Multibody Dynamics B - Assignment 7 **Rick Staa**

ME41055 #4511328

Prof. Arend L. Schwab Lab Date: 26/04/2018

Head TA: Simon vd. Helm 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 Acknoledgements

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

3 Setup overview

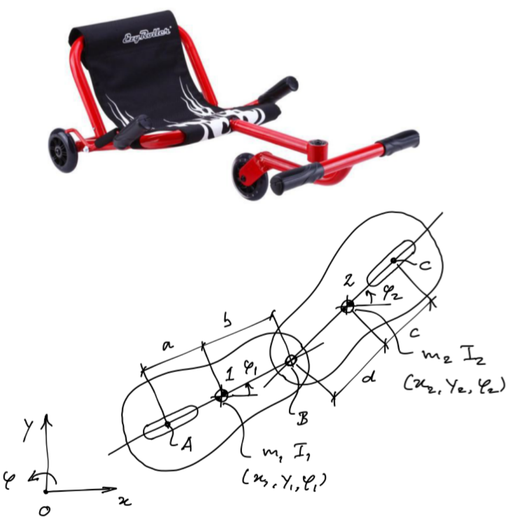

Figure 2: Quick return mechanism as depicted in assignment 7

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:

$$
\begin{align}
a &= 0.5m \tag{1}\\
b &= 0.5m \tag{2}\\
c &= 0.125m \tag{3}\\
d &= 0.125m \tag{4}\\
m1 &= 1kg \tag{5}\\
m2 &= 0kg \tag{6}\\
J1 &= 0.1kgm^2 \tag{7}\\
J2 &= 0kgm^2 \tag{8}\\
g &= 9.81m/s^2 \tag{9}\\
& \tag{10}
\end{align}
$$

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.

$$x0 = \begin{bmatrix} x_1 & y_1 & phi_1 & x_2 & y_2 & phi_2 \end{bmatrix} \tag{11}$$

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:

$$x0 = \begin{bmatrix} x_1 & y_1 & phi_1 & x_2 & y_2 & phi_2 & \dot{x_1} & \dot{y_1} & \dot{phi_1} & \dot{x_2} & \dot{y_2} & \dot{phi_2} \end{bmatrix} \tag{12}$$

$$x0 = \begin{bmatrix} a & 0 & 0 & a+b & d & \pi/2 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{13}$$

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. The full derivation will be explained below.

4.1 Equations of motion(EOM)

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

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

Figure 3: Vector product of the EOM

In this $M_{i,j}$ depicts the mass matrix, $C_{k,j}$ the Jacobean of the holonomic constraints (position constraint) and $S_{k,j}$ 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:

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

and B matrix is equal to:

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

$$F = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{16}$$

4.1.1 holonomic constraints

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

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

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:

$$V_A = V_{COM1} + \omega \times r_{A/COM1} \tag{18}$$

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 axile to 0 we get the following velocity constraints:

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

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 integration method

To get the movement of EzyRoller in time we will use a 4^{th} order Runge-Kuta integration method combined with a Gauß-Newton correction for position and speed. This correction is done to compensate for integration 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:

$$k_1 = f(t_n, y_n) \tag{20}$$

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

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

$$k_4 = f(t_n + h, y_n + hk_3) \tag{23}$$

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

Gauß-Newton corrections

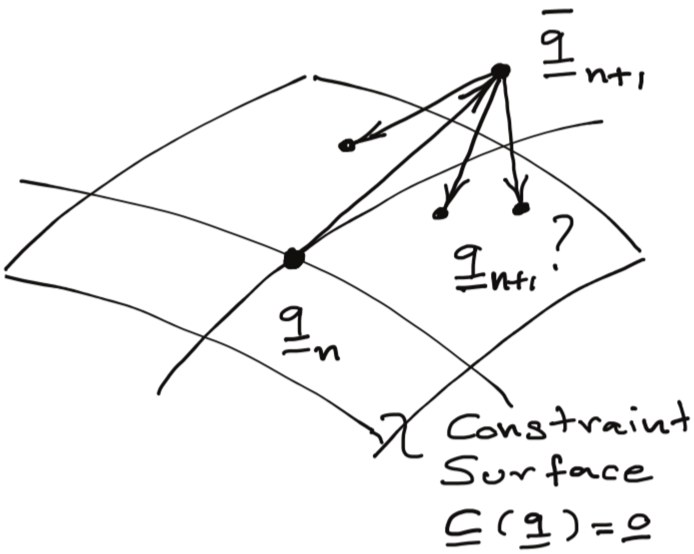

Figure 4: A depiction of the constraint surface and Gauß-Newton method as displayed in [1]. This picture was not modified in any sense

The Gauß-Newton we are using here is a non-linear leas-square constraint optimization method. In our problem we the following optimization problem:

$$\|\bar{q}_{n+1} - q_{n+1}\|_2 = \min_{q_{n+1}}, \quad \forall \quad \{q_{n+1} | C(q_{n+1}) = 0\} \tag{25}$$

In words what your doing with the Gausß-Newton method is you look at the solution and see how much it deviates from the constraint surface. You then look for the point on the constraint surface that is closest to our original point. This point searching is what is done by the optimization (see 4).Above 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 q:

$$q_{n+1} = \bar{q}_{n+1} + \Delta q_{n+1} \tag{26}$$

When you fill this in in the original

$$\Delta q_{n+1} = 0, \quad \forall \quad \{\Delta q_{n+1} | C(\bar{q}_{n+1}) \Delta q_{n+1} = 0\} \tag{27}$$

This leads to the following system of equations:

$$\begin{pmatrix} I & C^T \\ C0 & \end{pmatrix} \begin{pmatrix} \Delta \\ \mu \end{pmatrix} = \begin{pmatrix} 0 \\ e \end{pmatrix} \tag{28}$$

In which:

$$-CC^T \mu = e \tag{29}$$

$$\mu = -(CC^T)^{-1} \tag{30}$$

$$\Delta = C^T (CC^T)^{-1} e \tag{31}$$

In the end you obtain:

$$\Delta = C{+}e \tag{32}$$

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

$$q_{new} = q_{old} + \Delta \tag{33}$$

Following you can the recalculate the C and $\dot{C}$ 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:

```
% Solve non-linear constraint least-square problem
while (max(abs(C)) > parms.tol)&& (n_iter < parms.nmax)
    x_tmp            = x(1:6);
    n_iter = n_iter + 1;
    x_del  = Cd*inv(Cd.'*Cd)*-C.';
    x(1:6) = x_tmp+ x_del.';

    % Recalculate constraint
    [C,Cd,~,~]       = constraint_calc(x,parms);
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:

```
% Calculate the corresponding speeds
q_tmp_vel            = q(4:6);
Dqd_n1               = -Cd*inv(Cd.'*Cd)*Cd.'*q_tmp_vel.';
q(4:6)               = q_tmp_vel + Dqd_n1.';
```

Figure 6: Matlab code doing the velocity correction

In these MATLAB scripts C depicts the position constraints, Cd the Jacobean of these constraints, D the velocity constraints and Dd the Jacobean of these velocity constraints. Sd is simply the matrix of both the holonomic and non-holonomic velocityconstraints 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.

$$x0 = \begin{bmatrix} a & 0 & 0 & a+b & d & \pi/2 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{34}$$
$$x0 = \begin{bmatrix} a & 0 & 0 & a+b & d & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{35}$$
$$x0 = \begin{bmatrix} a & 0 & pi/2 & a+b & d & \pi/2 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{36}$$

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 34) is shown in figure 7. From the figure we that as we put a input x-velocity on the COM of the first body while the second body is under a 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 othe r 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.

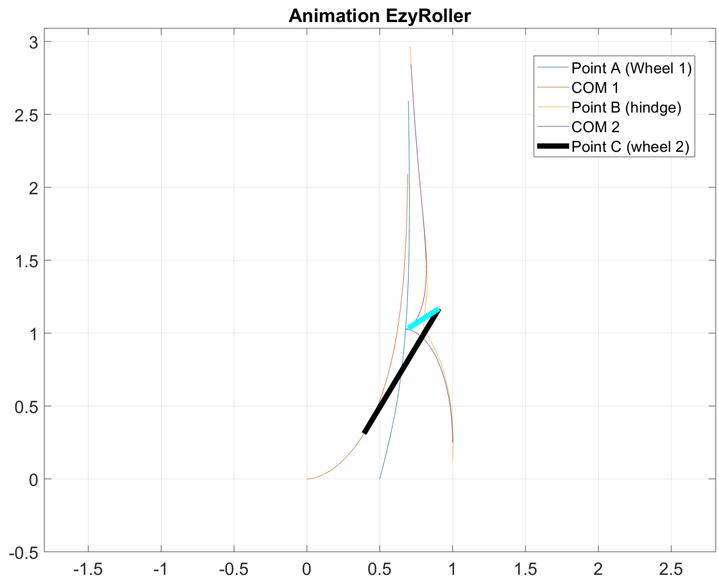

Figure 7: Path of the points on the EzyRoller

5.2 Powered mechanism

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

$$F = \begin{bmatrix} 0 & 0 & T1 & 0 & 0 & T2 \end{bmatrix} \tag{37}$$

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

$$B = \begin{pmatrix} 0 \\ 0 \\ -\frac{\cos(\pi\, t)}{10} \\ 0 \\ 0 \\ \frac{\cos(\pi\, t)}{10} \\ b\, \cos\left(\varphi_1\right)\, \dot{\mathrm{phi}}_1{}^2 + d\, \cos\left(\varphi_2\right)\, \dot{\mathrm{phi}}_2{}^2 \\ b\, \sin\left(\varphi_1\right)\, \dot{\mathrm{phi}}_1{}^2 + d\, \sin\left(\varphi_2\right)\, \dot{\mathrm{phi}}_2{}^2 \\ \dot{\mathrm{phi}}_1\, \left(\dot{\mathrm{x}}_1\, \cos\left(\varphi_1\right) + \dot{\mathrm{y}}_1\, \sin\left(\varphi_1\right)\right) \\ \dot{\mathrm{phi}}_2\, \left(\dot{\mathrm{x}}_2\, \cos\left(\varphi_2\right) + \dot{\mathrm{y}}_2\, \sin\left(\varphi_2\right)\right) \end{pmatrix} \tag{38}$$

Further we are instructed to use the following initial state.

$$x0 = \left[\begin{array}{cccccccccccc} a & 0 & 0 & a+b & d & \pi & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \tag{39}$$

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.

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 **??**.

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.

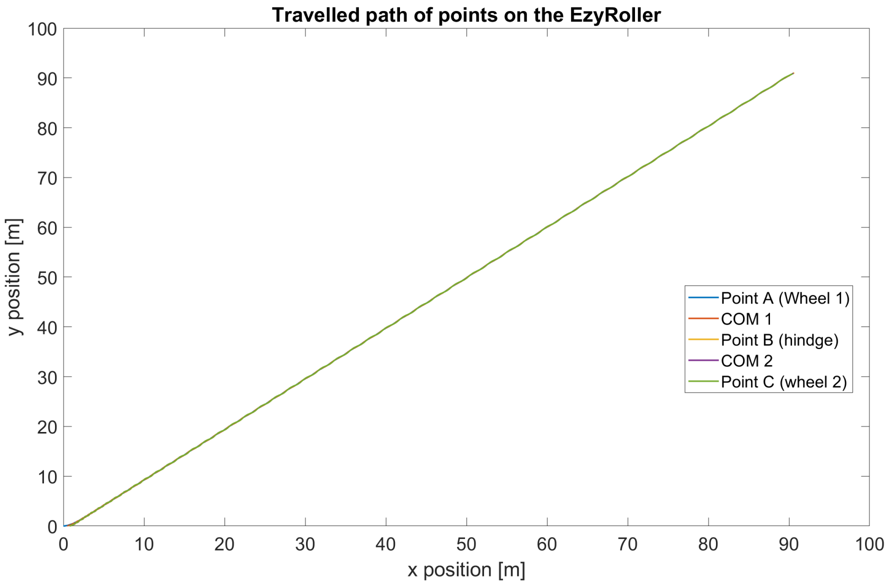

Figure 8: Path of the points on the EzzRoller

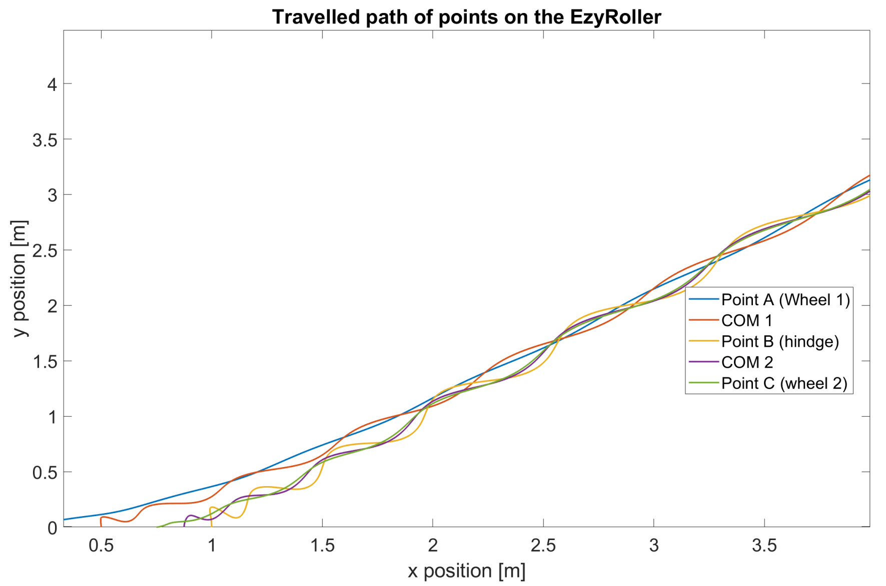

Figure 9: Path of the points on the EzzRoller Zoomed in

5.2.3 kinetic energy and Torque work

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

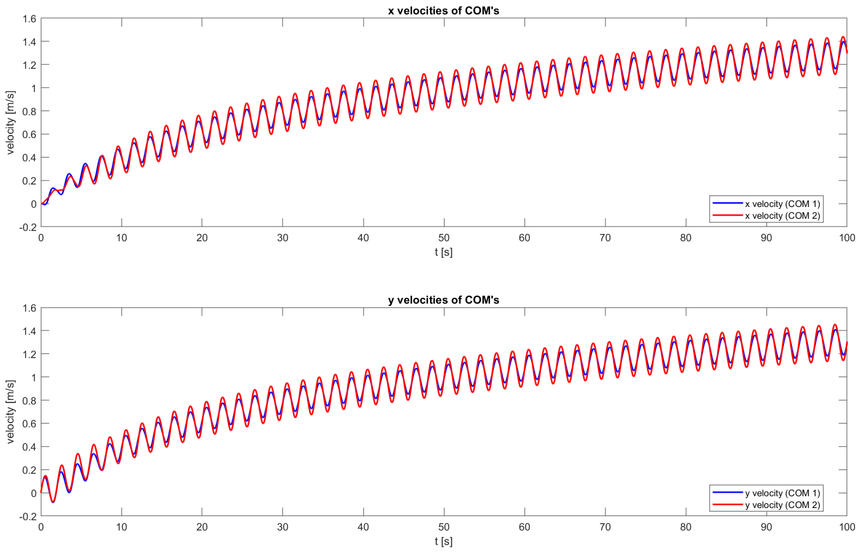

Figure 10: Linear velocities of EzzRoller

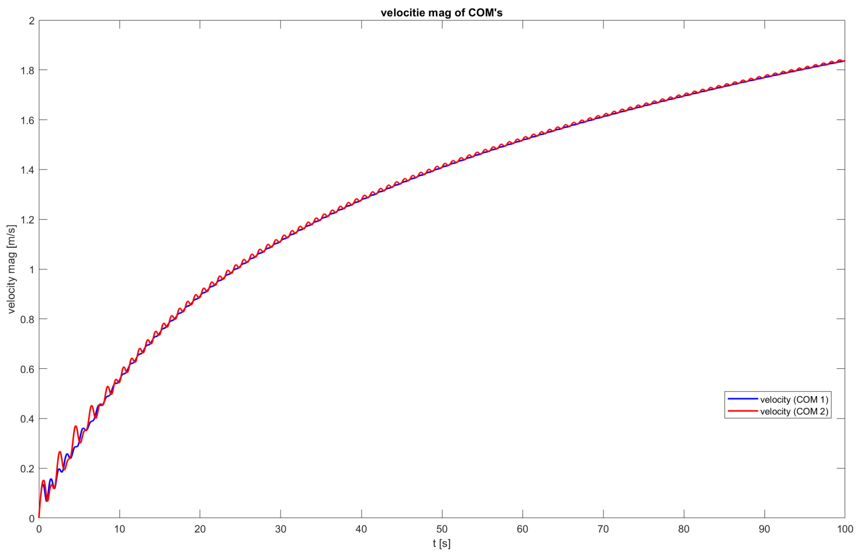

Figure 11: Magnitudes of Linear velocities of EzzRoller

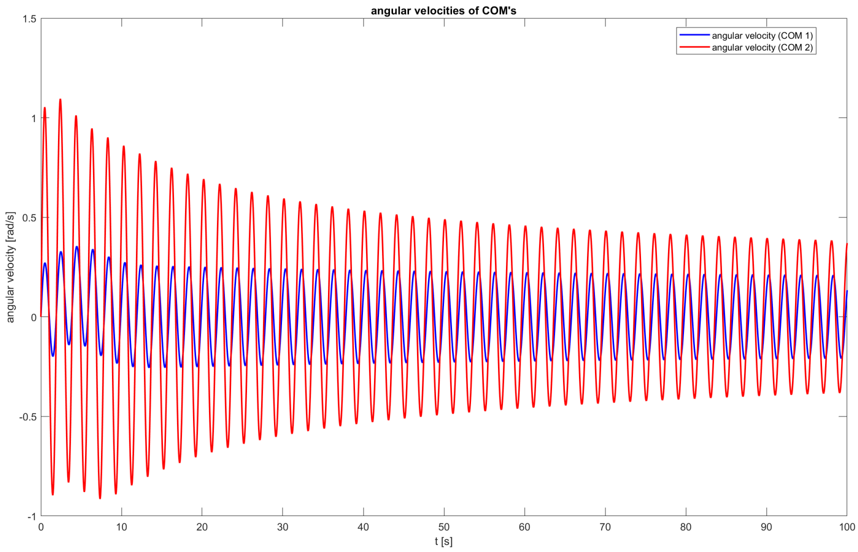

Figure 12: Angular velocities of EzzRoller

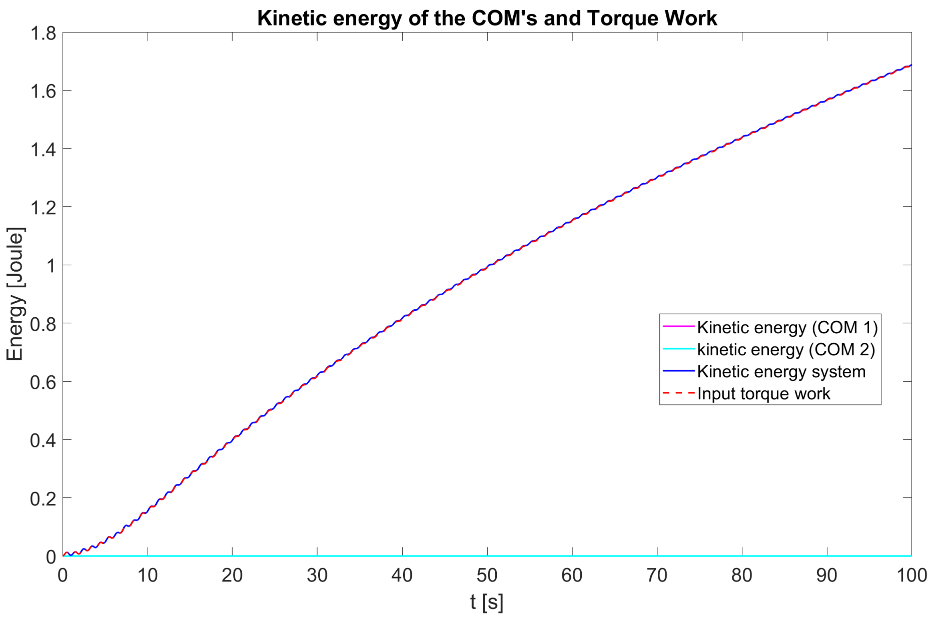

Figure 13: Kinetic energy of the system plotted together with the work supplied by the external torque.

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. Further it might seem strange that body 2 has no kinetic energy but this is caused by the fact that the mass and inertia of body 2 were said to be zero.

Appendix A: Accompanying MAT LAB scripts

```
%% MBD_B: Assignment 8 - EzyRoller
%  Rick Staa (4511328)
%  Last edit: 29/05/2018

%% - - Pre processing operations --
clear all; close all; clc;
fprintf('--- A8 ---\n');
animate_bool = 0;                               % Set on 1 if you want to see an animation

%% Script settings
parms.accuracy_bool          = 0;               % If set to 1 A\b will be performed instead of inv(A)*B this is more accurate but slower
parms.question_bool          = 1;               % Set on 0 for the first part of the question and 1 for the second part

%% Model parameters
% Set up needed symbolic parameters
syms x1 y1 phi1 x2 y2 phi2 x1d y1d phi1d x2d y2d phi2d t

% State
parms.syms.x1                = x1;
parms.syms.y1                = y1;
parms.syms.phi1              = phi1;
parms.syms.x2                = x2;
parms.syms.y2                = y2;
parms.syms.phi2              = phi2;
parms.syms.t                 = t;

% State derivative
parms.syms.x1d                = x1d;
parms.syms.y1d                = y1d;
parms.syms.phi1d              = phi1d;
parms.syms.x2d                = x2d;
parms.syms.y2d                = y2d;
parms.syms.phi2d              = phi2d;

%% -- Set model/simulation parameters and initial states --
%% Intergration parameters
```

```
sim_time                    = 100;                                        % Intergration time
parms.h                     = 1e-3;                                       % Intergration step size
parms.tol                   = 1e-12;                                      % Intergration constraint error tolerance
parms.nmax                  = 10;                                         % Maximum number of Gauss-Newton drift correction iterations

%% Model Parameters
% Lengths and distances
parms.a                     = 0.5;                                        % Length wheel first segment to COM segment 1
parms.b                     = 0.5;                                        % Length COM to revolute joint B
parms.c                     = 0.125;                                      % Length revolute jonit B to COM segment 2
parms.d                     = 0.125;                                      % Length COM segment 2 to wheel 2

% Masses and inertias
parms.m1                    = 1;                                          % Body 1 weight [kg]
parms.m2                    = 0;                                          % Body 2 weight [kg]
parms.J1                    = 0.1;                                        % Moment of inertia body 1 [kgm^2]
parms.J2                    = 0;                                          % Moment of inertia body 2 [kgm^2]

% Create mass matrix (Segment 1 and 2)
parms.M                     = diag([parms.m1,parms.m1,parms.J1,parms.m2,parms.m2,parms.J2]);

```

```
% Torque and force variables (See assignment)
parms.M0                    = 0.1;
parms.omega                 = pi;

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

if parms.question_bool == 0
    %% states for Question 1
    x1_0                        = parms.a;
    y1_0                        = 0;
    phi1_0                      = 0;
    x2_0                        = parms.a+parms.b;
    y2_0                        = parms.d;
    phi2_0                      = pi/2;

    % Phi1d
    x1d_0                       = 1;
    y1d_0                       = 0;
    phi1d_0                     = 0;
    x2d_0                       = 0;
    y2d_0                       = 1;
    phi2d_0                     = 0;

    % Set forces
    F                           = [0 0 0 0 0 0].';
                                % No torque applied
    parms.F                     = F;
    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];
else
    %% States Question 2
    % In this the generalised coordinates x1_init and y1_init
        are assumed to be
    % defined so that wheel 1 is in the origin.
    phi1_0                      = 0;
                                                    % Angle of first
        body with horizontal
    phi2_0                      = pi;
                                                    % Angle of second
        body with horizontal
```

```
    % Calculate other dependent initial positions and angles
    x1_0                        = parms.a*cos(phi1_0);
    y1_0                        = parms.b*sin(phi1_0);
    x2_0                        = (parms.a+parms.b)*cos(phi1_0)+parms.d*cos(phi2_0);
    y2_0                        = (parms.a+parms.b)*sin(phi1_0)+parms.d*sin(phi2_0);

    % Velocity inititial states (Make sure that the are admissable)
    % Phi1d
    x1d_0                       = 0;
    y1d_0                       = 0;
    phi1d_0                     = 0;
    x2d_0                       = 0;
    y2d_0                       = 0;
    phi2d_0                     = 0;

    % Create full state for optimization
    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];

    % Set Forces and torques
    % F=[F1_x,F1_y,M1,F2_x,F2_y,M2];
    F                           = [0 0 -parms.M0*cos(parms.omega*t) 0 0 parms.M0*cos(parms.omega*t)].';  % Torque applied

    % Store F in function
    parms.F                     = F;
end

%% -- Derive equation of motion --
%% Calculate EOM by means of Newton-Euler equations
EOM_calc(parms);                                                            % Calculate symbolic equations of motion and put in parms struct

%% -- Perform simulation --
%% Calculate movement by mean sof a Runge-Kuta 4th order intergration method
```

```
tic
[t,x]                           = RK4_custom(x0,sim_time,parms);
toc

%% -- Post Processing --
%% Calculate com velocities
% xd                            = diff(x)/parms.h;
xdd                             = state_deriv(x,parms);

%% Calculate position of point A B and C
[A,B,C]                         = point_calc(x,parms);

%% Calculate kinetic energy and torque wo[ekin] = ekin_calc(x
    ,parms);
[ekin]                          = ekin_calc(x,parms);
[tw]                            = tw_calc(x,parms);

%% -- ANIMATE --
if animate_bool == 1
    % Adapted from A. Schwab's animation code

    % Rename data
    X1 = x(:,1); Y1 = x(:,2); P1 = x(:,3);
    DX1 = x(:,7); DY1 = x(:,8); DP1 = x(:,9);
    X2 = x(:,4); Y2 = x(:,5); P2 = x(:,6);
    DX2 = x(:,10); DY2 = x(:,11); DP2 = x(:,12);

    % Rename Points
    XA = A(:,1); YA = A(:,2);
    XB = B(:,1); YB = B(:,2);
    XC = C(:,1); YC = C(:,2);

    % Create figure
    figure
    plot(X1,Y1)
    hold on
    plot(XA,YA)
    hold on
    plot(X2,Y2)
    hold on
    plot(XC,YC)
    grid on
    set(gca,'fontsize',16)
```

```
    title('Animation EzyRoller')
    axis([min(X1)-parms.a max(X1)+parms.a min(Y1)-parms.a max
       (Y1)+parms.a]);
    axis equal
    l = plot([X1(1) XA(1)],[Y1(1) YA(1)]);
    k = plot([X2(1) XC(1)],[Y2(1) YC(1)]);
    j = plot([X1(1) XB(1)],[Y1(1) YB(1)]);
    m = plot([XB(1) X2(1)],[YB(1) Y2(1)]);
    set(l,'LineWidth',5);
    set(l,'Color','K')
    set(k,'LineWidth',5);
    set(k,'Color','C')
    set(j,'LineWidth',5);
    set(j,'Color','K')
    set(m,'LineWidth',5);
    set(m,'Color','C')
    nstep = length(t);
    nskip = 10;
    for istep = 2:nskip:nstep
        set(l,'XData',[X1(istep) XA(istep)])
        set(l,'YData',[Y1(istep) YA(istep)])
        set(k,'XData',[X2(istep) XC(istep)])
        set(k,'YData',[Y2(istep) YC(istep)])
        set(j,'XData',[X1(istep) XB(istep)])
        set(j,'YData',[Y1(istep) YB(istep)])
        set(m,'XData',[XB(istep) X2(istep)])
        set(m,'YData',[YB(istep) Y2(istep)])
        drawnow
        pause(1e-10)
    end

end

%% - - Create plots - -
%% Plot path of points on the robot
figure;
plot(A(:,1),A(:,2),x(:,1),x(:,2),B(:,1),B(:,2),x(:,4),x(:,5),
   C(:,1),C(:,2),'linewidth',1.5);
set(gca,'fontsize',18);
title('Travelled path of points on the EzyRoller');
xlabel('x position [m]');
ylabel('y position [m]');
```

```
legend('Point A (Wheel 1)','COM 1','Point B (hindge)','COM 2','Point C (wheel 2)','Location', 'Best');

%% Plot linear velocities COM's
figure;
subplot(2,1,1);
plot(t,x(:,7),'b',t,x(:,10),'r','Linewidth',1.5);
title("x velocities of COM's");
xlabel('t [s]');
ylabel('velocity [m/s]');
legend('x velocity (COM 1)','x velocity (COM 2)','Location','Best');
subplot(2,1,2);
plot(t,x(:,8),'b',t,x(:,11),'r','Linewidth',1.5);
title("y velocities of COM's");
xlabel('t [s]');
ylabel('velocity [m/s]');
legend('y velocity (COM 1)','y velocity (COM 2)','Location','Best');

%% Plot linear magnitude velocities COM's
% Calculate velocity magnitudes
v_com1 = sqrt(x(:,7).^2+x(:,8).^2);
v_com2 = sqrt(x(:,10).^2+x(:,11).^2);

% Plot figure
figure;
plot(t,v_com1,'b',t,v_com2,'r','Linewidth',1.5);
title("velocitie mag of COM's");
xlabel('t [s]');
ylabel('velocity mag [m/s]');
legend('velocity (COM 1)','velocity (COM 2)','Location', 'Best');

%% Plot angular velocities
figure;
plot(t,x(:,9),'b',t,x(:,12),'r','Linewidth',1.5);
title("angular velocities of COM's");
xlabel('t [s]');
ylabel('angular velocity [rad/s]');
legend('angular velocity (COM 1)','angular velocity (COM 2)','Location', 'Best');

```

```
%% Plot linear and angular accelerations COM's
figure;
subplot(2,1,1);
plot(t,xdd(:,7),'b',t,xdd(:,10),'r','Linewidth',1.5);
title("x accelerations of COM's");
xlabel('t [s]');
ylabel('accelleration [m/s^2]');
legend('x accelleration (COM 1)','x celleration (COM 2)','Location', 'Best');
subplot(2,1,2);
plot(t,xdd(:,8),'b',t,xdd(:,11),'r','Linewidth',1.5);
title("y accellerations of COM's");
xlabel('t [s]');
ylabel('Accelleration [m/s^2]');
legend('y accelleration (COM 1)','y accelleration (COM 2)','Location', 'Best');

%% Plot angular accelerations-
figure;
plot(t,xdd(:,9),'b',t,xdd(:,12),'r','Linewidth',1.5);
title("Angular velocities of COM's");
xlabel('t [s]');
ylabel('Angular acceleration [rad/s^2]');
legend('Angular acceleration (COM 1)','Angular acceleration (COM 2)','Location', 'Best');

%% Plot reaction forces
figure;
plot(t,x(:,13:end),'Linewidth',1.5);
title("Reaction forces in the constraints");
xlabel('t [s]');
ylabel('Reaction Force [N]');
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');

%% Plot kinetic energy
figure;
plot(t,ekin(:,1),'-b',t,ekin(:,2),'-r',t,ekin(:,3),'-g','Linewidth',1.5)
set(gca,'fontsize',18);
title("Kinetic energy of the COM's");
xlabel('t [s]');
```

```
ylabel('Kinetic energy[Joule]');
legend('Kinetic energy (COM 1)','kinetic energy (COM 2)','Kinetic energy system','Location', 'Best');

%% Plot Kinetic energy plus torque energy
figure;
plot(t,ekin(:,1),'-m',t,ekin(:,2),'-c',t,ekin(:,3),'-b',t,tw,'--r','Linewidth',1.5)
set(gca,'fontsize',18);
title("Kinetic energy of the COM's and Torque Work");
xlabel('t [s]');
ylabel('Energy [Joule]');
legend('Kinetic energy (COM 1)','kinetic energy (COM 2)','Kinetic energy system','Input torque work','Location', 'Best');

%% FUNCTIONS

%% Post processing functions
% These functions are used to calculate quantaties that are not calculated
% during the simulation. This regards quantaties which are not state
% variables

% Calculate second derivative
function [xdd] = state_deriv(x,parms)

% preallocate memory for xdd vector
xdd          = zeros(size(x,1),12);

% Create time vector
time         = 0:parms.h:((parms.h*size(x,1))-parms.h);

% Loop through states
for ii = 1:size(x,1)
    % Set time
    t = time(ii);

    % Calculate xdd
    x_now_tmp     = x(ii,1:end-4);
    x_now_input   = num2cell([x(ii,[3 6 7:12]),t],1);
    xdd_tmp       = subs_xdd(x_now_input{:}).';
```

```
    xdd(ii,:)     = [x_now_tmp(7:12),xdd_tmp(1:6)];
end
end

% Calculation points on EzyRoller
function [A,B,C] = point_calc(x,parms)

%% Calculate Point A, B, C out of the state
A_x                = x(:,1)-parms.a*cos(x(:,3));
A_y                = x(:,2)-parms.a*sin(x(:,3));
B_x                = x(:,1)+parms.b*cos(x(:,3));
B_y                = x(:,2)+parms.b*sin(x(:,3));
C_x                = x(:,4)+parms.c*cos(x(:,6));
C_y                = x(:,5)+parms.c*sin(x(:,6));

% Put them in their corresponding vector
A                  = [A_x A_y];
B                  = [B_x B_y];
C                  = [C_x C_y];

end

% Calculate kinetic energy of COM's
function [ekin] = ekin_calc(x,parms)

% preallocate memory for ekin vector
ekin               = zeros(size(x,1),1);

% Loop through states
% State is x = [x1 y1 phi1 x2 y2 phi2 x1p y1p phi1p x2p y2p
    phi2p
for ii = 1:size(x,1)
    ekin(ii,1) = 0.5*x(ii,7:9)*parms.M(1:3,1:3)*x(ii,7:9).';
    ekin(ii,2) = 0.5*x(ii,10:12)*parms.M(4:6,4:6)*x(ii,10:12)
        .';
    ekin(ii,3) = 0.5*x(ii,7:12)*parms.M*x(ii,7:12).';
end
end

% Calculate kinetic energy of COM's
function [tw] = tw_calc(x,parms)

% Calculate the applied torque for the whole movement
```

```
% preallocate memory for xdd vector
tw               = zeros(size(x,1),1);

% Create time vector
time             = 0:parms.h:((parms.h*size(x,1))-parms.h);

% Create W vector
if parms.question_bool == 0
    for ii = (2:size(x,1))
        tw(ii)      = tw(ii-1) + sum(subs_F.'.*(x(ii,1:6)-x((ii-1),1:6)));
    end
else
    for ii = (2:size(x,1))
        tw(ii)      = tw(ii-1) + sum((subs_F(time(ii))).'.*(x(ii,1:6)-x((ii-1),1:6)));
    end
end

end

%% 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] = RK4_custom(x0,sim_time,parms)

% Initialise variables
time                = (0:parms.h:sim_time).';                                  % Create time array
x                   = zeros(length(time),16);                                  % Create empty state array
x(1,1:length(x0))   = x0;                                                      % Put initial state in array

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

```matlab

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

    % Add time constant
    t = time(ii);

    % Perform RK 4
    x_now_tmp           = x(ii,1:end-4);                                % Create cell for subs function function
    x_input             = num2cell([x(ii,[3 6 7:12]),t],1);             % Add time to state
    K1                  = [x_now_tmp(1,end-5:end),subs_xdd(x_input{:}).'];      % Calculate the second derivative at the start of the step
    x1_tmp              = x_now_tmp + (parms.h*0.5)*K1(1:end-4);        % Create cell for subs function function
    x1_input            = num2cell([x1_tmp([3 6 7:12]),t],1);           % Add time to state
    K2                  = [x1_tmp(1,end-5:end),subs_xdd(x1_input{:}).'];        % Calculate the second derivative halfway the step
    x2_tmp              = x_now_tmp + (parms.h*0.5)*K2(1:end-4);        % Refine value calculation with new found derivative
    x2_input            = num2cell([x2_tmp([3 6 7:12]),t],1);           % Add time to state
    K3                  = [x2_tmp(1,end-5:end),subs_xdd(x2_input{:}).'];        % Calculate new derivative at the new refined location
    x3_tmp              = x_now_tmp + (parms.h)*K3(1:end-4);            % Calculate state at end step with refined derivative
    x3_input            = num2cell([x3_tmp([3 6 7:12]),t],1);           % Add time to state
```

```
    K4                   = [x3_tmp(1,end-5:end),subs_xdd(x3_input{:}).'];                          % Calculate last second derivative
    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
    x(ii+1,1:end-4)      = x_now_tmp + (parms.h/6)*(K1(1:end-4)+2*K2(1:end-4)+2*K3(1:end-4)+K4(1:end-4));    % Perform euler intergration step

    % Calculate last acceleration
    if ii == (size(time,1)-1)
        x_now_tmp            = x(ii+1,1:end-4);                  % Create cell for subs function function
        x_input              = num2cell([x(ii+1,[3 6 7:12]),t],1);                                     % Add time to state
        K1                   = [x_now_tmp(1,end-5:end),subs_xdd(x_input{:}).'];                        % Calculate the second derivative at the start of the step
        x1_tmp               = x_now_tmp + (parms.h*0.5)*K1(1:end-4);         % Create cell for subs function function
        x1_input             = num2cell([x1_tmp([3 6 7:12]),t],1);                                      % Add time to state
        K2                   = [x1_tmp(1,end-5:end),subs_xdd(x1_input{:}).'];                          % Calculate the second derivative halfway the step
        x2_tmp               = x_now_tmp + (parms.h*0.5)*K2(1:end-4);         % Refine value calculation with new found derivative
        x2_input             = num2cell([x2_tmp([3 6 7:12]),t],1);                                      % Add time to state
```

```
        K3                      = [x2_tmp(1,end-5:end),subs_xdd(x2_input{:}).'];                      % Calculate new derivative at the new refined location
        x3_tmp                  = x_now_tmp + (parms.h)*K3(1:end-4);                                       % Calculate state at end step with refined derivative
        x3_input                = num2cell([x3_tmp([3 6 7:12]),t],1);                                      % Add time to state
        K4                      = [x3_tmp(1,end-5:end),subs_xdd(x3_input{:}).'];                      % Calculate last second derivative
        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
    end

    % Correct for intergration drift
    x_now_tmp = x(ii+1,:);
    [x_new,~] = gauss_newton(x_now_tmp,parms);

    % Update the constraint forces
    x_new_input         = num2cell([x(ii,[3 6 7:12]),t],1);
    x_update            = subs_xdd(x_new_input{:}).';

    % Overwrite position coordinates
    x(ii+1,:)       = [x_new(1:end-4) x_update(end-3:end)];

end
end

%% Constraint calculation function
function [C,Cd,D,Dd] = constraint_calc(x,parms)

% Get needed angles out
x_now_tmp       = num2cell(x,1);

%% Calculate position constraint
C               = subs_C(x_now_tmp{1:6}).';
```

```

% Calculate constraint derivative
Cd                  = subs_Cd(x_now_tmp{[3 6]}).';

%% Calculate velocity constraint
D                   = subs_D(x_now_tmp{[3 6:12]}).';

% Calculate velocity constraint derivative
Dd                  = subs_Dd(x_now_tmp{[3 6]}).';
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
% Calculate the two needed constraints
[C,Cd,~,~] = constraint_calc(x,parms);

%% Guass-Newton position constraint correction
n_iter              = 0;

   % Set iteration counter

   % Get position data out

% Solve non-linear constraint least-square problem
while (max(abs(C)) > parms.tol)&& (n_iter < parms.nmax)
    x_tmp               = x(1:6);
    n_iter = n_iter + 1;
    x_del  = Cd*inv(Cd.'*Cd)*-C.';
    x(1:6) = x_tmp+ x_del.';

    % Recalculate constraint
    [C,Cd,~,~]        = constraint_calc(x,parms);
end

% % Calculate the corresponding speeds
% x_tmp_vel           = x(7:12);
% Dxd_n1              = -Cd*inv(Cd.'*Cd)*Cd.'*x_tmp_vel.';
% x(7:12)             = x_tmp_vel + Dxd_n1.';
%
```

```
%% Gaus-newton velocity constraint correction
n_iter             = 0;

   % Set iteration counter

   % Get position data out

% % Calculate the two needed constraints
% [~,~,D,Dd] = constraint_calc(x,parms);

% % Solve non-linear constraint least-square problem
% while (max(abs(D)) > parms.tol)&& (n_iter < parms.nmax)
%     x_tmp           = x(7:12);
%     n_iter = n_iter + 1;
%     x_del  = Dd*inv(Dd.'*Dd)*-D.';
%     x(7:12) = x_tmp+ x_del.';
%
%     % Recalculate constraint
%     [~,~,D,Dd]      = constraint_calc(x,parms);
% end


% Calculate constraints
[~,Cd,D,Dd]         = constraint_calc(x,parms);
Sd                  = [Cd Dd];

% Calculate new velocities
x_tmp_vel           = x(7:12);
Dxd_n1              = -Sd*inv(Sd.'*Sd)*Sd.'*x_tmp_vel.';
x(7:12)             = x_tmp_vel + Dxd_n1.';

%% Recalculate error
[C,~,D,~]         = constraint_calc(x,parms);
C_error = C;
D_error = D;

% Store full error
error = [C_error D_error];
end

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

```
%% -- The code between this lines is done to obtain the latex
    formulas --
% % Create model parameters in symbolic form
% syms a b c d m1 m2 J1 J2 g;

% Overwrite with real values if you don't want the full
    symbolic expresion
a                = parms.a;
b                = parms.b;
c                = parms.c;
d                = parms.d;
m1               = parms.m1;
m2               = parms.m2;
J1               = parms.J1;
J2               = parms.J2;
g                = parms.g;

%% -- The code between this lines is done to create the latex
    formulas --

% Unpack symbolic variables from parms
x1               = parms.syms.x1;
y1               = parms.syms.y1;
phi1             = parms.syms.phi1;
x2               = parms.syms.x2;
y2               = parms.syms.y2;
phi2             = parms.syms.phi2;
t                = parms.syms.t;

% Generalised state derivative
x1d              = parms.syms.x1d;
y1d              = parms.syms.y1d;
phi1d            = parms.syms.phi1d;
x2d              = parms.syms.x2d;
y2d              = parms.syms.y2d;
phi2d            = parms.syms.phi2d;

% Create generalized coordinate vectors
x                = [x1;y1;phi1;x2;y2;phi2];
xd               = [x1d;y1d;phi1d;x2d;y2d;phi2d];

% Calculate Position constraints
```

```
C                   = [x1+b*cos(phi1)-x2+d*cos(phi2); ...
     y1+b*sin(phi1)-y2+d*sin(phi2)];

% Calculate Velocity constraints
v1                  = [x1d y1d 0;x2d y2d 0].';
omega               = [0 0 phi1d;0 0 phi2d].';
R_A_COM             = [-a*cos(phi1) -a*sin(phi1) 0; c*cos(phi2) c*sin(phi2) 0].';
Va                  = v1 + cross(omega,R_A_COM);
eA                  = [-sin(phi1) cos(phi1) 0; -sin(phi2) cos(phi2) 0].';
D_x                 = simplify([Va(:,1).'*eA(:,1);Va(:,2).'*eA(:,2)]);

% Split constraint in matrix vector product
D                   = equationsToMatrix(D_x,[x1d y1d phi1d x2d y2d phi2d]);

% Compute the jacobian of the (non-)holonomic constraints
JC_x                = simplify(jacobian(C,x.'));
JD_x                = simplify(jacobian(D_x,xd.'));

% Calculate convective component
JC_xd               = jacobian(JC_x*xd,x);
JD_xd               = jacobian(D*xd,x);

% Create system of DAE
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))];
B = [parms.F ;-JC_xd*xd;-JD_xd*xd];

% Calculate result expressed in generalized coordinates
if parms.accuracy_bool == 0
     xdd                 = inv(A)*B;        % Less accurate but in our case faster
else
     xdd                 = A\B;             % More accurate but it is slow
end
```

```
%% Convert to function handles
matlabFunction(simplify(xdd),'vars',[x1 y1 phi1 x2 y2 phi2],'vars',[phi1 phi2 x1d y1d phi1d x2d y2d phi2d t],'File','subs_xdd');

% Position constraint function handle
matlabFunction(simplify(C),'vars',[x1 y1 phi1 x2 y2 phi2],'File','subs_C');

% Position constraint derivative function handle
matlabFunction(simplify(JC_x),'File','subs_Cd');

% Velocity constraint  function handle
matlabFunction(simplify(D_x),'vars',[phi1 phi2 x1d y1d phi1d x2d y2d phi2d],'File','subs_D');

% Velocity constraint derivative function handle
matlabFunction(simplify(JD_x),'File','subs_Dd');

% Force torque volocity handle
if parms.question_bool == 0
    parms.F = sym(parms.F);
    matlabFunction(parms.F,'File','subs_F');
else
    matlabFunction(parms.F,'File','subs_F');
end

end
```

References

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