Statement of integrity

My homework is completely in accordance with the Academic Integrity

Figure 1: My handwritten statement of integrity

Acknoledgements

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

Setup overview

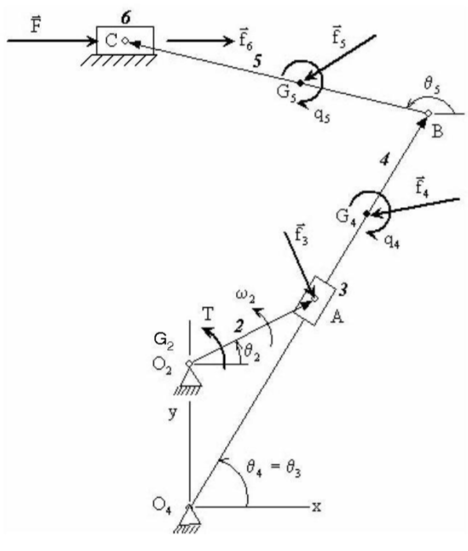

Figure 2: Quick return mechanism as depicted in assignment 7

Problem Statement

In this assignment we were asked to determine the motion of the quick return mechanism depicted in homework 7 (see 2). This quick return mechanism consisted of 3 bars connected by 2 slider joints and 2 revolute joints. As a result of these joints the quick return mechanism has 1 degree of freedom (3*3 -2 *2 -2*2 = 1 DOF). The qucik retrun mechanism has the following parameters:

$$O_2A = 0.2m \tag{1}$$
$$O_4 = 0.7m \tag{2}$$
$$O_4O_2 = 0.3m \tag{3}$$
$$O_4G_4 = 0.4m \tag{4}$$
$$BG_5 = 0.3m \tag{5}$$
$$y_c = 0.9m \tag{6}$$
$$m_3 = 0.5kg \tag{7}$$
$$m_4 = 6kg \tag{8}$$
$$m_6 = 2kg \tag{9}$$
$$J_4 = 10kgm^2 \tag{10}$$
$$J_5 = 6kgm^2 \tag{11}$$

On this mechanism the following forces and moments work:

$$F = 1000N \tag{12}$$
$$T = 0Nm \tag{13}$$

We further assume no friction and gravity.

Equations of motion (EOM)

The examine the motion of the mechanism described above we will use the TMT method explained in Chapter 5 of the reader [1] to derive the equations of motion (EOM).

TMT method

The Virtual Power TMT method is like the Lagrange method but differs in the fact that it does not go into the energy domain. Instead it stays in the forces domain and uses an incremental approach to obtain the equations of motion. Like the Lagrange method the TMT method uses generalized coordinates. The generalized coordinates for our problem are:

$$q = \begin{bmatrix} \phi_2 & \phi_4 & \phi_5 & \dot{\phi_2} & \dot{\phi_4} & \dot{\phi_5} \end{bmatrix} \tag{14}$$

The other angles in the quick return mechanism are all dependent on these 3 generalized coordinates as:

$$\phi_2 init = 0 \tag{15}$$

$$\phi_4 init = \tan^{-1}(\frac{O402 + O2Asin(\phi_2 init)}{O2Acos(\phi_2 init)}) \tag{16}$$

$$\phi_5 init = \pi - sin^{-1}(\frac{Y_c - O_4 Bsin(\phi_4 init)}{BC}) \tag{17}$$

$$\dot{\phi_2} init = \frac{(150\pi)}{60} \tag{18}$$

$$\dot{\phi_4} init = cos(\phi_4)^2 \dot{\phi_2} \tag{19}$$

$$\dot{\phi_5} init = \frac{O_4 Bcos(\phi_4)\dot{\phi_4}}{-BCcos(\phi_5)} \tag{20}$$

With these generalized coordinates we can derive the TMT method by first looking at the virtual power equation in which the D'Alembert forces are included:

$$\delta P = (F - M\ddot{x})\delta\dot{x} \tag{21}$$

The TMT method makes use of a transformation matrix T to transform the virtual COM velocities in this equation into generalized virtual velocities. The transformation is done follows:

$$\dot{x}_i = T_{i,j}\dot{q}_j \tag{22}$$

The kinematic-ally acceptable velocities now become:

$$\delta\dot{x}_i = T_{i,j}\delta\dot{q}_j \tag{23}$$

To obtain the Transformation matrix $T_{i,j}$ we first need to express the COM coordinates in terms of generalized coordinates. For our problem this results in the following expression:

$$x_2 = 0 \tag{24}$$
$$y_2 = O_4O_2 \tag{25}$$
$$x_3 = O_2Acos(\phi_2) \tag{26}$$
$$y_3 = O_4O_2 + O_2Asin(\phi_2) \tag{27}$$
$$x_4 = O_4G_4cos(\phi_4) \tag{28}$$
$$y_4 = O_4G_4sin(\phi_4) \tag{29}$$
$$x_5 = O_4Bcos(\phi_4) + BG_5cos(\phi_5) \tag{30}$$
$$y_5 = O_4Bsin(\phi_4) + BG_5sin(\phi_5) \tag{31}$$
$$x_6 = O_4Bcos(\phi_4) + BCcos(\phi 5) \tag{32}$$
$$y_6 = O_4Bsin(\phi_4) + BCsin(\phi 5) \tag{33}$$

The Transformation matrix can then be obtained by taking the Jacobean of this expression with respect to the generalized coordinates. This Jacobean was derived using the MATLAB symbolic toolbox. The script that was used in doing this can be found in appendix appendix A. When we fill equation 23 in in equation 21 we now get the following expression for the virtual power:

$$\delta P = (F - M\ddot{x})T_{i,j}\delta\dot{q}_j + \delta\dot{q}_jQ_j \tag{34}$$

Since we introduced a new set of coordinates q_j we now also get generalized forces Q_j. With these generalized forces and virtual velocities the virtual power expression is now almost fully expressed in terms of generalized coordinates. The only thing we need to do next is get rid of the COM accelerations $\ddot{x}$. We can do this by taking the derivative of expression (22):

$$\ddot{x}_k = T_{k,l}\ddot{q}_l + T_{k,lm}\dot{q}_l\dot{q}_m \tag{35}$$

When we fill this in to the virtual power expression in equation 34 we obtain the following result:

$$\delta P = \delta\dot{x}_i(F_i - M_{ik}T_{k,l}\ddot{q}_l - M_{ik}T_{k,lm}\dot{q}_l\dot{q}_m) + \delta\dot{q}_kQ_j \tag{36}$$

In this $T_{k,lm}$ is the derivative of the $T_{k,l}$ matrix with respect to the first derivative of the generalized coordinate state $\dot{q}_j$. After noting that this equation must hold for all virtual velocities and rearranging the equation a bit we obtain:

$$\bar{M}\ddot{q} = \bar{f} \tag{37}$$

Where:

$$\bar{M} = T^T M T and \bar{f} = T^T(F - mG) \tag{38}$$

In M represents the mass matrix and the F contains the applied forces and moment at the COM's of the segments:

$$M = \begin{bmatrix} J_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & m_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & J_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & m_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & m_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & J_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & m_5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & m_5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & J_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & m_6 \end{bmatrix} \tag{39}$$

$$x0 = \begin{bmatrix} T & 0 & -m_3 g & 0 & 0 & -m_4 g & 0 & 0 & -m_5 g & 0 & F \end{bmatrix} \tag{40}$$

The new G matrix is a convective term that arises due to deriving the virtual accelerations this term was also derived using the symbolic toolbox and will not be displayed here. The MATLAB code implementing the TMT method can be found in Appendix A.

Cut the loop method

Since our mechanism is closed loop we need to make use of the "Cut the loop method" to get these equations of motion. In this method we first make two cuts, one at sliding joint 6 and one at revolute joint 2, as a result we now have a open loop system with 3 degrees of freedom. This system now has 3 generalized coordinates ϕ_2,ϕ_4 and ϕ_5. To get back to the original DOF of the system we need to add 2 extra constraints:

$$C = \begin{bmatrix} x_3 - sqrt(x_3^2 + y_3^2)cos(\phi_4) \\ y_6 - Y_c \end{bmatrix} \tag{41}$$

Full system

If we combine the open loop system with the close loop system we get the following system of equation which can be solved in MATLAB:

$$\begin{pmatrix} T_{i,j} M_{ij} T_{j,k} & C_{c,l} \\ C_{c,k} & 0_{cc} \end{pmatrix} \begin{pmatrix} \ddot{q}_k \\ \lambda_c \end{pmatrix} = \begin{pmatrix} Q_l + T_{i,l}(F_i - M_{ij} q_j)) \\ -C_{c,kl} \dot{q}_k \dot{q}_l \end{pmatrix} \tag{42}$$

In this the $T_{i,l}$ is the Jacobean of state x w.r.t the generalized coordinates, $C_{c,l}$ is the Jacobean of the constraints w.r.t. the generalized coordinates and $C_{c,kl}$ is a convective term that comes from taking the second derivative of the constraint. Since all these are calculated with the symbolic toolbox they are not depicted here. To see what their structure is one is referred to the matlab script in appendix A.

Numerical integration method

To get the movement of the quick release mechanism 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.

Runge-Kutta 4th order method (RK4)

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

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

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

$$k_4 = f(t_n + h, y_n + h k_3) \tag{46}$$

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

Gauß-Newton corrections

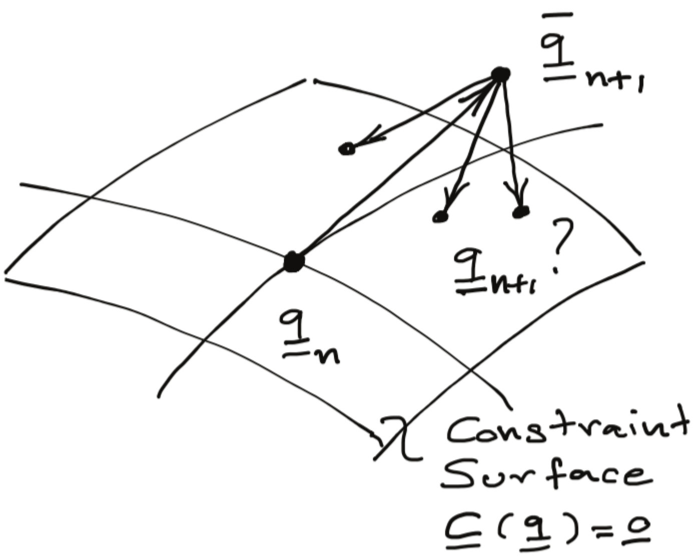

Figure 3: A depiction of the constraint surface and Gauß-Newton method as displayed in [1]. This picture was not modified in any sense

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

$$\|\bar{q}_{n+1} - q_{n+1}\|_2 = \min_{q_{n+1}}, \quad \forall \quad \{q_{n+1} | C(q_{n+1}) = 0\} \tag{48}$$

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 3).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:

$$q_{n+1} = \bar{q}_{n+1} + \Delta q_{n+1} \tag{49}$$

When you fill this in in the original

$$\Delta q_{n+1} = 0, \quad \forall \quad \{\Delta q_{n+1} | C(\bar{q}_{n+1}) \Delta q_{n+1} = 0\} \tag{50}$$

This leads to the following system of equations:

$$\begin{pmatrix} I & C^T \\ C0 & \end{pmatrix} \begin{pmatrix} \Delta \\ \mu \end{pmatrix} = \begin{pmatrix} 0 \\ e \end{pmatrix} \tag{51}$$

In which:

$$-CC^T\mu = e \tag{52}$$

$$\mu = -(CC^T)^{-1} \tag{53}$$

$$\Delta = C^T(CC^T)^{-1}e \tag{54}$$

In the end you obtain:

$$\Delta = C{+}e \tag{55}$$

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

$$q_{new} = q_{old} + \Delta \tag{56}$$

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. In determining the speeds that fulfill the constraints we however only need to solve a linear least square problem which can be solved by performing only one step. The MATLAB code performing the Gausß-Newton method for the positions and velocities are shown below in figure 4 5.

```
% Solve non-linear constraint least-square problem
while (max(abs(C)) > parms.tol)&& (n_iter < parms.nmax)
    q_tmp           = q(1:3);
    n_iter = n_iter + 1;
    q_del  = Cd*inv(Cd.'*Cd)*-C.';
    q(1:3) = q_tmp+ q_del.';

    % Recalculate constraint
    [C,Cd]        = constraint_calc(q,parms);
end
```

Figure 4: Code Gauß newton algorithm for the positions

```
% 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 5: Code Gauß newton algorithm for the velocities

Results

Below the results of the simulation are discussed.

Angular speed of crank 2, rocker 4 and connecting bar 5

From figure 6 we can see that both the crank, rocker and connecting bar oscillate around their axis of rotation. The rocker has the biggest amplitude while the crank the smallest this is what we would expect of the lengths of the segments.

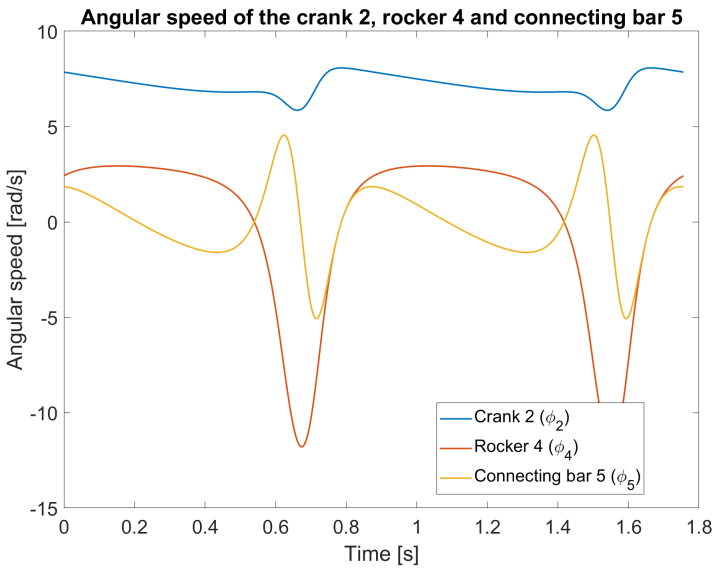

Figure 6: Plot of the angular velocity of crank 2, rocker 4 and connecting bar 5

Sliding speed of 3 relative to rocker 4

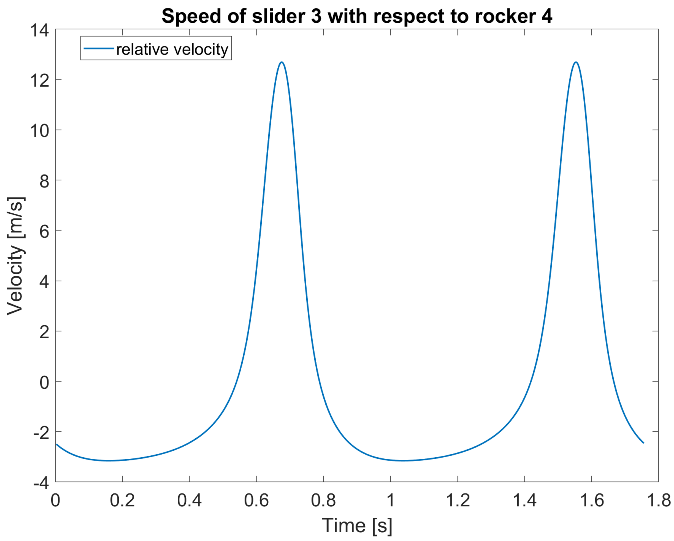

Figure 7: Speed slider 3 relative to rocker 3

Position slider 6

Now we look at the velocity position and acceleration of slider 6:

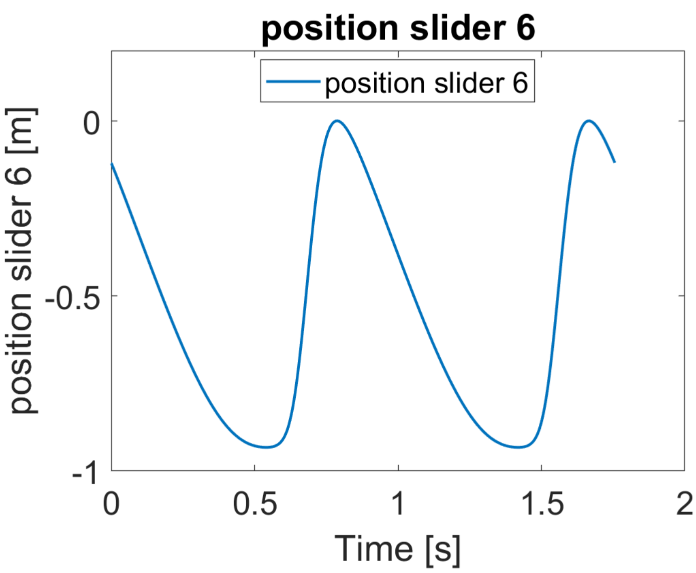

Figure 8: position slider 6

Velocity slider 6

Velocity of slider 6 can be found in figure 9.

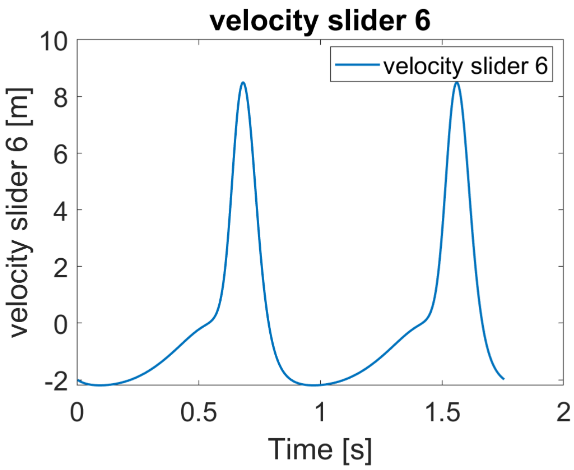

Figure 9: velocity slider 6

Acceleration slider 6

The acceleration of slider 6 can be found in figure 10

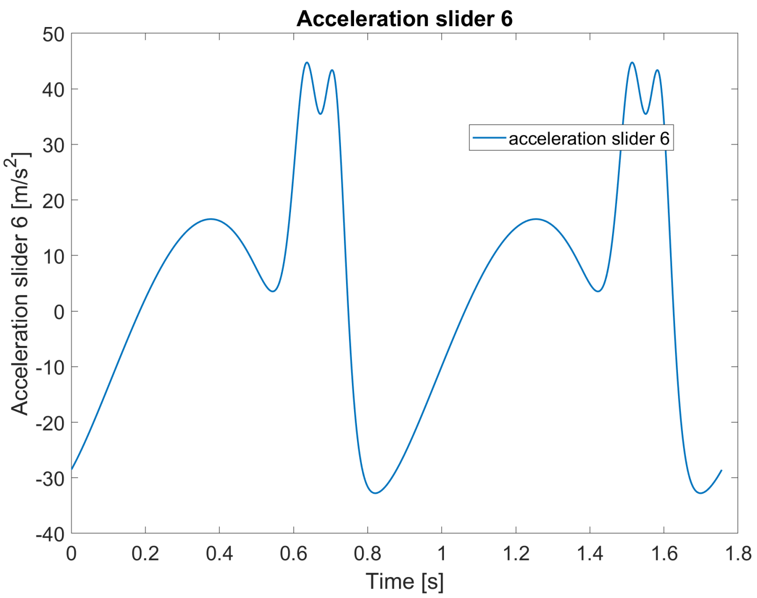

Figure 10: acceleration slider 6

Reaction forces in slider 6 and 3

In this section you will find the reaction forces experienced in slider 6 and 3. These reaction forces are shown in figure 11. From this figure we can see two things. First we also clearly see that the quick release mechanism in our simulation experience a oscillatory motion. Second the reaction force of the slider on the ground is way bigger than the reaction force of slider 3 on rocker 4. Lastly we see that also the amplitude of the reaction force of slider 6 on the ground is bigger. These results can be explained by the lower relative impact angle between slider 3 and rocker 4 compared to the impact angle between slider 6 and the ground.

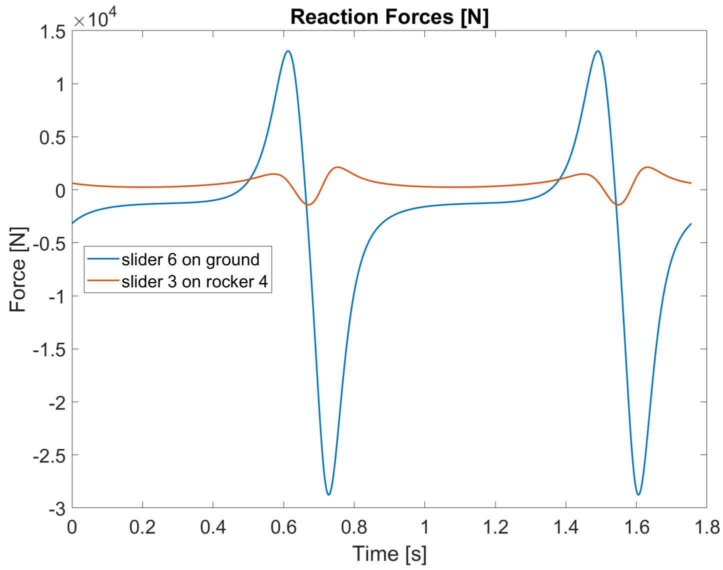

Figure 11: Reaction forces experienced in the quick release mechanism during the simulated motion

Validation checks

First of all I checked if the motion was cyclic since this is what I would expect based on intuition. That this is the case can be clearly seen from the plots. Secondly I used a open-source four bar mechanism plotter by "Mohammad Saadeh", which can be found on the MATLAB file exchange server, to check if the motion looked reasonable. Other possible checks would be calculating the kinetic and potential energy of the mechanism to see if energy is lost during the simulation. Lastly one can also calculate the static forces and torques which cause equilibrium in the mechanism and apply these to the model to see If our quick release mechanism is modelled the right way. The last two checks I unfortunately couldn't perform due to a recent bug in the MATLAB symbolic toolbox.

Appendix A: Accompanying MAT LAB scripts

```
%% MBD_B: Assignment 7 - Quick return mechanism
%  Rick Staa (4511328)
%  Last edit: 09/05/2018
clear all;tic; % close all; clc;
fprintf('--- A7 ---\n');

%% Script parameters
parms.accuracy_bool = 0;                          % If set to 1 A\b will be performed instead of inv(A)*B this is more accurate but slower
animate_bool = 0;                                 % Set on 1 if you want to see an animation

%% Set up needed symbolic parameters
% Create needed symbolic variables
syms phi2 phi4 phi5 phi2d phi4d phi5d

% Put in parms struct for easy function handling
parms.syms.phi2            = phi2;
parms.syms.phi4            = phi4;
parms.syms.phi5            = phi5;
parms.syms.phi2d           = phi2d;
parms.syms.phi4d           = phi4d;
parms.syms.phi5d           = phi5d;

%% Intergration parameters
time                       = 3;                                           % 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
parms.rot_sim              = 4*pi;                                        % How many rotations you want to simulate

```

```
%% Model Parameters
% Lengths and distances
parms.O2A                    = 0.2;                % Length segment 2 [m]
parms.O4B                    = 0.7;                % Length segment 4 [m]
parms.BC                     = 0.6;                % Length segment 5 [m]
parms.O4O2                   = 0.3;                % Distance between joint 4 and joint 2 [m]
parms.O4G4                   = 0.4;                % Distance bewteen COM4 and joint 4 [m]
parms.BG5                    = 0.3;                % Distance joint 5 and COM 5 [m]
parms.Yc                     = 0.9;                % Height joint C (COM body 6) [m]

% Masses and inertias
parms.m3                     = 0.5;                % Body 3 weight [kg]
parms.m4                     = 6;                  % Body 4 weight [kg]
parms.m5                     = 4;                  % Body 5 weight [kg]
parms.m6                     = 2;                  % Body 6 weight [kg]
parms.J2                     = 100;                % Moment of inertia body 2 [kgm^2]
parms.J3                     = 0;                  % Moment of inertia body 3 [kgm^2] - Put on 0 because no moment possible
parms.J4                     = 10;                 % Moment of inertia body 4 [kgm^2]
parms.J5                     = 6;                  % Moment of inertia body 5 [kgm^2]

```

```
%% World parameters
% Gravity
parms.g                     = 0;                                            % [parms.m/s^2]

% Forces
% Add forces F=[M2,F3_x,F3_y,M3,F4_x,F4_y,M4,F5_x,F5_y,M5,F6_x];
parms.F6_x                  = 1000;                                     % x force on body 6 [N]%
parms.T2                    = 0;                                        % Torque around joint 6 [Nm]
parms.F                     = [parms.T2, 0, -parms.m3*parms.g, 0, 0, -parms.m4*parms.g, 0, 0, -parms.m5*parms.g, 0, parms.F6_x];
parms.Q                     = [0;0;0];                                  % The generalised forces and torques

%% Calculate Initial states
phi2_init                   = 0;
phi4_init                   = atan2(parms.O4O2,parms.O2A);
phi5_init                   = pi-asin((parms.Yc-parms.O4B*sin(phi4_init))/parms.BC);
phi2d_init                  = (150*pi)/60;
phi4d_init                  = cos(phi4_init)^2*phi2d_init;              % Not real value but failed to calculate
phi5d_init                  = (parms.O4B*cos(phi4_init)*phi4d_init)/(-parms.BC*cos(phi5_init));  % Not real value but failed to calculate
q0                          = [phi2_init phi4_init phi5_init phi2d_init phi4d_init phi5d_init];

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

```

```
%% Calculate movement by mean sof a Runge-Kuta 4th order intergration method
[t_RK4,q_RK4,x_RK4,xdp_RK4]                              =
    RK4_custom(q0,parms);

%% Calculate com velocities
xp = diff(x_RK4)/parms.h;

%% Create plots

%% Plot Angular speed crank as a function of time
figure;
plot(t_RK4,q_RK4(:,4:6),'linewidth',1.5);
set(gca,'fontsize',18);
title('Angular speed of the crank 2, rocker 4 and connecting bar 5');
xlabel('Time [s]');
ylabel('Angular speed [rad/s]');
legend('Crank 2 (\phi_2)','Rocker 4 (\phi_4)','Connecting bar 5 (\phi_5)','Location', 'Best');

%% Plot the sliding speedof slider 3 with respect to rocker 4
v_slider_rel = xp(2:end,4).*cos(q_RK4(3:end,3)) + xp(2:end,5)
    .*sin(q_RK4(3:end,3));

figure;
plot(t_RK4,xdp_RK4(:,end),'linewidth',1.5);
set(gca,'fontsize',18);
xlabel('Time [s]');
ylabel('Acceleration slider 6 [m/s^2]');
title('Acceleration slider 6');
legend('acceleration slider 6','Location', 'Best');

figure;
plot(t_RK4,x_RK4(:,end),'linewidth',1.5);
set(gca,'fontsize',18);
xlabel('Time [s]');
ylabel('position slider 6 [m]');
title('position slider 6');
legend('position slider 6','Location', 'Best');

figure;
plot(t_RK4(4:end),xp(3:end,end),'linewidth',1.5);
```

```
set(gca,'fontsize',18);
xlabel('Time [s]');
ylabel('velocity slider 6 [m]');
title('velocity slider 6');
legend('velocity slider 6','Location', 'Best');

plot(t_RK4(4:end),v_slider_rel(2:end),'linewidth',1.5);
set(gca,'fontsize',18);
xlabel('Time [s]');
ylabel('Velocity [m/s]');
title('Speed of slider 3 with respect to rocker 4');
legend('relative velocity','Location', 'Best');

%% Normal forces
figure;
plot(t_RK4,q_RK4(:,end-1:end),'linewidth',1.5);
set(gca,'fontsize',18);
xlabel('Time [s]');
ylabel('Force [N]');
title('Reaction Forces [N]');
legend('slider 6 on ground','slider 3 on rocker 4','Location', 'Best')
toc

%% -- ANIMATE --
if animate_bool == 1
    % Adapted from A. Schwab's animation code and a animation script posted
    % on github by BitNide
    y = q_RK4.';
    g = figure;
    filename = 'testAnimated.gif';

    %% Loop through all time steps
    l = line([0, 0.2*cos(y(1,1))],[0.3, 0.3+0.2*sin(y(1,1))]);
    k = line([0, 0.7*cos(y(2,1))],[0, 0.7*sin(y(2,1))]);
    j = line([0.7*cos(y(2,1)), 0.7*cos(y(2,1)) + cos(y(3,1))*0.6],[0.7*sin(y(2,1)), 0.7*sin(y(2,1))+sin(y(3,1))*0.6]);
    xlim([-1.5 1])
    ylim([0 1])

```

```
    nstep = length(y);
    nskip = 10;

    for istep = 2:nskip:nstep
        set(l,'XData',[0, 0.2*cos(y(1,istep))])
        set(l,'YData',[0.3, 0.3+0.2*sin(y(1,istep))])
        set(k,'XData',[0, 0.7*cos(y(2,istep))])
        set(k,'YData',[0, 0.7*sin(y(2,istep))])
        set(j,'XData',[0.7*cos(y(2,istep)), 0.7*cos(y(2,istep
            )) + cos(y(3,istep))*0.6])
        set(j,'YData',[0.7*sin(y(2,istep)), 0.7*sin(y(2,istep
            ))+sin(y(3,istep))*0.6])
        drawnow
        pause(parms.h)
    end
end

%% 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 [t,q,x,xdd] = RK4_custom(q0,parms)

% Calculate x0
q_new_tmp                   = num2cell(q0,1);
x0                          = subs_x(q_new_tmp{1:3}).';
xdd0                        = subs_xdp(q_new_tmp{:}).';

% Initialise variables
t                           = 0;
                                                          % Initiate
   time
q                           = [q0 0 0 0 0 0];
                                              % Put initial state in
   array
x                           = x0;
xdd                         = xdd0;
```

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

% Perform intergration till two full rotations of the crank
ii = 1;                                                            % Create counter
while abs(q(ii,1)) < parms.rot_sim

    % Calculate the next state by means of a RK4 method
    q_now_tmp           = num2cell(q(ii,1:end-5),1);                                        % Create cell for subs function function
    K1                  = [cell2mat(q_now_tmp(1,end-2:end)), subs_qdp(q_now_tmp{:}).'];                 % Calculate the second derivative at the start of the step
    q1_tmp              = num2cell(cell2mat(q_now_tmp) + (parms.h*0.5)*K1(1,1:end-2));                  % Create cell for subs function function
    K2                  = [cell2mat(q1_tmp(1,end-2:end)), subs_qdp(q1_tmp{:}).'];                       % Calculate the second derivative halfway the step
    q2_tmp              = num2cell(cell2mat(q_now_tmp) + (parms.h*0.5)*K2(1:end-2));                    % Refine value calculation with new found derivative
    K3                  = [cell2mat(q2_tmp(1,end-2:end)), subs_qdp(q2_tmp{:}).'];                       % Calculate new derivative at the new refined location
    q3_tmp              = num2cell(cell2mat(q_now_tmp) + (parms.h)*K3(1:end-2));                        % Calculate state at end step with refined derivative
    K4                  = [cell2mat(q3_tmp(1,end-2:end)), subs_qdp(q3_tmp{:}).'];                       % Calculate last second derivative
    q_now_d             = (1/6)*(K1(end-4:end)+2*K2(end-4:end)+2*K3(end-4:end)+K4(end-4:end));          % Estimated current derivative
    q_next              = cell2mat(q_now_tmp) + (parms.h/6)*(K1(1:6)+2*K2(1:6)+2*K3(1:6)+K4(1:6));      % Perform euler intergration step

    % Save reaction forces and current derivative in state
```

```
    q(ii,end-4:end)      = q_now_d;

    % Save full state back in q array
    q                    = [q;[q_next 0 0 0 0 0]];

    % Correct for intergration drift
    q_now_tmp = q(ii+1,:);
    [q_new,~] = gauss_newton(q_now_tmp,parms);

    % Update the second derivative and the constraint forces
    q_new_tmp            = num2cell(q(ii,1:end-5),1);
    q_update             = subs_qdp(q_new_tmp{:}).';

    % Overwrite position coordinates
    q(ii+1,:)            = [q_new(1:6) q_update];

    % Create time array
    t                    = [t;t(ii)+parms.h];                % Perform Gauss-Newton drift correction
    ii                   = ii + 1;                            % Append counter

    % Calculate COM coordinates
    % Calculate COM coordinates
    x_tmp                = subs_x(q_new_tmp{1:3}).';
    xdd_tmp              = subs_xdp(q_new_tmp{:}).';

    % Save x in state
    x                    = [x;x_tmp];
    xdd                  = [xdd;xdd_tmp];

end
end

%% Constraint calculation function
function [C,Cd] = constraint_calc(q)

% Get needed angles out
q_now_tmp        = num2cell(q,1);

% Calculate the two needed constraints
```

```
C                  = subs_C(q_now_tmp{1:3});

% Calculate constraint derivative
Cd                 = subs_Cd(q_now_tmp{1:3}).';

end

%% Speed correct function
function [q,error] = gauss_newton(q,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(q);

%% Guass-Newton position correction
n_iter             = 0;

    % Set iteration counter

% Solve non-linear constraint least-square problem
while (max(abs(C)) > parms.tol)&& (n_iter < parms.nmax)
    q_tmp              = q(1:3);
    n_iter             = n_iter + 1;
    q_del              = Cd*inv(Cd.'*Cd)*-C;
    q(1:3)             = q_tmp+ q_del.';

    % Recalculate constraint
    [C,Cd]             = constraint_calc(q);
end

% 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.';

error = C;
end

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

```

```
% Unpack symbolic variables from varargin
phi2                = parms.syms.phi2;
phi4                = parms.syms.phi4;
phi5                = parms.syms.phi5;
phi2d               = parms.syms.phi2d;
phi4d               = parms.syms.phi4d;
phi5d               = parms.syms.phi5d;

% Create generalized coordinate vectors
q                   = [phi2;phi4;phi5];
qd                  = [phi2d;phi4d;phi5d];

% COM of the bodies expressed in generalised coordinates
x2                  = 0;
y2                  = parms.O4O2;
x3                  = parms.O2A*cos(phi2);
y3                  = parms.O4O2+parms.O2A*sin(phi2);
x4                  = parms.O4G4*cos(phi4);
y4                  = parms.O4G4*sin(phi4);
x5                  = parms.O4B*cos(phi4)+parms.BG5*cos(phi5);
y5                  = parms.O4B*sin(phi4)+parms.BG5*sin(phi5);
x6                  = parms.O4B*cos(phi4)+parms.BC*cos(phi5);
y6                  = parms.O4B*sin(phi4)+parms.BC*sin(phi5);

% Create mass matrix
% x2 = 0, y2 = 0 and y5 =0 also no moments around slider 3
   and 6
parms.M             = diag([parms.J2,parms.m3,parms.m3,parms.J3,
   parms.m4,parms.m4,parms.J4,parms.m5,parms.m5,parms.J5,parms
   .m6]);

% Put in one state vector
x                   = [phi2;x3;y3;phi4;x4;y4;phi4;x5;y5;phi5;x6];

% Calculate the two needed constraints
C                   = [x3-sqrt(x3^2+y3^2)*cos(phi4);
   ...
    y6-parms.Yc];

% Compute the jacobian of state and constraints
Jx_q                = simplify(jacobian(x,q.'));
JC_q                = simplify(jacobian(C,q));

```

```
%% Calculate convective component
Jx_dq           = jacobian(Jx_q*qd,q);
JC_dq           = jacobian(JC_q*qd,q);

% Solve with virtual power
M_bar           = simplify(Jx_q.'*parms.M*Jx_q);

% Create system of DAE
A               = [M_bar JC_q.'; JC_q zeros(size(JC_q,1))];
B               = [parms.Q + Jx_q.'*(parms.F.'-parms.M*Jx_dq*qd); ...
    -JC_dq*qd];

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

% % Get result back in COM coordinates
% xdp     = simplify(jacobian(xp,qp.'))*qdp + simplify(jacobian(xp,q.'))*qp;

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

% Constraint function handle
matlabFunction(simplify(C),'File','subs_C');                                   % Create function handle of EOM in terms of generalised coordinates

% Constraint derivative function handle
matlabFunction(simplify(JC_q),'File','subs_Cd');                              % Create function handle of EOM in terms of generalised coordinates
```

```
% Get back to COM coordinates
matlabFunction(simplify(x),'File','subs_x');                                        % Create function handle of EOM in terms of generalised coordinates

% Get xdp COM coordinates
xd                = Jx_q*qd;
xdd               = simplify(jacobian(xd,qd))*qdd(1:3)+simplify(jacobian(xd,q))*qd(1:3);
matlabFunction(simplify(xdd),'File','subs_xdd');                                    % Create function handle of EOM in terms of generalised coordinates

end
```

References

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