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

Statement of integrity

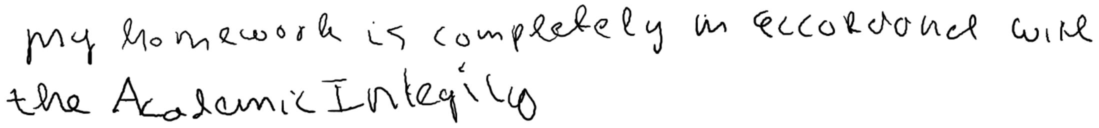

Figure 0.1: My handwritten statement of integrity

Acknoledgements

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

Setup overview

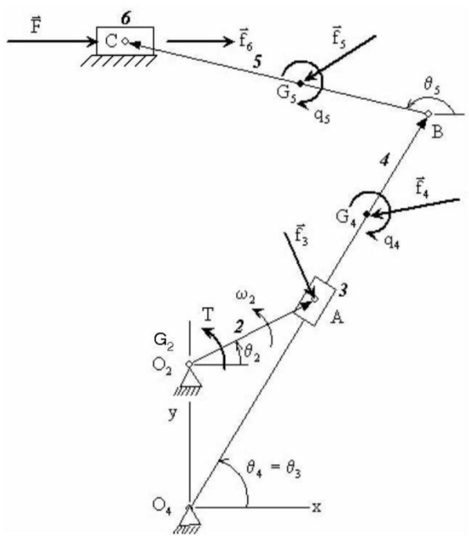

Figure 0.2: Quick return mechanism as depicted in assignment 7

Problem Statement

In this assignment we were asked to determine the motion of the quick return mechanism depicted in homework 7 (see 0.2). 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:

$$O_2A = 0.2m \tag{0.1}$$
$$O_4 = 0.7m \tag{0.2}$$
$$O_4O_2 = 0.3m \tag{0.3}$$
$$O_4G_4 = 0.4m \tag{0.4}$$
$$BG_5 = 0.3m \tag{0.5}$$
$$y_c = 0.9m \tag{0.6}$$
$$m_3 = 0.5kg \tag{0.7}$$
$$m_4 = 6kg \tag{0.8}$$
$$m_6 = 2kg \tag{0.9}$$
$$J_4 = 10kgm^2 \tag{0.10}$$
$$J_5 = 6kgm^2 \tag{0.11}$$

On this mechanism the following forces and moments work:

$$F = 1000N \tag{0.12}$$
$$T = 0Nm \tag{0.13}$$

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 doesn't go into the energy domain. Instead it stays in the forces domain and uses an incremental approach to obtain the equations of motion. We can derive the method by first looking at the virtual power equation with included D'Alembert forces:

$$\delta P = (F - M\ddot{x})\delta\dot{x} \tag{0.14}$$

Following the TMT method makes use of a transformation matrix T to transform the COM positions to the generalized coordinates. As a result we obtain the following equation:

$$\delta P = \delta\dot{x}_i(F_i - M_{ik}\ddot{x_k}) + \delta\dot{q}_k Q_j \tag{0.15}$$

Following we can derive the virtual accelerations which fulfill the constraints in equation 3 and fill this in in equation 8. After noting that this equation must hold for all virtual velocities and rearranging the equation a bit we obtain:

$$\bar{M}\ddot{q} = \bar{f} \tag{0.16}$$

Where:

$$\bar{M} = T^T M T and \bar{f} = T^T(F - mG) \tag{0.17}$$

In this T represents the transformation matrix which can be calculated by taking the Jacobian of the states x w.r.t. the general coordinates. In our problem the state is equal to:

$$x_2 = 0 \tag{0.18}$$
$$y_(2) = O_4O_2 \tag{0.19}$$
$$x_3 = O_2Acos(\phi_2) \tag{0.20}$$
$$y_3 = O_4O_2 + O_2Asin(\phi_2) \tag{0.21}$$
$$x_4 = O_4G_4cos(\phi_4) \tag{0.22}$$
$$y_4 = O_4G_4sin(\phi_4) \tag{0.23}$$
$$x_5 = O_4Bcos(\phi_4) + BG_5cos(\phi_5) \tag{0.24}$$
$$y_5 = O_4Bsin(\phi_4) + BG_5sin(\phi_5) \tag{0.25}$$
$$x_6 = O_4Bcos(\phi_4) + BCcos(\phi 5) \tag{0.26}$$

The M is again the mass matrix and the F contains the applied forces and moment at the COM's of the segments.

$$M = \begin{bmatrix} J_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 \\ 0 & m_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 \\ 0 & 0 & m_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 \\ 0 & 0 & 0 & J_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 \\ 0 & 0 & 0 & 0 & m_4 & 0 & 0 & 0 & 0 & 0 \\ 0 \\ 0 & 0 & 0 & 0 & 0 & m_4 & 0 & 0 & 0 & 0 \\ 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & J_4 & 0 & 0 & 0 \\ 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & m_5 & 0 & 0 \\ 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & m_5 & 0 \\ 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & J_5 \\ 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ m_6 \end{bmatrix} \tag{0.27}$$

$$F = \begin{bmatrix} T & 0 & -m_3 g & 0 & 0 & -m_4 g & 0 & 0 & -m_5 g & 0 \\ F \end{bmatrix} \tag{0.28}$$

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

$$q = \begin{bmatrix} \phi_2 & \phi_4 & \phi_5 & \dot{\phi_2} & \dot{\phi_4} & \dot{\phi_5} \end{bmatrix} \tag{0.29}$$

The other angles are all dependent on these 3 generalised coordinates. The intial state of the system is:

$$\phi_2 init = 0 \tag{0.30}$$

$$\phi_4 init = \tan^{-1}(\frac{O_4O_2}{O_2A}) \tag{0.31}$$

$$\pi - sin^{-1}(\frac{Y_c - O_4Bsin(\phi_4 init)}{BC}) \tag{0.32}$$

$$\dot{\phi_2} init = \frac{(150\pi)}{60} \tag{0.33}$$

$$\dot{\phi_4} init = cos(\phi_4)^2\dot{\phi_2} \tag{0.34}$$

$$\dot{\phi_5} init = \frac{O_4Bcos(\phi_4)\dot{\phi_4}}{-BCcos(\phi_5)} \tag{0.35}$$

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:

$$C = \begin{bmatrix} O_4Acos(\phi_4) + O_2Acos(\phi_2 \\ O_4Bsin(\phi_4 + BCsin(\phi_5) - Y_c \end{bmatrix} \tag{0.36}$$

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:

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

Figure 0.3: the end result

In this the $T_{i,l}$ is the jacobian of state x w.r.t the generalised coordinates, $C_{c,l}$ is the jacobian 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 refered to the matlab script in appendix A.

Numerical intergration method

To get the movent of the quick release mechanism in time we will use a 4^{th} order Runge-Kuta intergration method combined with a Gauß-Newton correction for position and speed. This correction is done to compensate for intergration drift.

Runge-Kutta 4th order method (RK4)

$$k_1 = f(t_n, y_n) \tag{0.37}$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \tag{0.38}$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \tag{0.39}$$

$$k_4 = f(t_n + h, y_n + hk_3) \tag{0.40}$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{0.41}$$

Gauß-Newton corrections

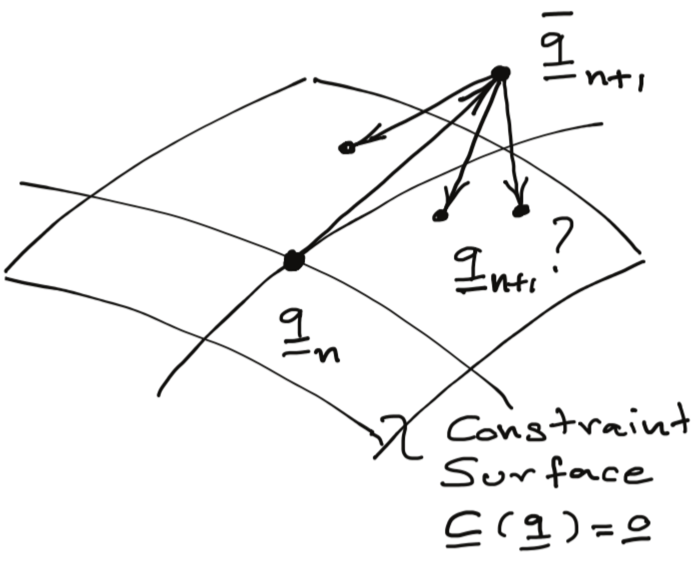

Figure 0.4: A depiction of the constraint surface and Gauß-Newton method as displayed in [1]. This picture was not modified in any sense

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

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

In words what your doing with the Gausß 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 0.4).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:

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

When you fill this in in the original

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

This leads to the following system of equations:

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

In which:

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

$$\boldsymbol{\mu} = -\left(\mathbf{C}\mathbf{C}^{\mathrm{T}}\right)^{-1}\boldsymbol{e},$$
$$\boldsymbol{\Delta} = \mathbf{C}^{\mathrm{T}}\left(\mathbf{C}\mathbf{C}^{\mathrm{T}}\right)^{-1}\boldsymbol{e}.$$

In the end you obtain:

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

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

$$q_new = q_old + \Delta \tag{0.42}$$

Following you can recalcaulte 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
The matlab code doing this operation is shown below:

```
% 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 0.5: Matlab code doing the position correction

velocity
The matlab code doing this operation 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 0.6: Matlab code doing the velocity correction

Results

Below the results of the simulation are discussed.

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

From figure 0.7 we can see that both the crank, rocker and connecting bar oscilate around their axis of roation. The rocker has the bigest amplitude while the crank the smallest this is what whe would expect of the lengths of the segments.

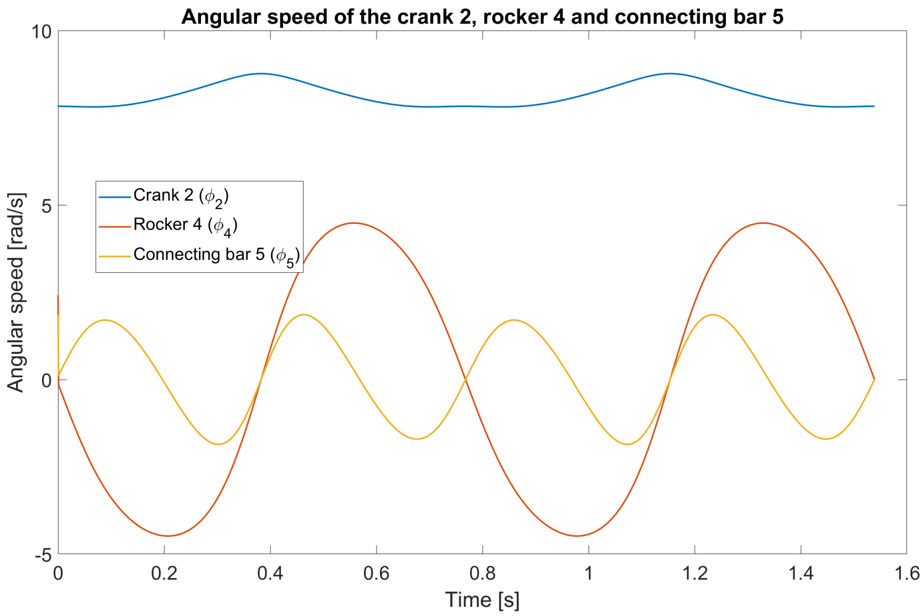

Figure 0.7: Plot of the angular velocity of crank 2, rocker 4 and connecting bar 5

Sliding speed of 3 relative to rocker 4

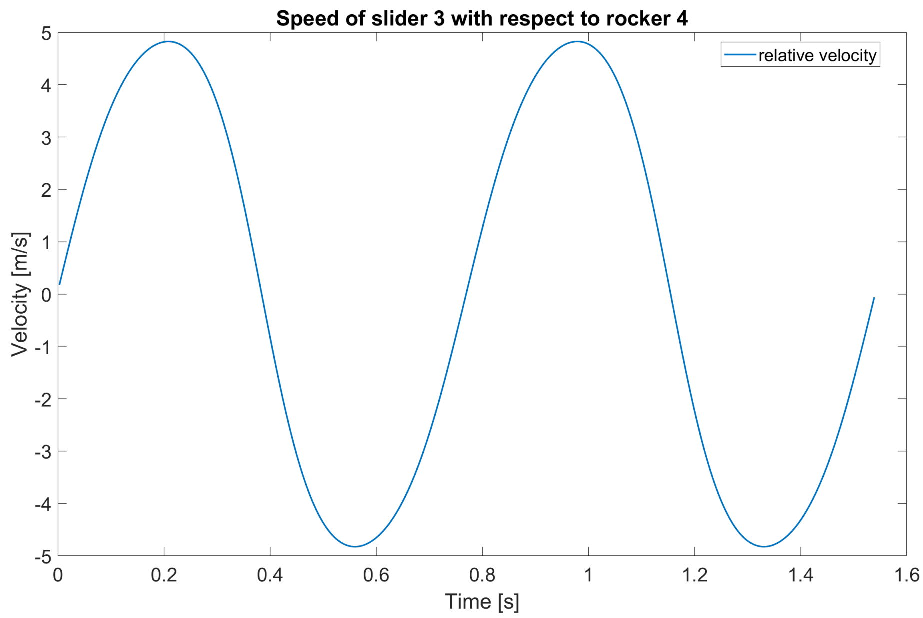

Figure 0.8: Speed slider 3 relative to rocker 3

Position slider 6

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

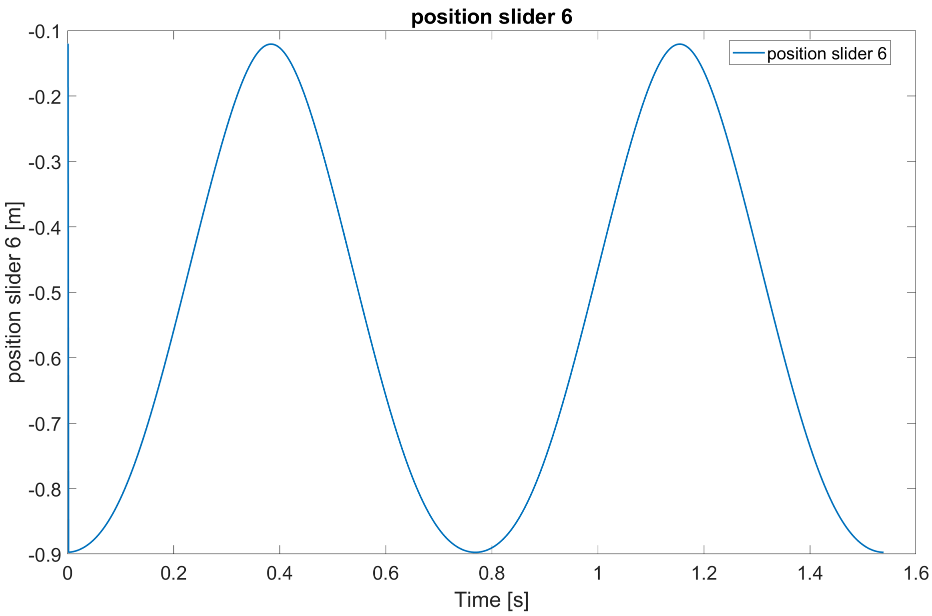

Figure 0.9: position slider 6

Velocity slider 6

Velocity of slider 6 can be found in figure 0.10.

Acceleration slider 6

The acceleration of slider 6 can be found in figure 0.11

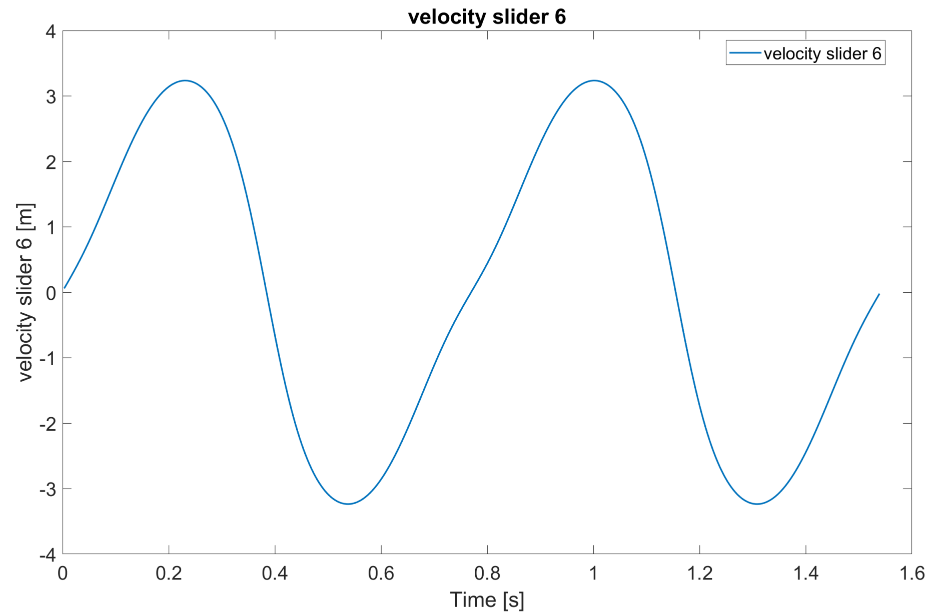

Figure 0.10: velocity slider 6

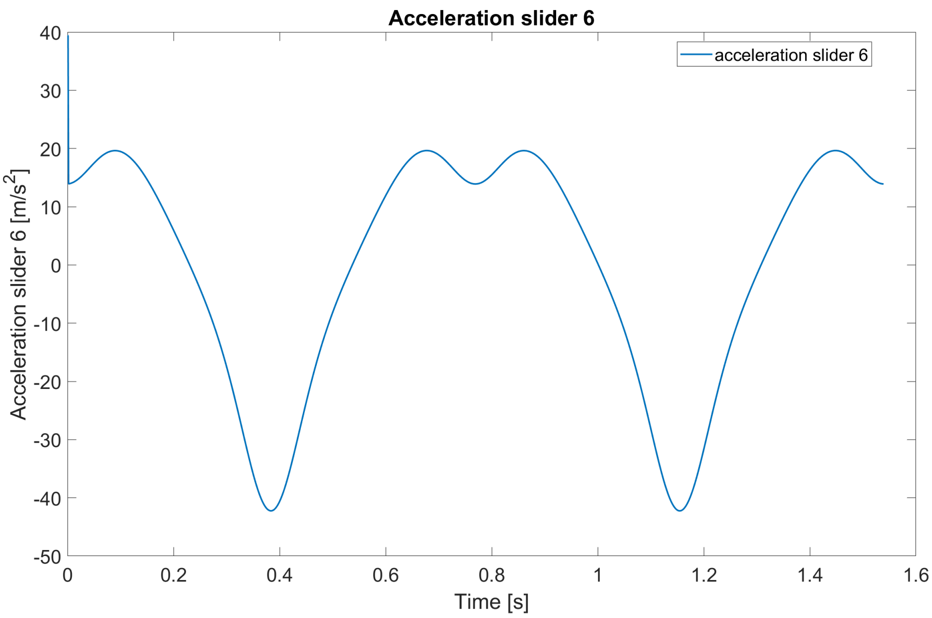

Figure 0.11: acceleration slider 6

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 0.12. 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.

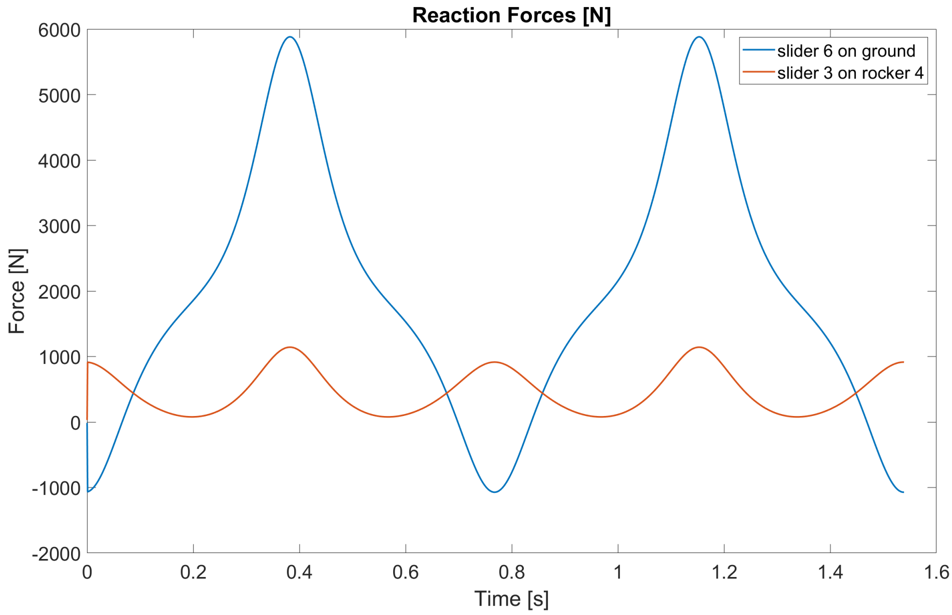

Figure 0.12: Reaction forces experienced in the quick release mechanism during the simulated motion

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 unfortunatelly couldn't perform due to a recent bug in the MATLAB symbolic toolbox.

Appendix A

The main MATLAB script

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

%% 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

%% 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]
parms.O4A                    = sqrt(parms.O2A^2+parms.O4O2^2);             % Distance between joint 4 and joint 3 [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                      = 9.81;           % [parms.m/s^2]

% Forces
```

```
parms.F6_x                  = 1000;                      % x force on body 6 [N]
parms.T2                    = 0;                            % Torque around joint 6 [Nm]

%% 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_qdp,C_handle,Cd_handle,X_handle,Xp_handle] = EOM_calc(parms);                   % Calculate symbolic equations of motion and put in parms struct
parms.C_handle               = C_handle;
parms.Cd_handle              = Cd_handle;
parms.X_handle               = X_handle;
parms.Xp_handle              = Xp_handle;

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

%% 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(2:end),xp(:,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')

%% 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,xdp] = RK4_custom(EOM,q0,parms)

% Calculate x0
q_new_tmp        = num2cell(q0,1);
x0   = feval(parms.X_handle,q_new_tmp{1:3}).';
xdp0 = feval(parms.Xp_handle,q_new_tmp{:}).';

% Initialise variables
t                     = 0;                                                         % Initiate time
q                     = [q0 0 0 0 0 0];                               % Put initial state in array
x                     = x0;
xdp                   = xdp0;
% Caculate the motion for the full simulation time by means of a
% Runge-Kutta4 method

% Perform intergration till two full rotations of the crank
```

```matlab
ii = 1;                                                              % Create counter
while abs(q(ii,1)) < (4*pi)

    % Calculate the next state by means of a RK4 method
    q_now_tmp          = num2cell(q(ii,1:end-5),1);                                                            % Create cell for feval function
    K1                 = [cell2mat(q_now_tmp(1,end-2:end)),feval(EOM,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 feval function
    K2                 = [cell2mat(q1_tmp(1,end-2:end)),feval(EOM,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)),feval(EOM,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)),feval(EOM,q3_tmp{:}).'];                  % Calculate last second derivative                % Take weighted sum of K1, K2, K3
    q_now_p            = (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_p;

    % 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,error] = 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            = feval(EOM,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
t(ii)
q(ii,1)

% Calculate COM coordinates
% Calculate COM coordinates
x_tmp    = feval(parms.X_handle,q_new_tmp{1:3}).';
xdp_tmp = feval(parms.Xp_handle,q_new_tmp{:}).';

% Save x in state
x        = [x;x_tmp];
xdp      = [xdp;xdp_tmp];

end
end

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

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

% Calculate the two needed constraints
C                = [parms.O4A*cos(q(2))+parms.O2A*cos(q(1)) ...
```

```
                     parms.O4B*sin(q(2))+parms.BC*sin(q(3))-parms.Yc];

C_test            = feval(parms.C_handle,q_now_tmp{1:3}).';

% Calculate constraint derivative
Cd                = feval(parms.Cd_handle,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,parms);

%% Guass-Newton position 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)
    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

% 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;
```

```matlab
end

%% Calculate (symbolic) Equations of Motion four our setup
function [qdp_handle,C_handle,Cd_handle,X_handle,Xd_handle] = 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];
qp                  = [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                   = [parms.O4A*cos(phi4)+parms.O2A*cos(phi2) ...
```

```
                   parms.O4B*sin(phi4)+parms.BC*sin(phi5)-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*qp,q);
JC_dq               = jacobian(JC_q*qp,q);

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

% Add forces F=[M2,F3_x,F3_y,M3,F4_x,F4_y,M4,F5_x,F5_y,M5,F6_x];
F                   = [parms.T2, 0, -parms.m3*parms.g, 0, 0, -parms.m4*parms.g, 0, 0, -parms.m5*parms.g, 0, parms.F6_x];

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

% Calculate result expressed in generalized coordinates
qdp                 = A\B;

% % 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
qdp_handle          = matlabFunction(simplify(qdp));         % Create function handle of EOM in terms of generalised coordinates
% matlabFunction(qdp,'file',qdp_cal')

% Constraint function handle
C_handle            = matlabFunction(simplify(C));

```

```
% Constraint derivative function handle
Cd                = JC_q;
Cd_handle         = matlabFunction(simplify(Cd));

% Get back to COM coordinates
X_handle          = matlabFunction(simplify(x));
xp                = Jx_q*qp;
xdp               = simplify(jacobian(xp,qp))*qdp(1:3)+simplify
   (jacobian(xp,q))*qp(1:3);
Xd_handle         = matlabFunction(simplify(xdp));

end
```

References

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