8.25

6/6: answered all questions, matlab provided
0.25/0.5: a) spring loaded at the top
0.5/1: b) no stabilization of euler mentioned -0.25. Spring forces and damper always active? -0.25

0/0.5 c) no approach given. -0.5
0.25/0.5: d) rising total energy, -0.25
0.25/0.5: e) bad evaluation
0.5/0.5: f)
0.5/0.5: g)

Multibody Dynamics B - Assignment

ME41055 #4511328
Prof. Arend L. Schwab Lab Date: 07/06/2018
Head TA: Simon vd. Helm Due Date: 14/06/2018

1 Statement of integrity

my homework is completely in accordance with the Academic Integrity

Figure 1: My handwritten statement of integrity

2 Acknowledgements

I used [1] in making this assignment when finished I compared my values with Prajish Kumar (4743873). He then pointed out that my Q matrix was expressed in the Inertial frame instead of the Body fixed frame.

3 Setup overview

In this assignment we were asked to evaluate the motion of a ejection seat. An overview of the ejection seat is given in figure 2. In this figure h represents the height of the pillars and w the width between the tops ends of the pillars. Further for this setup we know the moment of inertia J, the mas m, the vehicle radius r, the gravity constant g, the relative distance between the center of the vehicle and the COM c and the relative distance between the vehicle center and the spring attachment sites p_s. The values for these parameters are as follows:

Vehicle, frame and world parameters

$$J = \begin{bmatrix} 170 & 0 & 0 \\ 0 & 120 & 0 \\ 0 & 0 & 140 \end{bmatrix} \ kgm^2 \tag{1}$$

$$m = 420 \ kg \tag{2}$$

$$h = 25 \ m \tag{3}$$

$$w = 18 \ m \tag{4}$$

$$r = 1 \ m \tag{5}$$

$$g = 9.81 \ m/s^2 \tag{6}$$

$$\rho = 1.25 \ kg/m^3 \tag{7}$$

$$c_d = 0.5 \tag{8}$$

Known spring parameters

$$\zeta = 0.1 \tag{9}$$

4 Problem Statement

In this assignment we were asked to derive the motion of a ejection seat by means of Euler parameters. To be able to do this we first need to compute some spring parameters that were not given, namely the spring stiffness k, the damping coefficient b and the spring resting length l_0. These parameters can be calculated by making use of the principle of conservation of energy and the newtons second law.

Conservation of energy
First we derive the sum of the potential and kinetic energies for both the top and the down positions:

$$E_t = E_b \tag{10}$$

$$E_{kin,t} + E_{pot,t} = E_{kin,b} + E_{pot,b} \tag{11}$$

In these formulas E_t represent the energy at the top while E_b represents the energy at the bottom. Since at the top and the bottom the vehicle has zero speed the kinetic energy is 0. Further since set the height at the bottom position to be zero we at the bottom only have the potential energy of the spring. As a result we get the following equation:

$$mg(h + 0.5w) + k(l_t - l_0)^2 - k(l_b - l_0)^2 = 0 \tag{12}$$

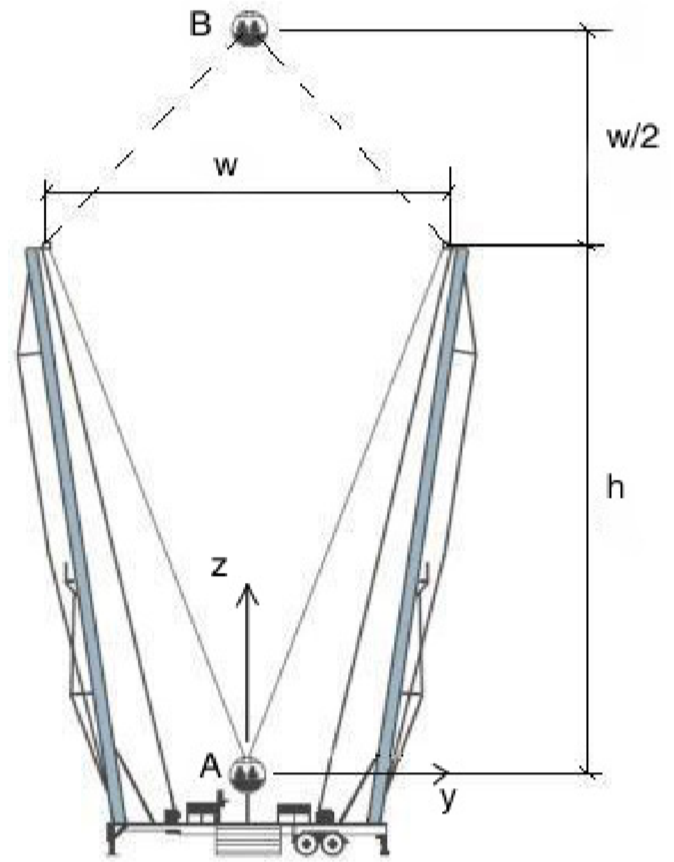

Figure 2: Overview of the ejection seat of assignment 10

Second law of newton
For the second law of newton we know the spring forces and we know that the acceleration at the bottom is equal to 4.8g. With this information and some simple geometry we get the following equation:

$$2k * (l_b - l0) * cos(tan^{-1}(\frac{0.5w - r}{h})) - 5.8mg \tag{13}$$

We now have two equations with two unknowns which can be solved by Gaussian elimination. When we use the symbolic toolbox of MATLAB to do this we get the following result:

0.25

$$k = 1.0720e + 03\ N/m \tag{14}$$
$$l_0 = 14.5463\ m \tag{15}$$
$$\tag{16}$$

Spring gives force when Lf<L0 -0.25

with these parameters we now can also calculate the damping coefficient b by approximating the spring as a simple spring damper system. For a simple spring damping system we get the following equation:

$$b = \zeta\sqrt{k * m} = 134.2020\ Ns/m \tag{17}$$

4.1 Rotation matrix

Before we can calculate the equations of motion we need to make sure that all our variables are specified w.r.t. the same coordinate frame. To do this we need to derive the Rotation matrix. As earlier stated to get rid of singularities we can use the Euler parameters for this. These Euler parameters make use of Eulers principal axis of rotation theorem. This theorem says the following:

> "Any rotation in 3D can be represented by a rotation about a fixed axis at a given angle."

With the help of this theorem the Euler parameters are defined as follows [1]:

$$q_0 = cos(0.5\phi) \tag{18}$$
$$\mathbf{q} = \begin{bmatrix} q1 \\ q2 \\ q3 \end{bmatrix} = sin(0.5\phi)\hat{n} \tag{19}$$

In this ϕ is the angle of rotation and $\hat{n}$ the axis about which is rotated. After a long derivation we can get the following rotation matrix to rotate from the body fixed frame $\mathcal{B}$ to the inertial frame $\mathcal{N}$:

$$R^{\mathcal{N}}_{\mathcal{B}} = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_2q_1 - q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_3q_2 - q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} \tag{20}$$

4.2 derivation of EOM

To derive the equations of motion we need to use the principle of virtual work that was explained in chapter 3 of the reader [1]. In this example the following forces contribute to the virtual work of the system:

1. Elastic components of the springs F_{s1} and F_{s2}
2. The damper components of the two springs F_{b1} and F_{b2}
3. The air drag F_d
4. The d'alambert forces $-m\ddot{a}$

With all these components we get the following virtual power expression expressed in the body fixed frame:

$$M\ddot{x} = F\partial\dot{x} - \sigma_{s1}\, l_{s1,i}\, \partial\dot{x} - \sigma_{s2}\, l_{s2,i}\, \partial\dot{x} - \sigma_{b1}\, l_{s1,i}\, \partial\dot{x} - \sigma_{b2}\, l_{s1,i}\, \partial\dot{x} - \sigma_d\, \partial\dot{x} \tag{21}$$

Since this has to hold for all virtual velocities that obey the constraints the EOM equation becomes:

$$M\ddot{x} = F - \sigma_{s1}\, l_{s1,i} - \sigma_{s2}\, l_{s2,i} - \sigma_{b1}\, l_{s1,i} - \sigma_{b2}\, l_{s1,i} - \sigma_d\, \partial\dot{x} \tag{22}$$

In this:

$$M = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix} \quad \text{and} \quad F = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} \tag{23}$$

In this the σ_s is the spring force magnitude, σ_b the damper force magnitude and σ_d the drag magnitude. Further the $L_{s1,i}$ is the Jacobean of the spring length. This Jacobean is used to translate the Force magnitude to their x,y and z components in the inertial frame. Below the derivation of each of the component will be discussed:

4.2.1 Elastic components

To get the virtual power of the spring we need to first get the spring elongation. To get the spring elongation we first need to compute the position of the center of the vehicle in the inertial frame and the positions of the spring attachment points. The center of the vehicle in the inertial frame can be calculated as:

$$r_c = r_{COM} + R^{\mathcal{N}}_{\mathcal{B}}\, p^{\mathcal{B}}_c = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + R^{\mathcal{N}}_{\mathcal{B}} * p^{\mathcal{B}}_c \begin{bmatrix} 0.01 \\ -0.01 \\ 0.1 \end{bmatrix} \tag{24}$$

With this the attachment points of the springs expressed in the N frame can be calculated. The attachment point of the left spring is:

$$r_{s1} = r_c + R_{\mathcal{B}}^{\mathcal{N}}\, r_{s1}^{\mathcal{B}} = r_c + R_{\mathcal{B}}^{\mathcal{N}} \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} \tag{25}$$

and the attachment point of the right spring is:

$$r_{s2} = r_c + R_{\mathcal{B}}^{\mathcal{N}}\, r_{s2}^{\mathcal{B}} = r_c + R_{\mathcal{B}}^{\mathcal{N}} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{26}$$

With these attachment points we can now create two vectors which point along the springs for the left spring this is done as:

$$l_{s1} = r_{s1} - \begin{bmatrix} 0 \\ -0.5w \\ h \end{bmatrix} \quad \text{and} \quad l_s1 = r_{s1} - \begin{bmatrix} 0 \\ 0.5w \\ h \end{bmatrix} \tag{27}$$

From these attachment point the spring length $|l_s|$ and also the change in spring length $\Delta l_s = l_s - l_0$. We now only need one more ingredient to get to the virtual power of the 2 springs, namely the Jacobean of the spring length l_s w.r.t. to the reduced state $x_{small} = \begin{bmatrix} xyz \end{bmatrix}^T$. This Jacobean is calculated with MATLABS symbolic tool box. The script doing this can be found in appendix A. The resulting virtual power of the spring becomes:

$$\partial F_s = \sigma_{s1}\, l_{s1,i}\, \partial \dot{x} +_{\sigma\ s2}\, l_{s2,i}\, \partial \dot{x} \tag{28}$$

4.2.2 Damping action of spring elements

The damping action of the springs gets added to the virtual power equation in a similar way.

$$\partial F_b = \sigma_{b1}\, l_{s1,i}\, \partial \dot{x} - \sigma_{b2}\, l_{s1,i}\, \partial \dot{x} \tag{29}$$

In which:

$$\sigma_{b1} = b\frac{dl_{s1}}{dt} \quad \text{and} \quad l_{s1,i} = \partial l_{s1} \partial x_{small} \tag{30}$$

For the right spring the procedure is the same.

4.2.3 Drag force

As given in the assignment the drag force can be calculated as:

$$\partial F_d = \sigma_d \partial \dot{x}_i = 0.5 \rho A C_d |\dot{x}| \dot{x} \partial \dot{x} \tag{31}$$

4.2.4 Linear accelerations

We now have all the ingredients to calculate the linear accelerations. These accelerations can be calculated by solving the following system of equations:

$$M\ddot{x} = F\partial\dot{x} - \sigma_{s1}\, l_{s1,i}\, \partial\dot{x} - \sigma_{s2}\, l_{s2,i}\, \partial\dot{x} - \sigma_{b1}\, l_{s1,i}\, \partial\dot{x} - \sigma_{b2}\, l_{s1,i}\, \partial\dot{x} - \sigma_d\, \partial\dot{x} \tag{32}$$

$$\ddot{x} = M^{-1}\, F\partial\dot{x} - \sigma_{s1}\, l_{s1,i}\, \partial\dot{x} - \sigma_{s2}\, l_{s2,i}\, \partial\dot{x} - \sigma_{b1}\, l_{s1,i}\, \partial\dot{x} - \sigma_{b2}\, l_{s1,i}\, \partial\dot{x} - \sigma_d\, \partial\dot{x} \tag{33}$$

This system can be solved by means of a Gaussian elimination. The procedure for this was done in MATLAB and can be found in appendix A.

4.2.5 Angular acceleration

To get the full motion of the system in 3D we now also have to look at the newton-Euler equations. To describe the motion of the body we use the earlier mentioned Euler parameters and the following state:

$$q = \begin{bmatrix} x & y & z & q0 & q1 & q2 & q3 & \dot{x} & \dot{y} & \dot{z} & \omega_x & \omega y & omegaz \end{bmatrix} \tag{34}$$

In our example as a result the initial state becomes:

$$q = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{35}$$

The time derivative of the state becomes:

$$\dot{q} = \begin{bmatrix} \dot{x} & \dot{y} & \dot{z} & \dot{q0} & \dot{q1} & \dot{q2} & \dot{q3} & \ddot{x} & \ddot{y} & \ddot{z} & \dot{\omega_x} & \dot{\omega y} & \dot{\omega z} \end{bmatrix} \tag{36}$$

To be able to solve for the full EOM we therefore need to calculate the derivatives of the Euler parameters q. As stated in [1] we can calculate theses derivatives out of the body fixed angular velocities $\mathcal{B}^{\omega}$ as follows:

$$\begin{bmatrix} \dot{q0} \\ \dot{q1} \\ \dot{q2} \\ \dot{q3} \end{bmatrix} = 0.5 \begin{bmatrix} q0 & -q1 & -q2 & q3 \\ q1 & q0 & -q3 & q2 \\ q2 & q3 & q0 & -q1 \\ q3 & -q2 & q1 & q0 \end{bmatrix} \begin{bmatrix} 0 \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

The zero on the top row comes from the constraint that the norm of axis of rotation has to be equal to 1. Following to get the angular acceleartions we use the Newton-Euler with al its compenents expressed in the body fixed frame B. This equation is:

$$\sum M_{\mathcal{B}} = I_{\mathcal{B}}\dot{\omega} + \omega \times I_{\mathcal{B}}\omega \tag{37}$$

In which the second part of the equation contains the convective term, the $\omega = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$ and the moment of inertia is diagonal in nature:

$$I_{\mathcal{B}} = \begin{bmatrix} 170 & 0 & 0 \\ 0 & 120 & 0 \\ 0 & 0 & 140 \end{bmatrix} \tag{38}$$

With these equation the angular accelerations can be calculated as:

$$\dot{\omega} = I_{\mathcal{B}}^{-1}(\sum M_{\mathcal{B}} - \omega \times I_{\mathcal{B}}\omega) \tag{39}$$

As the drag and Gravity forces work on the COM they do not contribute to the moments. The forces that contribute to the moments are:

$$F_1 =_{s1} l_{s1,i} + \sigma_{b1} l_{s1,i} \quad \text{and} \quad F_2 =_{s2} l_{s1,i} + \sigma_{b2} l_{s1,i} \tag{40}$$

The moment arms can be calculated as:

$$r1 = -r_c - \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \text{and} \quad r2 = -r_c - \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{41}$$

Following the moments can be calculated as:

$$M = r_1 \times F_1 + r_2 \times F_2 \tag{42}$$

4.3 Numerical intergration method

To get the movement of the ejection seat in time we will use a 4^{th} order Runge-Kuta intergration method combined with a Gauß-Newton correction for position and speed. This correction is done to compensate for intergration drift. In this correction we use the position constraints and the velocity constraints.

4.3.1 Runge-Kutta 4th order method (RK4)

The Runge-Kutta 4th order method has the following iteration scheme:

$$k_1 = f(t_n, y_n) \tag{43}$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \tag{44}$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \tag{45}$$

$$k_4 = f(t_n + h, y_n + hk_3) \tag{46}$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{47}$$

4.3.2 Drift correction

In this assignment we only have one constraint 48, namely that axis of rotation $\hat{n}$ needs to have a length equal to one. The drift correction therefore is simply done by normalizing the Euler parameters by means of a simple coordinate projection the method. This coordinate projection method is implemented in appendix A.

$$C = q0^2 + q1^2 + q2^2 + q3^2 \tag{48}$$

4.4 Results and discussion

The the results of a simulation of 60 seconds are shown in figures below (figure (3, 4,5 and 6). From these figures we can see the following things:

- From figure 3 we can clearly see that the vehicle doesn't reach the max height of (h+0.5w = 34m) this is probably due to our drag and damping forces. From figure **??** it can be seen that the Drag and Damping forces are quite big and active at the beginning of the movement.
- From figure 4 we can see that the vehicle has very high oscilations in the Z directino while it has low oscillations in the x and y directions. This is to be expected due to the initial orientation of the vehicle and the direction of the springs.

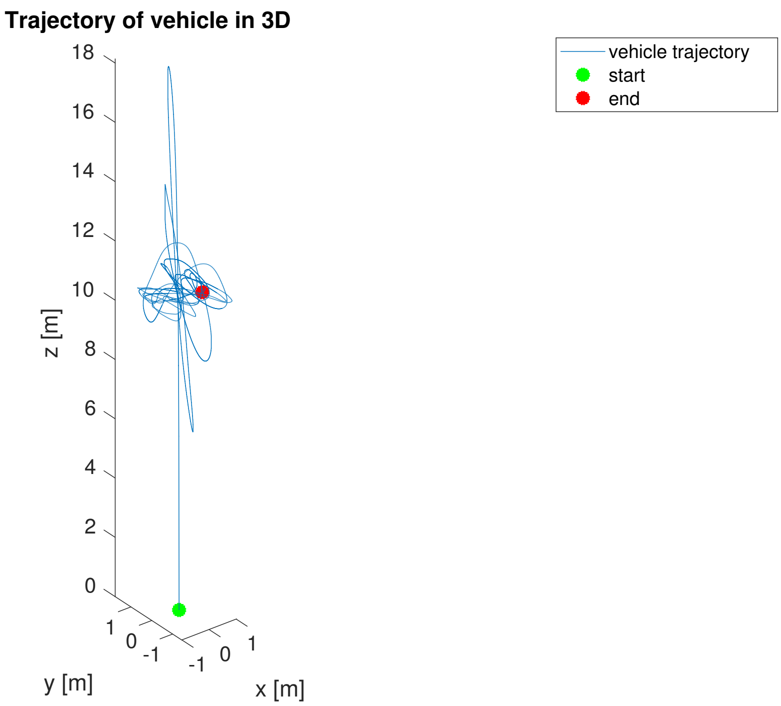

Figure 3: The Vehicle trajectory

- From figure 5 we can see that the angular velocities are roughly equal in magnitude in each direction. this is probably cased by the fact that the inertia matrix has similar values on the diagonal and that the COM is not displaced by a very much amount compared to the center of rotation.

4.5 Global intergration error

In figure 7 the global error is shown versus the time. We can see that for the whole simulation this global error stays below the accepted value of $1^{-}12$. We can however not rule out that this value could however eventually rise above $1^{-}12$.

No clear approach how the error is calculated. -0.5

4.6 Energy of the system

To get to know more about the behavoir of the system we can also look at the energies present in the system. The total energy of the system is made up by the following energies (gravitational potential energy E_p, spring elastic potential energy E_e, linear kinetic energy E_k and rotational kinetic energy E_r). These energies can be calculated as follows:

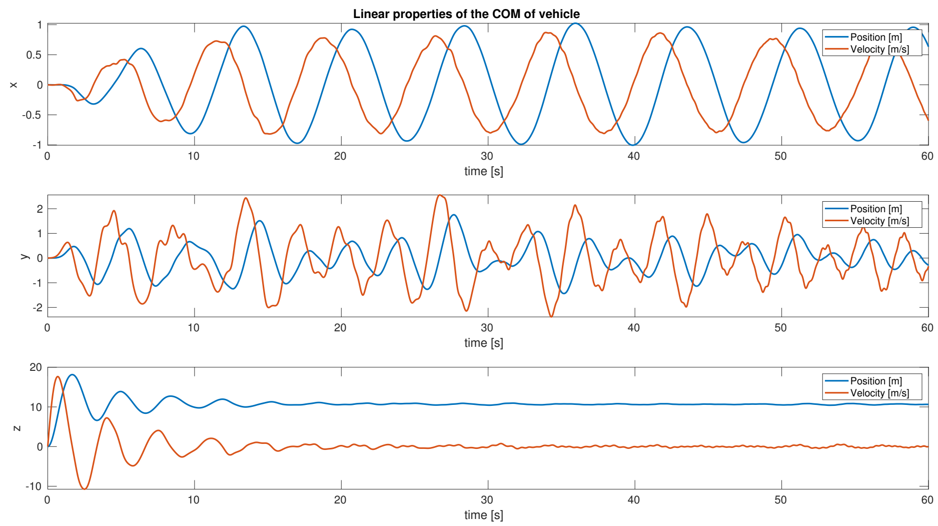

Figure 4: Linear positions and velocities

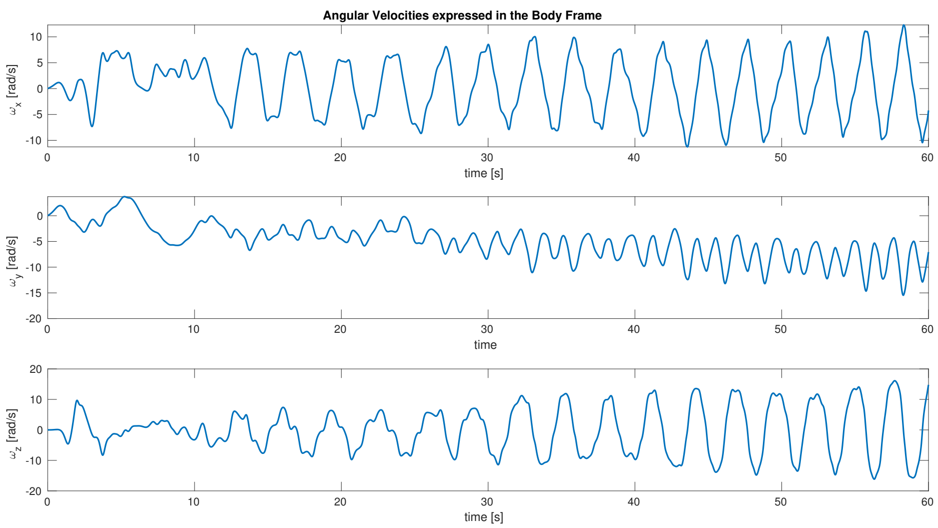

Figure 5: Angular velocities

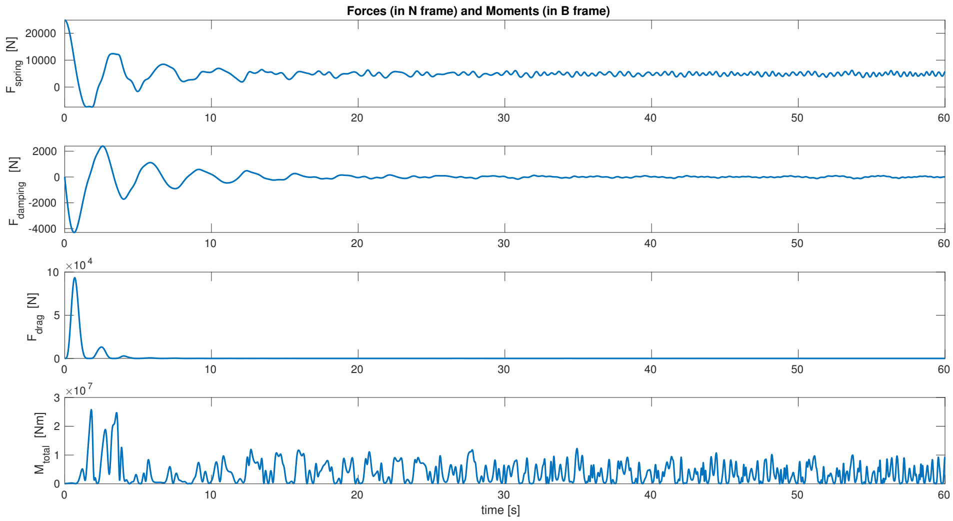

Figure 6: Forces on the vehicle

Figure 7: Global error versus the time

$$E_p = mgz \tag{49}$$
$$E_e = 0.5k(l_{s1} - l_0)^2 + -0.5k(l_{s2} - l_0)^2 \tag{50}$$
$$E_t = 0.5m(\dot{x}^2 + \dot{y}^2 + \dot{z}^2); E_r = 0.5(\dot{\omega}^T J \dot{\omega}) \tag{51}$$

The result of these energy calculations are shown in figure 8 and figure **??**. From this figure we can see the following things:

- From figure 8 we see that all energies are relatively high compared to the other assignments in this course. This is cased by the higher velocities and the higher scale of the setup.
- We further see that the spring energy and gravitational energy are the biggest contributors to the total energy.
- Lastly from examining the total energy in figure **??** we see that a considerably amount of energy is lost due to drag and damming forces.

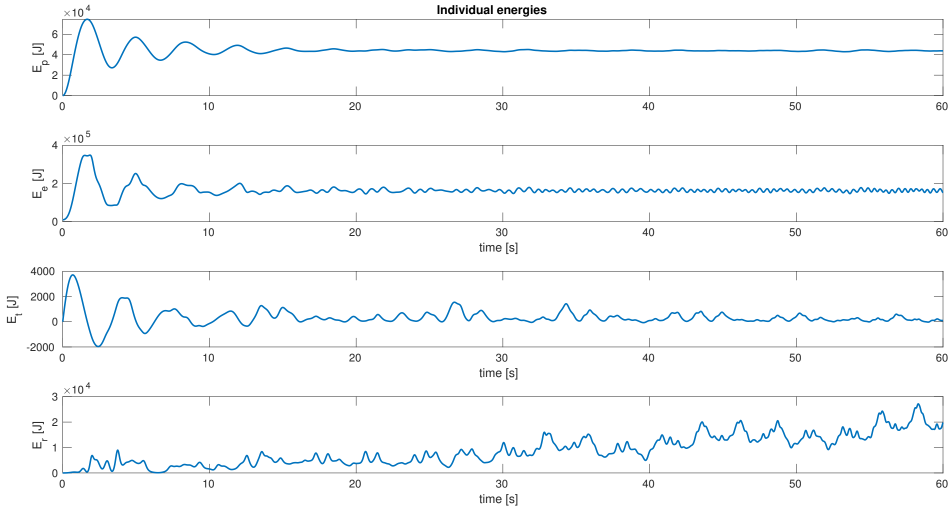

Figure 8: Figure of the individual energies

Total energy cannot rise due to energy dissipation. -0.25

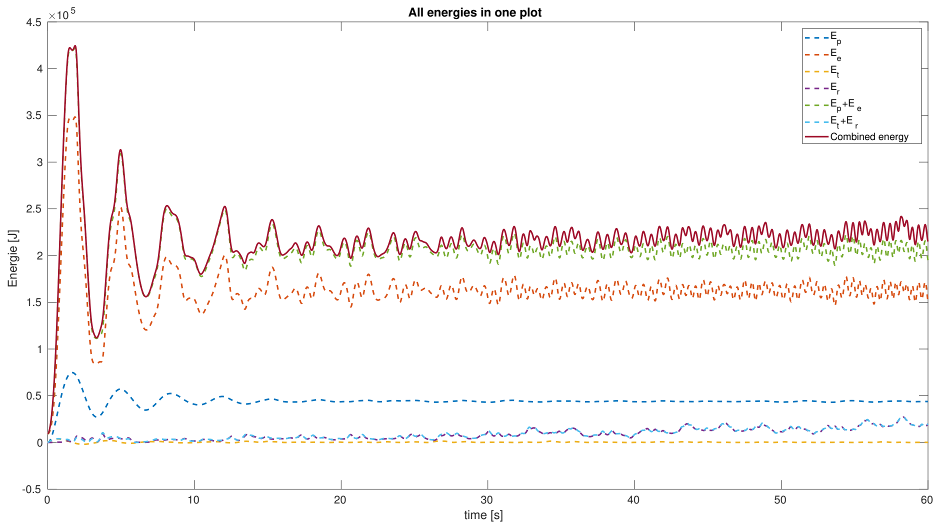

Figure 9: Figure of the individual energies

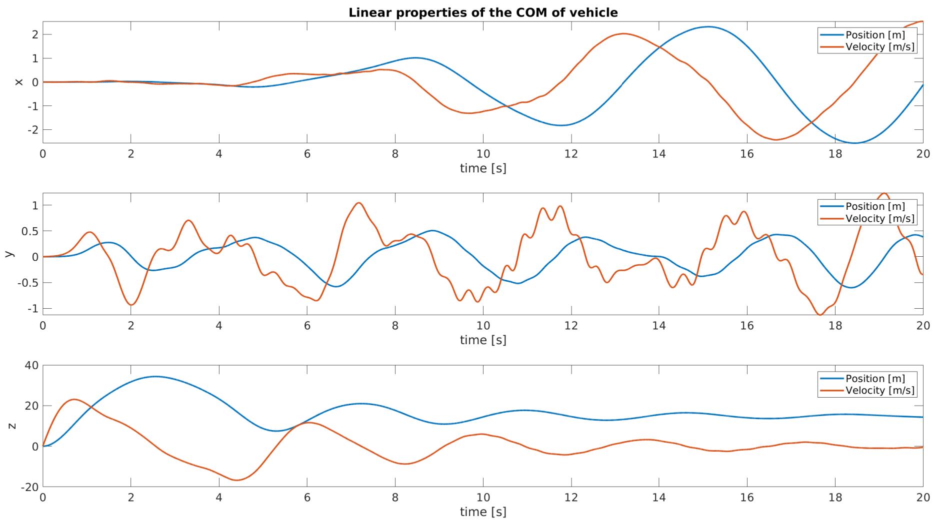

Figure 10: Figure of linear translations for optimum parameters

4.7 Further optimize the spring and damping parameters

To get a more fun customer experience the owner of the attraction hired us to create a more thrilling but yet not lethal experience. We therefore perform an optimization on the spring stiffness k and spring rest length l_0. to see if we can get a little higher. The script performing this optimization can be found in Appendix A. In this optimization the following parameters seemed to give a height of $h + 0.5 * w = 34m$

$$l_0 = 11\ m \tag{52}$$
$$k = 1000N/m \tag{53}$$
$$b = 129.6148Ns/m \tag{54}$$
$$\tag{55}$$

The result of simulating with these parameters is found in figure 10, 11, 12 13. From these figures we can see that duet to the lower stiffness k there are less oscillations in both the transnational and rotational movement. We also clearly see that in the new trajectory (figure 12). Lastly we see now that the contribution from the gravitational potential energy is higher.

extreme high elatic energy
not mentioned, -0,25

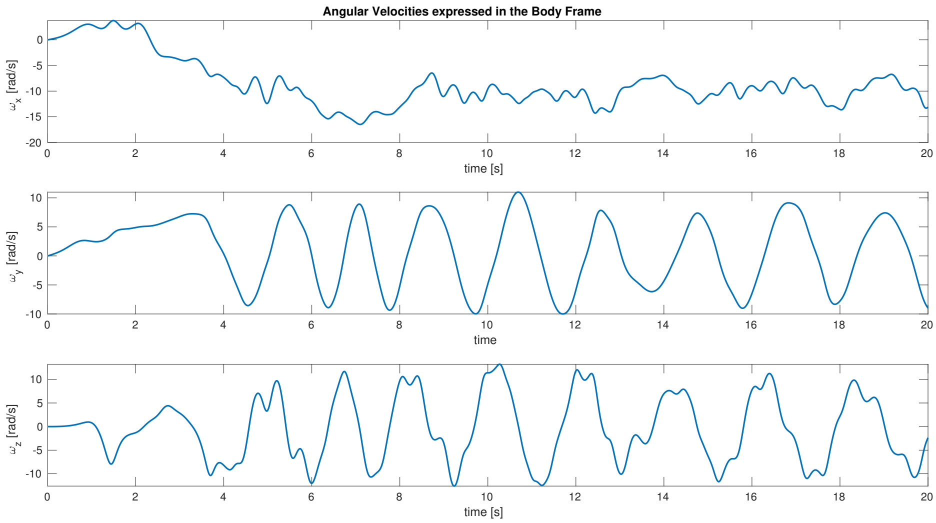

Figure 11: Figure of angular velocities for optimum parameter

.

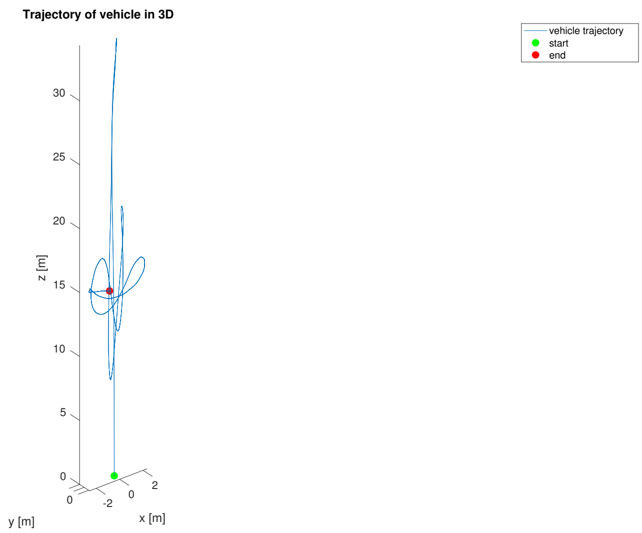

Figure 12: Trajectory of vehicle for optimium parameters

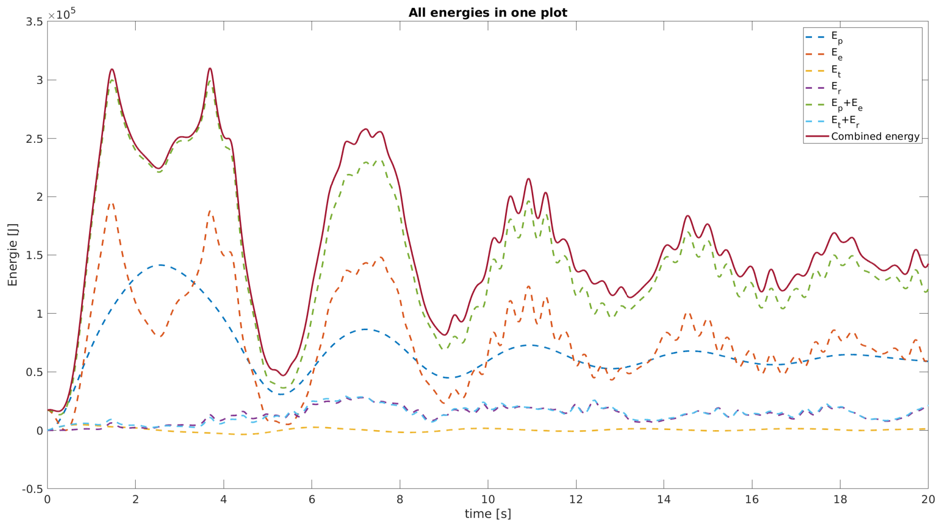

Figure 13: Energies for optimum parameters

4.8 But does the vehicle loop

Since the Moment of inertia matrix is diagonal and express relative to the COM we can use the vector pointing from the center of the vehicle to the COM to examine if the body is rotating around its axis. These needs to be expressed into the global frame $\mathcal{M}$ so the rotation matrix $R_{\mathcal{B}}^{\mathcal{N}}$ is needed to translate $r_c^{\mathcal{B}}$ to the inetial frame;

$$r_c^{\mathcal{N}} = R_{\mathcal{B}}^{\mathcal{N}} r_c^{\mathcal{B}} \tag{56}$$

The simulation results are shown in figure 15. From the body we can see that since it changes sign in a non smooth way the y direction it is looping around the y direction.

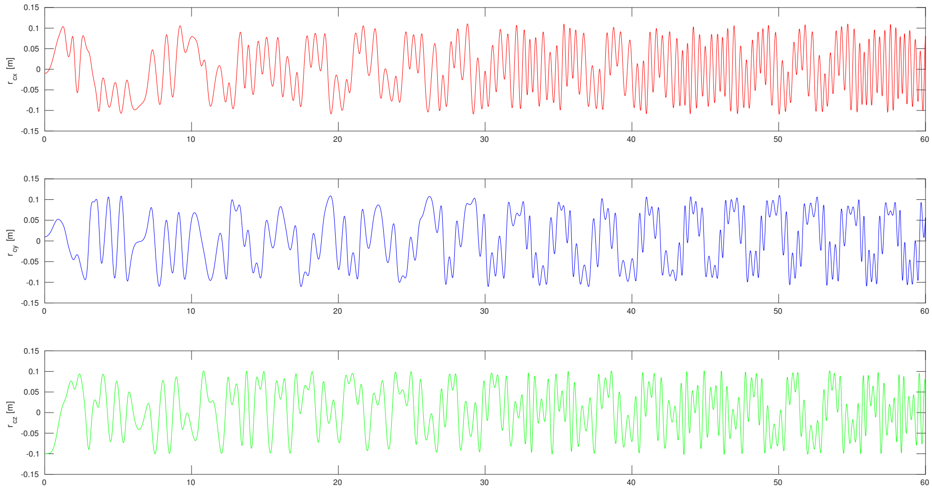

Figure 14: Behavoir of axis pointing from the center to the COM of the body

4.9 Can we also use Euler angles?

We can examine this question by using a cans in series representation. As we know from section 4.8 the vehicle mostly rotates around the global y axis.

When we rotate more around the y axis it is saver to take the [z-y-z] Euler angels since with not much x rotation there will be less change on singularities

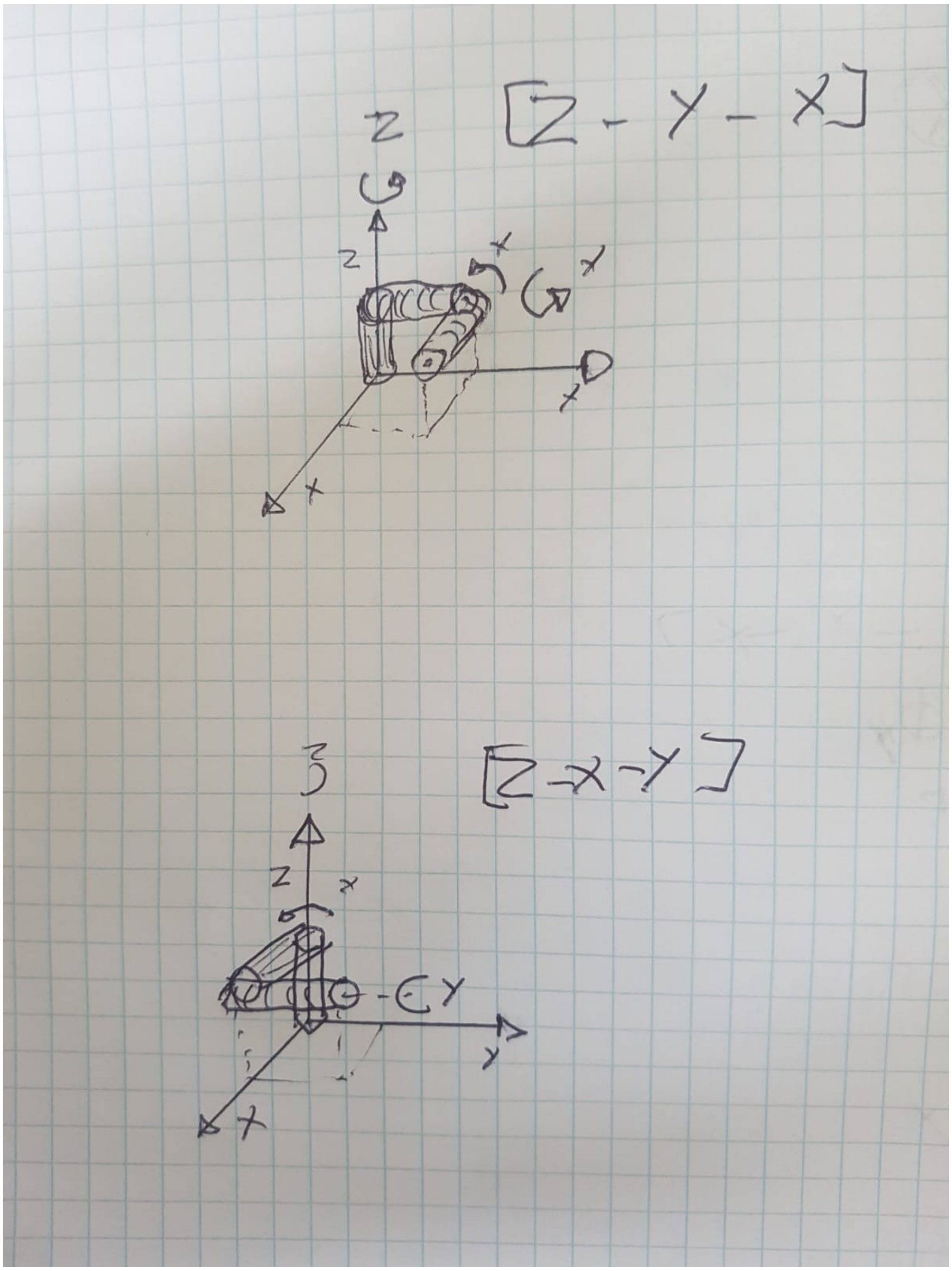

Figure 15: Behavoir of axis pointing from the center to the COM of the body

Appendix A

The main MATLAB script

```
%% MBD_B: Assignment 9 - Euler angles and the human arm
%  Rick Staa (4511328)
% clear all; close all; clc;
fprintf('--- A10 ---\n');

%% Simulation settings
EOM_calc_bool                = 0;                                    % Set on 1 if you want to recalculate the EOM

%% Intergration parameters
parms.sim.sim_time           = 60;                                   % Intergration time
parms.sim.dt                 = 1e-3;                                 % Intergration step size
parms.sim.nmax               = 10;                                   % Max number of coordinate projections
parms.sim.tol                = 1e-12;                                % Maximum allowable drift

%% Model Parameters
% Vehicle parameters
m                 = 420;                           % mass vehicle [kg]
J                 = diag([170 120 140]);           % moment of inertia of vehicle [kg]
r                 = 1;                             % Radius of vehicle [m]
c                 = [-0.01;0.01;-0.1];             % Relative COM position to vehicle center [m]
p_s               = [0;r;0];                       % Connection of bungie cords relative 2 sphere center
A                 = pi*r^2;                        % Frontal area of the vehicle [m]

% Paramters of support collumns
h                 = 25;                            % Hight of supporting collumns [m]
```

```
w                   = 18;                               % Width between the supporting collums [m]
zeta                = 0.1;                              % Damping ratio

%% World parameters
g                   = 9.81;                             % N/kg
rho                 = 1.25;                             % kg/m^3
cd                  = 0.5;                              % Drag coefficient

%% put parameters in struct
parms.m             = m;
parms.J             = J;
parms.r             = r;
parms.c             = c;
parms.A             = A;
parms.h             = h;
parms.w             = w;
parms.zeta          = zeta;
parms.g             = g;
parms.rho           = rho;
parms.cd            = cd;
parms.p_s           = p_s;

% calculate the missing spring parameters
[parms.k,parms.l_0,parms.b] = spring_param_calc(parms);

% Create xtra symbolic variables
syms x y z q0 q1 q2 q3 x_d y_d z_d omega_x omega_y omega_z                 % In this q0 = lambda 0 this was done for code Readability

% Put symbolic variables in struct
parms.syms.x            = x;
parms.syms.y            = y;
parms.syms.z            = z;
parms.syms.q0           = q0;
parms.syms.q1           = q1;
parms.syms.q2           = q2;
parms.syms.q3           = q3;
parms.syms.x_d          = x_d;
parms.syms.y_d          = y_d;
```

```
parms.syms.z_d       = z_d;
parms.syms.omega_x   = omega_x;
parms.syms.omega_y   = omega_y;
parms.syms.omega_z   = omega_z;

%% Set Initial states
% Set euler parameters (In the initial state the axis of ration can said to
% be alighed with the axis through the spring attachment sites.
n                    = [0;1;0];                                              % Axis through spring attachment sites
phi                  = 0;                                                          % No rotation

% Calculate initial states
x0                   = 0;
y0                   = 0;
z0                   = 0;
q0                   = cos(0.5*phi);
q1                   = sin(0.5*phi)*n(1);
q2                   = sin(0.5*phi)*n(2);
q3                   = sin(0.5*phi)*n(3);
x_d                  = 0;
y_d                  = 0;
z_d                  = 0;
omega_x              = 0;
omega_y              = 0;
omega_z              = 0;
x0                   = [x0;y0;z0;q0;q1;q2;q3;x_d;y_d;z_d;omega_x;omega_y;omega_z];

%% Calculate equations of motion
if (EOM_calc_bool == 1)
    EOM_calc(parms);
end

%% Calculate movement by mean sof a Runge-Kuta 4th order intergration method
[t,x,error,r_axis]          = RK4_custom(x0,parms);

```

```matlab
%% Play sound
load gong
sound(y,Fs)

%% Optimize k b and c values
% Try to find the optimal values that get the highest height

% % Create arrays
% flag = 1;
% h_max_val = 0;
% l_0_array = 13:-0.5:8;
% k_array   = 500:50:1000;
% parms.sim.sim_time = 20; % Set to 20 seconds that should be more than enough
% for ii = 1:length(l_0_array)
%     % set l_0
%     parms.l_0 = l_0_array(ii);
%
%     for jj = 1:length(k_array)
%         % Set k
%         parms.k = k_array(jj);
%
%         % Calculate the b that corresponds to a zeta of 0.1 and the given k
%         parms.b           = 2*parms.zeta*sqrt(parms.k*m);
%
%         % Plot for new values
%         [~,x_opt,~,~] = RK4_custom(x0,parms);
%
%         % Get height and save in array
%         if max(x_opt(:,3)) > h_max_val
%             h_max_val   = max(x_opt(:,3));
%             h_max.k    = parms.k;
%             h_max.b    = parms.b;
%             h_max.l_0 = parms.l_0;
%
%             % Exit loop if
%             if h_max_val > (parms.h+0.5*parms.w)
%                 flag = 0;
%             end
%
%             % break out of loop if flag = 0;
%             if flag == 0
```

```
%                 break
%             end
%         end
%         % break out of loop if flag = 0;
%         if flag == 0
%             break
%         end
%     end
%     % break out of loop if flag = 0;
%     if flag == 0
%         break
%     end
% end


%% Calculate energies
%% Energies

% Potential Energy
E_p     = m*g*x(:,3);   % Global axis is at the ground and z is relative to N frame

% Spring Energy
l_s1    = subs_l_s1(parms.l_0,x(:,1),x(:,2),x(:,3),x(:,4),x(:,5), ...
    x(:,6),x(:,7));
l_s2    = subs_l_s2(parms.l_0,x(:,1),x(:,2),x(:,3),x(:,4),x(:,5), ...
    x(:,6),x(:,7));

E_e     = 0.5*parms.k*(l_s1 - parms.l_0).^2 + 0.5*parms.k*(l_s2 - parms.l_0).^2;

% Translational Kinetic Energy
E_t     = 0.5*m*(x(:,8).^2 + x(:,9).^2 + x(:,10));

% Rotational Kinetic Energy
E_r     = 0.5*parms.J(1,1)*x(:,11).^2 + 0.5*parms.J(2,2)*x(:,12).^2 + 0.5*parms.J(3,3)*x(:,13).^2;

%% Create plots

%% Plots for the vehicle movement
```

```
% Plot x y z positions and velocities of the vehicle
figure;
subplot(3,1,1)
plot(t,x(:,1),t,x(:,8),'linewidth',2);
set(gca,'fontsize',14);
title('Linear properties of the COM of vehicle')
xlabel('time [s]');
ylabel('x');
legend('Position [m]','Velocity [m/s]');
subplot(3,1,2)
plot(t,x(:,2),t,x(:,9),'linewidth',2);
set(gca,'fontsize',14);
xlabel('time [s]');
ylabel('y');
legend('Position [m]','Velocity [m/s]');
subplot(313)
plot(t,x(:,3),t,x(:,10),'linewidth',2);
set(gca,'fontsize',14);
xlabel('time [s]');
ylabel('z');
legend('Position [m]','Velocity [m/s]');

% Plot angular velocities of the vehicle
figure(2)
subplot(3,1,1)
plot(t,x(:,11),'linewidth',2);
set(gca,'fontsize',14);
title('Angular Velocities expressed in the Body Frame')
xlabel('time [s]');
ylabel('\omega_x [rad/s]');
subplot(3,1,2)
plot(t,x(:,12),'linewidth',2);
set(gca,'fontsize',14);
xlabel('time');
ylabel('\omega_y [rad/s]');
subplot(3,1,3)
plot(t,x(:,13),'linewidth',2);
set(gca,'fontsize',14);
xlabel('time [s]');
ylabel('\omega_z [rad/s]');

% Plot trajectory of vehicle in 3D
figure;
```

```
plot3(x(:,1),x(:,2),x(:,3));
hold on;
plot3(x(1,1),x(1,2),x(1,3),'g*','linewidth',8); % Plot beginning
plot3(x(end,1),x(end,2),x(end,3),'r*','linewidth',8);
set(gca,'fontsize',14);
title('Trajectory of vehicle in 3D')
xlabel('x [m]');
ylabel('y [m]');
zlabel('z [m]');
legend('vehicle trajectory','start','end');
axis equal;

% Plot spring force magnitude
figure;
subplot(4,1,1)
plot(t,subs_F_spring(parms.k,parms.l_0,x(:,1),x(:,2),x(:,3),x(:,4), ...
    x(:,5),x(:,6),x(:,7)),'linewidth',2);
set(gca,'fontsize',14);
title('Forces (in N frame) and Moments (in B frame)')
ylabel('F_{spring} [N]');
subplot(4,1,2)
plot(t,subs_F_damp(parms.b,x(:,1),x(:,2),x(:,3),x(:,4),x(:,5),x(:,6), ...
    x(:,7),x(:,8),x(:,9),x(:,10)),'linewidth',2);
set(gca,'fontsize',14);
ylabel('F_{damping} [N]');
subplot(4,1,3)
plot(t,sum(subs_F_drag(x(:,8)',x(:,9)',x(:,10)').^2),'linewidth',2);
set(gca,'fontsize',14);
ylabel('F_{drag} [N]');
subplot(4,1,4)
plot(t,sum(subs_M(parms.k,parms.l_0,parms.b,x(:,1)',x(:,2)',x(:,3)', ...
    x(:,4)',x(:,5)',x(:,6)',x(:,7)',x(:,8)',x(:,9)',x(:,10)').^2) ...
    ,'linewidth',2);
set(gca,'fontsize',14);
ylabel('M_{total} [Nm]');
xlabel('time [s]');

```

```
% Plot the accuracy of the solution
figure;
plot(t,abs(error));
hold on;
plot([0 parms.sim.sim_time],[1e-12 1e-12],'linewidth',2,'linestyle','--');
set(gca,'fontsize',14);
ylim([0 1.15e-12]);
xlabel('time [s]');
ylabel('Intergration drift (Error on normality constraint)');
title('Intergration accuracy of the solution against the time')
legend('Intergration error','max allowed error line','location','best');

%% Plot for the energies

% First plot the individual energies
figure;
subplot(4,1,1)
plot(t,E_p,'linewidth',2);
set(gca,'fontsize',14);
title('Individual energies')
ylabel('E_p [J]');
subplot(4,1,2)
plot(t,E_e,'linewidth',2);
set(gca,'fontsize',14);
ylabel('E_e [J]');
xlabel('time [s]');
subplot(4,1,3)
plot(t,E_t,'linewidth',2);
set(gca,'fontsize',14);
ylabel('E_t [J]');
xlabel('time [s]');
subplot(4,1,4)
plot(t,E_r,'linewidth',2);
set(gca,'fontsize',14);
ylabel('E_r [J]');
xlabel('time [s]');

% Now plot interesting combinations of energy
figure;
```

```
plot(t,E_p,'linewidth',2,'linestyle','--');hold on %
    Gravitational
plot(t,E_e,'linewidth',2,'linestyle','--');hold on %
    Gravitational
plot(t,E_t,'linewidth',2,'linestyle','--');hold on;
plot(t,E_r,'linewidth',2,'linestyle','--');hold on;
plot(t,E_p+E_e,'linewidth',2,'linestyle','--'); hold on; %
    Full potential energy
plot(t,E_t+E_r,'linewidth',2,'linestyle','--'); hold on;
plot(t,E_p+E_e+E_t+E_r,'linewidth',2,'linestyle','-');
set(gca,'fontsize',14);
title('All energies in one plot');
legend('E_p','E_e','E_t','E_r','E_p+E_e','E_t+E_r','Combined
    energy');
ylabel('Energie [J]');
xlabel('time [s]');

%% Plot xomponents of the axis through the spring attachment
    sites expressed in the global fram
figure;
subplot(3,1,1);
set(gca,'fontsize',14)
title('Check for looping')
plot(t,r_axis(:,1),'r');
ylabel('r_{cx} [m]');
subplot(3,1,2);
plot(t,r_axis(:,2),'b');
ylabel('r_{cy} [m]');
subplot(3,1,3);
plot(t,r_axis(:,3),'g');
ylabel('r_{cz} [m]');

%% FUNCTIONS

%% Runge-Kuta numerical intergration function
% This function calculates the motion of the system by means
    of a
% Runge-Kuta 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 [time,x,error,r_axis] = RK4_custom(x0,parms)
```

```
% Initialise variables
time                    = (0:parms.sim.dt:parms.sim.sim_time).';                    % Create time array
x                       = zeros(length(time),length(x0));                  % Create empty state array
x(1,1:length(x0))       = x0;                                                % Put initial state in array
error                   = zeros(length(time),1);

% preallocate memory for rotation axis
R_tmp = subs_R_B_N(x0(4),x0(5),x0(6),x0(7));
r_axis                  = zeros(length(time),3);
r_axis(1,:)             = (R_tmp*parms.c)';

% Caculate the motion for the full simulation time by means of a
% Runge-Kutta4 method

% Perform intergration till end of set time
for ii = 1:(size(time,1)-1)

    % Perform RK 4
    x_now_tmp               = x(ii,:);                          % Create cell for subs function function
    x_input                 = num2cell([parms.k,parms.l_0,parms.b,x(ii,:)],1);                  % Add time to state
    K1                      = subs_Xdd(x_input{:}).';                         % Calculate the second derivative at the start of the step
    x1_tmp                  = x_now_tmp + (parms.sim.dt*0.5)*K1;              % Create cell for subs function function
    x1_input                = num2cell([parms.k,parms.l_0,parms.b,x1_tmp],1);                  % Add time to state
    K2                      = subs_Xdd(x1_input{:}).';                        % Calculate the second derivative halfway the step
    x2_tmp                  = x_now_tmp + (parms.sim.dt*0.5)*K2;              % Refine value calculation with new found derivative
```

```
    x2_input              = num2cell([parms.k,parms.l_0,parms.b,x2_tmp],1);                  % Add time to state
    K3                    = subs_Xdd(x2_input{:}).';                                         % Calculate new derivative at the new refined location
    x3_tmp                = x_now_tmp + (parms.sim.dt)*K3;                                   % Calculate state at end step with refined derivative
    x3_input              = num2cell([parms.k,parms.l_0,parms.b,x3_tmp],1);                  % Add time to state
    K4                    = subs_Xdd(x3_input{:}).';                                         % Calculate last second derivative
    x(ii+1,:)             = x_now_tmp + (parms.sim.dt/6)*(K1+2*K2+2*K3+K4);                  % Perform euler intergration step

    % Correct for intergration drift (Renormalise the axis of rotation)

    %% Coordinate projection method (Gaus-newton method)
    % Correct for intergration drift
    x_now_tmp = x(ii+1,:);
    [x_new,error_tmp] = gauss_newton(x_now_tmp,parms);

    % Calculate the roation of the vector pointing from the center to one
    % of the springs

    % For Question f
    R_tmp = subs_R_B_N(x_new(4),x_new(5),x_new(6),x_new(7));
    r_axis(ii,:) = (R_tmp*parms.c)';

    % Overwrite position coordinates
    x(ii+1,:)         = x_new;
    error(ii+1,:)     = error_tmp;
end
end

%% Speed correct function
function [x,error] = gauss_newton(x,parms)

```

```
% Get rid of the drift by solving a non-linear least square problem by
% means of the Gaus-Newton method

%% Gaus-newton velocity constraint correction
n_iter              = 0;                                % Set iteration counter           % Get position data out

% % Calculate the two needed constraints
D                       = subs_D(x(4),x(5),x(6),x(7));
Dd                      = subs_Dd(x(4),x(5),x(6),x(7));

% Solve drift using gaus-newton iteration
while (max(abs(D)) > parms.sim.tol)&& (n_iter < parms.sim.nmax)
    x_tmp               = x;
    n_iter              = n_iter + 1;
    x_del               = Dd.'*inv(Dd*Dd.')*-D;
    x                   = x_tmp + x_del.';

    % Recalculate constraint
    D                       = subs_D(x(4),x(5),x(6),x(7));
    Dd                      = subs_Dd(x(4),x(5),x(6),x(7));
end


% Store full error
error = D;
end

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

%% Get parameters and variables

% create symbolic variables
syms k b l_0;

% Unpack variables for clarity
m                   = parms.m;
J                   = parms.J;
```

```
r                   = parms.r;
c                   = parms.c;
A                   = parms.A;
h                   = parms.h;
w                   = parms.w;
g                   = parms.g;
rho                 = parms.rho;
c_d                 = parms.cd;
p_s                 = parms.p_s;

% Unpack symbolic variables from parms
x                   = parms.syms.x;
y                   = parms.syms.y;
z                   = parms.syms.z;
q0                  = parms.syms.q0;
q1                  = parms.syms.q1;
q2                  = parms.syms.q2;
q3                  = parms.syms.q3;
x_d                 = parms.syms.x_d;
y_d                 = parms.syms.y_d;
z_d                 = parms.syms.z_d;
omega_x             = parms.syms.omega_x;
omega_y             = parms.syms.omega_y;
omega_z             = parms.syms.omega_z;

% Create small generalised spring state
x_state             = [x;y;z];
xd_state            = [x_d;y_d;z_d];
omega_state         = [omega_x;omega_y;omega_z];

%% Calculate Rotation Matrix

R_B_N = [q0^2+q1^2-q2^2-q3^2, 2*(q1*q2-q0*q3),      2*(q1*q3-q0*q2);      ...
   2*(q2*q1-q0*q3),     q0^2-q1^2+q2^2-q3^2 , 2*(q2*q3-q0*q1);      ...
   2*(q3*q1-q0*q2),     2*(q3*q2-q0*q1),      q0^2-q1^2-q2^2+q3^2];

matlabFunction(R_B_N,'File','subs_R_B_N');

%% Calculate Spring contribution to virtual work
r_c                 = [x;y;z] + R_B_N*-parms.c;
```

```
r_s1              = r_c + R_B_N*-parms.p_s;
r_s2              = r_c + R_B_N*parms.p_s;

% Create vetors along the spring
l_s1              = r_s1 - [0; -w/2; h];
l_s2              = r_s2 - [0; w/2; h];

% Calculate delta spring lenght
l_s2              = sqrt(sum(l_s2.^2)) - l_0;
l_s1              = sqrt(sum(l_s1.^2)) - l_0;

% Calculate spring forces
sigma_s1          = k*l_s1;
sigma_s2          = k*l_s2;

%% Damping Components
sigma_c1          = b*jacobian(l_s1,x_state)*xd_state;
sigma_c2          = b*jacobian(l_s2,x_state)*xd_state;

%% Air Drag
sigma_a           = 0.5*rho*A*c_d*sqrt(sum(xd_state.^2))*
   xd_state;

%% Add the contributions of all forces together

% External intertial forces
F_ext             = [0;0;-m*g];

% Spring, damper and drag forces
F                 = [F_ext - sigma_s1*jacobian(l_s1,x_state)' -
    sigma_s2*jacobian(l_s2,x_state)' - ...
    sigma_c1*jacobian(l_s1,x_state)' - sigma_c2*jacobian(l_s2
       ,x_state)' - sigma_a];

%% Now finaly calculate the linear accelerations by means of
   gaussian elimination
M                 = diag([m m m]);
xdd_lin           = inv(M)*F;

%% Now lets find the angular accelerations
% Calculate forces
F1                = sigma_s1*jacobian(l_s1,x_state).' +
   sigma_c1*jacobian(l_s1,x_state).';
```

```
F2                 = sigma_s2*jacobian(l_s2,x_state).' +
    sigma_c2*jacobian(l_s2,x_state).';

% Calculate moment arms
r1                 = -parms.c-parms.p_s;
r2                 = -parms.c+parms.p_s;

% Calculate moments
M                  = cross(r1,R_B_N'*F1) + cross(r2,R_B_N'*F2);

%% Calculate angular accelerations

omega_d_state     = inv(J)*(M - cross(omega_state,J*omega_state
    ));

%% Calculate the derivative of the omegas and put them in one
     big state variable
X_state            = [x,y,z,q0,q1,q2,q3,x_d,y_d,z_d,omega_x,
    omega_y,omega_z].';

dlambda            = 0.5*[q0 -q1 -q2 -q3;...
     q1 q0 -q3 q2;...
     q2 q3 q0 -q1;...
     q3 -q2 q1 q0]*[0;omega_state];
Xdd                = [xd_state;dlambda;xdd_lin;omega_d_state];

%% Calculate the constraint
D                  = q0^2 + q1^2 + q2^2 + q3^2 - 1;
Dd                 = jacobian(D,X_state);

% Save all the symbolic expressions in function files
matlabFunction(D,'File','subs_D','vars',[q0 q1 q2 q3]);
matlabFunction(Dd,'File','subs_Dd','vars',[q0 q1 q2 q3]);
matlabFunction(l_s1,'File','subs_l_s1','vars',[l_0,x,y,z,q0,
    q1,q2,q3]);
matlabFunction(l_s2,'File','subs_l_s2','vars',[l_0,x,y,z,q0,
    q1,q2,q3]);
matlabFunction((sigma_s1+sigma_s2),'File','subs_F_spring','
    vars',[k,l_0,x,y,z,q0,q1,q2,q3]);
matlabFunction((sigma_c1+sigma_c2),'File','subs_F_damp','vars
    ',[b,x,y,z,q0,q1,q2,q3,x_d,y_d,z_d]);
matlabFunction((sigma_a),'File','subs_F_drag');
```

```
matlabFunction(M,'File','subs_M','vars',[k,l_0,b,x,y,z,q0,q1,
    q2,q3,x_d,y_d,z_d]);
matlabFunction(X_state,'File','subs_X_state','Vars',[x,y,z,q0
    ,q1,q2,q3,x_d,y_d,z_d,omega_x,omega_y,omega_z]);
matlabFunction(Xdd,'File','subs_Xdd','Vars',[k,l_0,b,x,y,z,q0
    ,q1,q2,q3,x_d,y_d,z_d,omega_x,omega_y,omega_z]);

end

%% Calculate spring-damper paramters

function [k,l0,b] = spring_param_calc(parms)
% Unpack variables for clarity
m                = parms.m;
J                = parms.J;
r                = parms.r;
c                = parms.c;
A                = parms.A;
zeta             = parms.zeta;
h                = parms.h;
w                = parms.w;
g                = parms.g;
rho              = parms.rho;
cd               = parms.cd;
p_s              = parms.p_s;

%% A) Calculate k and L0 paramters
% This can be done by using the principle of conservation of
    energy and
% newtons second law of motion.

% Create symbolic variables
syms k l0

% Calculate needed spring lengths
l_t         = sqrt((0.5*w)^2 + (0.5*w-r)^2);
l_b         = sqrt(h^2 + (0.5*w-r)^2);

% Conservation of energy equation
eq_con       = k*(l_t - l0)^2+m*g*(0.5*w+h)-k*(l_b-l0)^2;

% Force equation (Newtons second law of motion)
eq_F         = 2*k*(l_b-l0)*cos(atan2((0.5*w-r),h))-5.8*m*g;
```

```

% Put equations in one vector
eq          = [eq_con; eq_F];

% Solve these two equations for two unknowns with the
    symbolic toolbox
sol         = solve(eq,[k;l0]);
k           = double(sol.k);
l0          = double(sol.l0);

% Calculate damping ratio
b           = 2*zeta*sqrt(k*m);
end
```

References

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