UNIVERSITÀ DEGLI STUDI DI TRENTO

Department of Industrial Engineering

Master's degree in *Mechatronics Engineering*

Course of Introduction to robotics

Optional Lab exercises

Professors:
Michele Focchi
Octavio Antonio Villareal Magana

Student:
Giorgio Checola

Academic Year 2020 - 2021

1 MATLAB Exercise: Model the elasticity of the transmissions

We started by modeling the dynamics of the DC motor and the load. In order to increase the torque and decrease the speed we added a gearbox between motor and load. At this point we model the elasticity of the transmission because it can affect the dynamics of the load. We can use a spring damper system (we will consider only the stiffness contribution K_t).
The equations of motion are:

$$\tau_m - \frac{1}{n}\left[k_t\left(\frac{\theta_m}{n} - \theta_l\right)\right] = J_m\ddot{\theta}_m + b_m\dot{\theta}_m$$

$$\tau_l + k_t\left(\frac{\theta_m}{n} - \theta_l\right) = J_l\ddot{\theta}_l + b_l\dot{\theta}_l$$

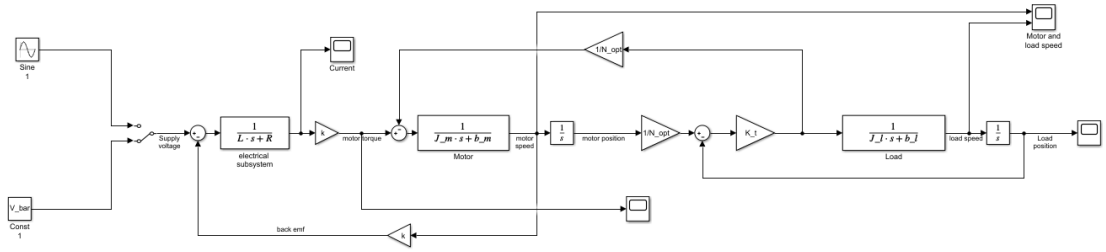
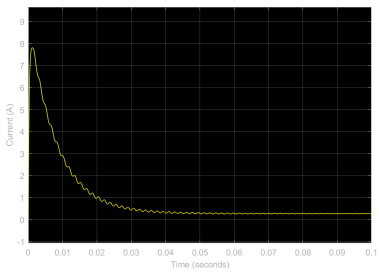
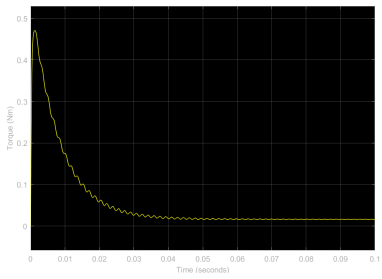


Figure 1: Model of the beam

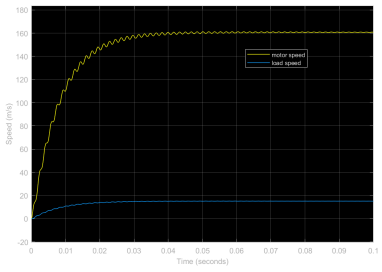
In the Figure 1 the Simulink model is shown. I imposed the external torque $\tau_l = 0$
Setting $K_t = 10000\ Nm/rad$ we get the following results:



(a) Current



(b) Motor torque

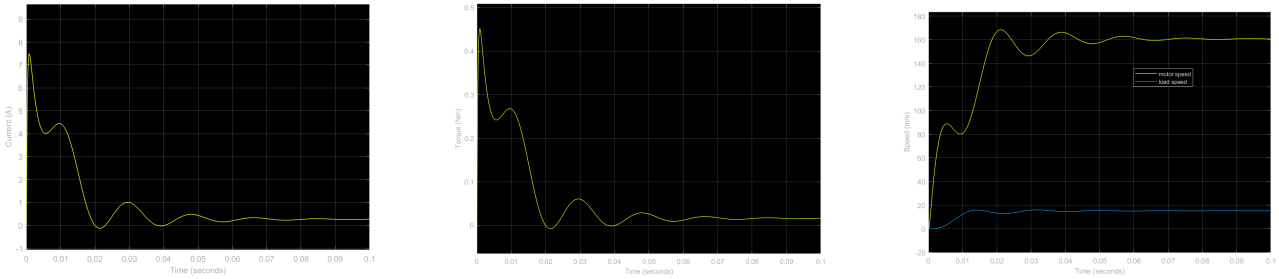


(c) Motor and load speed

Figure 2: K_t high: parameters behavior

From the picture above little oscillations are visible in all 3 plots. To the eye they are bigger in motor speed respect to motor torque and current.

Let's try to reduce the stiffness to 100 Nm/rad:



(a) Current (b) Motor torque (c) Motor and load speed

Figure 3: K_t low: parameters behavior

Reducing the stiffness of transmission oscillation increases significantly because the system becomes less stiff and more compliant.

2 PYTHON Exercise: Design a trajectory at the Cartesian level

Instead of computing the trajectory of the joints, we can do it at the end-effector level and then map to joint space by using inverse kinematics.
As initial and final point I selected respectively the one we obtained with direct kinematics of `conf.q0` and the desired point used later for inverse kinematics imposing $\Phi = 0$ for both of them.

```
start_p = np.array([-0.68538, -0.11015, 0.5216, 0.0])
end_p = np.array([-0.5, -0.2, 0.5, 0.0])
```

Velocity and acceleration has been initialized to 0.

```
start_p = np.array([-0.68538, -0.11015,  0.5216,   0])
end_p   = np.array([-0.5,          -0.2,     0.5, 0.0])
distance = end_p - start_p

j = 0
p = start_p
pd = np.array([0.0, 0.0, 0.0, 0.0])
pdd = np.array([0.0, 0.0, 0.0, 0.0])

while np.count_nonzero(p - end_p) :
#
    for i in range(3):
        a = coeffTraj_cart(3,start_p[i],end_p[i])
        pdd[i] = 2*a[2] + 6*a[3]*time + 12*a[4]*time**2 + 20*a[5]*time**3
        pd[i]  = a[1]   + 2*a[2]*time + 3*a[3]*time**2  + 4*a[4]*time**3 + 5*a[5]*time**4
        p[i]   = a[0]   + a[1]*time   + a[2]*time**2    + a[3]*time**3   + a[4]*time**4 + a[5]*time**5

#   p[:3] = start_p[:3] + distance[:3]*time
#   pd = distance

    q   = ik(p, q)
    qd  = (q-q_log[j])/conf.dt
    qdd = (qd-qd_log[j])/conf.dt

    q_des = ik(p,q)
    qd_des = (q-q_log[j])/conf.dt
    qdd_des = (q-q_log[j])/conf.dt
```

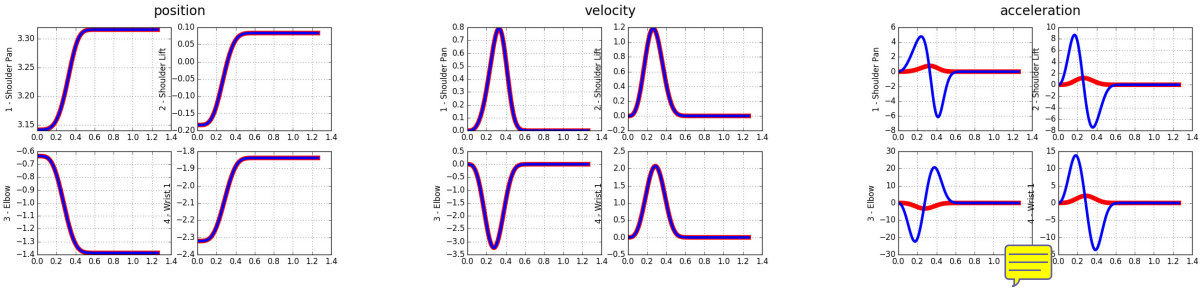
(a) Main code

```
def fifthOrderPolynomialTrajectory_cartesianspace(tf,p0,pf):

    # Matrix used to solve the linear system of equations for the polynomial trajectory
    polyMatrix = np.array([[1,  0,          0,          0,          0,          0],
                           [1, tf, np.power(tf,2),  np.power(tf,3),     np.power(tf,4),   np.power(tf,5)],
                           [0,  1,          0,          0,          0,          0],
                           [0,  1,        2*tf, 3*np.power(tf,2),  4*np.power(tf,3), 5*np.power(tf,4)],
                           [0,  0,          2,          0,          0,          0],
                           [0,  0,          2,        6*tf, 12*np.power(tf,2), 20*np.power(tf,3)]])

    polyVector = np.array([p0, pf, 0, 0, 0, 0])
    matrix_inv = np.linalg.inv(polyMatrix)
    polyCoeff = matrix_inv.dot(polyVector)

    return polyCoeff
```

(b) Cartesian fifth order polynomial

Figure 4: Python code exercise 2

In order to design the trajectory, I tried two different approach: interpolating two points in the workspace keeping constant the velocity, and manage the function of the fifth order polynomial for the end-effector, but I am not sure of the results, honestly. Moreover velocity and acceleration was computed in a simplified manner with the difference of two consecutive values and dividing by the time step.



(a) E[illegible]Position (b) E-E[illegible]locity (c) E-E Acceleration

Figure 5: Python code exercise 2

The plots show that position and velocity are tracked very well. You cannot say the same for the acceleration in which peaks which differ from the reference curve are present in the transient.

3 PYTHON Exercise: Inverse kinematics approach

I want to implement the end-effector tracking using inverse kinematics (with line search) to obtain the desired joint trajectory, and the pseudo-inverse Jacobian for velocity and acceleration.
In Figure 6 the code is shown: end-effector position and jacobian are computed with Pinocchio library; the Jacobian is a 3x6 fat matrix (x, y and z coordinates of the end-effector and 6 joints). For this reason the computation of Moore-Penrose pseudoinverse is $A^{\dagger} = A^T(AA^T)^{-1}$. Finally I used a PD control with feed-forward term and evaluate the tracking error.

```
# Inverse kinematics with line search
while np.linalg.norm(e_bar) >= epsilon and iter < max_iter:
    # compute end-effector Jacobian
    J6   = robot.frameJacobian(q0, frame_ee, False, pin.ReferenceFrame.LOCAL_WORLD_ALIGNED)
    J = J6[:3,:]
    # compute end-effector position with direct kinematics
    p_e = robot.framePlacement(q0, frame_ee, True).translation + np.array([0.0, 0.0, 0.0])
    # compute Newton step
    e_bar = p_e - p
    JtJ     = np.dot(J.T,J) + np.identity(J.shape[1])*lambda_
    JtJ_inv = np.linalg.inv(JtJ)
    P =-JtJ_inv.dot(J.T)
    dq = -P.dot(e_bar)

    # Update
    q1 = q0 + dq*alpha

    #Compute error of next step
    p_e1 = robot.framePlacement(q1, frame_ee, True).translation
    e_bar1 = p_e - p_e1
    e_bar_check = -np.linalg.norm(e_bar) + np.linalg.norm(e_bar1)
    threshold = gamma*alpha*np.linalg.norm(e_bar)
```

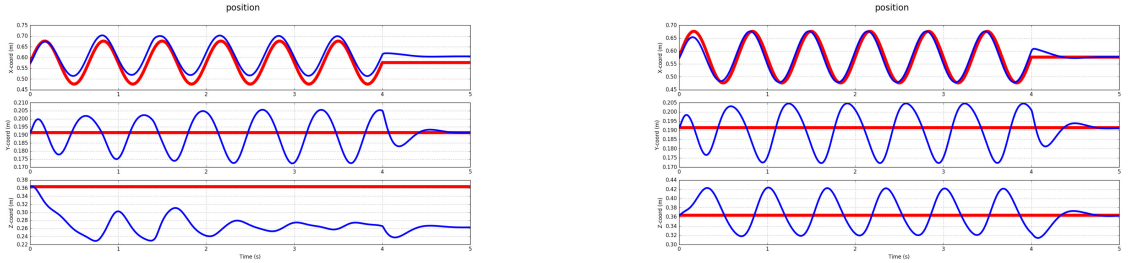
(a) Main code

```
##### Inverse kinematics approach (extra es)

    q_des   = ik_5(p_des, q, robot)

    Jpinv = J.T.dot(np.linalg.inv(J.dot(J.T))) # pseudo inverse J fat
    qd_des  = Jpinv.dot(pd_des)
    qdd_des = Jpinv.dot(pdd_des - Jdqd)

    # PD control + feedforward term
    M_des = robot.mass(q_des)
    tau =  np.multiply(np.diag(M_des), qdd_des) + kp.dot(q_des-q) + kd.dot(qd_des-qd)
```

(b) Inverse kinematics function

Figure 6: Python code exercise 3

I assessed the performance of my controller with the predefined sinusoidal reference trajectory. In the next plots we can see the corresponding results.



(a) PD control + FD term (b) PD control + FD term + gravity compensation

Figure 7: End-effector position

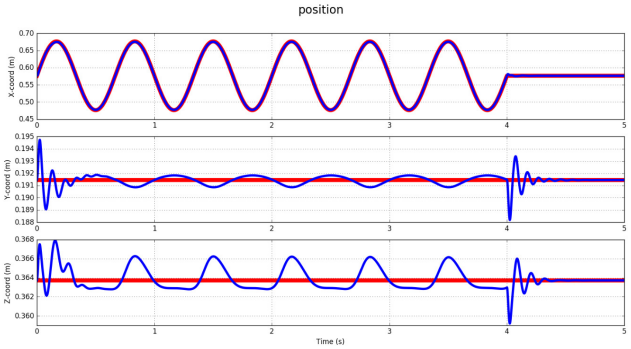


Figure 8: $K_p = 10000$

Observing that for the z-coordinate the initial result was bad, I added a gravity compensation (Figure 7 (b)) which made the manipulator follow the trajectory. Not completely satisfied I increase the K_p gain to 10000 and get the last plot.

List of Figures