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 1: My handwritten statement of integrity

Acknoledgements

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

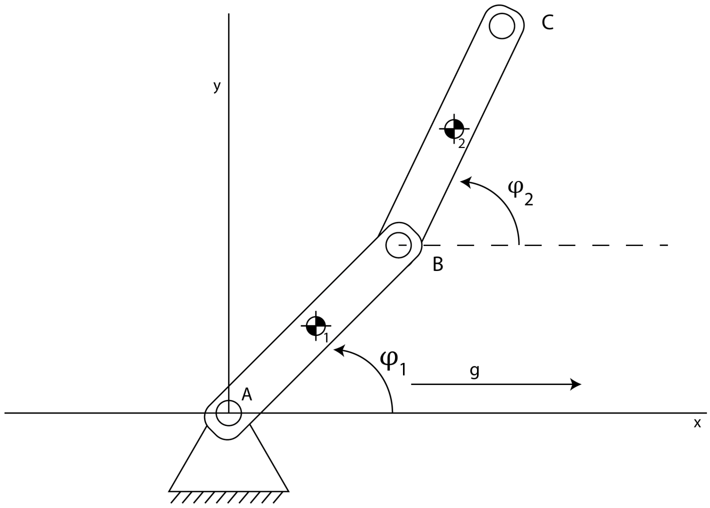

Figure 2: Double pendulum used in this assignment. In this g depicts the gravity field and φ the angle of the bar with the horizontal.

Problem Statement

In this assignment we were asked to determine the motion of the double pendulum from homework assignment 1 (see figure 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:

$$\frac{\partial V}{\partial x} = -F \tag{1}$$

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

$$P = m\ddot{x}\dot{x} \tag{2}$$

So the energy of the system becomes:

$$\int F\dot{x}dt = \int m\ddot{x}\dot{x}dt \tag{3}$$

$$\int Fdx = \int m\dot{x}d\dot{x} \tag{4}$$

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:

$$T + V = constant \tag{5}$$

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:

$$\frac{d}{dt}(\frac{\partial T}{\partial \dot{x}_i}) - \frac{\partial V}{\partial x_i} = F_i \tag{6}$$

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 (ϕ_1, ϕ_2). The new equation now becomes:

$$\frac{d}{dt}(\frac{\partial T}{\partial \dot{q}}) - \frac{\partial T}{\partial q_j} + \frac{\partial V}{\partial q_j} = Q_j \tag{7}$$

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:

$$\frac{d}{dt}(\frac{\partial T}{\partial \dot{q}}) = \frac{\partial}{\partial \dot{q}}(\frac{\partial T}{\partial \dot{q}})\ddot{q} + \frac{\partial}{\partial q}(\frac{\partial T}{\partial \dot{q}_i})\dot{q} \tag{8}$$

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:

$$\frac{\partial}{\partial \dot{q}}(\frac{\partial T}{\partial \dot{q}_i})\ddot{q} = Qi - \frac{\partial}{\partial q}(\frac{\partial T}{\partial \dot{q}_i})\dot{q} - \frac{\partial V}{\partial q_i} + \frac{\partial T}{\partial q_i} \tag{9}$$

This results in the following matrix vector product:

$$M_{ij}\ddot{q}_j = F_i \tag{10}$$

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 to big to show them here.

Goal

In this assignment the initial conditions of the bars were $\begin{bmatrix}\pi/2 & \pi/2 & 0 & 0\end{bmatrix}$, meaning be both bars vertically up with zero speed. Further the gravitational field was said to work in the horizontal direction with a field strength of $g = 9.81 \; \left[N/kg\right]$. 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 were 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:
  $$y_{n+1} = y_n + hf(t_n, y_n) \tag{11}$$
  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:
  $$y^*_{n+1} = y_n + hf(t_n, y_n) \tag{12}$$
  $$y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y^*_{n+1})) \tag{13}$$
- Runge-Kutta method RK
  Like the Heun method could be viewed as a 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 oder Runge-Kutta method (RK3)

$$k_1 = f(t_n, y_n) \tag{14}$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \tag{15}$$

$$k_3 = f(t_n + \frac{3h}{4}, y_n + \frac{3h}{4}k_2) \tag{16}$$

$$y_{n+1} = y_n + \frac{h}{9}(2k_1 + 3k_2 4 + k_3) \tag{17}$$

– 4rd oder Runge-Kutta method (RK4)

$$k_1 = f(t_n, y_n) \tag{18}$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \tag{19}$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \tag{20}$$

$$k_4 = f(t_n + h, y_n + hk_3) \tag{21}$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{22}$$

Results

Euler, Heun, RK3 adn RK4

First lets 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:

$$D_n = |y_n - y_{n+1}| \tag{23}$$

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]:

$$h = \frac{T}{2^n} \tag{24}$$

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

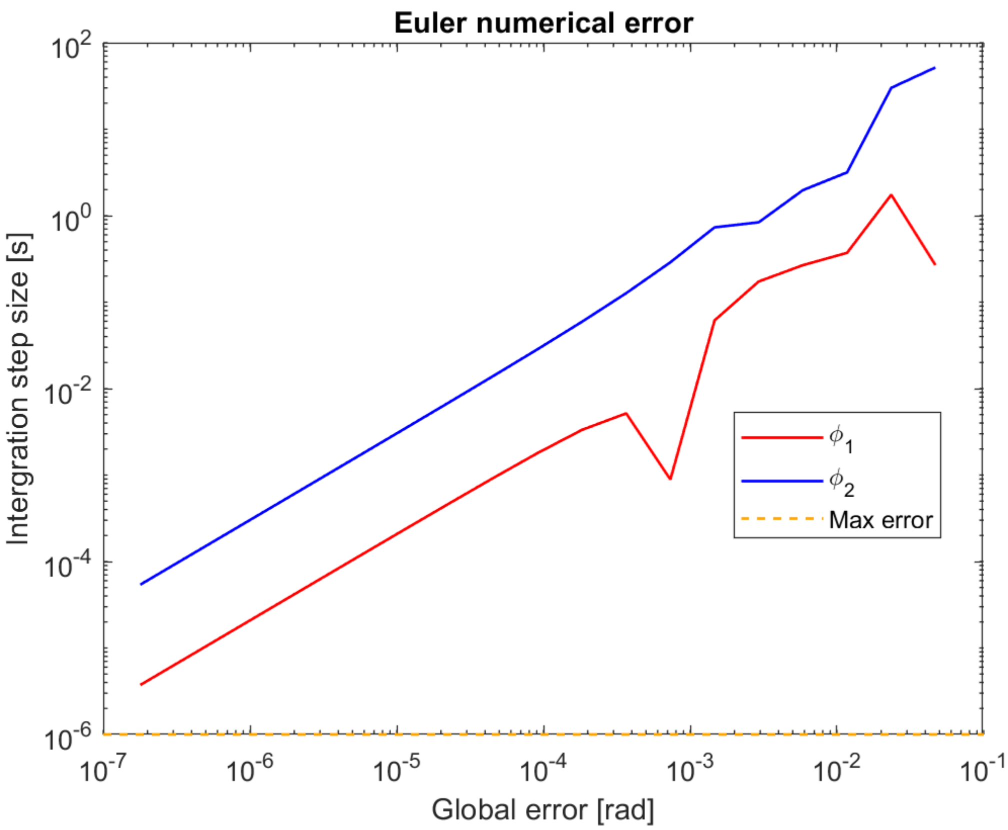

Figure 3: The global error [E] plotted on a log log scale vs the step size [h] for the Euler method.

From figure 3 we can see that with the Euler method we need a very small step size $h << 9 * 10^{-8}$ s to obtain a max global error lower than $10^{-6}s$ 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 $h = 9 * 10^{-8}$ s since it already took very long to run.

Heun

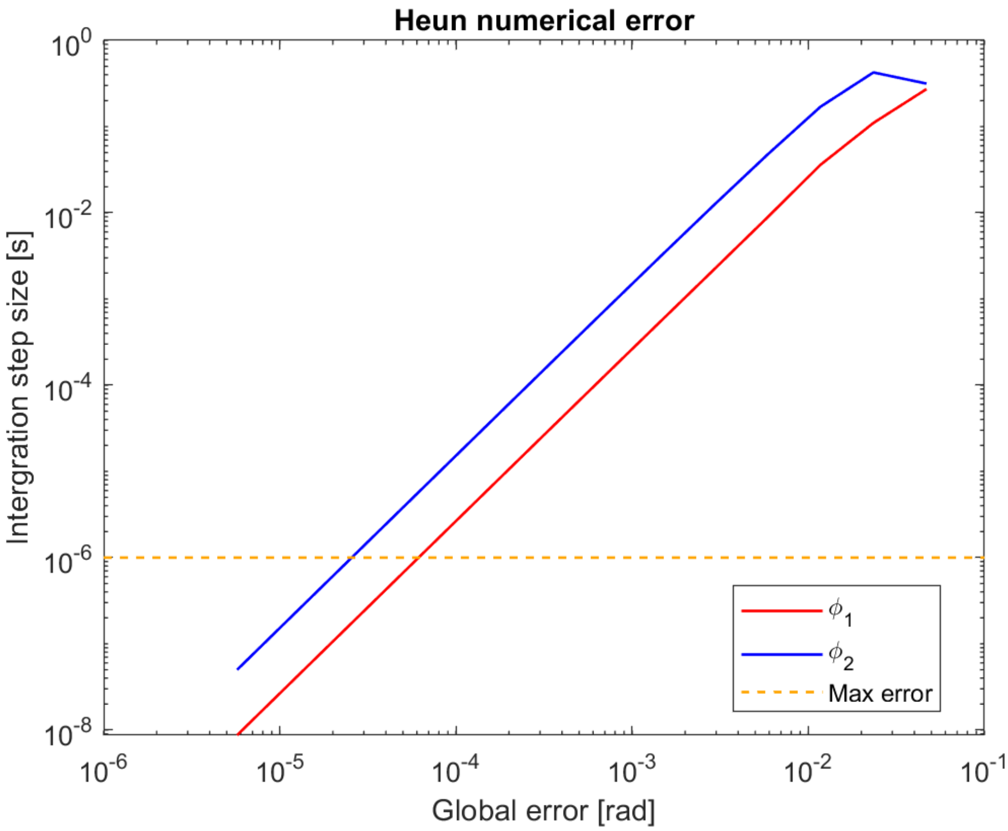

Figure 4: The global error [E] plotted on a log log scale vs the step size [h] for the Heun method.

From figure 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 $h = 2.861022949218750 * 10^{-6} s$ due to long run times. From the figure and the results we can see that a step size of $h = 2.5 * 10^{-5}$ s already achieved the desired accuracy. We will use this step size in the comparison in the next question.

3rd order Runge-Kutta (RK3)

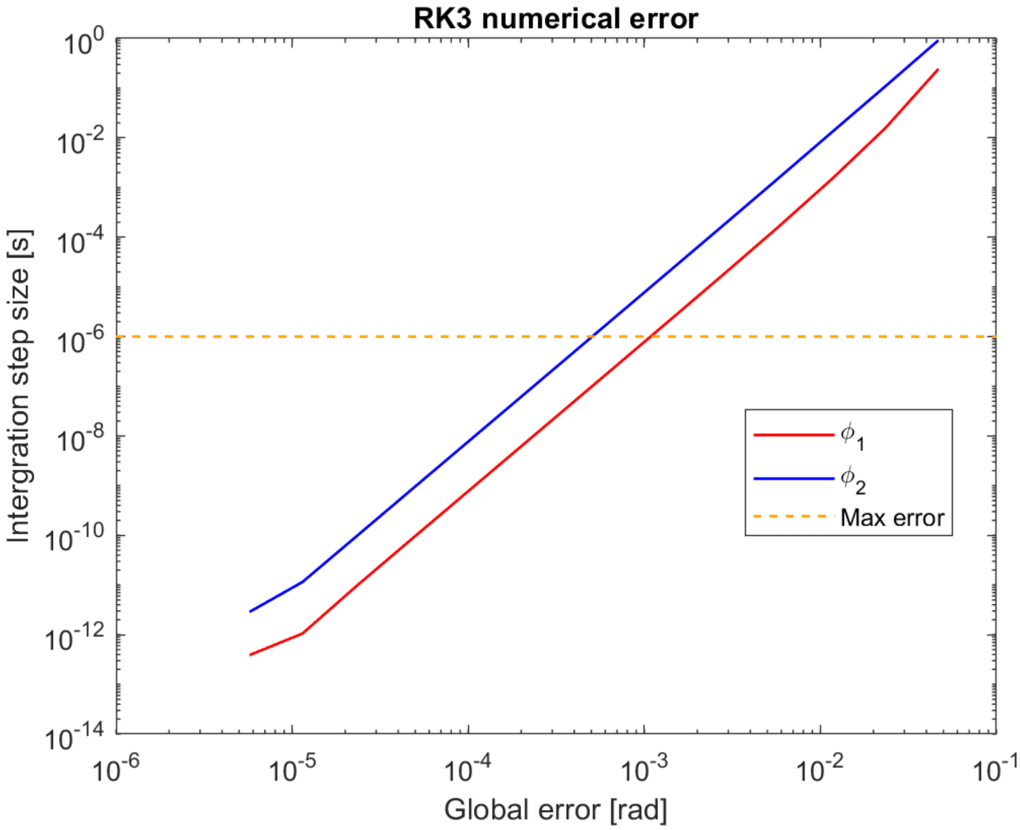

Figure 5: The global error [E] plotted on a log log scale vs the step size [h] for the 3rd order Runge-Kutta method.

From figure 5 we can see that the RK3 method n 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 $h = 4 * 10^{-4}$ s already achieved the desired accuracy. We will use this step size in the comparison in the next question.

4th order Runge-Kutta (RK4)

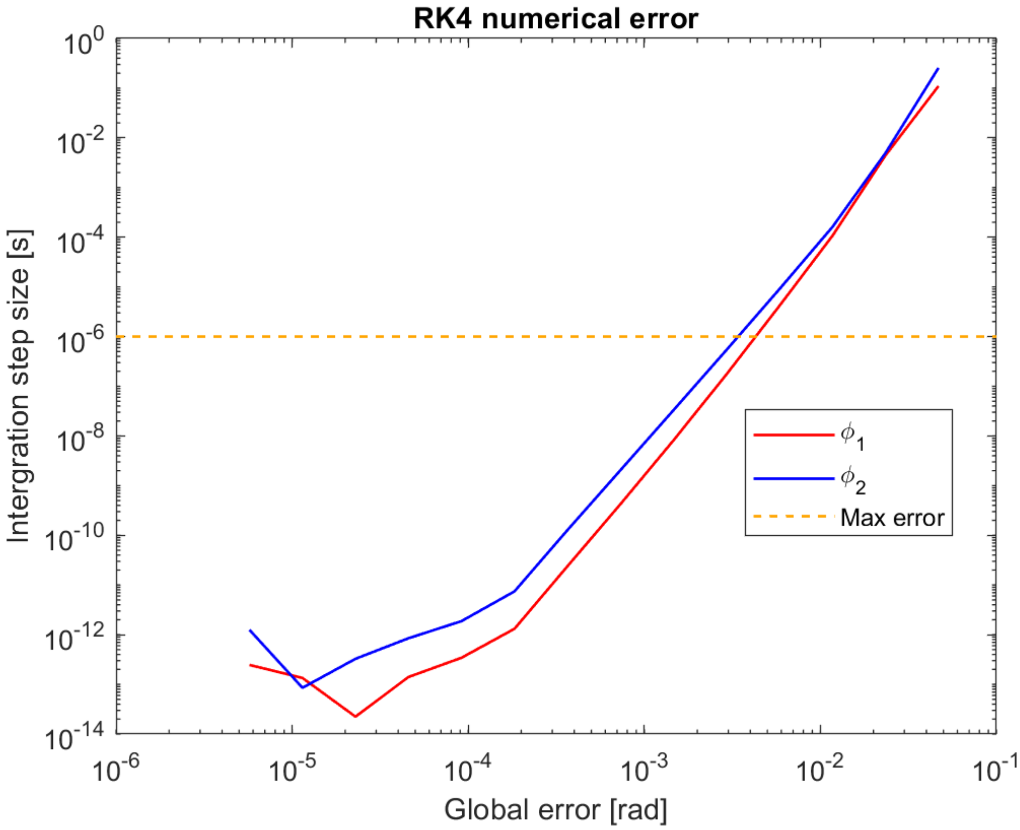

Figure 6: The global error [E] plotted on a log log scale vs the step size [h] for the 4th order Runge-Kutta method.

From figure 6 we can see that the RK4 method n 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 $h = 2 * 10^{-3}$ s 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

	φ_1	φ_2	**Function iterations**	**Mean step size [s]**	**Run time [s]**
Euler	-1.211821 rad	-2.675981 rad	300001	1e-5	8.04 s
Heun	-1.212237 rad	-2.669923 rad	120001	2.5-5	15.41s
RK3	-1.212237 rad	-2.669924 rad	7501	4e-4	1.47 s
RK4	-1.212237 rad	-2.669925 rad	1501	2e-3	0.381s
ODE23	-1.212252 rad	-2.669807 rad	4378	0.0021	0.28 s
ODE45	-1.212248 rad	-2.669863 rad	1033	0.0048	0.11 s
ODE113	-1.212231 rad	-2669953 rad	798	0.0077	0.13 s

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 cased since for the higher However as we see from [tableref] one should take into account the trade off between the number of steps taken and the ovnteerall executation time. In our example the ODE23 due to the large number off steps takes the longes 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 manly 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: Accompanying MATLAB scripts

```
%% MBD_B: Assignment 6 - Double pendulum numerical intergration
%  Rick Staa (4511328)
%  Last edit: 02/05/2018
%  In this script I append the acceleration to the generalised state q so
%  q = [phi1 phi2 phi1p phi2p phi1dp phi2dp]. This was done to save space.
clear all; close all; clc;
fprintf('--- A6 ---\n');

%% Set up needed symbolic parameters
% Create needed symbolic variables
syms phi1 phi2 phi1p phi2p

% Put in parms struct for easy function handling
parms.syms.phi1                 = phi1;
parms.syms.phi2                 = phi2;
parms.syms.phi1p                = phi1p;
parms.syms.phi2p                = phi2p;

%% Intergration parameters
time                            = 3;                            % Intergration time
parms.h                         = 0.001;                        % Intergration step size

%% Model Parameters
% Segment 1
parms.L                         = 0.55;                         % [parms.m]
parms.w                         = 0.05;                         % [parms.m]
parms.t                         = 0.004;                        % [parms.m]
parms.p                         = 1180;                         % [kg/parms.m^3]
parms.m                         = parms.p * parms.w * parms.t * parms.L;     % [kg]
```

```
parms.I                         = (1/12) * parms.m * parms.L^2;                 % [kg*parms.m^2]

% World parameters
parms.g                         = 9.81;                                         % [parms.m/s^2]

%% Initial state
q0                              = [0.5*pi 0.5*pi 0 0];

%% Derive equation of motion
EOM_calc(parms);                                                                % Calculate symbolic equations of motion and put in parms struct

%% Calculate GLOBAL ERROR of the numerical intergration methods specified in the assignment
% 1). Euler (Euler)
% 2). Heun  (Heun)
% 3). Runge-Kutta 3th order (RK3)
% 4). Runge-Kutta 4th order (RK4)

tic
%% Euler intergration
% Calculate the error per step size for euler

% Loop h and calculate global error
n_range    = 6:1:25;
h_range    = time./(2.^n_range);
q_end_h_euler    = zeros(length(h_range),6);
for kk = 1:length(h_range)
    parms.h                         = h_range(kk);
    [t,q]                           = ODE_custom(time,q0,'euler',parms);
    % bar_animate(t,q,parms);                                                   % Animate Bar
    q_end_h_euler(kk,:)             = q(end,:);
end
glob_error                          = abs(q_end_h_euler(2:end,:)-q_end_h_euler(1:end-1,:));    % Calculate global error

```

```matlab
% Create error plot
figure;
loglog(h_range(1:end-1),glob_error(:,1),'Color','red','LineWidth',1);hold on;
loglog(h_range(1:end-1),glob_error(:,2),'Color','blue','LineWidth',1);hold on;
line(xlim,[1e-6 1e-6],'Color',[1 0.6471 0],'LineStyle','--','LineWidth',1);
legend('\phi_1','\phi_2','Max error','Location', 'Best');
title('Euler numerical error');
xlabel('Global error [rad]')
ylabel('Intergration step size [s]')

%% Heun intergration
% Calculate the error per step size for heun
% Calculate the error per step size for euler

% Loop h and calculate global error
n_range    = 6:1:20;
h_range    = time./(2.^n_range);
q_end_h_heun    = zeros(length(h_range),6);
for kk = 1:length(h_range)
    parms.h                             = h_range(kk);
    [t,q]                               = ODE_custom(time,q0,'heun',parms);
    % bar_animate(t,q,parms);                % Animate Bar
    q_end_h_heun(kk,:)                  = q(end,:);
end
glob_error_heun                         = abs(q_end_h_heun(2:end,:)-q_end_h_heun(1:end-1,:));    % Calculate global error

% Create error plot
figure;
loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_heun(:,1)'),'Color','red','LineWidth',1);hold on;
loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_heun(:,2)'),'Color','blue','LineWidth',1);hold on;
line(xlim,[1e-6 1e-6],'Color',[1 0.6471 0],'LineStyle','--','LineWidth',1);
legend('\phi_1','\phi_2','Max error','Location', 'Best');
title('Heun numerical error');
```

```matlab
xlabel('Global error [rad]')
ylabel('Intergration step size [s]')

%% Runge-Kutta 3th order (RK3)
% Calculate the error per step size for RK3

% Loop h and calculate global error
n_range    = 6:1:20;
h_range    = time./(2.^n_range);
q_end_h_RK3    = zeros(length(h_range),6);
for kk = 1:length(h_range)
    parms.h                         = h_range(kk);
    [t,q]                           = ODE_custom(time,q0,'RK3',parms);
    % bar_animate(t,q,parms);                                    % Animate Bar
    q_end_h_RK3(kk,:)              = q(end,:);
end
glob_error_RK3                     = abs(q_end_h_RK3(2:end,:)-q_end_h_RK3(1:end-1,:));      % Calculate global error

% Create error plot
figure;
loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_RK3(:,1)'),'Color','red','LineWidth',1);hold on;
loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_RK3(:,2)'),'Color','blue','LineWidth',1);hold on;
line(xlim,[1e-6 1e-6],'Color',[1 0.6471 0],'LineStyle','--','LineWidth',1);
legend('\phi_1','\phi_2','Max error','Location', 'Best');
title('RK3 numerical error');
xlabel('Global error [rad]')
ylabel('Intergration step size [s]')

%% Runge-Kutta 4th order (RK4)
% Calclate the error per step size for RK4

% Loop h and calculate global error
n_range    = 6:1:20;
h_range    = time./(2.^n_range);
q_end_h_RK4    = zeros(length(h_range),6);
for kk = 1:length(h_range)
```

```matlab
    parms.h                         = h_range(kk);
    [t,q]                           = ODE_custom(time,q0,'RK4',parms);
    % bar_animate(t,q,parms);                                         % Animate Bar
    q_end_h_RK4(kk,:)               = q(end,:);
end
glob_error_RK4                      = abs(q_end_h_RK4(2:end,:)-q_end_h_RK4(1:end-1,:));       % Calculate global error

% Create error plot
figure;
loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_RK4(:,1)'),'Color','red','LineWidth',1);hold on;
loglog(fliplr(h_range(1:end-1)),fliplr(glob_error_RK4(:,2)'),'Color','blue','LineWidth',1);hold on;
line(xlim,[1e-6 1e-6],'Color',[1 0.6471 0],'LineStyle','--','LineWidth',1);
legend('\phi_1','\phi_2','Max error','Location', 'Best');
title('RK4 numerical error');
xlabel('Global error [rad]')
ylabel('Intergration step size [s]')
toc;

%% Perform methods at maxstepsize
%% Euler method at step-size 1e-5
tic
parms.h = 1e-5;
[t_euler,q_euler]                           = ODE_custom(time,q0,'euler',parms);
toc

%% Heun method at step-size 1e-5
tic
parms.h = 2.5e-5;
[t_heun,q_heun]                             = ODE_custom(time,q0,'heun',parms);
toc

%% Runge-Kutta 3th order (RK4)
tic
parms.h = 4e-4;
```

```
[t_RK3,q_RK3]                           = ODE_custom(time,q0,'RK3',parms);
toc

%% Runge-Kutta 4th order (RK4)
tic
parms.h = 2e-3;
[t_RK4,q_RK4]                           = ODE_custom(time,q0,'RK4',parms);
toc

%% Calculate motion withODE functions
% ODE 23
tic
opt = odeset('AbsTol',1e-6,'RelTol',1e-6,'Stats','on');
[t23,q23] = ode23(@(t,q) ODE_func(t,q), [0 time], q0',opt);
t23_mean    = mean(diff(t23));                                           % Caculate mean step size
disp(t23_mean);
toc

% ODE 45
tic
opt = odeset('AbsTol',1e-6,'RelTol',1e-6,'Stats','on');
[t45,q45] = ode45(@(t,q) ODE_func(t,q), [0 time], q0',opt);
t45_mean    = mean(diff(t45));                                           % Caculate mean step size
disp(t45_mean);
toc

% ODE 113
tic
opt = odeset('AbsTol',1e-6,'RelTol',1e-6,'Stats','on');
[t113,q113] = ode113(@(t,q) ODE_func(t,q), [0 time], q0',opt);
t113_mean    = mean(diff(t113));                                         % Caculate mean step size
disp(t113_mean);
toc

```

```matlab
%% FUNCTIONS

%% Bar animate
% This function creates a movie of the double pendulum. I did not find a
% way to do this with the real speed but it gives a nice impression of what
% is happening.
function bar_animate(t,q,parms)
figure;
h=plot(0,0,'MarkerSize',20,'Marker','.','LineWidth',2);
range=1.1*(parms.L+parms.L); axis([-range range -range range]); axis square;
set(gca,'nextplot','replacechildren');
a = tic;
for jj=1:length(q)-1
    if (ishandle(h)==1)
        tic
        phi1 = q(jj,1);
        phi2 = q(jj,2);
        Xcoord=[0,parms.L*cos(phi1),parms.L*cos(phi1)+parms.L*cos(phi2)];
        Ycoord=[0,parms.L*sin(phi1),parms.L*sin(phi1)+parms.L*sin(phi2)];
        set(h,'XData',Xcoord,'YData',Ycoord);
        b = toc(a);          % check timer
        if b > (1/30)
            drawnow          % update screen every 1/30 seconds
            a = tic;         % reset timer after updating
        end
        %               pause(t(jj+1)-t(jj));                                % Realtime
        toc;
    end
end
drawnow
end

%% ODE Function handle
function [qdp] = ODE_func(t,q)
q_now = num2cell(q',1);
qdp   = subs_qdp(q_now{:});
```

```
qdp    = [q(3);q(4);qdp];
end

%% Euler numerical intergration function
% This function calculates the motion of the system by means of a euler
% numerical intergration. This function takes as inputs the parameters of
% the system (parms), the EOM of the system (parms.EOM) and the initial
% state.
function [t,q] = ODE_custom(time,q0,method,parms)

% Initialise variables
t                   = (0:parms.h:time).';                      % Create time array
q                   = zeros(length(t),6);                      % Create empty state array
q(1,1:size(q0,2))   = q0;                                                          % Put initial state in array

% Caculate the motion for the full time by means of the 4 different
% numerical intergration methods
% 1). Euler
% 2). Heun
% 3). Runge-Kutta 3th order
% 4). Runge-Kutta 4th order
% See report for the Workings of each method.

% Euler method
switch method

    %% Euler method
    case 'euler'

        % Perform the full intergration with eulers method
        for ii = 1:(size(t,1)-1)
            q_now_tmp       = num2cell(q(ii,1:end-2),1);                % Create cell for subs function function
```

```
            qdp                = subs_qdp(q_now_tmp{:}).';                 % Calculate the second derivative of the generalised coordinates
            q(ii,end-1:end) = qdp;
            q(ii+1,1:end-2) = q(ii,1:end-2) + parms.h*q(ii,3:end);         % Perform euler intergration step

            % Calculate last acceleration
            if ii == (size(t,1)-1)
                q_next              = num2cell(q(ii+1,1:end-2),1);          % Create cell for subs function function
                q(ii+1,end-1:end) = subs_qdp(q_next{:}).';                  % Calculate the second derivative of the last step
            end
        end

        %% Heun method
    case 'heun'

        % Perform the full intergration with eulers method
        for ii = 1:(size(t,1)-1)
            % Step 1: Approximate the next state
            q_now              = [q(ii,1:end-2) 0 0];                                    % Read out current states
            q_now_tmp          = num2cell(q_now,1);                                       % Create cell for subs function function
            qdp_now_tmp        = subs_qdp(q_now_tmp{1:end-2}).';                 % Calculate the second derivative of the generalised coordinates
            qdp_now            = [cell2mat(q_now_tmp(end-3:end-2)),qdp_now_tmp];    % Add first derivative
            q_now(end-3:end)= qdp_now;
            q_star             = q(ii,1:end-2) + parms.h*q_now(3:end);             % Make a approximation of the next state by means of a euler step

            % Step 2: Calculate the state derivative at next state
```

```
            q_star_tmp      = num2cell(q_star,1);                            % Create cell for subs function function
            qdp_star_tmp    = subs_qdp(q_star_tmp{:}).';                      % Calculate the second derivative of the generalised coordinates
            qdp_star        = [cell2mat(q_star_tmp(end-1:end)),qdp_star_tmp];   % Add first derivative

            % Step3: Calculate the state at the next step using the mean
            % derivative.
            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)

            % Calculate last acceleration
            if ii == (size(t,1)-1)
                q_next_tmp        = num2cell(q(ii+1,1:end-2),1);                 % Create cell for subs function function
                q(ii+1,end-1:end) = subs_qdp(q_next_tmp{:}).';                    % Calculate the second derivative of the last step
            end
        end

        %% Runge-Kutta 3th order
    case 'RK3'
        for ii = 1:(size(t,1)-1)
            q_now_tmp          = num2cell(q(ii,1:end-2),1);                                                  % Create cell for subs function function
            K1                 = [cell2mat(q_now_tmp(1,end-1:end)),subs_qdp(q_now_tmp{:}).'];            % Calculate the second derivative at the start of the step
            q1_tmp             = num2cell(cell2mat(q_now_tmp) + (parms.h*0.5)*K1);                  % Create cell for subs function function
            K2                 = [cell2mat(q1_tmp(1,end-1:end)),subs_qdp(q1_tmp{:}).'];             %
```

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

        % Calculate last acceleration
        if ii == (size(t,1)-1)
            q_now_tmp             = num2cell(q(ii+1,1:end-2),1);                                   % Create cell for subs function function
            K1                    = [cell2mat(q_now_tmp(1,end-1:end)),subs_qdp(q_now_tmp{:}).'];    % Calculate the second derivative at the start of the step
            q1_tmp                = num2cell(cell2mat(q_now_tmp) + (parms.h*0.5)*K1);              % Create cell for subs function function
            K2                    = [cell2mat(q1_tmp(1,end-1:end)),subs_qdp(q1_tmp{:}).'];          % Calculate the second derivative halfway the step
            q2_tmp                = num2cell(cell2mat(q_now_tmp) + ((parms.h*0.75))*K2);           % Refine value calculation with new found derivative
            K3                    = [cell2mat(q2_tmp(1,end-1:end)),subs_qdp(q2_tmp{:}).'];          % Calculate new 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
            end
        end

        %% Runge-Kutta 4th order
    case 'RK4'
        for ii = 1:(size(t,1)-1)
            q_now_tmp           = num2cell(q(ii,1:end-2),1);                                       % Create cell for subs function function
            K1                  = [cell2mat(q_now_tmp(1,end-1:end)),subs_qdp(q_now_tmp{:}).'];          % Calculate the second derivative at the start of the step
            q1_tmp              = num2cell(cell2mat(q_now_tmp) + (parms.h*0.5)*K1);                     % Create cell for subs function function
            K2                  = [cell2mat(q1_tmp(1,end-1:end)),subs_qdp(q1_tmp{:}).'];                % Calculate the second derivative halfway the step
            q2_tmp              = num2cell(cell2mat(q_now_tmp) + (parms.h*0.5)*K2);                     % Refine value calculation with new found derivative
            K3                  = [cell2mat(q2_tmp(1,end-1:end)),subs_qdp(q2_tmp{:}).'];                % Calculate new derivative at the new refined location
            q3_tmp              = num2cell(cell2mat(q_now_tmp) + (parms.h)*K3);                         % Calculate state at end step with refined derivative
            K4                  = [cell2mat(q3_tmp(1,end-1:end)),subs_qdp(q3_tmp{:}).'];                % Calculate last second derivative
            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
        q(ii+1,1:end-2)    = cell2mat(q_now_tmp) + (parms.h/6)*(K1+2*K2+2*K3+K4);                          % Perform euler intergration step

        % Calculate last acceleration
        if ii == (size(t,1)-1)
            q_now_tmp           = num2cell(q(ii+1,1:end-2),1);                                              % Create cell for subs function function
            K1                  = [cell2mat(q_now_tmp(1,end-1:end)),subs_qdp(q_now_tmp{:}).'];               % Calculate the second derivative at the start of the step
            q1_tmp              = num2cell(cell2mat(q_now_tmp) + (parms.h*0.5)*K1);                          % Create cell for subs function function
            K2                  = [cell2mat(q1_tmp(1,end-1:end)),subs_qdp(q1_tmp{:}).'];                     % Calculate the second derivative halfway the step
            q2_tmp              = num2cell(cell2mat(q_now_tmp) + (parms.h*0.5)*K2);                          % Refine value calculation with new found derivative
            K3                  = [cell2mat(q2_tmp(1,end-1:end)),subs_qdp(q2_tmp{:}).'];                     % Calculate new derivative at the new refined location
            q3_tmp              = num2cell(cell2mat(q_now_tmp) + (parms.h)*K3);                              % Calculate state at end step with refined derivative
            K4                  = [cell2mat(q3_tmp(1,end-1:end)),subs_qdp(q3_tmp{:}).'];                     % Calculate last second derivative
            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
```

```
            end
        end
end
end

%% Calculate (symbolic) Equations of Motion four our setup
function EOM_calc(parms)

% Unpack symbolic variables from varargin
phi1                    = parms.syms.phi1;
phi2                    = parms.syms.phi2;
phi1p                   = parms.syms.phi1p;
phi2p                   = parms.syms.phi2p;

% Create generalized coordinate vectors
q                       = [phi1; phi2];
qp                      = [phi1p; phi2p];

% COM of the bodies expressed in generalised coordinates
x1                      = (parms.L/2)*cos(phi1);
y1                      = (parms.L/2)*sin(phi1);
x2                      = x1 + (parms.L/2)*cos(phi1) + (parms.L/2) * cos(phi2);
y2                      = y1 + (parms.L/2)*sin(phi1) + (parms.L/2) * sin(phi2);

% Calculate derivative of COM expressed in generalised coordinates (We need this for the energy equation)
x                       = [x1;y1;phi1;x2;y2;phi2];
Jx_q                    = simplify(jacobian(x,q));
xp                      = Jx_q*qp;

%% Compute energies
T                       = 0.5*xp.'*diag([parms.m;parms.m;parms.I;parms.m;parms.m;parms.I])*xp;             % Kinetic energy
V                       = -([parms.m*parms.g 0 0 parms.m*parms.g 0 0]*x);                                   % Potential energy

%% Calculate the terms of the jacobian
Q                       = 0;                                                                                % Non-conservative forces

```

```
% Partial derivatives of Kinetic energy
T_q                 = simplify(jacobian(T,q));
T_qp                = simplify(jacobian(T,qp));
T_qpqp              = simplify(jacobian(T_qp,qp));
T_qpq               = simplify(jacobian(T_qp,q));

% Partial derivatives of Potential energy
V_q                 = simplify(jacobian(V,q));
V_qp                = simplify(jacobian(V,qp));
V_qpqp              = simplify(jacobian(V_qp,qp));

% Make matrix vector product
M                   = T_qpqp;
F                   = Q + T_q.' - V_q.' - T_qpq*qp;

% Solve Mqdp=F to get the accelerations
qdp                 = M\F;

%% Get back to COM coordinates
% xdp              = simplify(jacobian(xp,qp))*qdp+simplify(jacobian(xp,q))*qp;

%% Convert to function handles
% xdp_handle       = matlabFunction(xdp,'File','subs_xdp');        % Create function handle of EOM in terms of COM positions
matlabFunction(simplify(qdp),'File','subs_qdp');           % Create function handle of EOM in terms of generalised coordinates
end
```

References

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