

Multibody Dynamics B - Assignment 7

ME41055

Prof. Arend L. Schwab

Head TA: Simon vd. Helm

Rick Staa

#4511328

Lab Date: 26/04/2018

Due Date: 03/05/2018

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 initial values with Prajish Kumar (4743873).

3 Setup overview

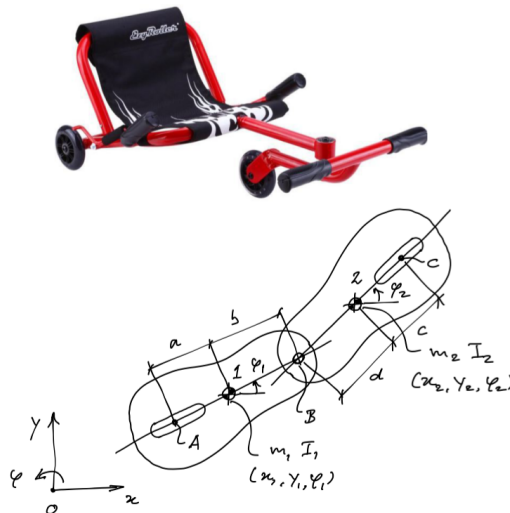


Figure 2: Quick return mechanism as depicted in assignment 7

4 Problem Statement

In this assignment we were asked to derive the motion of the EzyRoller Mechanism (see fig 2). This EzyRoller mechanism has the following parameters:

$$a = 0.5m \quad (1)$$

$$b = 0.5m \quad (2)$$

$$c = 0.125m \quad (3)$$

$$d = 0.125m \quad (4)$$

$$m1 = 1kg \quad (5)$$

$$m2 = 0kg \quad (6)$$

$$J1 = 0.1kgm^2 \quad (7)$$

$$J2 = 0kgm^2 \quad (8)$$

$$g = 9.81m/s^2 \quad (9)$$

$$(10)$$

Since the EOM were asked in the implicit form we will use the COM coordinates as the state. From this state the position of all the other points on the Ezyroller can be calculated.

$$x0 = [x_1 \ y_1 \ \phi_{i1} \ x_2 \ y_2 \ \phi_{i2}] \quad (11)$$

In the first part of the question there were no external forces or torques applied to the EzyRoller. We were asked to choose a set of initial states that comply with the given constraints. I choose the following initial states:

$$x0 = [x_1 \ y_1 \ \phi_{i1} \ x_2 \ y_2 \ \phi_{i2} \ \dot{x}_1 \ \dot{y}_1 \ \dot{\phi}_{i1} \ \dot{x}_2 \ \dot{y}_2 \ \dot{\phi}_{i2}] \quad (12)$$

$$x0 = [a \ 0 \ 0 \ a+b \ d \ \pi/2 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0] \quad (13)$$

With these initial states the EOM can be derived in implicit form by putting the Newton-Euler equations (explained in CH1-CH2 [1]) and the constraint equations in one big matrix vector product. Following to get the state derivative this system of equations can then be solved by using Gaussian elimination.

$$\begin{pmatrix} M_{ij} & C_{k,i} & S_{mi} \\ C_{k,j} & \mathbf{0} & \mathbf{0} \\ S_{mj} & \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \ddot{x}_j \\ \lambda_k \\ \lambda_m \end{pmatrix} = \begin{pmatrix} F_i \\ -C_{k,jl} \dot{x}_j \dot{x}_l \\ -S_{mj,l} \dot{x}_j \dot{x}_l \end{pmatrix},$$

Figure 3: Caption

4.1 Equations of motion(EOM)

After applying the earlier explained procedure we get the following system of equations:

In this $M_{i,j}$ depicts the mass matrix, $C_{k,j}$ the Jacobean of the holonomic constraints (position constraint) and $D_{k,j}$ the Jacobean of the non-holonomic constraints (velocity constraint). The right hand side of this system of equations contains the force vector F and the convective e terms. In our example the left hand size matrix A equal to:

$$A = \begin{pmatrix} m_1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -\sin(\varphi_1) & 0 \\ 0 & m_1 & 0 & 0 & 0 & 0 & 0 & 1 & \cos(\varphi_1) & 0 \\ 0 & 0 & J_1 & 0 & 0 & 0 & -b \sin(\varphi_1) & b \cos(\varphi_1) & -a & 0 \\ 0 & 0 & 0 & m_2 & 0 & 0 & -1 & 0 & 0 & -\sin(\varphi_2) \\ 0 & 0 & 0 & 0 & m_2 & 0 & 0 & -1 & 0 & \cos(\varphi_2) \\ 0 & 0 & 0 & 0 & 0 & J_2 & -d \sin(\varphi_2) & d \cos(\varphi_2) & 0 & -c \\ 1 & 0 & -b \sin(\varphi_1) & -1 & 0 & -d \sin(\varphi_2) & 0 & 0 & 0 & 0 \\ 0 & 1 & b \cos(\varphi_1) & 0 & -1 & d \cos(\varphi_2) & 0 & 0 & 0 & 0 \\ -\sin(\varphi_1) & \cos(\varphi_1) & -a & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\sin(\varphi_2) & \cos(\varphi_2) & -c & 0 & 0 & 0 & 0 \end{pmatrix} \quad (14)$$

and B matrix is equal to:

$$B = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ b \cos(\varphi_1) \dot{\varphi}_1^2 + d \cos(\varphi_2) \dot{\varphi}_2^2 \\ b \sin(\varphi_1) \dot{\varphi}_1^2 + d \sin(\varphi_2) \dot{\varphi}_2^2 \\ \dot{\varphi}_1 (\dot{x}_1 \cos(\varphi_1) + \dot{y}_1 \sin(\varphi_1)) \\ \dot{\varphi}_2 (\dot{x}_2 \cos(\varphi_2) + \dot{y}_2 \sin(\varphi_2)) \end{pmatrix} \quad (15)$$

$$F = [0 \ 0 \ 0 \ 0 \ 0 \ 0] \quad (16)$$

4.1.1 holonomic constraints

The mechanism of this example had 2 holonomic constraints in point B. These are defined as follos:

$$C = \begin{pmatrix} x_1 - x_2 + b \cos(\varphi_1) + d \cos(\varphi_2) \\ y_1 - y_2 + b \sin(\varphi_1) + d \sin(\varphi_2) \end{pmatrix} \quad (17)$$

To add this constraints to the EOM we need to differentiate it two times to get them in terms of accelerations. The Jacobean and Hessian of these constraints were calculated by symbolic toolbox and is therefore not displayed.

4.1.2 non-holonomic constraints

The non-holonomic constraints for this mechanism can be found in the two wheels. These constraint ensure that there is no lateral movement of the wheels. The non-holonomic constraints in our can be derived by calculating the x and y velocity in point A and C. This is done with the relative velocity theorhem:

$$V_A = V_{COM1} + \omega \times r_{A/COM1} \quad (18)$$

After the velocity of point A and C are calculated we can use the dot product to project them onto the tangential and normal wheel components. By following setting the normal velocity component (The component pointing out of the wheel axle to 0 we get the following velocity constraints:

$$D = \begin{pmatrix} \dot{y}_1 \cos(\varphi_1) - a \dot{\varphi}_1 - \dot{x}_1 \sin(\varphi_1) \\ c \dot{\varphi}_2 + \dot{y}_2 \cos(\varphi_2) - \dot{x}_2 \sin(\varphi_2) \end{pmatrix} \quad (19)$$

To add these constraints to the EOM we only need to calculate the first derivative. The jacobian of the velocity constraint was calculated by symbolic toolbox and is therefore not displayed. They can however be found in the A matrix (equation 14).

4.2 Numerical intergration method

To get the movement of EzyRoller in time we will use a 4th 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.

4.2.1 Runge-Kutta 4th order method (RK4)

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

$$k_1 = f(t_n, y_n) \quad (20)$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \quad (21)$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \quad (22)$$

$$k_4 = f(t_n + h, y_n + hk_3) \quad (23)$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (24)$$

4.2.2 Gauß-Newton corrections

The Gauß-Newton we are using here is a non-linear least-square constraint optimization method. This is also called the coordinate projection method since we project the state onto the constraints. In our problem we have the following optimization problem:

$$\|\bar{\mathbf{q}}_{n+1} - \mathbf{q}_{n+1}\|_2 = \min_{\mathbf{q}_{n+1}}, \quad \forall \quad \{\mathbf{q}_{n+1} | \mathbf{C}(\mathbf{q}_{n+1}) = \mathbf{0}\}.$$

In words, this method evaluates the constraint values (Both position and velocity) to see how much the integrated value deviates from the constraint surface. It then looks for the point on the constraint surface that is closest to our original point (see 4).

The earlier named non-linear constraint optimization problem is easily solved by an iterative method. The idea of this method is that you look at a small change around the current state \mathbf{q} :

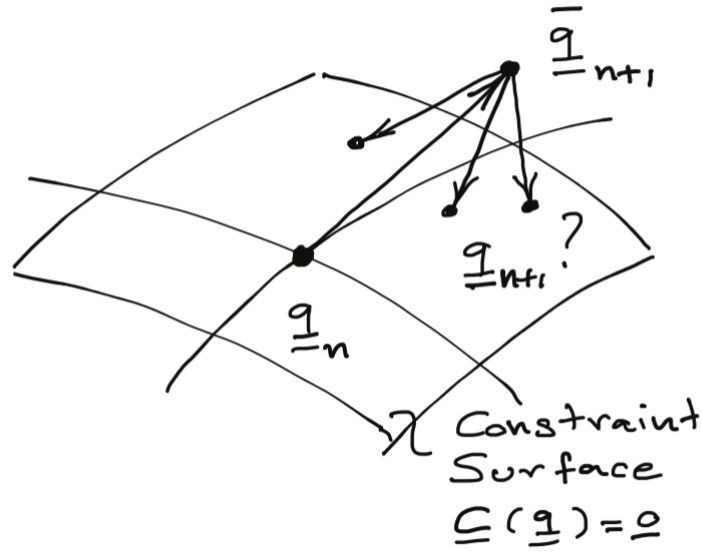


Figure 4: coordinate projection method explained

$$\mathbf{q}_{n+1} = \bar{\mathbf{q}}_{n+1} + \Delta \mathbf{q}_{n+1}.$$

When you fill this in in the original optimization problem you get the following equation:

$$\Delta \mathbf{q}_{n+1} = \mathbf{0}, \quad \forall \quad \{\Delta \mathbf{q}_{n+1} | \{ \mathbf{C}(\bar{\mathbf{q}}_{n+1}) + \mathbf{C}_{,q}(\bar{\mathbf{q}}_{n+1}) \Delta \mathbf{q}_{n+1} = \mathbf{0} \}.$$

This leads to the following system of equations:

$$\begin{pmatrix} \mathbf{I} & \mathbf{C}^T \\ \mathbf{C} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \Delta \\ \mu \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{e} \end{pmatrix}.$$

In which:

$$-\mathbf{C}\mathbf{C}^T \mu = \mathbf{e}.$$

$$\begin{aligned} \mu &= -(\mathbf{C}\mathbf{C}^T)^{-1} \mathbf{e}, \\ \Delta &= \mathbf{C}^T (\mathbf{C}\mathbf{C}^T)^{-1} \mathbf{e}. \end{aligned}$$

In the end you obtain:

$$\Delta = \mathbf{C}^+ \mathbf{e}.$$

With this you can calculate a new \mathbf{q} that is closer to the constraint surface as:

$$\mathbf{q}_{new} = \mathbf{q}_{old} + \Delta \quad (25)$$

When the \mathbf{q}_{new} is obtained you can recalculate the \mathbf{C} and $\dot{\mathbf{C}}$ and start the process over again. In our example we repeat this process till or 10 function iterations are done or the constraints are smaller than 10^{-12} . This procedure is applied to both the position and velocity of the quick return mechanism.

Position scheme explanation

For the position constraints since they are non-linear we will need to use the loop described above this was implemented in matlab as follows:

```

468 % Solve non-linear constraint least-square problem
469 - while (max(abs(C)) > parms.tol) && (n_iter < parms.nmax)
470 -     x_tmp = x(1:6);
471 -     n_iter = n_iter + 1;
472 -     x_del = Cd*inv(Cd.*Cd)*-C.';
473 -     x(1:6) = x_tmp+ x_del.';
474
475 % Recalculate constraint
476 - [C,Cd,~,~] = constraint_calc(x,parms);
477 - end

```

Figure 5: Matlab code doing the position correction

velocity

Since the velocity constraints are linear to compensate for the velocity drift we can do this more easily. The MATLAB code implementing this is shown below:

```

183 % Calculate the corresponding speeds
184 - q_tmp_vel = q(4:6);
185 - Dqd_n1 = -Cd*inv(Cd.*Cd)*Cd.*q_tmp_vel.';
186 - q(4:6) = q_tmp_vel + Dqd_n1.';
187

```

Figure 6: Matlab code doing the velocity correction

In these MATLAB scripts \mathbf{C} depicts the position constraints, \mathbf{C}_d the Jacobean of these constraints, \mathbf{D} the velocity constraints and \mathbf{D}_d the Jacobean of these velocity constraints.

Sd is simply the matrix of both the holonomic and non-holonomic velocity constraints together.

5 Results

5.1 Non powered mechanism

First we were asked to implement a non-powered version of the Ezyroller. The full MATLAB code implementing the model can be found in appendix A. After this model was created I tested the model with three initial conditions.

$$x0 = \begin{bmatrix} a & 0 & 0 & a+b & d & \pi/2 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad (26)$$

$$x0 = \begin{bmatrix} a & 0 & 0 & a+b & d & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad (27)$$

$$x0 = \begin{bmatrix} a & 0 & \pi/2 & a+b & d & \pi/2 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \quad (28)$$

While doing this I used intuition to see if the motion of the EzyRoller was the one expected. I did this by looking at the animation and the plot of the path. The path of the most interesting condition (equation 26) is shown in figure 7. From the figure we see that as we put an input x-velocity on the COM of the first body while the second body is under an angle of $\pi/2$ the first body will push the second body upwards. Further since the second body applies a reaction force on the first body the whole mechanism will go upwards. The other initial conditions also displayed expected behavior (the mechanism moves in a horizontal or vertical straight line). From the animation we can also see that our drift correction works correctly since the Mechanism doesn't fall apart.

5.2 Powered mechanism

To get the powered mechanism we add the following torques to the force matrix F:

$$F = \begin{bmatrix} 0 & 0 & T1 & 0 & 0 & T2 \end{bmatrix} \quad (29)$$

In this $T1 = -M1 * \cos(\pi * t)$ and $T2 = M1 * \cos(\pi * t)$. The new B matrix now becomes:

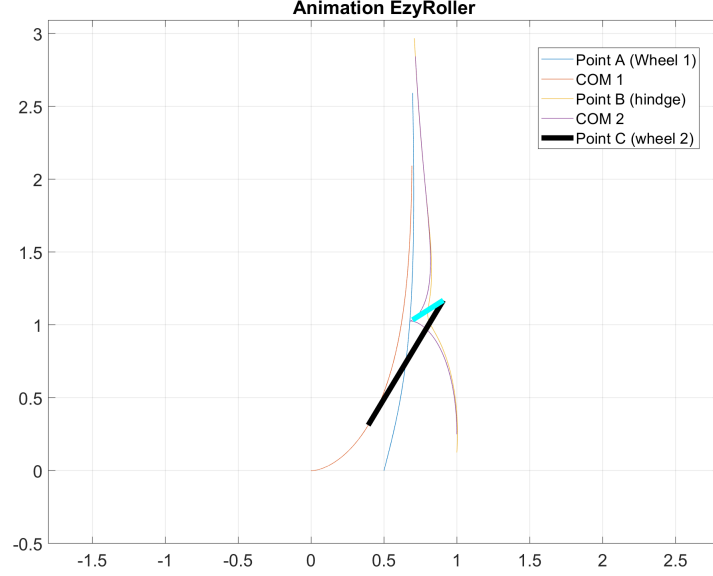


Figure 7: Path of the points on the EzyRoller

$$B = \begin{pmatrix} 0 \\ 0 \\ -\frac{\cos(\pi t)}{10} \\ 0 \\ 0 \\ \frac{\cos(\pi t)}{10} \\ b \cos(\varphi_1) \dot{\varphi}_1^2 + d \cos(\varphi_2) \dot{\varphi}_2^2 \\ b \sin(\varphi_1) \dot{\varphi}_1^2 + d \sin(\varphi_2) \dot{\varphi}_2^2 \\ \dot{\varphi}_1 (\dot{x}_1 \cos(\varphi_1) + \dot{y}_1 \sin(\varphi_1)) \\ \dot{\varphi}_2 (\dot{x}_2 \cos(\varphi_2) + \dot{y}_2 \sin(\varphi_2)) \end{pmatrix} \quad (30)$$

Further we are instructed to use the following initial state.

$$x_0 = [a \ 0 \ 0 \ a+b \ d \ \pi \ 0 \ 0 \ 0 \ 0 \ 0 \ 0] \quad (31)$$

5.2.1 Path of the mechanism

In figure 8 the path of the mechanism is plotted. From this figure we can see that with input torques The EzyRoller follows a path that goes slightly upwards. As we have put segment 1 of the roller aligned with the horizontal and the second segment aligned with

the vertical this path is to be expected. We can further notice that this path looks linear, however when we Zoom in (see figure 9)we see that it actually is comprised of small oscillations around this linear path.

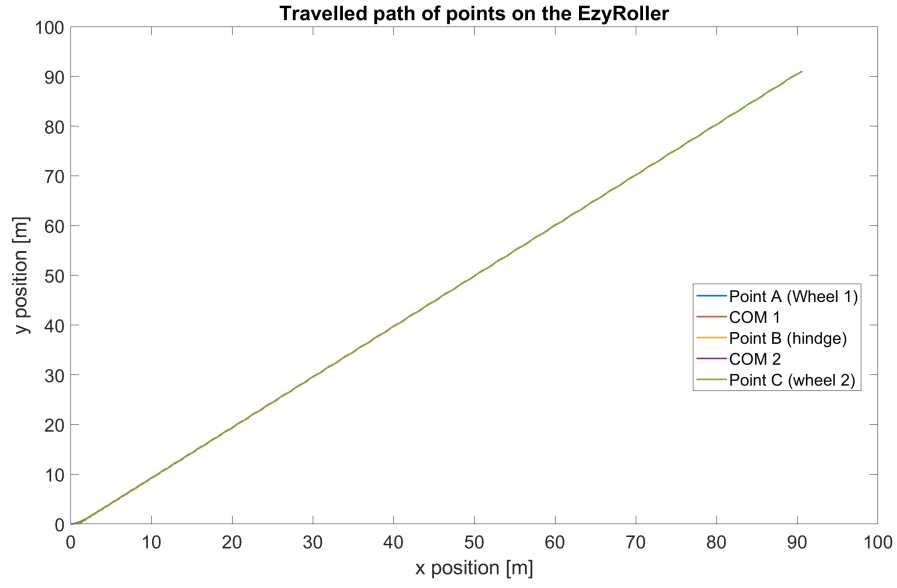


Figure 8: Path of the points on the EzzRoller

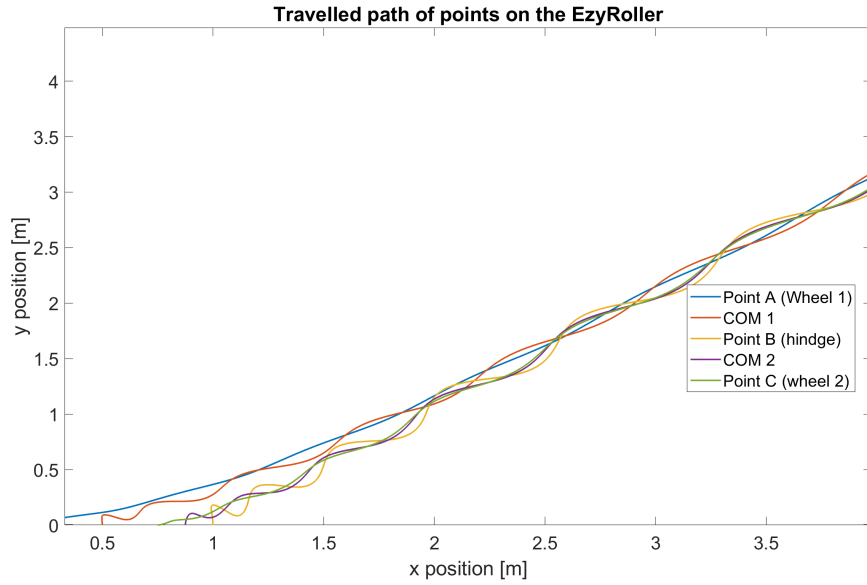


Figure 9: Path of the points on the EzzRoller Zoomed in

5.2.2 Linear and angular velocities

In figure 11 the linear velocities of the COM's of the two segments are shown. From the figure we can see that the mechanism displays oscillatory behavior and that both the x and y velocities of the COM's are oscillating around the a given velocity magnitude ??.

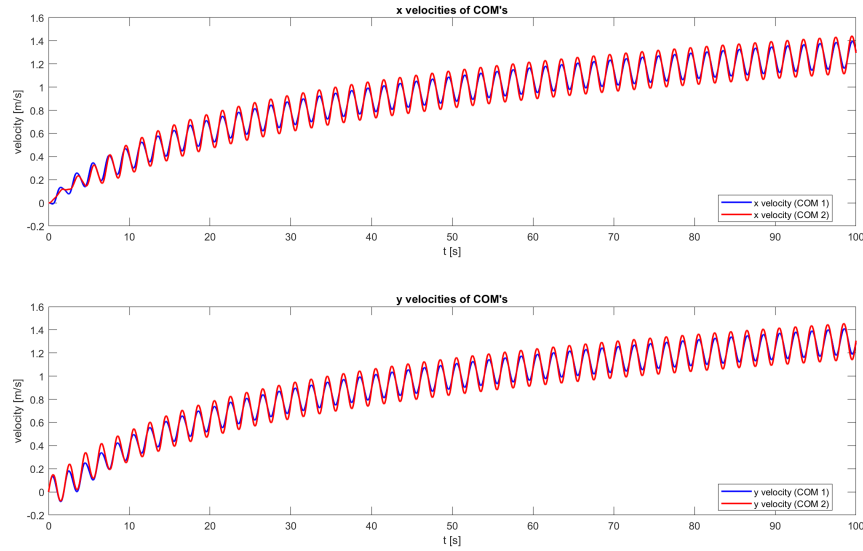


Figure 10: Linear velocities of EzzRoller

In figure 12 the angular velocities are shown. We can see from the figure that both segments display oscillatory behavior and that the amplitude segment 2 is bigger than segment 1. This is probably due to the difference in segment parameters.

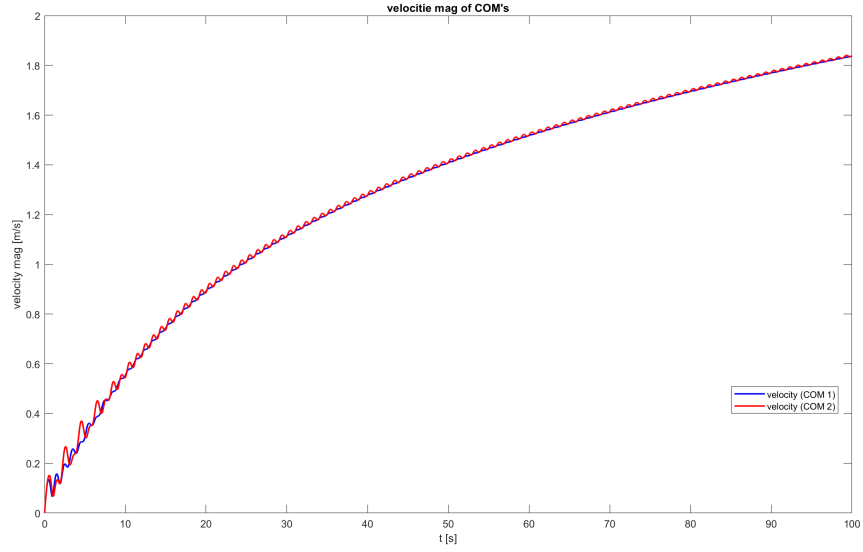


Figure 11: Magnitudes of Linear velocities of EzzRoller

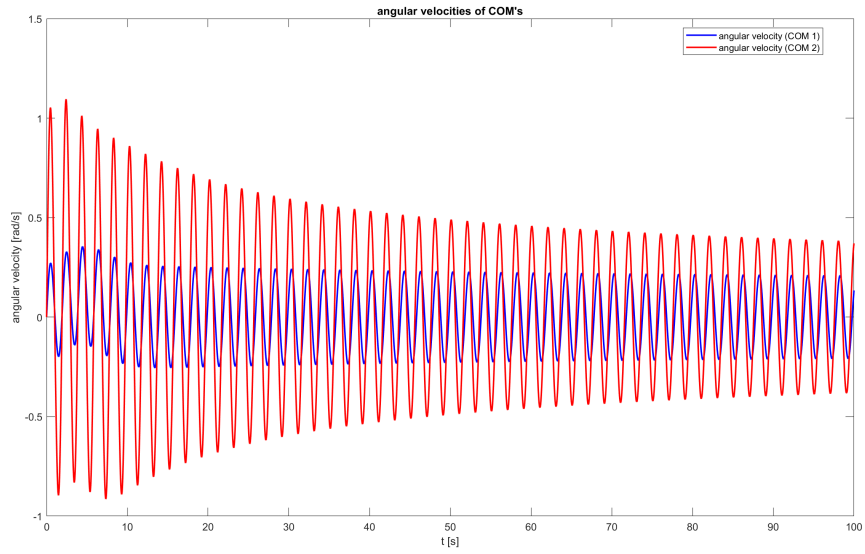


Figure 12: Angular velocities of EzzRoller

5.2.3 kinetic energy and Torque work

In figure 13 the kinetic energy and the work created by the torque are shown.

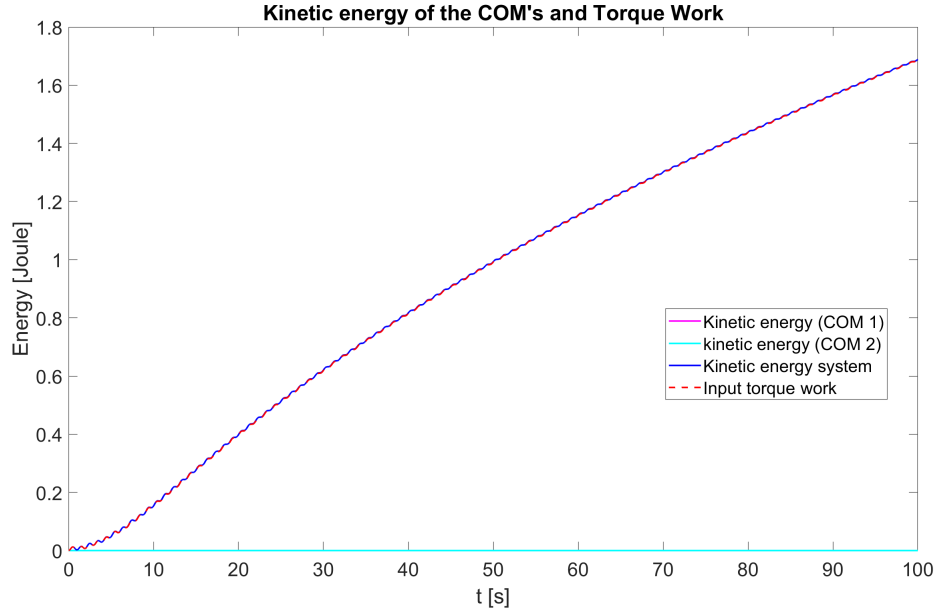


Figure 13: Kinetic energy of the system plotted together with the work supplied by the external torque.

6 Discussion

From the results above we see that the work done by the torque is equal to the kinetic energy. This is to be expected since due to the absence of other external forces there is no potential or dissipate component. All the work done on the segment is therefore transformed into kinetic energy of the system.

Appendix A

The main MATLAB script

```
1 %% MBD_B: Assignment 8 - EzyRoller
2 % Rick Staa (4511328)
3 % Last edit: 29/05/2018
4
5 %% NOTES
6 %% 1: Check if solution is alright
7 %     - EOM
8 %     - RK4
9 %     - Gaus method
10 %     - Create function which calculates the initial states
11 %% 2: Add torque
12 %     - Finish last part of assignment
13
14 %% - - Pre processing operations --
15 clear all; close all; clc;
16 fprintf('--- A8 ---\n');
17
18 % Set up needed symbolic parameters
19 syms x1 y1 phi1 x2 y2 phi2 x1d y1d phi1d x2d y2d phi2d t
20
21 % State
22 parms.syms.x1          = x1;
23 parms.syms.y1          = y1;
24 parms.syms.phi1        = phi1;
25 parms.syms.x2          = x2;
26 parms.syms.y2          = y2;
27 parms.syms.phi2        = phi2;
28 parms.syms.t           = t;
29
30 % State derivative
31 parms.syms.x1d         = x1d;
32 parms.syms.y1d         = y1d;
33 parms.syms.phi1d       = phi1d;
34 parms.syms.x2d         = x2d;
35 parms.syms.y2d         = y2d;
36 parms.syms.phi2d       = phi2d;
37
38 %% -- Set model/simulation parameters and initial states --
39 %% Intergration parameters
```

```

40 | sim_time                = 100;
                                     % Intergration time
41 | parms.h                = 1e-3;
                                     % Intergration step
    | size
42 | parms.tol              = 1e-12;
                                     % Intergration
    | constraint error tolerance
43 | parms.nmax             = 10;
                                     % Maximum number of
    | Gauss-Newton drift correction iterations
44 |
45 | %% Model Parameters
46 | % Lengths and distances
47 | parms.a                = 0.5;
                                     % Length wheel first
    | segment to COM segment 1
48 | parms.b                = 0.5;
                                     % Length COM to
    | revolute joint B
49 | parms.c                = 0.125;
                                     % Length revolute jonit
    | B to COM segment 2
50 | parms.d                = 0.125;
                                     % Length COM segment 2
    | to wheel 2
51 |
52 | % Masses and inertias
53 | parms.m1               = 1;
                                     % Body 1 weight [kg]
54 | parms.m2               = 0;
                                     % Body 2 weight [kg]
55 | parms.J1               = 0.1;
                                     % Moment of inertia
    | body 1 [kgm^2]
56 | parms.J2               = 0;
                                     % Moment of inertia
    | body 2 [kgm^2]
57 |
58 | % Create mass matrix (Segment 1 and 2)
59 | parms.M                = diag([parms.m1,parms.m1,parms.
    | J1,parms.m2,parms.m2,parms.J2]);
60 |

```

```

61 % Torque and force variables (See assignment)
62 parms.M0 = 0.1;
63 parms.omega = pi;
64
65 %% World parameters
66 % Gravity
67 parms.g = 9.81; % [parms.m/s^2]
68
69 % %% states for Question 1
70 % x1_0 = parms.a;
71 % y1_0 = 0;
72 % phi1_0 = 0;
73 % x2_0 = parms.a+parms.b;
74 % y2_0 = parms.d;
75 % phi2_0 = pi/2;
76 %
77 % % Phi1d
78 % x1d_0 = 1;
79 % y1d_0 = 0;
80 % phi1d_0 = 0;
81 % x2d_0 = 0;
82 % y2d_0 = 1;
83 % phi2d_0 = 0;
84 %
85 % % Set forces
86 % F = [0 0 0 0 0 0].';
% No torque applied
87 % parms.F = F;
88 % x0 = [x1_0 y1_0 phi1_0 x2_0 y2_0
    phi2_0 x1d_0 y1d_0 phi1d_0 x2d_0 y2d_0 phi2d_0];
89
90 %% States Question 2
91 % In this the generalised coordinates x1_init and y1_init are
    assumed to be
92 % defined so that wheel 1 is in the origin.
93 phi1_0 = 0; % Angle of first
    body with horizontal
94 phi2_0 = pi; % Angle of second
    body with horizontal
95

```



```

96 % Calculate other dependent initial positions and angles
97 x1_0 = parms.a*cos(phi1_0);
98 y1_0 = parms.b*sin(phi1_0);
99 x2_0 = (parms.a+parms.b)*cos(phi1_0)+
    parms.d*cos(phi2_0);
100 y2_0 = (parms.a+parms.b)*sin(phi1_0)+
    parms.d*sin(phi2_0);
101
102 % Velocity initital states (Make sure that the are admissable
    )
103 % Phi1d
104 x1d_0 = 0;
105 y1d_0 = 0;
106 phi1d_0 = 0;
107 x2d_0 = 0;
108 y2d_0 = 0;
109 phi2d_0 = 0;
110
111 % Create full state for optimization
112 x0 = [x1_0 y1_0 phi1_0 x2_0 y2_0
    phi2_0 x1d_0 y1d_0 phi1d_0 x2d_0 y2d_0 phi2d_0];
113
114 %% Set Forces and torques
115 % F=[F1_x,F1_y,M1,F2_x,F2_y,M2];
116 F = [0 0 -parms.M0*cos(parms.omega*
    t) 0 0 parms.M0*cos(parms.omega*t)].';
    % Torque applied
117
118 % Store F in function
119 parms.F = F;
120
121 %% -- Derive equation of motion --
122 %% Calculate EOM by means of Newton-Euler equations
123 [xdd_handle,C_handle,Cd_handle,D_handle,Dd_handle,F_handle] =
    EOM_calc(parms); % Calculate symbolic equations of
    motion and put in parms struct
124 parms.C_handle = C_handle;
125 parms.Cd_handle = Cd_handle;
126 parms.D_handle = D_handle;
127 parms.Dd_handle = Dd_handle;
128 parms.xdd_handle = xdd_handle;
129 parms.EOM_xdd = xdd_handle;
130

```

```

131 %% -- Perform simulation --
132 %% Calculate movement by mean sof a Runge-Kuta 4th order
    intergration method
133 tic
134 [t,x] = RK4_custom(parms.EOM_xdd,x0,
    sim_time,parms);
135 toc
136
137 %% -- Post Processing --
138 %% Calculate com velocities
139 % xd = diff(x)/parms.h;
140 xdd = state_deriv(x,parms);
141
142 %% Calculate position of point A B and C
143 [A,B,C] = point_calc(x,parms);
144
145 %% Calculate kinetic energy and torque wo[ekin] = ekin_calc(x
    ,parms);
146 [ekin] = ekin_calc(x,parms);
147 [tw] = tw_calc(x,parms);
148
149 % %% -- ANIMATE --
150 % % Adapted from A. Schwab's animation code
151 %
152 % % Rename data
153 % X1 = x(:,1); Y1 = x(:,2); P1 = x(:,3);
154 % DX1 = x(:,7); DY1 = x(:,8); DP1 = x(:,9);
155 % X2 = x(:,4); Y2 = x(:,5); P2 = x(:,6);
156 % DX2 = x(:,10); DY2 = x(:,11); DP2 = x(:,12);
157 %
158 % % Rename Points
159 % XA = A(:,1); YA = A(:,2);
160 % XB = B(:,1); YB = B(:,2);
161 % XC = C(:,1); YC = C(:,2);
162 %
163 % % Create figure
164 % figure
165 % plot(X1,Y1)
166 % hold on
167 % plot(XA,YA)
168 % hold on
169 % plot(X2,Y2)
170 % hold on

```

```

171 % plot(XC,YC)
172 % grid on
173 % set(gca,'fontsize',16)
174 % title('Animation EzyRoller')
175 % axis([min(X1)-parms.a max(X1)+parms.a min(Y1)-parms.a max(
    Y1)+parms.a]);
176 % axis equal
177 % l = plot([X1(1) XA(1)],[Y1(1) YA(1)]);
178 % k = plot([X2(1) XC(1)],[Y2(1) YC(1)]);
179 % j = plot([X1(1) XB(1)],[Y1(1) YB(1)]);
180 % m = plot([XB(1) X2(1)],[YB(1) Y2(1)]);
181 % set(l,'LineWidth',5);
182 % set(l,'Color','K')
183 % set(k,'LineWidth',5);
184 % set(k,'Color','C')
185 % set(j,'LineWidth',5);
186 % set(j,'Color','K')
187 % set(m,'LineWidth',5);
188 % set(m,'Color','C')
189 % nstep = length(t);
190 % nskip = 10;
191 % for istep = 2:nskip:nstep
192 %     set(l,'XData',[X1(istep) XA(istep)])
193 %     set(l,'YData',[Y1(istep) YA(istep)])
194 %     set(k,'XData',[X2(istep) XC(istep)])
195 %     set(k,'YData',[Y2(istep) YC(istep)])
196 %     set(j,'XData',[X1(istep) XB(istep)])
197 %     set(j,'YData',[Y1(istep) YB(istep)])
198 %     set(m,'XData',[XB(istep) X2(istep)])
199 %     set(m,'YData',[YB(istep) Y2(istep)])
200 %     drawnow
201 %     pause(1e-10)
202 % end
203
204 %% -- Create plots --
205 %% Plot path of points on the robot
206 figure;
207 plot(A(:,1),A(:,2),x(:,1),x(:,2),B(:,1),B(:,2),x(:,4),x(:,5),
    C(:,1),C(:,2),'linewidth',1.5);
208 set(gca,'fontsize',18);
209 title('Travelled path of points on the EzyRoller');
210 xlabel('x position [m]');
211 ylabel('y position [m]');

```

```

212 legend('Point A (Wheel 1)', 'COM 1', 'Point B (hidge)', 'COM 2'
        , 'Point C (wheel 2)', 'Location', 'Best');
213
214 %% Plot linear velocities COM's
215 figure;
216 subplot(2,1,1);
217 plot(t,x(:,7), 'b', t,x(:,10), 'r', 'Linewidth', 1.5);
218 title("x velocities of COM's");
219 xlabel('t [s]');
220 ylabel('velocity [m/s]');
221 legend('x velocity (COM 1)', 'x velocity (COM 2)', 'Location',
        'Best');
222 subplot(2,1,2);
223 plot(t,x(:,8), 'b', t,x(:,11), 'r', 'Linewidth', 1.5);
224 title("y velocities of COM's");
225 xlabel('t [s]');
226 ylabel('velocity [m/s]');
227 legend('y velocity (COM 1)', 'y velocity (COM 2)', 'Location',
        'Best');
228
229 %% Plot linear magnitude velocities COM's
230 % Calculate velocity magnitudes
231 v_com1 = sqrt(x(:,7).^2+x(:,8).^2);
232 v_com2 = sqrt(x(:,10).^2+x(:,11).^2);
233
234 % Plot figure
235 figure;
236 plot(t,v_com1, 'b', t,v_com2, 'r', 'Linewidth', 1.5);
237 title("velocity mag of COM's");
238 xlabel('t [s]');
239 ylabel('velocity mag [m/s]');
240 legend('velocity (COM 1)', 'velocity (COM 2)', 'Location', '
        Best');
241
242 %% Plot angular velocities
243 figure;
244 plot(t,x(:,9), 'b', t,x(:,12), 'r', 'Linewidth', 1.5);
245 title("angular velocities of COM's");
246 xlabel('t [s]');
247 ylabel('angular velocity [rad/s]');
248 legend('angular velocity (COM 1)', 'angular velocity (COM 2)',
        'Location', 'Best');
249

```

```

250 %% Plot linear and angular accelerations COM's
251 figure;
252 subplot(2,1,1);
253 plot(t,xdd(:,7),'b',t,xdd(:,10),'r','Linewidth',1.5);
254 title("x accelerations of COM's");
255 xlabel('t [s]');
256 ylabel('acceleration [m/s^2]');
257 legend('x acceleration (COM 1)','x celleration (COM 2)', '
    Location', 'Best');
258 subplot(2,1,2);
259 plot(t,xdd(:,8),'b',t,xdd(:,11),'r','Linewidth',1.5);
260 title("y accelerations of COM's");
261 xlabel('t [s]');
262 ylabel('Accelleration [m/s^2]');
263 legend('y acceleration (COM 1)','y acceleration (COM 2)', '
    Location', 'Best');
264
265 %% Plot angular accelerations
266 figure;
267 plot(t,xdd(:,9),'b',t,xdd(:,12),'r','Linewidth',1.5);
268 title("Angular velocities of COM's");
269 xlabel('t [s]');
270 ylabel('Angular acceleration [rad/s^2]');
271 legend('Angular acceleration (COM 1)','Angular acceleration (
    COM 2)', 'Location', 'Best');
272
273 %% Plot reaction forces
274 figure;
275 plot(t,x(:,13:end),'Linewidth',1.5);
276 title("Reaction forces in the constraints");
277 xlabel('t [s]');
278 ylabel('Reaction Force [N]');
279 legend('X reaction force in joint B (FB_x)','Y reaction force
    in joint B (FB_y)','Wheel A friction force (no slip)', '
    Wheel C friction force (no slip)', 'Location', 'Best');
280
281 %% Plot kinetic energy
282 figure;
283 plot(t,ekin(:,1),'-b',t,ekin(:,2),'-r',t,ekin(:,3),'-g', '
    Linewidth',1.5)
284 set(gca,'fontsize',18);
285 title("Kinetic energy of the COM's");
286 xlabel('t [s]');

```

```

287 ylabel('Kinetic energy[Joule]');
288 legend('Kinetic energy (COM 1)','kinetic energy (COM 2)','
        Kinetic energy system','Location', 'Best');
289
290 %% Plot Kinetic energy plus torque energy
291 figure;
292 plot(t,ekin(:,1),'-m',t,ekin(:,2),'-c',t,ekin(:,3),'-b',t,tw,
        '--r','Linewidth',1.5)
293 set(gca,'fontsize',18);
294 title("Kinetic energy of the COM's and Torque Work");
295 xlabel('t [s]');
296 ylabel('Energy [Joule]');
297 legend('Kinetic energy (COM 1)','kinetic energy (COM 2)','
        Kinetic energy system','Input torque work','Location', '
        Best');
298
299 %% FUNCTIONS
300
301 %% Post processing functions
302 % These functions are used to calculate quantities that are
    not calculated
303 % during the simulation. This regards quantities which are
    not state
304 % variables
305
306 % Calculate second derivative
307 function [xdd] = state_deriv(x,parms)
308
309 % preallocate memory for xdd vector
310 xdd      = zeros(size(x,1),12);
311
312 % Create time vector
313 time = 0:parms.h:((parms.h*size(x,1))-parms.h);
314
315 % Loop through states
316 for ii = 1:size(x,1)
317     % Set time
318     t = time(ii);
319
320     % Calculate xdd
321     x_now_tmp    = num2cell(x(ii,1:end-4),1);
322     x_now_full   = num2cell([x(ii,1:end-4),t],1);

```

```

323     xdd_tmp    = feval(parms.EOM_xdd,x_now_full{[3 6 7:13]})
324     .';
325     xdd(ii,:) = [cell2mat(x_now_tmp(7:12)) xdd_tmp(1:6)];
326 end
327
328 % Calculation points on EzyRoller
329 function [A,B,C] = point_calc(x,parms)
330
331 %% Calculate Point A, B, C out of the state
332 A_x      = x(:,1)-parms.a*cos(x(:,3));
333 A_y      = x(:,2)-parms.a*sin(x(:,3));
334 B_x      = x(:,1)+parms.b*cos(x(:,3));
335 B_y      = x(:,2)+parms.b*sin(x(:,3));
336 C_x      = x(:,4)+parms.c*cos(x(:,6));
337 C_y      = x(:,5)+parms.c*sin(x(:,6));
338
339 % Put them in their corresponding vector
340 A = [A_x A_y];
341 B = [B_x B_y];
342 C = [C_x C_y];
343
344 end
345
346 % Calculate kinetic energy of COM's
347 function [ekin] = ekin_calc(x,parms)
348
349 % preallocate memory for ekin vector
350 ekin      = zeros(size(x,1),1);
351
352 % Loop through states
353 for ii = 1:size(x,1)
354     ekin(ii,1) = 0.5*x(ii,7:9)*parms.M(1:3,1:3)*x(ii,7:9).';
355     ekin(ii,2) = 0.5*x(ii,10:12)*parms.M(4:6,4:6)*x(ii,10:12)
356     .';
357     ekin(ii,3) = 0.5*x(ii,7:12)*parms.M*x(ii,7:12).';
358 end
359
360 % Calculate kinetic energy of COM's
361 function [tw] = tw_calc(x,parms)
362
363 % Calculate the applied torque for the whole movement

```

```

364 % preallocate memory for xdd vector
365 tw          = zeros(size(x,1),1);
366
367 % Create time vector
368 time        = 0:parms.h:((parms.h*size(x,1))-parms.h);
369
370 % Create W vector
371 for ii = (2:size(x,1))
372     tw(ii)    = tw(ii-1) + sum((subs_F(time(ii))).'*(x(ii
        ,1:6)-x((ii-1),1:6)));
373 end
374 end
375
376 %% Runge-Kuta numerical intergration function
377 % This function calculates the motion of the system by means
    of a
378 % Runge-Kuta numerical intergration. This function takes as
    inputs the
379 % parameters of the system (parms), the EOM of the system (
    parms.EOM)
380 % and the initial state.
381 function [time,x] = RK4_custom(EOM,x0,sim_time,parms)
382
383 % Initialise variables
384 time        = (0:parms.h:sim_time).';
                                     % Create time array
385 x            = zeros(length(time),16);
                                     % Create empty state array
386 x(1,1:length(x0)) = x0;
                                     % Put
        initial state in array
387
388 % Caculate the motion for the full simulation time by means
    of a
389 % Runge-Kutta4 method
390
391 % Perform intergration till end of set time
392 for ii = 1:(size(time,1)-1)
393
394     % Add time constant
395     t = time(ii);
396
397     % Perform RK 4

```



```

398     x_now_tmp          = num2cell(x(ii,1:end-4),1);

        % Create cell for feval function
399     x_full_tmp         = num2cell([x(ii,1:end-4),t],1);

        % Add time to state
400     K1                 = [cell2mat(x_now_tmp(1,end-5:end)),
        feval(EOM,x_full_tmp{[3 6 7:13]})].'];
        % Calculate the second derivative
        at the start of the step
401     x1_tmp             = num2cell(cell2mat(x_now_tmp) + (parms
        .h*0.5)*K1(1:end-4));
        % Create cell for feval function
402     x1_full            = num2cell([cell2mat(x1_tmp),t],1);

        % Add time to state
403     K2                 = [cell2mat(x1_tmp(1,end-5:end)),feval(
        EOM,x1_full{[3 6 7:13]})].'];
        % Calculate the second derivative halfway the step
404     x2_tmp             = num2cell(cell2mat(x_now_tmp) + (parms
        .h*0.5)*K2(1:end-4));
        % Refine value calculation with new found derivative
405     x2_full            = num2cell([cell2mat(x2_tmp),t],1);

        % Add time to state
406     K3                 = [cell2mat(x2_tmp(1,end-5:end)),feval(
        EOM,x2_full{[3 6 7:13]})].'];
        % Calculate new derivative at the new refined location
407     x3_tmp             = num2cell(cell2mat(x_now_tmp) + (parms
        .h)*K3(1:end-4));
        % Calculate state at end step with refined derivative
408     x3_full            = num2cell([cell2mat(x3_tmp),t],1);

        % Add time to state
409     K4                 = [cell2mat(x3_tmp(1,end-5:end)),feval(
        EOM,x3_full{[3 6 7:13]})].'];
        % Calculate last second derivative
410     x(ii,end-3:end)    = (1/6)*(K1(end-3:end)+2*K2(end-3:end)
        +2*K3(end-3:end)+K4(end-3:end));
        % Take weighted sum of K1, K2,
        K3
411     x(ii+1,1:end-4)    = cell2mat(x_now_tmp) + (parms.h/6)*(K1
        (1:end-4)+2*K2(1:end-4)+2*K3(1:end-4)+K4(1:end-4));

```

```

412         % Perform euler intergration step
413     % Calculate last acceleration
414     if ii == (size(time,1)-1)
415         x_now_tmp          = num2cell(x(ii+1,1:end-4),1);

         % Create cell for feval function
416     x_full_tmp            = num2cell([x(ii+1,1:end-4),t],1);

         % Add time to state
417     K1                    = [cell2mat(x_now_tmp(1,end-5:end))
        ,feval(EOM,x_full_tmp{[3 6 7:13]})].'];
        % Calculate the second
        derivative at the start of the step
418     x1_tmp                = num2cell(cell2mat(x_now_tmp) + (
        parms.h*0.5)*K1(1:end-4));

        % Create cell for
        feval function
419     x1_full               = num2cell([cell2mat(x1_tmp),t],1);

        % Add time to state
420     K2                    = [cell2mat(x1_tmp(1,end-5:end)),
        feval(EOM,x1_full{[3 6 7:13]})].'];
        % Calculate the second
        derivative halfway the step
421     x2_tmp                = num2cell(cell2mat(x_now_tmp) + (
        parms.h*0.5)*K2(1:end-4));

        % Refine value
        calculation with new found derivative
422     x2_full               = num2cell([cell2mat(x2_tmp),t],1);

        % Add time to state
423     K3                    = [cell2mat(x2_tmp(1,end-5:end)),
        feval(EOM,x2_full{[3 6 7:13]})].'];
        % Calculate new
        derivative at the new refined location
424     x3_tmp                = num2cell(cell2mat(x_now_tmp) + (
        parms.h)*K3(1:end-4));

        % Calculate
        state at end step with refined derivative

```

```

425         x3_full = num2cell([cell2mat(x3_tmp),t],1);

        % Add time to state
426         K4 = [cell2mat(x3_tmp(1,end-5:end)),
                feval(EOM,x3_full{[3 6 7:13]})].';
                % Calculate last second
                derivative
427         x(ii+1,end-3:end) = (1/6)*(K1(end-3:end)+2*K2(end
                -3:end)+2*K3(end-3:end)+K4(end-3:end));
                % Take weighted sum of K1,
                K2, K3
428     end
429
430     % Correct for intergration drift
431     x_now_tmp = x(ii+1,:);
432     [x_new,~] = gauss_newton(x_now_tmp,parms);
433
434     % Update the constraint forces
435     x_new_full = num2cell([x(ii,1:end-4),t],1);
436     x_update = feval(EOM,x_new_full{[3 6:13]})'.';
437
438     % Overwrite position coordinates
439     x(ii+1,:) = [x_new(1:end-4) x_update(end-3:end)];
440
441 end
442 end
443
444 %% Constraint calculation function
445 function [C,Cd,D,Dd] = constraint_calc(x,parms)
446
447 % Get needed angles out
448 x_now_tmp = num2cell(x,1);
449
450 %% Calculate position constraint
451 C = feval(parms.C_handle,x_now_tmp{1:6})'.';
452
453 % Calculate constraint derivative
454 Cd = feval(parms.Cd_handle,x_now_tmp{[3 6]})'.';
455
456 %% Calculate velocity constraint
457 D = feval(parms.D_handle,x_now_tmp{[3 6:12]})
    .';

```

```

458
459 % Calculate velocity constraint derivative
460 Dd          = feval(parms.Dd_handle,x_now_tmp{[3 6]}).';
461 end
462
463 %% Speed correct function
464 function [x,error] = gauss_newton(x,parms)
465
466 % Get rid of the drift by solving a non-linear least square
    problem by
467 % means of the Gaus-Newton method
468 % Calculate the two needed constraints
469 [C,Cd,~,~] = constraint_calc(x,parms);
470
471 %% Guass-Newton position constraint correction
472 n_iter      = 0;
473
474     % Set iteration counter
475
476     % Get position data out
477
478 % Solve non-linear constraint least-square problem
479 while (max(abs(C)) > parms.tol)&& (n_iter < parms.nmax)
480     x_tmp          = x(1:6);
481     n_iter = n_iter + 1;
482     x_del  = Cd*inv(Cd.*Cd)*-C.';
483     x(1:6) = x_tmp+ x_del.';
484
485     % Recalculate constraint
486     [C,Cd,~,~] = constraint_calc(x,parms);
487 end
488
489 % % Calculate the corresponding speeds
490 % x_tmp_vel          = x(7:12);
491 % Dxd_n1             = -Cd*inv(Cd.*Cd)*Cd.*x_tmp_vel.';
492 % x(7:12)            = x_tmp_vel + Dxd_n1.';
493 %
494
495 %% Gaus-newton velocity constraint correction
496 n_iter      = 0;
497
498     % Set iteration counter

```

```

493     % Get position data out
494 % % Calculate the two needed constraints
495 % [~,~,D,Dd] = constraint_calc(x,parms);
496
497 % % Solve non-linear constraint least-square problem
498 % while (max(abs(D)) > parms.tol)&& (n_iter < parms.nmax)
499 %     x_tmp          = x(7:12);
500 %     n_iter = n_iter + 1;
501 %     x_del  = Dd*inv(Dd.'*Dd)*-D.';
502 %     x(7:12) = x_tmp+ x_del.';
503 %
504 %     % Recalculate constraint
505 %     [~,~,D,Dd]      = constraint_calc(x,parms);
506 % end
507
508
509 % Calculate constraints
510 [~,Cd,D,Dd]          = constraint_calc(x,parms);
511 Sd                    = [Cd Dd];
512
513 % Calculate new velocities
514 x_tmp_vel            = x(7:12);
515 Dxd_n1               = -Sd*inv(Sd.'*Sd)*Sd.'*x_tmp_vel.';
516 x(7:12)              = x_tmp_vel + Dxd_n1.';
517
518 %% Recalculate error
519 [C,~,D,~]            = constraint_calc(x,parms);
520 C_error = C;
521 D_error = D;
522
523 % Store full error
524 error = [C_error D_error];
525 end
526
527 %% Calculate (symbolic) Equations of Motion four our setup
528 function [xdd_handle,C_handle,Cd_handle,D_handle,Dd_handle,
529         F_handle] = EOM_calc(parms)
529
530 %% -- The code between this lines is done to obtain the latex
531      formulas --
531 % % Create model parameters in symbolic form
532 % syms a b c d m1 m2 J1 J2 g;

```

```

533
534 % Overwrite with real values if you don't want the full
      symbolic expression
535 a      = parms.a;
536 b      = parms.b;
537 c      = parms.c;
538 d      = parms.d;
539 m1     = parms.m1;
540 m2     = parms.m2;
541 J1     = parms.J1;
542 J2     = parms.J2;
543 g      = parms.g;
544
545 %% -- The code between this lines is done to create the latex
      formulas --
546
547 % Unpack symbolic variables from parms
548 x1      = parms.syms.x1;
549 y1      = parms.syms.y1;
550 phi1    = parms.syms.phi1;
551 x2      = parms.syms.x2;
552 y2      = parms.syms.y2;
553 phi2    = parms.syms.phi2;
554 t       = parms.syms.t;
555
556 % Generalised state derivative
557 x1d     = parms.syms.x1d;
558 y1d     = parms.syms.y1d;
559 phi1d   = parms.syms.phi1d;
560 x2d     = parms.syms.x2d;
561 y2d     = parms.syms.y2d;
562 phi2d   = parms.syms.phi2d;
563
564 % Create generalized coordinate vectors
565 x       = [x1;y1;phi1;x2;y2;phi2];
566 xd      = [x1d;y1d;phi1d;x2d;y2d;phi2d];
567
568 % Calculate Position constraints
569 C       = [x1+b*cos(phi1)-x2+d*cos(phi2); ...
570           y1+b*sin(phi1)-y2+d*sin(phi2)];
571
572 % Calculate Velocity constraints
573 v1      = [x1d y1d 0;x2d y2d 0].';

```

```

574 omega          = [0 0 phi1d;0 0 phi2d].';
575 R_A_COM         = [-a*cos(phi1) -a*sin(phi1) 0; c*cos(phi2) c
    *sin(phi2) 0].';
576 Va             = v1 + cross(omega,R_A_COM);
577 eA              = [-sin(phi1) cos(phi1) 0; -sin(phi2) cos(
    phi2) 0].';
578 D_x            = simplify([Va(:,1).'*eA(:,1);Va(:,2).'*eA
    (:,2)]);
579
580 % Split constraint in matrix vector product
581 D              = equationsToMatrix(D_x,[x1d y1d phi1d x2d
    y2d phi2d]);
582
583 % Compute the jacobian of the (non-)holonomic constraints
584 JC_x            = simplify(jacobian(C,x.'));
585 JD_x            = simplify(jacobian(D_x,xd.'));
586
587 % Calculate convective component
588 JC_xd           = jacobian(JC_x*xd,x);
589 JD_xd           = jacobian(D*xd,x);
590
591 % Create system of DAE
592 A = [parms.M JC_x.' D.'
    ; ...
    JC_x zeros(size(JC_x,1),size(JC_x.',2)) zeros(size(D,1),
    size(D.',2)); ...
    D zeros(size(D,1),size(JC_x.',2)) zeros(size(D,1),size(D
    .',2))];
593 B = [parms.F ; -JC_xd*xd; -JD_xd*xd];
594
595 % Calculate result expressed in generalized coordinates
596 xdd             = A\B;
597
598 %% Convert to function handles
599 % xdp_handle     = matlabFunction(xdp);
600 % Create function
    handle of EOM in terms of COM positions
601 xdd_handle      = matlabFunction(simplify(xdd),'vars',[
    phi1 phi2 x1d y1d phi1d x2d y2d phi2d t]);
    % Create function handle of EOM in
    terms of generalised coordinates
602 % matlabFunction(qdp,'file',qdp_cal')
603
604

```

```

605 % Position constraint function handle
606 C_handle      = matlabFunction(simplify(C),'vars',[x1 y1
        phi1 x2 y2 phi2]);
607
608 % Position constraint derivative function handle
609 Cd            = JC_x;
610 Cd_handle     = matlabFunction(simplify(Cd));
611
612 % Velocity constraint function handle
613 D_handle      = matlabFunction(simplify(D_x),'vars',[phi1
        phi2 x1d y1d phi1d x2d y2d phi2d]);
614
615 % Velocity constraint derivative function handle
616 Dd            = simplify(JD_x);
617 Dd_handle     = matlabFunction(Dd);
618
619 % Force torque velocity handle
620 F_handle = matlabFunction(parms.F,'File','subs_F');
621
622 end

```


References

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