.^n_range);
137 q_end_h_RK4 = zeros(length(h_range),6);

```

```

131 for kk = 1:length(h_range)
132     parms.h = h_range(kk);
133     [t,q] = ODE_custom(EOM_qdp,time,
134         q0,'RK4',parms);
135     % bar_animate(t,q,parms);
136
137     % Animate Bar
138     q_end_h_RK4(kk,:) = q(end,:);
139 end
140 glob_error_RK4 = abs(q_end_h_RK4(2:end,:)-
141     q_end_h_RK4(1:end-1,:)); % Calculate global error
142
143 % Create error plot
144 figure;
145 loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_RK4(:,1)'),
146     'Color','red','LineWidth',1);hold on;
147 loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_RK4(:,2)'),
148     'Color','blue','LineWidth',1);hold on;
149 line(xlim,[10e-6 10e-6],'Color',[1 0.6471 0],'LineStyle','--',
150     'LineWidth',1);
151 legend('\phi_1','\phi_2','Max error','Location','Best');
152 title('RK4 numerical error');
153 xlabel('Global error [rad]')
154 ylabel('Intergration step size [s]')
155 toc;
156
157 %% Perform methods at maxstepsize
158 %% Euler method at step-size 1e-5
159 tic
160 parms.h = 1e-5;
161 [t_euler,q_euler] = ODE_custom(
162     EOM_qdp,time,q0,'euler',parms);
163 toc
164
165 %% Heun method at step-size 1e-5
166 tic
167 parms.h = 2.5e-5;
168 [t_heun,q_heun] = ODE_custom(EOM_qdp,
169     time,q0,'heun',parms);
170 toc
171
172 %% Runge-Kutta 3th order (RK4)
173 tic

```

```

165 | parms.h = 4e-4;
166 | [t_RK3,q_RK3] = ODE_custom(EOM_qdp,
    |     time,q0,'RK3',parms);
167 | toc
168 |
169 | %% Runge-Kutta 4th order (RK4)
170 | tic
171 | parms.h = 2e-3;
172 | [t_RK4,q_RK4] = ODE_custom(EOM_qdp,
    |     time,q0,'RK4',parms);
173 | toc
174 |
175 | %% Calculate motion with ODE functions
176 | % ODE 23
177 | tic
178 | opt = odeset('AbsTol',1e-6,'RelTol',1e-6,'Stats','on');
179 | [t23,q23] = ode23(@(t,q) ODE_func(t,q,EOM_qdp), [0 time], q0
    |     ',opt);
180 | t23_mean = mean(diff(t23));
    |                                     % Caculate mean
    |     step size
181 | disp(t23_mean);
182 | toc
183 |
184 | % ODE 45
185 | tic
186 | opt = odeset('AbsTol',1e-6,'RelTol',1e-6,'Stats','on');
187 | [t45,q45] = ode45(@(t,q) ODE_func(t,q,EOM_qdp), [0 time], q0
    |     ',opt);
188 | t45_mean = mean(diff(t45));
    |                                     % Caculate mean
    |     step size
189 | disp(t45_mean);
190 | toc
191 |
192 | % ODE 113
193 | tic
194 | opt = odeset('AbsTol',1e-6,'RelTol',1e-6,'Stats','on');
195 | [t113,q113] = ode113(@(t,q) ODE_func(t,q,EOM_qdp), [0 time],
    |     q0',opt);
196 | t113_mean = mean(diff(t45));
    |                                     % Caculate mean
    |     step size

```



```

197 disp(t113_mean);
198 toc
199
200 %% FUNCTIONS
201
202 %% Bar animate
203 % This function creates a movie of the double pendulum. I did
    not find a
204 % way to do this with the real speed but it gives a nice
    impression of what
205 % is happening.
206 function bar_animate(t,q,parms)
207 figure;
208 h=plot(0,0,'MarkerSize',20,'Marker','.','LineWidth',2);
209 range=1.1*(parms.L+parms.L); axis([-range range -range range
    ]); axis square;
210 set(gca,'nextplot','replacechildren');
211 a = tic;
212 for jj=1:length(q)-1
213     if (ishandle(h)==1)
214         tic
215         phi1 = q(jj,1);
216         phi2 = q(jj,2);
217         Xcoord=[0,parms.L*cos(phi1),parms.L*cos(phi1)+parms.L
            *cos(phi2)];
218         Ycoord=[0,parms.L*sin(phi1),parms.L*sin(phi1)+parms.L
            *sin(phi2)];
219         set(h,'XData',Xcoord,'YData',Ycoord);
220         b = toc(a); % check timer
221         if b > (1/30)
222             drawnow % update screen every 1/30 seconds
223             a = tic; % reset timer after updating
224         end
225         %
            pause(t(jj+1)-t(jj));
            % Realtime
226         toc;
227     end
228 end
229 drawnow
230 end
231
232 %% ODE Function handle
233 function [qdp] = ODE_func(t,q,EOM_qdp)

```

```

234 q_now = num2cell(q',1);
235 qdp   = feval(EOM_qdp,q_now{:});
236 qdp   = [q(3);q(4);qdp];
237 end
238
239 %% Euler numerical intergration function
240 % This function calculates the motion of the system by means
    of a euler
241 % numerical intergration. This function takes as inputs the
    parameters of
242 % the system (parms), the EOM of the system (parms.EOM) and
    the initial
243 % state.
244 function [t,q] = ODE_custom(EOM,time,q0,method,parms)
245
246 % Initialise variables
247 t           = (0:parms.h:time).';
                                % Create time array
248 q           = zeros(length(t),6);
                                % Create empty state array
249 q(1,1:size(q0,2)) = q0;
                                % Put
                                initial state in array
250
251 % Caculate the motion for the full time by means of the 4
    different
252 % numerical intergration methods
253 % 1). Euler
254 % 2). Heun
255 % 3). Runge-Kutta 3th order
256 % 4). Runge-Kutta 4th order
257 % See report for the Workings of each method.
258
259 % Euler method
260 switch method
261
262     %% Euler method
263     case 'euler'
264
265         % Perform the full intergration with eulers method
266         for ii = 1:(size(t,1)-1)
267             q_now_tmp       = num2cell(q(ii,1:end-2),1);
                                % Create cell for feval

```

```

268         function
269         qdp = feval(EOM,q_now_tmp{:}).';
270             % Calculate the second
271             derivative of the generalised coordinates
272         q(ii,end-1:end) = qdp;
273         q(ii+1,1:end-2) = q(ii,1:end-2) + parms.h*q(ii,3:
274             end); % Perform euler intergration step
275
276         % Calculate last acceleration
277         if ii == (size(t,1)-1)
278             q_next = num2cell(q(ii+1,1:end-2)
279                 ,1); % Create cell for feval
280             function
281                 q(ii+1,end-1:end) = feval(EOM,q_next{:}).';
282                 % Calculate the second
283                 derivative of the last step
284         end
285     end
286
287     %% Heun method
288     case 'heun'
289
290         % Perform the full intergration with eulers method
291         for ii = 1:(size(t,1)-1)
292             % Step 1: Approximate the next state
293             q_now = [q(ii,1:end-2) 0 0];
294             % Read out current
295             states
296             q_now_tmp = num2cell(q_now,1);
297             % Create cell
298             for feval function
299                 qdp_now_tmp = feval(EOM,q_now_tmp{1:end-2})
300                 .'; % Calculate the second
301                 derivative of the generalised coordinates
302                 qdp_now = [cell2mat(q_now_tmp(end-3:end
303                     -2)),qdp_now_tmp]; % Add first derivative
304                 q_now(end-3:end)= qdp_now;
305                 q_star = q(ii,1:end-2) + parms.h*q_now
306                 (3:end); % Make a approximation of
307                 the next state by means of a euler step
308
309             % Step 2: Calculate the state derivative at next
310             state

```

```

293     q_star_tmp      = num2cell(q_star,1);
                                % Create cell for
                                feval function
294     qdp_star_tmp    = feval(EOM,q_star_tmp{:}).';
                                % Calculate the second
                                derivative of the generalised coordinates
295     qdp_star        = [cell2mat(q_star_tmp(end-1:end)
                                ),qdp_star_tmp];    % Add first derivative
296
297     % Step3: Calculate the state at the next step
                using the mean
298     % derivative.
299     q(ii+1,1:end-2) = q(ii,1:end-2) + (parms.h*0.5)*(
                qdp_now+qdp_star);    % Calculate
                state of next step (I use both the approximated
                new velocity and acceleration)
300
301     % Calculate last acceleration
302     if ii == (size(t,1)-1)
303         q_next_tmp      = num2cell(q(ii+1,1:end-2)
                ,1);    % Create cell for feval
                function
304         q(ii+1,end-1:end) = feval(EOM,q_next_tmp{:})
                .';    % Calculate the second
                derivative of the last step
305     end
306 end
307
308 %% Runge-Kutta 3th order
309 case 'RK3'
310     for ii = 1:(size(t,1)-1)
311         q_now_tmp      = num2cell(q(ii,1:end-2),1);
312
                % Create cell for feval function
313         K1              = [cell2mat(q_now_tmp(1,end-1:
                end)),feval(EOM,q_now_tmp{:}).'];    %
                Calculate the second derivative at the start
                of the step
314         q1_tmp          = num2cell(cell2mat(q_now_tmp)
                + (parms.h*0.5)*K1);    %
                Create cell for feval function
315         K2              = [cell2mat(q1_tmp(1,end-1:end)
                ),feval(EOM,q1_tmp{:}).'];    %

```

```

    Calculate the second derivative halfway the
    step
315 q2_tmp = num2cell(cell2mat(q_now_tmp)
    + ((parms.h*0.75))*K2); %
    Refine value calculation with new found
    derivative
316 K3 = [cell2mat(q2_tmp(1,end-1:end)
    ),feval(EOM,q2_tmp{:}).']; %
    Calculate new derivative at the new refined
    location
317 q(ii,end-1:end) = (1/9)*(2*K1(3:4)+3*K2(1:2)+4*
    K3(3:4));

    % Take weighted sum of K1, K2, K3
318 q(ii+1,1:end-2) = cell2mat(q_now_tmp) + (parms.
    h/9)*(2*K1+3*K2+4*K3); %
    Perform euler intergration step

319
320 % Calculate last acceleration
321 if ii == (size(t,1)-1)
322     q_now_tmp = num2cell(q(ii+1,1:end-2)
        ,1);

        % Create cell for feval function
323 K1 = [cell2mat(q_now_tmp(1,end
    -1:end)),feval(EOM,q_now_tmp{:}).'];
        % Calculate the second
        derivative at the start of the step
324 q1_tmp = num2cell(cell2mat(
    q_now_tmp) + (parms.h*0.5)*K1);
        % Create cell for
        feval function
325 K2 = [cell2mat(q1_tmp(1,end-1:
    end)),feval(EOM,q1_tmp{:}).'];
        % Calculate the second
        derivative halfway the step
326 q2_tmp = num2cell(cell2mat(
    q_now_tmp) + ((parms.h*0.75))*K2);
        % Refine value
        calculation with new found derivative
327 K3 = [cell2mat(q2_tmp(1,end-1:
    end)),feval(EOM,q2_tmp{:}).'];
        % Calculate new

```

```

328         derivative at the new refined location
q(ii+1,end-1:end) = (1/9)*(2*K1(3:4)+3*K2
    (3:4)+4*K3(3:4));
                                                    % Take
        weighted sum of K1, K2, K3

        % Take weighted sum of K1, K2, K3
329     end
330 end
331
332 %% Runge-Kutta 4th order
333 case 'RK4'
334     for ii = 1:(size(t,1)-1)
335         q_now_tmp = num2cell(q(ii,1:end-2),1);

        % Create cell for feval function
336         K1 = [cell2mat(q_now_tmp(1,end-1:
end)),feval(EOM,q_now_tmp{:}).']; %
        Calculate the second derivative at the start
        of the step
337         q1_tmp = num2cell(cell2mat(q_now_tmp)
+ (parms.h*0.5)*K1); %
        Create cell for feval function
338         K2 = [cell2mat(q1_tmp(1,end-1:end)
),feval(EOM,q1_tmp{:}).']; %
        Calculate the second derivative halfway the
        step
339         q2_tmp = num2cell(cell2mat(q_now_tmp)
+ (parms.h*0.5)*K2); %
        Refine value calculation with new found
        derivative
340         K3 = [cell2mat(q2_tmp(1,end-1:end)
),feval(EOM,q2_tmp{:}).']; %
        Calculate new derivative at the new refined
        location
341         q3_tmp = num2cell(cell2mat(q_now_tmp)
+ (parms.h)*K3); %
        Calculate state at end step with refined
        derivative
342         K4 = [cell2mat(q3_tmp(1,end-1:end)
),feval(EOM,q3_tmp{:}).']; %
        Calculate last second derivative

```

```

343 q(ii,end-1:end)    = (1/6)*(K1(3:4)+2*K2(3:4)+2*K3
      (3:4)+K4(3:4)); %
      Take weighted sum of K1, K2, K3
344 q(ii+1,1:end-2)   = cell2mat(q_now_tmp) + (parms.
      h/6)*(K1+2*K2+2*K3+K4); %
      Perform euler intergration step

345 % Calculate last acceleration
346 if ii == (size(t,1)-1)
347     q_now_tmp      = num2cell(q(ii+1,1:end-2)
348         ,1);

      % Create cell for feval function
349 K1                = [cell2mat(q_now_tmp(1,end
      -1:end)),feval(EOM,q_now_tmp{:}).'];
      % Calculate the second
      derivative at the start of the step
350 q1_tmp            = num2cell(cell2mat(
      q_now_tmp) + (parms.h*0.5)*K1);
      % Create cell for
      feval function
351 K2                = [cell2mat(q1_tmp(1,end-1:
      end)),feval(EOM,q1_tmp{:}).'];
      % Calculate the second
      derivative halfway the step
352 q2_tmp            = num2cell(cell2mat(
      q_now_tmp) + (parms.h*0.5)*K2);
      % Refine value
      calculation with new found derivative
353 K3                = [cell2mat(q2_tmp(1,end-1:
      end)),feval(EOM,q2_tmp{:}).'];
      % Calculate new
      derivative at the new refined location
354 q3_tmp            = num2cell(cell2mat(
      q_now_tmp) + (parms.h)*K3);
      % Calculate
      state at end step with refined derivative
355 K4                = [cell2mat(q3_tmp(1,end-1:
      end)),feval(EOM,q3_tmp{:}).'];
      % Calculate last second
      derivative
356 q(ii,end-1:end)    = (1/6)*(K1(3:4)+2*K2(3:4)
      +2*K3(3:4)+K4(3:4));

```

```

357                                     % Take
                                     weighted sum of K1, K2, K3
358                                     end
359 end
360 end
361
362 %% Calculate (symbolic) Equations of Motion four our setup
363 function [qdp_handle] = EOM_calc(parms)
364
365 % Unpack symbolic variables from varargin
366 phi1          = parms.syms.phi1;
367 phi2          = parms.syms.phi2;
368 phi1p         = parms.syms.phi1p;
369 phi2p         = parms.syms.phi2p;
370
371 % Create generalized coordinate vectors
372 q             = [phi1; phi2];
373 qp            = [phi1p; phi2p];
374
375 % COM of the bodies expressed in generalised coordinates
376 x1            = (parms.L/2)*cos(phi1);
377 y1            = (parms.L/2)*sin(phi1);
378 x2            = x1 + (parms.L/2)*cos(phi1) + (parms.L/2) *
379               cos(phi2);
380 y2            = y1 + (parms.L/2)*sin(phi1) + (parms.L/2) *
381               sin(phi2);
382
383 % Calculate derivative of COM expressed in generalised
384   coordinates (We need this for the energy equation)
385 x             = [x1;y1;phi1;x2;y2;phi2];
386 Jx_q          = simplify(jacobian(x,q));
387 xp            = Jx_q*qp;
388
389 %% Compute energies
390 T             = 0.5*xp.'*diag([parms.m;parms.m;parms.I;
391                               parms.m;parms.m;parms.I])*xp; % Kinetic energy
392 V             = -([parms.m*parms.g 0 0 parms.m*parms.g 0
393                   0]*x); % Potential energy
394
395 %% Calculate the terms of the jacobian
396 Q             = 0; % Non-
397               conservative forces

```



```

392
393 % Partial derivatives of Kinetic energy
394 T_q          = simplify(jacobian(T,q));
395 T_qp         = simplify(jacobian(T,qp));
396 T_qpqp       = simplify(jacobian(T_qp,qp));
397 T_qpq        = simplify(jacobian(T_qp,q));
398
399 % Partial derivatives of Potential energy
400 V_q          = simplify(jacobian(V,q));
401 V_qp         = simplify(jacobian(V,qp));
402 V_qpqp       = simplify(jacobian(V_qp,qp));
403
404 % Make matrix vector product
405 M            = T_qpqp;
406 F            = Q + T_q.' - V_q.' - T_qpq*qp;
407
408 % Solve Mqdp=F to get the accelerations
409 qdp          = M\F;
410
411 %% Get back to COM coordinates
412 % xdp        = simplify(jacobian(xp,qp))*qdp+simplify(
    jacobian(xp,q))*qp;
413
414 %% Convert to function handles
415 % xdp_handle  = matlabFunction(xdp);
    % Create function handle
    of EOM in terms of COM positions
416 qdp_handle   = matlabFunction(qdp);
    % Create function handle
    of EOM in terms of generalised coordinates
417 % matlabFunction(qdp,'file',qdp_cal')
418 end

```

## References

- [1] Arend L. Schwab. Reader: MultiBody Dynamics B. In *Multibody Dynamics*, chapter 3. TU Delft, Delft, The Netherlands, 2018.