University of São Paulo

São Carlos School of engineering

Legged Robots

Professor: Thiago Boaventura

Student: Jhon Charaja

Laboratory 3

Task Space Motion Control

São Carlos - Brasil

2021 - 2

1 Decentralized task space control

1.1 Generate sinusoidal reference

The objective of this activity is to generate a sinusoidal reference trajectory on the x-axis for 4 seconds and then change to a constant reference trajectory. Likewise, the reference trajectory should start in position $p_0 = \begin{bmatrix} 0.5765 & 0.1915 & 0.3637 \end{bmatrix}$ m. For this purpose, Algorithm 1 describes a function to generate a trajectory that change from sinusoidal to constant reference and consider initial end-effector position. Finally, Figure 1 shows the sinusoidal reference trajectories that robot end-effector will track in next activities.

```
def sinusoidal_reference_generator(q0, a, f, t_change, t):
    """
    @info: generates a sine signal.

    @inputs:
    ------
        - q0: initial joint/cartesian position   [or rad or m]
        - a: amplitude [rad]
        - f: frecuency [hz]
        - t_change: time to make the change  [sec]
        - t: simulation time [sec]
    @outputs:
    -------
        - q, dq, ddq: joint/carteisan position, velocity and acceleration
    """
    w = 2*np.pi*f                # [rad/s]
    if t<=t_change:
        q = q0 + a*np.sin(w*t)       # [rad]
        dq = a*w*np.cos(w*t)         # [rad/s]
        ddq = -a*w*w*np.sin(w*t)     # [rad/s^2]
    else:
        q = q0 + a*np.sin(w*t_change)   # [rad]
        dq = 0                          # [rad/s]
        ddq = 0                         # [rad/s^2]
    return q, dq, ddq
```

Algorithm 1: Function to generate a sinusoidal reference trajectory for some seconds and then change to a constant reference trajectory.

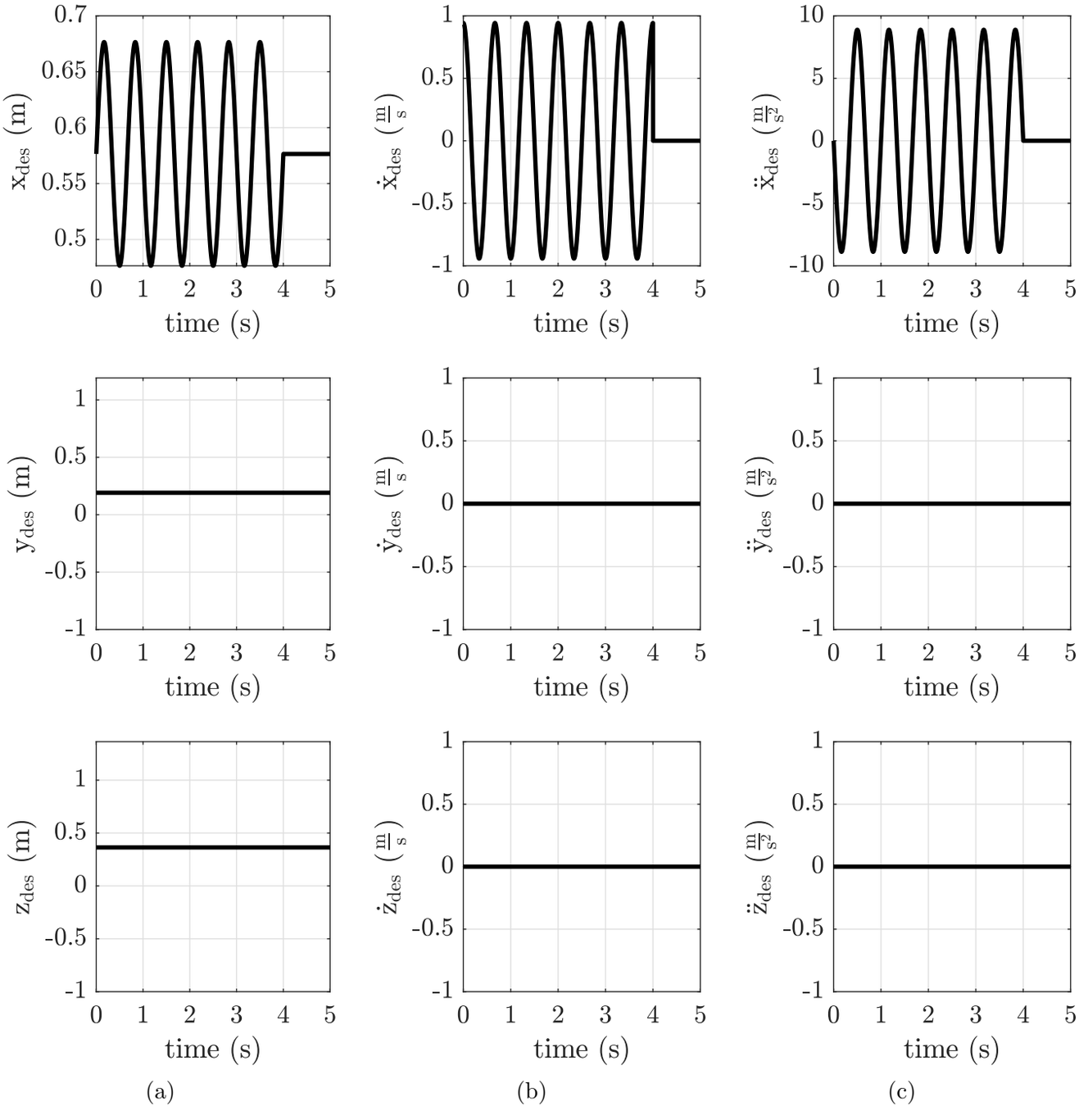


Figure 1: Cartesian reference trajectories for the robot end-effector with Algorithm 1: (a) position, (b) velocity and (c) acceleration.

1.2 Generate step reference

The objective of this activity is to generate a step reference trajectory on the z-axis after 2 seconds of simulation. Likewise, reference trajectory should start in position $p_0 = \begin{bmatrix} 0.5765 & 0.1915 & 0.3637 \end{bmatrix}$ m. For this purpose, Algorithm 2 describes a function to generate a step trajectory that consider initial end-effector position. Finally, Figure 2 shows the step reference trajectories that robot end-effector will track in next activities.

```
def step_reference_generator(q0, a, t_step, t):
    """
    @info: generate a constant reference.

    @inputs:
    ------
        - q0: initial joint/cartesian position
        - a: constant reference
        - t_step: start step [sec]
        - t: simulation time [sec]
    @outputs:
    -------
        - q, dq, ddq: joint/carteisan position, velocity and acceleration
    """
    if t>=t_step:
        q = q0 + a  # [rad]
        dq = 0      # [rad/s]
        ddq = 0     # [rad/s^2]
    else:
        q = copy(q0)    # [rad]
        dq = 0          # [rad/s]
        ddq = 0         # [rad/s^2]
    return q, dq, ddq
```

Algorithm 2: Function to generate a step reference trajectory.

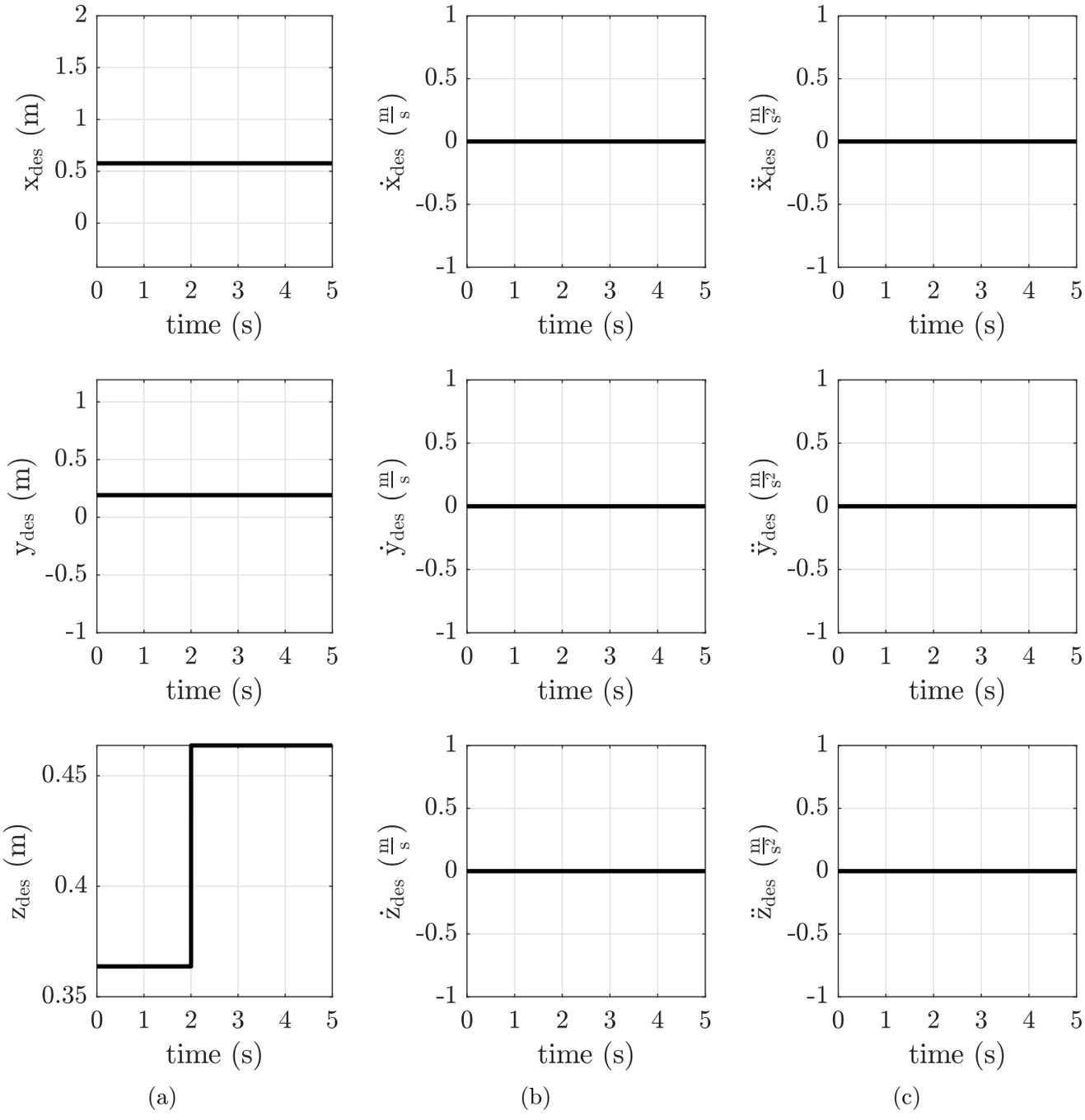


Figure 2: Cartesian reference trajectories for the robot end-effector with Algorithm 2: (a) position, (b) velocity and (c) acceleration.

1.3 Inverse kinematics approach

The objective of this activity is to control movement of the ur5 robot end-effector so that it follows the Cartesian sinusoidal reference trajectory of activity 1.1. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad and end-effector $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m. Likewise, the Cartesian sinusoidal reference trajectory starts at $\mathbf{p_0}$. Then, inverse kinematics computes joint reference points ($\mathbf{q_{des}}$) from the Cartesian sinusoidal trajectory ($\mathbf{p_{des}}$); Algorithm 3 describes a function to perform inverse kinematics using jacobian damped pseudo-inverse. Finally, movement of the ur5 robot is controlled with a proportional-derivative control method, at joint level, with feed-forward terms. Thus, control law can be computed as

$$\boldsymbol{\tau} = \mathbf{M}\ddot{\mathbf{q}}_{\mathbf{des}} + \mathbf{K_p}\mathbf{e} + \mathbf{K_d}\dot{\mathbf{e}}, \tag{1}$$

where $\mathbf{M}$ represent inertia matrix, $\mathbf{q_{des}}$ is desired joint trajectory, $\mathbf{e} = \mathbf{q_{des}} - \mathbf{q}$ is position error, and $\mathbf{K_p}, \mathbf{K_d}$ are the proportional and derivative gains respectively.

```
def inverse_kinematics_position(self, x_des, q0):
    """
    @info: computes inverse kinematics with jacobian damped pseudo-inverse.

    @inputs:
    -------
        - xdes  :   desired position vector
        - q0    :   initial joint configuration (it's very important)
    @outputs:
    --------
        - q_best  : joint position
    """
    best_norm_e     = 1e-6
    max_iter        = 10
    delta           = 1
    lambda_         = 0.0000001
    q               = copy(q0)

    for i in range(max_iter):
        p, _ = self.forward_kinematics(q) # current position
        e   = x_des - p      # position error
        J   = self.jacobian(q)[0:3, 0:self.ndof] # position jacobian [3x6]
        J_damped_inv =  self.jacobian_damped_pinv(J, lambda_) # inverse jacobian
           [6x3]
        dq  = np.dot(J_damped_inv, e)
```

```
        q   = q + delta*dq

        # evaluate convergence criterion
        if (np.linalg.norm(e)<best_norm_e):
            best_norm_e = np.linalg.norm(e)
            q_best = copy(q)
    return q_best
```

Algorithm 3: Function to compute inverse kinematics with jacobian damped psedo-inverse method.

The Algorithm 4 control the movements of ur5 robot end-effector to track the Cartesian sinusoidal reference trajectory of activity 1.1. In this file, the articular PD control method is configured with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$. On one hand, Figure 3 shows that trajectory tracking performance at the Cartesian space is poor with mean norm error at each axis $(||e_x||, ||e_y||, ||e_z||)$ of (0.0811, 0.0176, 0.062) cm respectively. The constant position error in z-axis could be reduced by adding gravity terms on control law (1). On the other hand, Figure 4 shows that trajectory tracking performance at joint space is poor because there are position error in all joints. The constant joint position error could be reduced by adding gravity terms on control law (1).

```
# =========================
#   Configuration of node
# =========================
# create a node:
rospy.init_node("inverse_kinematics_approach")
# public in topic /joint_states to send joint data
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)
# loop rate (in Hz)
rate  = rospy.Rate(1000)  # 1000 [Hz]
dt   = 1e-3     # 1  [ms]
# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
```

```
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
   wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof

# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ============================================
#   set initial cartesian configuration of UR5
# ============================================
# initial cartesian configuration: position, velocity and acceleration
p0 = ur5_robot.get_ee_position()
dp0 = np.array([0.0, 0.0, 0.0])
ddp0 = np.array([0.0, 0.0, 0.0])

# desired cartesian trajectory: position, velocity and acceleration
p_des = copy(p0)
dp_des = np.array([0.0, 0.0, 0.0])
ddp_des = np.array([0.0, 0.0, 0.0])

# measured cartesian trajectory: position, velocity and acceleration
p_med = copy(p0)
dp_med = np.array([0.0, 0.0, 0.0])
```

```
ddp_med = np.array([0.0, 0.0, 0.0])

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 600*np.ones(ndof)
# derivative gain
kd = 30*np.ones(ndof)
# control vector
tau = np.zeros(ndof)

#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0    # [sec]

while not rospy.is_shutdown():
    # desired cartesian trajectory
    p_des[0], dp_des[0], ddp_des[0] = sinusoidal_reference_generator(p0[0], 0.1,
       1.5, sine_duration, t)

    # jacobian: position xyz [3x6]
    J = ur5_robot.jacobian(q_des)[0:3, 0:6]
    # jacobian: dampend pseudo-inverse [6x3]
    J_pinv = ur5_robot.jacobian_damped_pinv(J)
    # jacobian: time derivative [3x6]
    dJ = ur5_robot.jacobian_time_derivative(q_des, dq_des)[0:3, 0:6]

    # desired joint trajectory
    q_des = ur5_robot.inverse_kinematics_position(p_des, q_des)
    dq_des = np.dot(J_pinv, p_des)
    ddq_des = np.dot(J_pinv, ddp_des - np.dot(dJ, dq_des))

    # error: position and velocity
    e  =  q_des - q_med
    de  =  dq_des - dq_med

    # compute inertia matrix
    M = ur5_robot.get_M()
```

```
    # control law: PD + feed-forward term
    tau_ff = M.dot(ddq_des)
    tau = tau_ff + np.multiply(kp, e) + np.multiply(kd, de)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q_med, dq_med, ddq_med = ur5_robot.read_joint_position_velocity_acceleration
        ()
    p_med, dp_med, ddp_med = ur5_robot.
        read_cartesian_position_velocity_acceleration()

    # publish message
    jstate.header.stamp = rospy.Time.now()
    jstate.name   = jnames   # Joints position name
    jstate.position  = q_med
    jstate.velocity  = dq_med
    pub.publish(jstate)

    # update time
    t = t + dt

    # stop simulation
    if t>=sim_duration:
        print("stopping rviz ...")
        break
    rate.sleep()
```

Algorithm 4: Move the ur5 robot end-effector using a articular proportional-derivative control method, (1), and inverse kinematics, Algorithm 3, to compute joint reference points from Cartesian sinusoidal reference trajectory of activity 1.1.

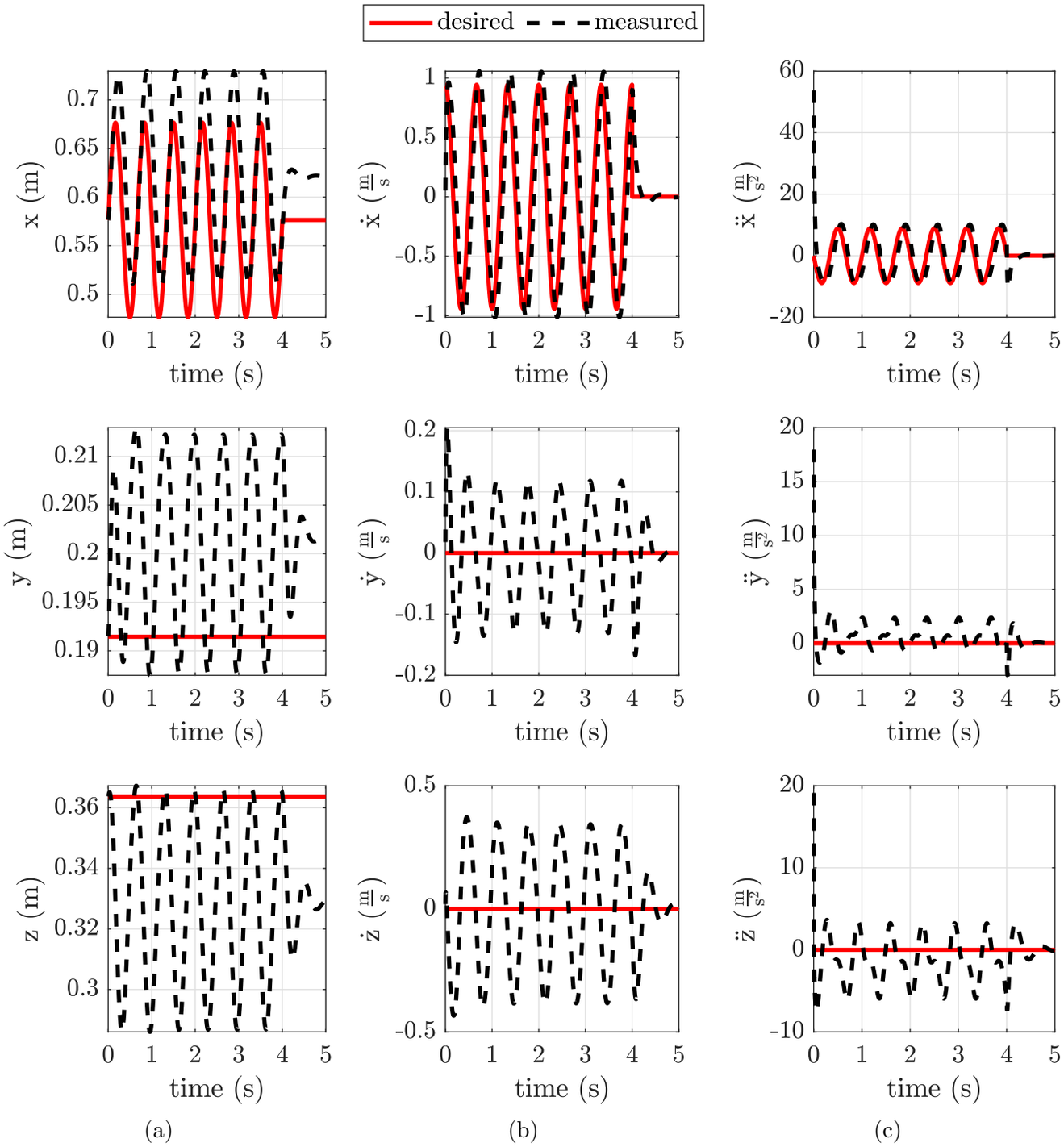


Figure 3: Cartesian trajectory tracking performances using articular proportional-derivative control method, (1), with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$: (a) position, (b) velocity and (c) acceleration.

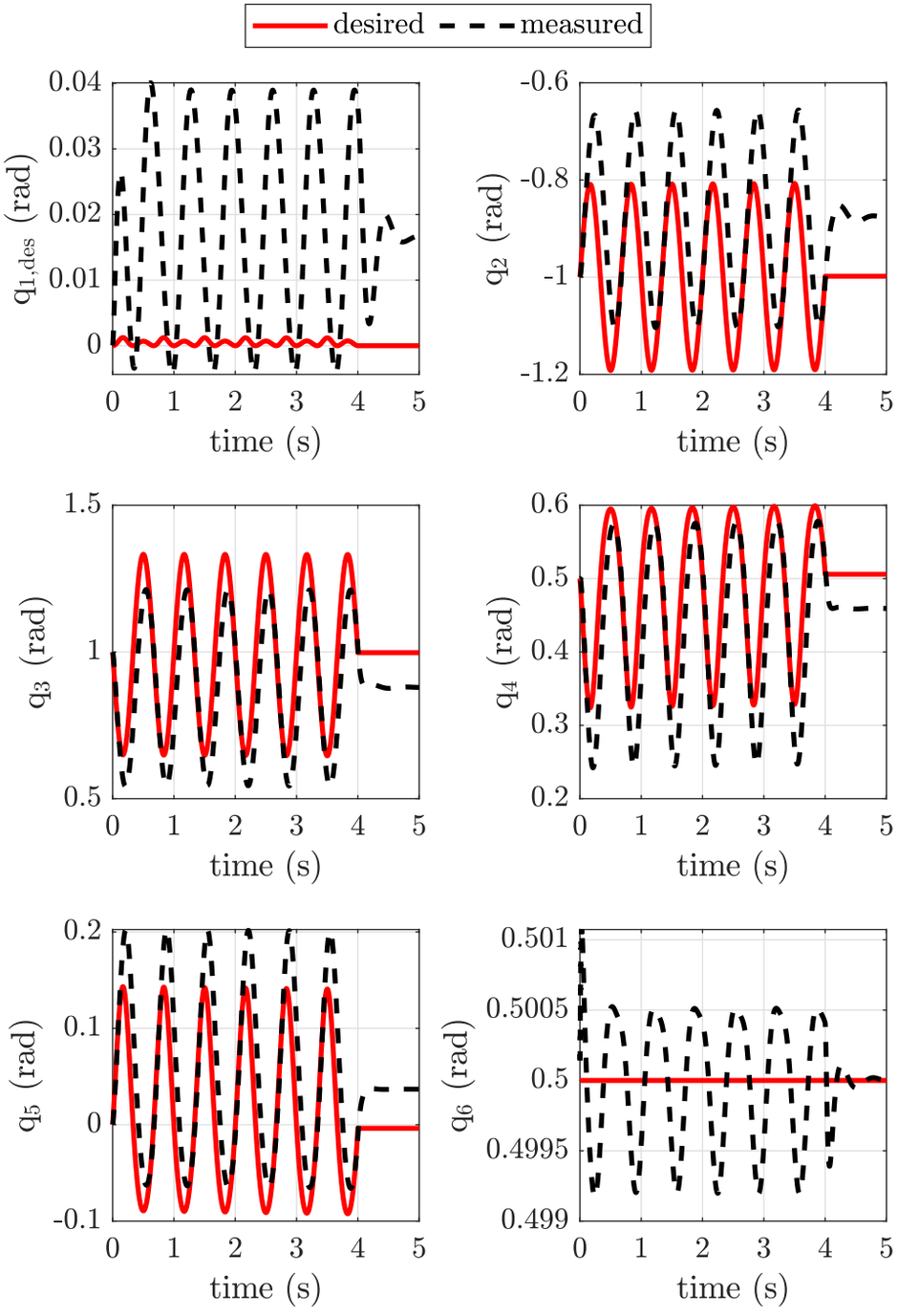


Figure 4: Angular trajectory tracking performances using articular proportional-derivative control method, (1), with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$.

1.4 Cartesian space PD control

The objective of this activity is to control movement of the ur5 robot end-effector so that it follows the Cartesian sinusoidal reference trajectory of activity 1.1. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad and end-effector $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m. Likewise, the Cartesian sinusoidal reference trajectory starts at $\mathbf{p_0}$. Finally, movement of the ur5 robot is controlled with a proportional-derivative control method at Cartesian level. Thus, control law can be computed as

$$\boldsymbol{\tau} = \mathbf{J^T}(\mathbf{K_p e} + \mathbf{K_d \dot{e}}), \tag{2}$$

where $\mathbf{J}$ is jacobian matrix, $\mathbf{e} = \mathbf{p_{des}} - \mathbf{p}$ is end-effector position error, and $\mathbf{K_p}, \mathbf{K_d}$ are the proportional and derivative gains respectively.

The Algorithm 5 control the movements of ur5 robot end-effector to track the Cartesian sinusoidal reference trajectory of activity 1.1. In this file, the Cartesian PD control method is configured with $K_p = 1000\ \frac{\text{N}}{\text{m}}$ and $K_d = 300\ \frac{\text{N.s}}{\text{m}}$. On one hand, Figure 5 shows that trajectory tracking performance at the Cartesian space is poor with mean norm error at each axis $(||e_x||, ||e_y||, ||e_z||)$ of (0.1853, 0.056, 0.107) cm respectively. The constant position error in z-axis could be reduced by adding gravity terms on control law (2); likewise, add feed-forward terms on control law (2) will reduce position error in x- and y-axis. On the other hand, Figure 6 shows that angular trajectory of each joint with Cartesian PD control method is different from angular trajectory obtained with articular PD control method in activity 1.3. This is because the control law focuses on reducing the Cartesian position error of the end-effector rather than position error of each joint. In this case, redundancy problem causes system to become unstable after 0.8 seconds of simulation.

```
# =========================
#   Configuration of node
# =========================
# create a node:
rospy.init_node("cartesian_PD_control")
# public in topic /joint_states to send joint data
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)
# loop rate (in Hz)
rate  = rospy.Rate(1000)  # 1000 [Hz]
dt    = 1e-3     # 1  [ms]
# object(message) type JointState
jstate = JointState()

# ========================================
```

```
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
   wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof

# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ============================================
#   set initial cartesian configuration of UR5
# ============================================
# initial cartesian configuration: position, velocity and acceleration
p0 = ur5_robot.get_ee_position()
dp0 = np.array([0.0, 0.0, 0.0])
```

```
ddp0 = np.array([0.0, 0.0, 0.0])

# desired cartesian trajectory: position, velocity and acceleration
p_des = copy(p0)
dp_des = np.array([0.0, 0.0, 0.0])
ddp_des = np.array([0.0, 0.0, 0.0])

# measured cartesian trajectory: position, velocity and acceleration
p_med = copy(p0)
dp_med = np.array([0.0, 0.0, 0.0])
ddp_med = np.array([0.0, 0.0, 0.0])

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = np.array([1000, 1000, 1000])   # N/m
# derivative gain
kd = np.array([300, 300, 300])      # N.s/m
# control vector
tau = np.zeros(ndof)

#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0    # [sec]

while not rospy.is_shutdown():
    # desired cartesian trajectory
    p_des[0], dp_des[0], ddp_des[0] = sinusoidal_reference_generator(p0[0], 0.1,
       1.5, sine_duration, t)

    # jacobian: position xyz [3x6]
    J = ur5_robot.jacobian(q_des)[0:3, 0:6]

    # error: position and velocity
    e  =  p_des - p_med
    de  =  dp_des - dp_med

    # PD control method (cartesian space)
    tau = J.T.dot( np.multiply(kp, e) + np.multiply(kd, de) )
```

```
# send control signal
ur5_robot.send_control_command(tau)
# update states
q_med,dq_med,ddq_med = ur5_robot.read_joint_position_velocity_acceleration()
p_med, dp_med, ddp_med = ur5_robot.
    read_cartesian_position_velocity_acceleration()

# publish message
jstate.header.stamp = rospy.Time.now()
jstate.name   = jnames   # Joints position name
jstate.position  = q_med
jstate.velocity  = dq_med
pub.publish(jstate)
# update time
t = t + dt
# stop simulation
if t>=0.8:
    print("stopping rviz ...")
    break
rate.sleep()
```

Algorithm 5: Move the ur5 robot end-effector using the Cartesian proportional-derivative control method, (2), so that it follows the Cartesian sinusoidal reference trajectory of activity 1.1.

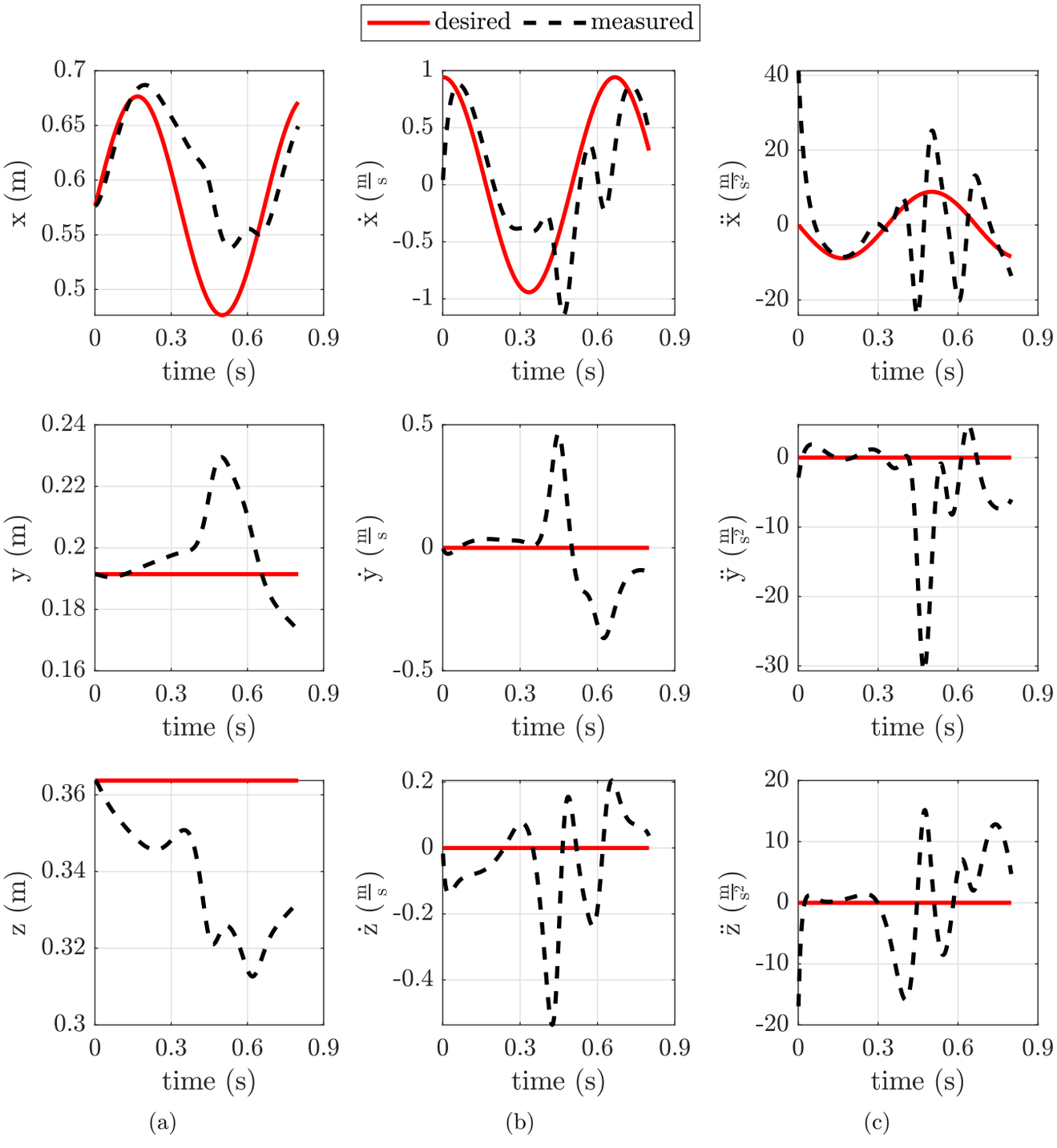


Figure 5: Cartesian trajectory tracking performances using Cartesian proportional-derivative control method, (2), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$ and $K_d = 300\ \frac{\text{N.s}}{\text{m}}$: (a) position, (b) velocity and (c) acceleration.

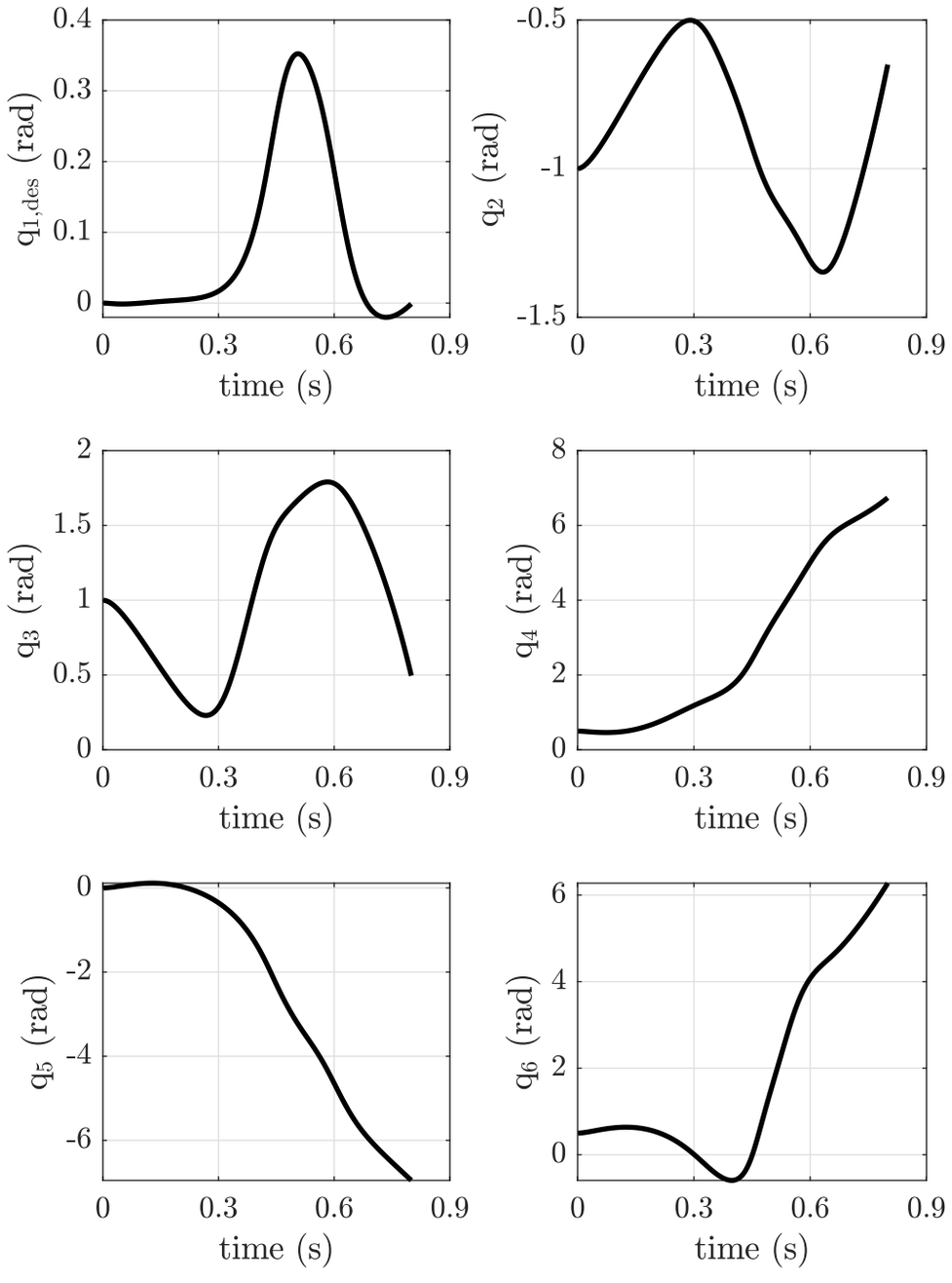


Figure 6: Angular position of each joint of UR5 robot with Algorithm 5.

1.5 Cartesian space PD control - postural task

The objective of this activity is to control movement of the ur5 robot end-effector so that it follows the Cartesian sinusoidal reference trajectory of activity 1.1. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad and end-effector $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m. Likewise, the Cartesian sinusoidal reference trajectory starts at $\mathbf{p_0}$. Motion control is made up of two approaches: Cartesian proportional-derivative and projection of the null space. In this sense, Cartesian PD control method focuses on reducing end-effector position error and the projection of null space maintains the articular position close to $\mathbf{q_0}$. Finally, control law can be computed as

$$\boldsymbol{\tau} = \mathbf{J^T}(\mathbf{K_p}\mathbf{e} + \mathbf{K_d}\dot{\mathbf{e}}) + \mathbf{N}\left(\mathbf{K_q}(\mathbf{q_0} - \mathbf{q}) - \mathbf{D_q}\dot{\mathbf{q}}\right), \tag{3}$$

$$\mathbf{N} = (\mathbf{I_{6\times 6}} - \mathbf{J}^{\#}\mathbf{J}),$$

where $\mathbf{J}$ is jacobian matrix, $\mathbf{e} = \mathbf{p_{des}} - \mathbf{p}$ is end-effector position error, $\mathbf{K_p}, \mathbf{K_d}$ are the proportional and derivative gains respectively, $\mathbf{J}^{\#}$ is jacobian damped pseudo-inverse, $\mathbf{N}$ is the null space projection of $\mathbf{J}^{\#}$, and $\mathbf{K_q}, \mathbf{D_q}$ are the proportional and derivative gains for null space projection.

The Algorithm 6 control the movements of ur5 robot end-effector to track the Cartesian sinusoidal reference trajectory of activity 1.1. In this file, the control law (3) is configured with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$. On one hand, Figure 7 shows that trajectory tracking performance at the Cartesian space is regular with mean norm error at each axis $(||e_x||, ||e_y||, ||e_z||)$ of (0.045, 0.013, 0.080) cm respectively. The constant position error in z-axis could be reduced by adding gravity terms on control law (3); likewise, add feed-forward terms on control law (3) will reduce position error in x- and y-axis. On the other hand, Figure 8 shows that angular trajectory of each joint with Cartesian PD control method is different from angular trajectory obtained with articular PD control method in activity 1.3. This is because the control law focuses on reducing the Cartesian position error of the end effector rather than position error of each joint. Finally, null space projection allows the smooth variation of the position of each joint, so the system remained stable for the entire simulation unlike activity 1.4.

```
# ========================
#   Configuration of node
# ========================
# create a node:
rospy.init_node("cartesian_space_PD_control_postural_task")
# public in topic /joint_states to send joint data
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)
# loop rate (in Hz)
```

```
rate  = rospy.Rate(1000)  # 1000 [Hz]
dt   = 1e-3     # 1  [ms]
# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
   wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof

# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)
```

```
# ================================================
#   set initial cartesian configuration of UR5
# ================================================
# initial cartesian configuration: position, velocity and acceleration
p0 = ur5_robot.get_ee_position()
dp0 = np.array([0.0, 0.0, 0.0])
ddp0 = np.array([0.0, 0.0, 0.0])

# desired cartesian trajectory: position, velocity and acceleration
p_des = copy(p0)
dp_des = np.array([0.0, 0.0, 0.0])
ddp_des = np.array([0.0, 0.0, 0.0])

# measured cartesian trajectory: position, velocity and acceleration
p_med = copy(p0)
dp_med = np.array([0.0, 0.0, 0.0])
ddp_med = np.array([0.0, 0.0, 0.0])

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = np.array([1000, 1000, 1000])   # N/m
# derivative gain
kd = np.array([300, 300, 300])      # N.s/m
# control vector
tau_PD = np.zeros(ndof)

# postural task: gains
Kq = 50*np.eye(ndof)
Dq = 10*np.eye(ndof)
# postural task: control term
tau_0 = np.zeros(ndof)

#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0    # [sec]

while not rospy.is_shutdown():
    # desired cartesian trajectory
```

```
    p_des[0], dp_des[0], ddp_des[0] = sinusoidal_reference_generator(p0[0], 0.1,
        1.5, sine_duration, t)

    # jacobian: position xyz [3x6]
    J = ur5_robot.jacobian(q_des)[0:3, 0:6]
    # jacobian: damped pseudo-inverse [6x3]
    J_inv = ur5_robot.jacobian_damped_pinv(J)

    # error: position and velocity
    e  =  p_des - p_med
    de  =  dp_des - dp_med

    # postural task: control term
    tau_0 = Kq.dot(q0-q_med) - Dq.dot(dq_med)
    N = np.eye(ndof) - J_inv.dot(J)

    # PD control method (cartesian space)
    tau_PD = J.T.dot( np.multiply(kp, e) + np.multiply(kd, de) )
    # control signal: PD control + null space control
    tau = tau_PD + N.dot(tau_0)
    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q_med,dq_med,ddq_med = ur5_robot.read_joint_position_velocity_acceleration()
    p_med, dp_med, ddp_med = ur5_robot.
        read_cartesian_position_velocity_acceleration()

    # publish message
    jstate.header.stamp = rospy.Time.now()
    jstate.name   = jnames   # Joints position name
    jstate.position  = q_med
    jstate.velocity  = dq_med
    pub.publish(jstate)
    # update time
    t = t + dt
    # stop simulation
    if t>=sim_duration:
        print("stopping rviz ...")
        break
    rate.sleep()
```

Algorithm 6: Move the ur5 robot end-effector using the Cartesian proportional-derivative control method and null space projection, (3) to follows the Cartesian sinusoidal reference of activity 1.1.

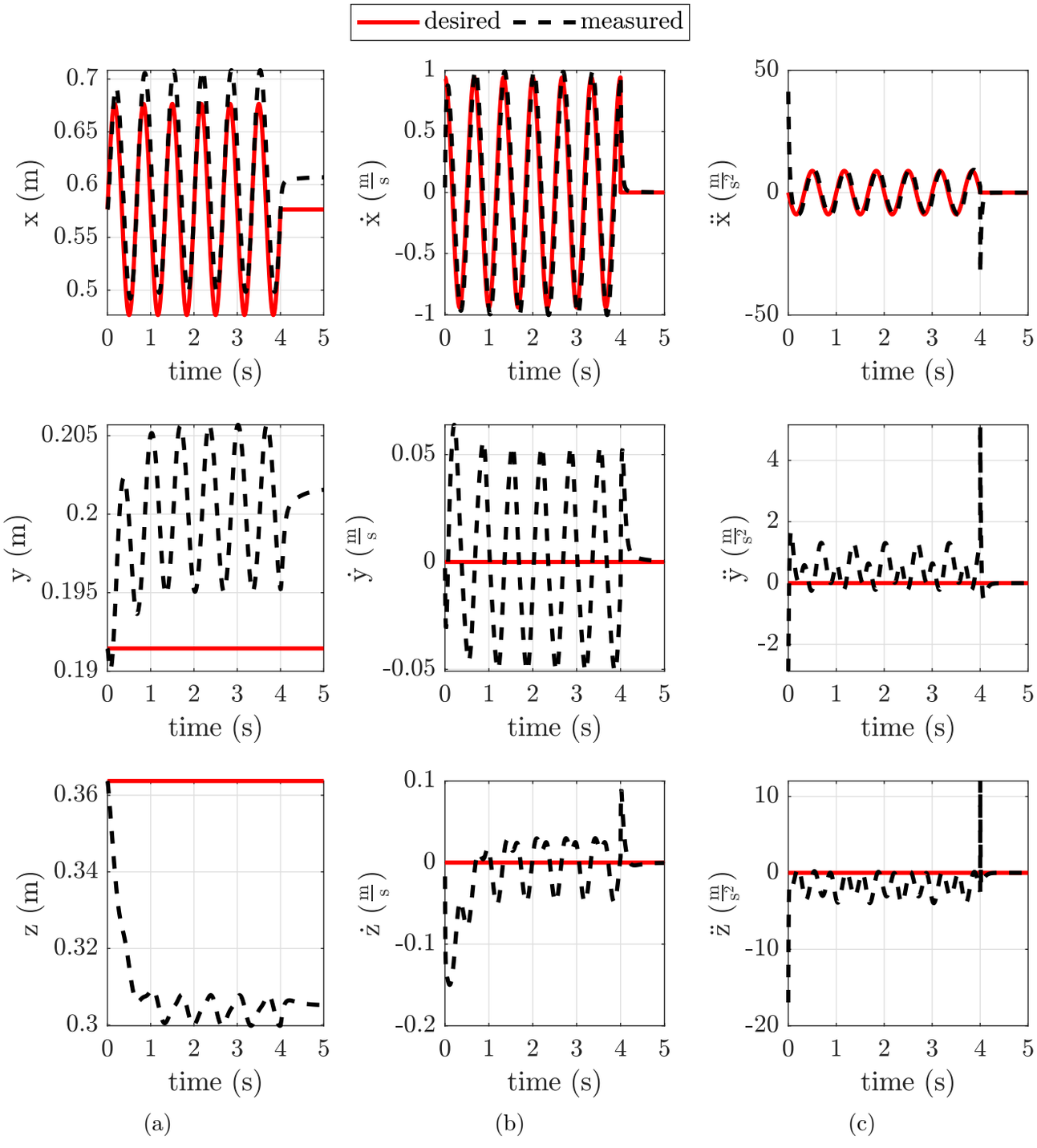


Figure 7: Cartesian trajectory tracking performances using Cartesian proportional-derivative control method and null space projection, (3), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$: (a) position, (b) velocity and (c) acceleration.

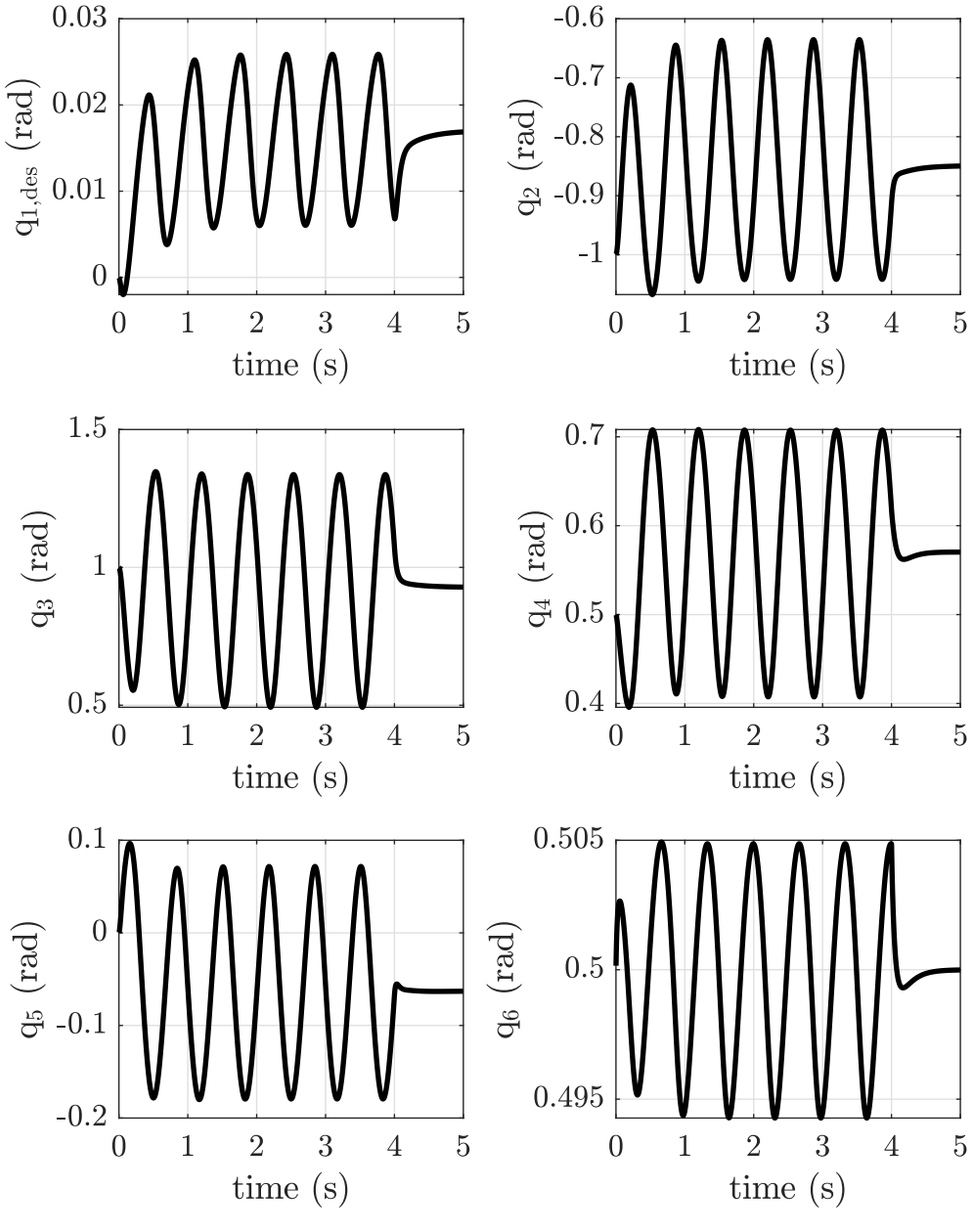


Figure 8: Angular position of each joint of UR5 robot with Algorithm 6.

1.6 Cartesian PD + gravity compensation

The objective of this activity is to control movement of the ur5 robot end-effector so that it follows the Cartesian step reference trajectory of activity 1.2. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad and end-effector $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m. Likewise, the Cartesian step reference trajectory starts at $\mathbf{p_0}$. Motion control is made up of two approaches: Cartesian proportional-derivative with gravity compensation (PD+g) and projection of the null space. In this sense, Cartesian PD+g focuses on reducing end-effector position error and the projection of null space maintains the articular position close to $\mathbf{q_0}$. Finally, control law can be computed as

$$\boldsymbol{\tau} = \mathbf{J^T}(\mathbf{K_p}\mathbf{e} + \mathbf{K_d}\dot{\mathbf{e}} + \mathbf{J^{T\#}}\mathbf{g}) + \mathbf{N}\left(\mathbf{K_q}(\mathbf{q_0} - \mathbf{q}) - \mathbf{D_q}\dot{\mathbf{q}}\right), \tag{4}$$

$$\mathbf{N} = (\mathbf{I_{6\times 6}} - \mathbf{J}^{\#}\mathbf{J}),$$

where $\mathbf{J}$ is jacobian matrix, $\mathbf{e} = \mathbf{p_{des}} - \mathbf{p}$ is end-effector position error, $\mathbf{K_p}, \mathbf{K_d}$ are the proportional and derivative gains respectively, $\mathbf{J}^{\#}$ is jacobian damped pseudo-inverse, $\mathbf{g}$ is gravity compensation term, $\mathbf{N}$ is the null space projection of $\mathbf{J}^{\#}$, and $\mathbf{K_q}, \mathbf{D_q}$ are the proportional and derivative gains for null space projection.

The Algorithm 7 control the movements of ur5 robot end-effector to track the Cartesian step reference trajectory of activity 1.2. In this file, the control law (4) is configured with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$. On one hand, Figure 9 shows that trajectory tracking performance at the Cartesian space is good with mean norm error at each axis $(||e_x||, ||e_y||, ||e_z||)$ of (0.0002, 0.0001, 0.025) cm respectively. The gravity term in control law (4) reduces the position error in z-axis from 0.08 cm (obtained in activity 1.5) to 0.025 cm; likewise, add feed-forward terms on control law (4) will reduce position error in x- and y-axis. On the other hand, Figure 10 shows shows that angular trajectory of each joint with Cartesian PD control method is different from angular trajectory obtained with articular PD control method in activity 1.3. This is because the control law focuses on reducing the Cartesian position error of the end effector rather than position error of each joint. Finally, null space projection allows the smooth variation of the position of each joint, so the system remained stable for the entire simulation unlike activity 1.4.

```
# ==========================
#   Configuration of node
# ==========================
# create a node:
rospy.init_node("cartesian_space_PD_control_postural_task_gravity_compensation")
# public in topic /joint_states to send joint data
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)
```

```
# loop rate (in Hz)
rate  = rospy.Rate(1000)  # 1000 [Hz]
dt    = 1e-3      # 1  [ms]
# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
   wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ================================================
#   set initial cartesian configuration of UR5
```

```
# ================================================
# initial cartesian configuration: position, velocity and acceleration
p0 = ur5_robot.get_ee_position()
dp0 = np.array([0.0, 0.0, 0.0])
ddp0 = np.array([0.0, 0.0, 0.0])
# desired cartesian trajectory: position, velocity and acceleration
p_des = copy(p0)
dp_des = np.array([0.0, 0.0, 0.0])
ddp_des = np.array([0.0, 0.0, 0.0])
# measured cartesian trajectory: position, velocity and acceleration
p_med = copy(p0)
dp_med = np.array([0.0, 0.0, 0.0])
ddp_med = np.array([0.0, 0.0, 0.0])

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = np.array([1000, 1000, 1000])   # N/m
# derivative gain
kd = np.array([300, 300, 300])      # N.s/m
# control vector
tau_PD = np.zeros(ndof)

# postural task: gains
Kq = 50*np.eye(ndof)
Dq = 10*np.eye(ndof)
# postural task: control term
tau_0 = np.zeros(ndof)

#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
step_start = 2.0    # [sec]

while not rospy.is_shutdown():
    # desired cartesian trajectory
    p_des[2], dp_des[2], ddp_des[2] = step_reference_generator(p0[2], 0.1,
        step_start, t)

    # jacobian: position xyz [3x6]
```

```
J = ur5_robot.jacobian(q_des)[0:3, 0:6]
# jacobian: damped pseudo-inverse [6x3]
J_inv = ur5_robot.jacobian_damped_pinv(J)

# error: position and velocity
e  =  p_des - p_med
de  =  dp_des - dp_med

# dynamics: gravity term
g = ur5_robot.get_g()

# postural task: control term
tau_0 = Kq.dot(q0-q_med) - Dq.dot(dq_med)
N = np.eye(ndof) - J_inv.dot(J)
# PD + gravity compensation (cartesian space)
tau_PD = J.T.dot( np.multiply(kp, e) + np.multiply(kd, de) + J_inv.T.dot(g))
# control signal: PD + g + null space projection
tau = tau_PD + N.dot(tau_0)

# send control signal
ur5_robot.send_control_command(tau)
# update states
q_med, dq_med, ddq_med = ur5_robot.read_joint_position_velocity_acceleration
    ()
p_med, dp_med, ddp_med = ur5_robot.
    read_cartesian_position_velocity_acceleration()
# publish message
jstate.header.stamp = rospy.Time.now()
jstate.name   = jnames   # Joints position name
jstate.position  = q_med
jstate.velocity  = dq_med
pub.publish(jstate)
# update time
t = t + dt
# stop simulation
if t>=sim_duration:
    print("stopping rviz ...")
    break
rate.sleep()
```

Algorithm 7: Move the ur5 robot end-effector using the Cartesian proportional-derivative control method with gravity compensation and null space projection, (4) to follows the Cartesian step reference trajectory of activity 1.2.

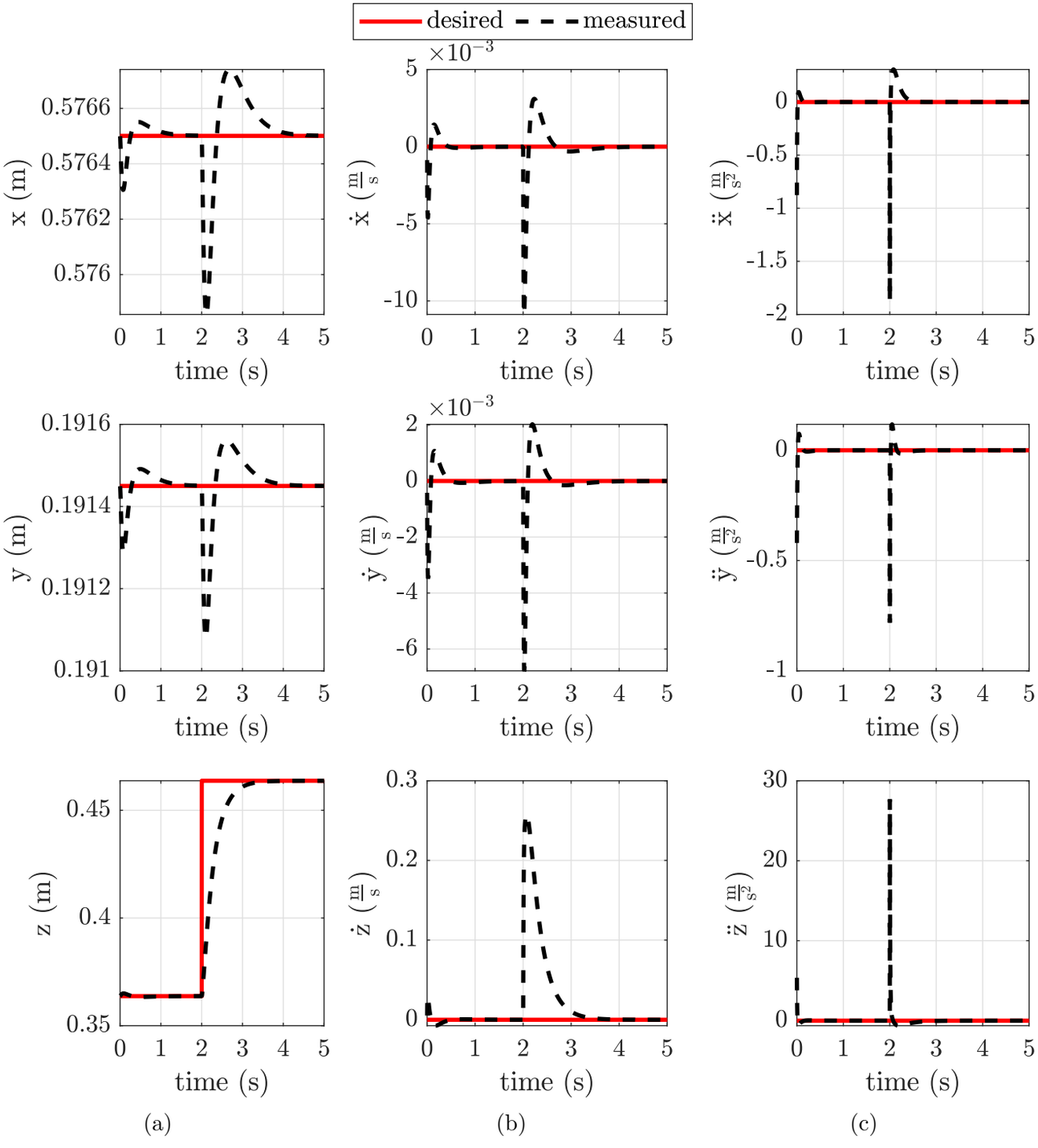


Figure 9: Cartesian trajectory tracking performances using Cartesian proportional-derivative control method with gravity compensation and null space projection, (4), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$: (a) position, (b) velocity and (c) acceleration.

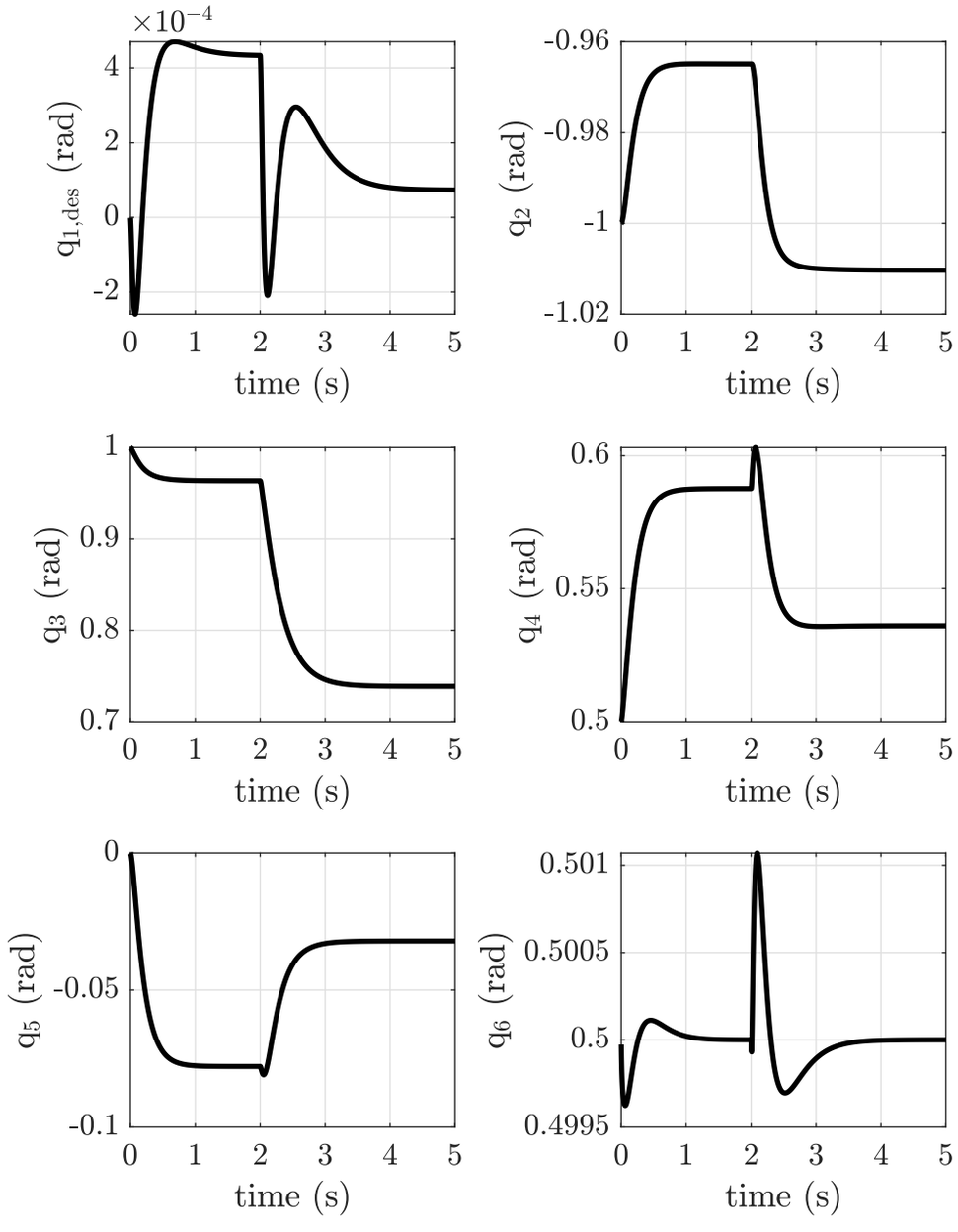


Figure 10: Angular position of each joint of UR5 robot with Algorithm 7.

1.7 Cartesian PD control + gravity compensation + feed-forward term

The objective of this activity is to control movement of the ur5 robot end-effector so that it follows the Cartesian sinusoidal reference trajectory of activity 1.1. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad and end-effector $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m. Likewise, the Cartesian sinusoidal reference trajectory starts at $\mathbf{p_0}$. Motion control is made up of two approaches: Cartesian proportional-derivative with gravity compensation, feed-forward term (PD+g+ff) and projection of the null space. In this sense, Cartesian PD+g+ff focuses on reducing end-effector position error and the projection of null space maintains the articular position close to $\mathbf{q_0}$. Finally, control law can be computed as

$$\boldsymbol{\tau} = \mathbf{J^T}(\boldsymbol{\Lambda}\ddot{\mathbf{p}}_{\mathbf{des}} + \mathbf{K_p e} + \mathbf{K_d}\dot{\mathbf{e}} + \mathbf{J^{T\#}g}) + \mathbf{N}\left(\mathbf{K_q}(\mathbf{q_0} - \mathbf{q}) - \mathbf{D_q}\dot{\mathbf{q}}\right), \tag{5}$$

$$\mathbf{N} = (\mathbf{I_{6\times 6}} - \mathbf{J^{\#}J}),$$

$$\boldsymbol{\Lambda} = (\mathbf{J M^{-1} J^T})^{-1},$$

where $\mathbf{J}$ is jacobian matrix, $\boldsymbol{\Lambda}$ is inertia matrix at Cartesian space, $\mathbf{e} = \mathbf{p_{des}} - \mathbf{p}$ is end-effector position error, $\mathbf{K_p}, \mathbf{K_d}$ are the proportional and derivative gains respectively, $\mathbf{J}^{\#}$ is jacobian damped pseudo-inverse, $\mathbf{g}$ is gravity compensation term, $\mathbf{N}$ is the null space projection of $\mathbf{J}^{\#}$, and $\mathbf{K_q}, \mathbf{D_q}$ are the proportional and derivative gains for null space projection.

The Algorithm 8 control the movements of ur5 robot end-effector to track the Cartesian sinusoidal reference trajectory of activity 1.1. In this file, the control law (5) is configured with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$. On one hand, Figure 11 shows that trajectory tracking performance at the Cartesian space is good with mean norm error at each axis ($||e_x||, ||e_y||, ||e_z||$) of (0.0087, 0.0016, 0.036) cm respectively. The gravity term in control law (5) reduces position error in z-axis from 0.08 cm (obtained in activity 1.5) to 0.036 cm. Likewise, feed-forward term in control law (5) reduces position error in x- and y-axis from 0.045, 0.013 cm (obtained in activity 1.5) to 0.0087, 0.0016 cm respectively. On the other hand, Figure 12 shows shows that angular trajectory of each joint with Cartesian PD control method is different from angular trajectory obtained with articular PD control method in activity 1.3. This is because the control law focuses on reducing the Cartesian position error of the end effector rather than position error of each joint. Finally, null space projection allows the smooth variation of the position of each joint, so the system remained stable for the entire simulation unlike activity 1.4.

```
# =========================
#   Configuration of node
# =========================
```

```
# create a node:
rospy.init_node("
    cartesian_space_PD_control_postural_task_gravity_compensation_feedforward_term
    ")
# public in topic /joint_states to send joint data
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)
# loop rate (in Hz)
rate  = rospy.Rate(1000)  # 1000 [Hz]
dt   = 1e-3     # 1  [ms]
# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
    wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof
```

```
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ================================================
#   set initial cartesian configuration of UR5
# ================================================
# initial cartesian configuration: position, velocity and acceleration
p0 = ur5_robot.get_ee_position()
dp0 = np.array([0.0, 0.0, 0.0])
ddp0 = np.array([0.0, 0.0, 0.0])

# desired cartesian trajectory: position, velocity and acceleration
p_des = copy(p0)
dp_des = np.array([0.0, 0.0, 0.0])
ddp_des = np.array([0.0, 0.0, 0.0])

# measured cartesian trajectory: position, velocity and acceleration
p_med = copy(p0)
dp_med = np.array([0.0, 0.0, 0.0])
ddp_med = np.array([0.0, 0.0, 0.0])

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = np.array([1000, 1000, 1000])   # N/m
# derivative gain
kd = np.array([300, 300, 300])      # N.s/m
# control vector
tau_PD = np.zeros(ndof)

# postural task: gains
Kq = 50*np.eye(ndof)
Dq = 10*np.eye(ndof)
# postural task: control term
tau_0 = np.zeros(ndof)

#===============
#   Simulation
```

```
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0    # [sec]

while not rospy.is_shutdown():
    # desired cartesian trajectory
    p_des[0], dp_des[0], ddp_des[0] = sinusoidal_reference_generator(p0[0], 0.1,
        1.5, sine_duration, t)

    # jacobian: position xyz [3x6]
    J = ur5_robot.jacobian(q_des)[0:3, 0:6]
    # jacobian: damped pseudo-inverse [6x3]
    J_inv = ur5_robot.jacobian_damped_pinv(J)

    # error: position and velocity
    e  =  p_des - p_med
    de  =  dp_des - dp_med

    # dynamics: inertia matrix
    M = ur5_robot.get_M()

    # dynamics: gravity term
    g = ur5_robot.get_g()

    # postural task: control term
    tau_0 = Kq.dot(q0-q_med) - Dq.dot(dq_med)
    N = np.eye(ndof) - J_inv.dot(J)

    # control signal: feedforward term
    M_x = np.linalg.inv(J.dot(np.linalg.inv(M).dot(J.T)))
    tau_ff = M_x.dot(ddp_des)
    # control signal: ff + PD + gravity compensation (cartesian space)
    tau_PD = J.T.dot( tau_ff + np.multiply(kp, e) + np.multiply(kd, de) + J_inv.T
        .dot(g))
    # control signal: PD + g + null space projection
    tau = tau_PD + N.dot(tau_0)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q_med, dq_med, ddq_med = ur5_robot.read_joint_position_velocity_acceleration
        ()
```

```
    p_med, dp_med, ddp_med = ur5_robot.
        read_cartesian_position_velocity_acceleration()

    # publish message
    jstate.header.stamp = rospy.Time.now()
    jstate.name   = jnames   # Joints position name
    jstate.position  = q_med
    jstate.velocity  = dq_med
    pub.publish(jstate)

    # update time
    t = t + dt

    # stop simulation
    if t>=sim_duration:
        print("stopping rviz ...")
        break
    rate.sleep()
```

Algorithm 8: Move the ur5 robot end-effector using the Cartesian proportional-derivative control method with gravity compensation, feed-forward term and null space projection, (5) to follows the Cartesian sinusoidal reference of activity 1.1.

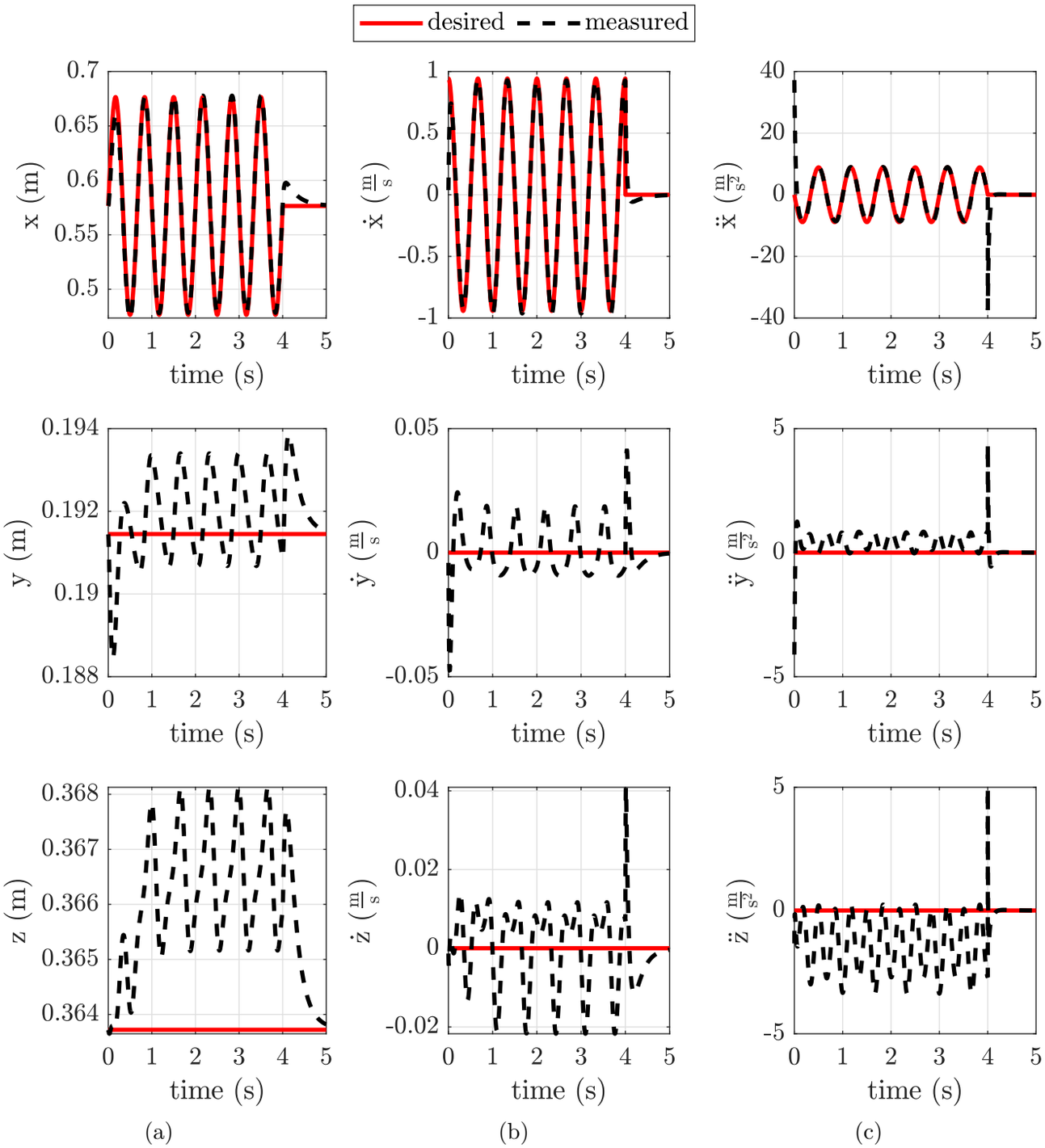


Figure 11: Cartesian trajectory tracking performances using Cartesian proportional-derivative control method with gravity compensation, feed-forward term and null space projection, (5), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$: (a) position, (b) velocity and (c) acceleration.

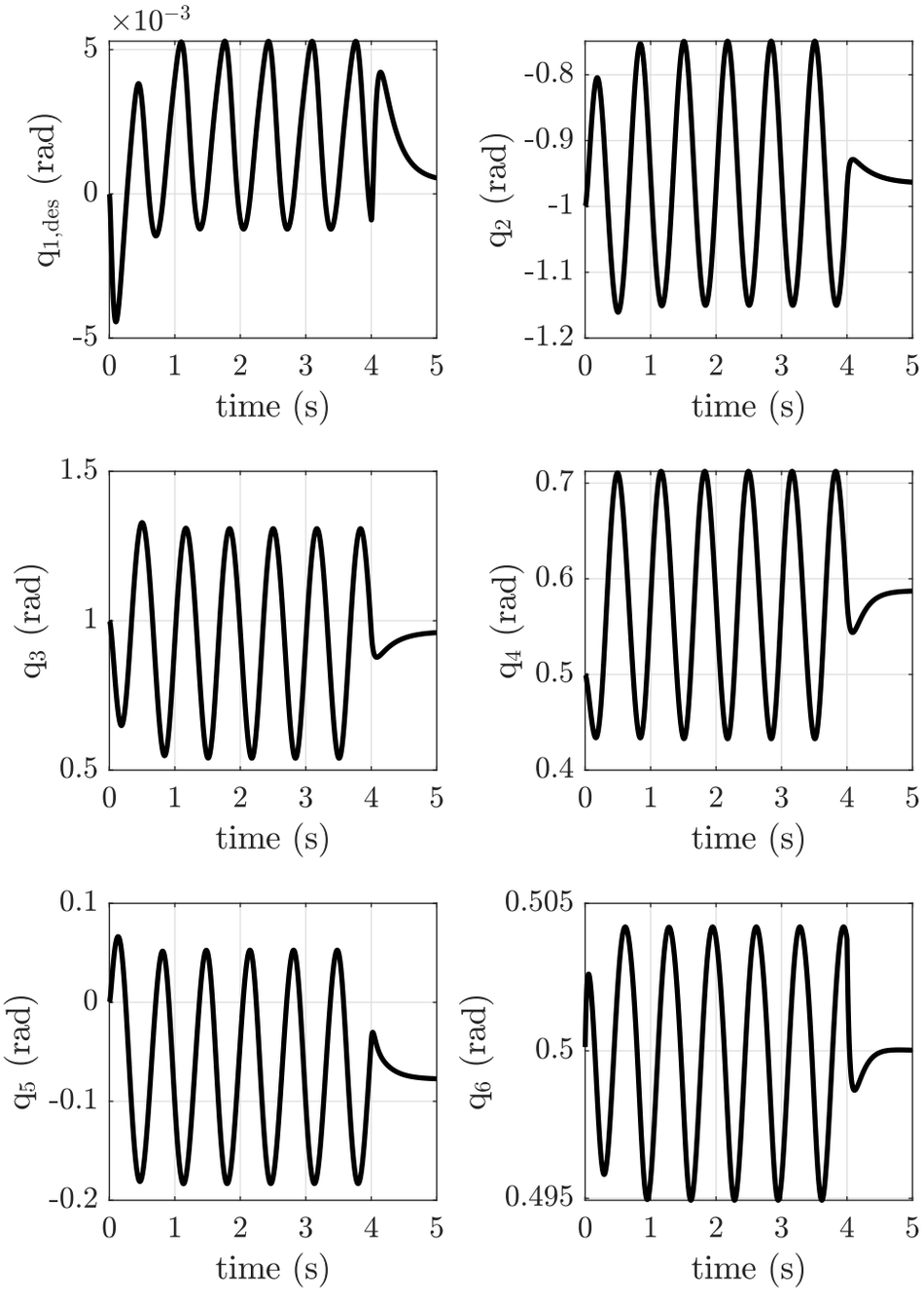


Figure 12: Angular position of each joint of UR5 robot with Algorithm 8.

2 Centralized task space control

2.1 Cartesian space inverse dynamics

The objective of this activity is to control movement of the ur5 robot end-effector so that it follows the Cartesian sinusoidal reference trajectory of activity 1.1. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad and end-effector $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m. Likewise, the Cartesian sinusoidal reference trajectory starts at $\mathbf{p_0}$. Motion control is made up of two approaches: Cartesian inverse dynamics and projection of the null space. In this sense, Cartesian inverse dynamics focuses on reducing end-effector position error and the projection of null space maintains the articular position close to $\mathbf{q_0}$. Finally, control law can be computed as

$$\boldsymbol{\tau} = \mathbf{J^T}(\boldsymbol{\Lambda}(\ddot{\mathbf{p}}_{\mathbf{des}} + \mathbf{K_p e} + \mathbf{K_d}\dot{\mathbf{e}}) + \boldsymbol{\mu}) + \mathbf{N}\left(\mathbf{K_q}(\mathbf{q_0} - \mathbf{q}) - \mathbf{D_q}\dot{\mathbf{q}}\right), \tag{6}$$

$$\begin{aligned} \boldsymbol{\Lambda} &= (\mathbf{J}\mathbf{M}^{-\mathbf{1}}\mathbf{J^T})^{-1}, \\ \boldsymbol{\mu} &= \mathbf{J}^{\mathbf{T}\#} - \boldsymbol{\Lambda}\dot{\mathbf{J}}\dot{\mathbf{q}}, \\ \mathbf{N} &= (\mathbf{I_{6\times 6}} - \mathbf{J}^{\#}\mathbf{J}), \end{aligned}$$

where $\mathbf{J}$ is jacobian matrix, $\boldsymbol{\Lambda}$ is inertia matrix at Cartesian space, $\mathbf{e} = \mathbf{p_{des}} - \mathbf{p}$ is end-effector position error, $\mathbf{K_p}, \mathbf{K_d}$ are the proportional and derivative gains respectively, $\mathbf{J}^{\#}$ is jacobian damped pseudo-inverse, $\boldsymbol{\mu}$ is nonlinear effects vector at Cartesian space, $\mathbf{N}$ is the null space projection of $\mathbf{J}^{\#}$, and $\mathbf{K_q}, \mathbf{D_q}$ are the proportional and derivative gains for null space projection.

The Algorithm 9 control the movements of ur5 robot end-effector to track the Cartesian sinusoidal reference trajectory of activity 1.1. In this file, the control law (6) is configured with $K_p = 1000$ $\frac{\text{N}}{\text{m}}$, $K_d = 300$ $\frac{\text{N.s}}{\text{m}}$, $K_q = 50$ $\frac{\text{N.m}}{\text{rad}}$ and $K_d = 10$ $\frac{\text{N.m.s}}{\text{rad}}$. On one hand, Figure 13 shows that trajectory tracking performance at the Cartesian space is excellent with mean norm error at each axis $(||e_x||, ||e_y||, ||e_z||)$ of (0.0019, 0.0005, 0.0004) cm respectively. Hence, ur5 robot with control law 6 has the best trajectory tracking performance compared with previous activities. On the other hand, Figure 14 shows angular trajectory of each joint of the ur5 robot. In this figure, the first and fifth joints present small peaks that were not observed in the previous activities. This may indicate that performing linear trajectories in Cartesian space requires small discontinuities in joint space. Finally, null space projection allows the smooth variation of the position of each joint, so the system remained stable for the entire simulation unlike activity 1.4.

```
# =========================
#   Configuration of node
```

```
# ========================
# create a node:
rospy.init_node("cartesian_space_inverse_dynamics")
# public in topic /joint_states to send joint data
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)
# loop rate (in Hz)
rate  = rospy.Rate(1000)  # 1000 [Hz]
dt   = 1e-3     # 1  [ms]
# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
   wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof

# create inertia matrix
```

```
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ================================================
#   set initial cartesian configuration of UR5
# ================================================
# initial cartesian configuration: position, velocity and acceleration
p0 = ur5_robot.get_ee_position()
dp0 = np.array([0.0, 0.0, 0.0])
ddp0 = np.array([0.0, 0.0, 0.0])

# desired cartesian trajectory: position, velocity and acceleration
p_des = copy(p0)
dp_des = np.array([0.0, 0.0, 0.0])
ddp_des = np.array([0.0, 0.0, 0.0])

# measured cartesian trajectory: position, velocity and acceleration
p_med = copy(p0)
dp_med = np.array([0.0, 0.0, 0.0])
ddp_med = np.array([0.0, 0.0, 0.0])

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = np.array([1000, 1000, 1000])   # N/m
# derivative gain
kd = np.array([300, 300, 300])      # N.s/m
# control vector
tau_PD = np.zeros(ndof)

# postural task: gains
Kq = 50*np.eye(ndof)
Dq = 10*np.eye(ndof)
# postural task: control term
tau_0 = np.zeros(ndof)

#===============
#   Simulation
#===============
```

```
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0    # [sec]

while not rospy.is_shutdown():
    # desired cartesian trajectory
    p_des[0], dp_des[0], ddp_des[0] = sinusoidal_reference_generator(p0[0], 0.1,
        1.5, sine_duration, t)

    # jacobian: position xyz [3x6]
    J = ur5_robot.jacobian(q_med)[0:3, 0:6]
    # jacobian: damped pseudo-inverse [6x3]
    J_inv = ur5_robot.jacobian_damped_pinv(J)
    # jacobian: time-derivative [3x6]
    dJ = ur5_robot.jacobian_time_derivative(q_med, dq_med)[0:3, 0:6]

    # error: position and velocity
    e  =  p_des - p_med
    de  =  dp_des - dp_med

    # dynamics: inertia matrix
    M = ur5_robot.get_M()
    M_x = np.linalg.inv(J.dot(np.linalg.inv(M).dot(J.T)))
    # dynamics: nonlinear effects vector
    b = ur5_robot.get_b()
    b_x = J_inv.T.dot(b) - M_x.dot(dJ.dot(dq_med))

    # control signal: null space projection
    tau_0 = Kq.dot(q0-q_med) - Dq.dot(dq_med)
    N = np.eye(ndof) - J_inv.dot(J)
    # control signal: Cartesian inverse_dyanmics
    F_d = ddp_des + np.multiply(kp, e) + np.multiply(kd, de)
    tau_PD = J.T.dot( M_x.dot(F_d) + b_x )
    # control signal: Cartesian inverse_dyanmics + null space projection
    tau = tau_PD + N.dot(tau_0)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q_med, dq_med, ddq_med = ur5_robot.read_joint_position_velocity_acceleration
        ()
    p_med, dp_med, ddp_med = ur5_robot.
        read_cartesian_position_velocity_acceleration()
```

```
    # publish message
    jstate.header.stamp = rospy.Time.now()
    jstate.name   = jnames   # Joints position name
    jstate.position  = q_med
    jstate.velocity  = dq_med
    pub.publish(jstate)

    # update time
    t = t + dt

    # stop simulation
    if t>=sim_duration:
        print("stopping rviz ...")
        break
    rate.sleep()
```

Algorithm 9: Move the ur5 robot end-effector using the Cartesian space inverse dynamics and null space projection, (6) to follows the Cartesian sinusoidal reference of activity 1.1.

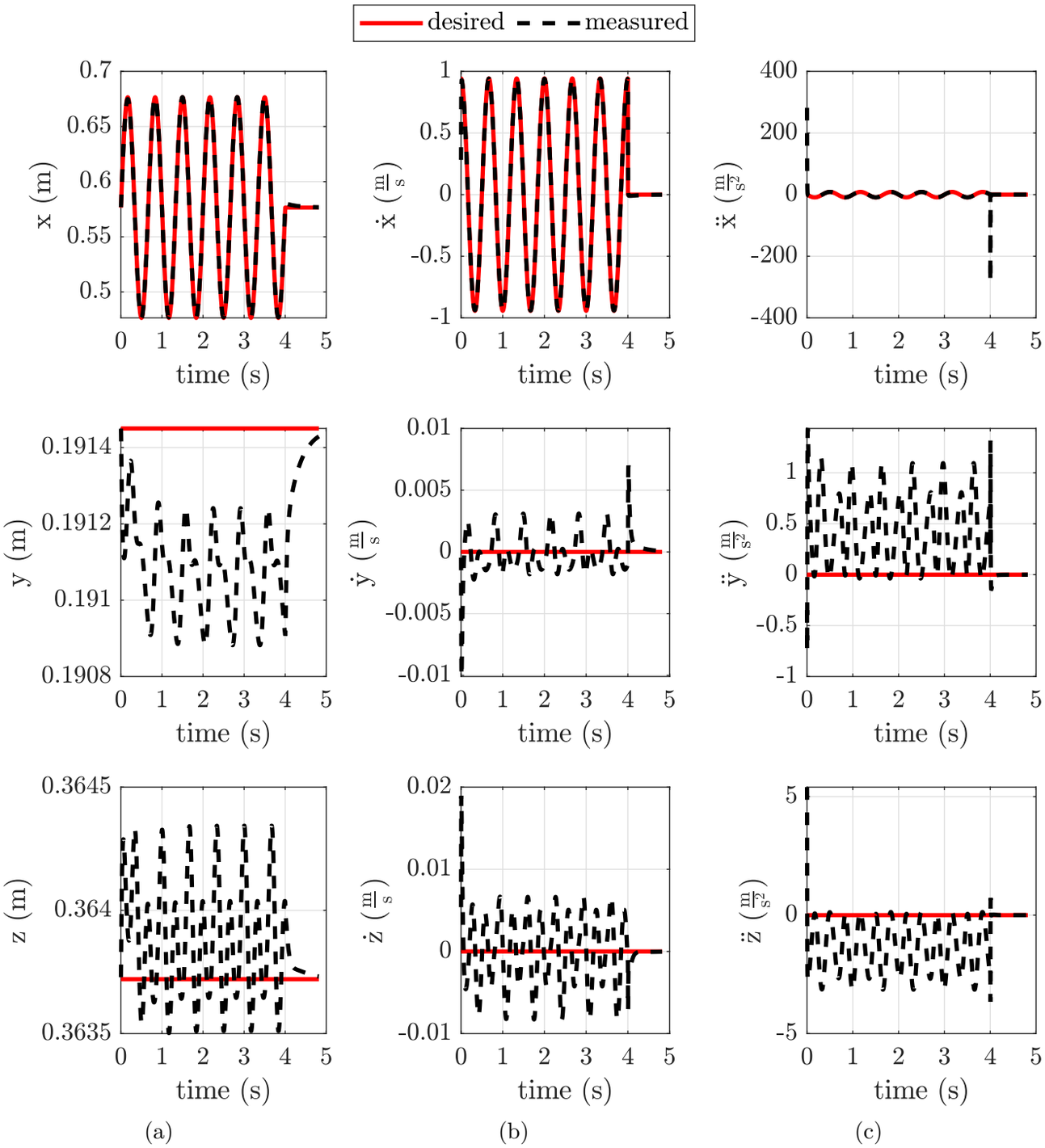


Figure 13: Cartesian trajectory tracking performances using Cartesian space inverse dynamics and null space projection, (6), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$: (a) position, (b) velocity and (c) acceleration.

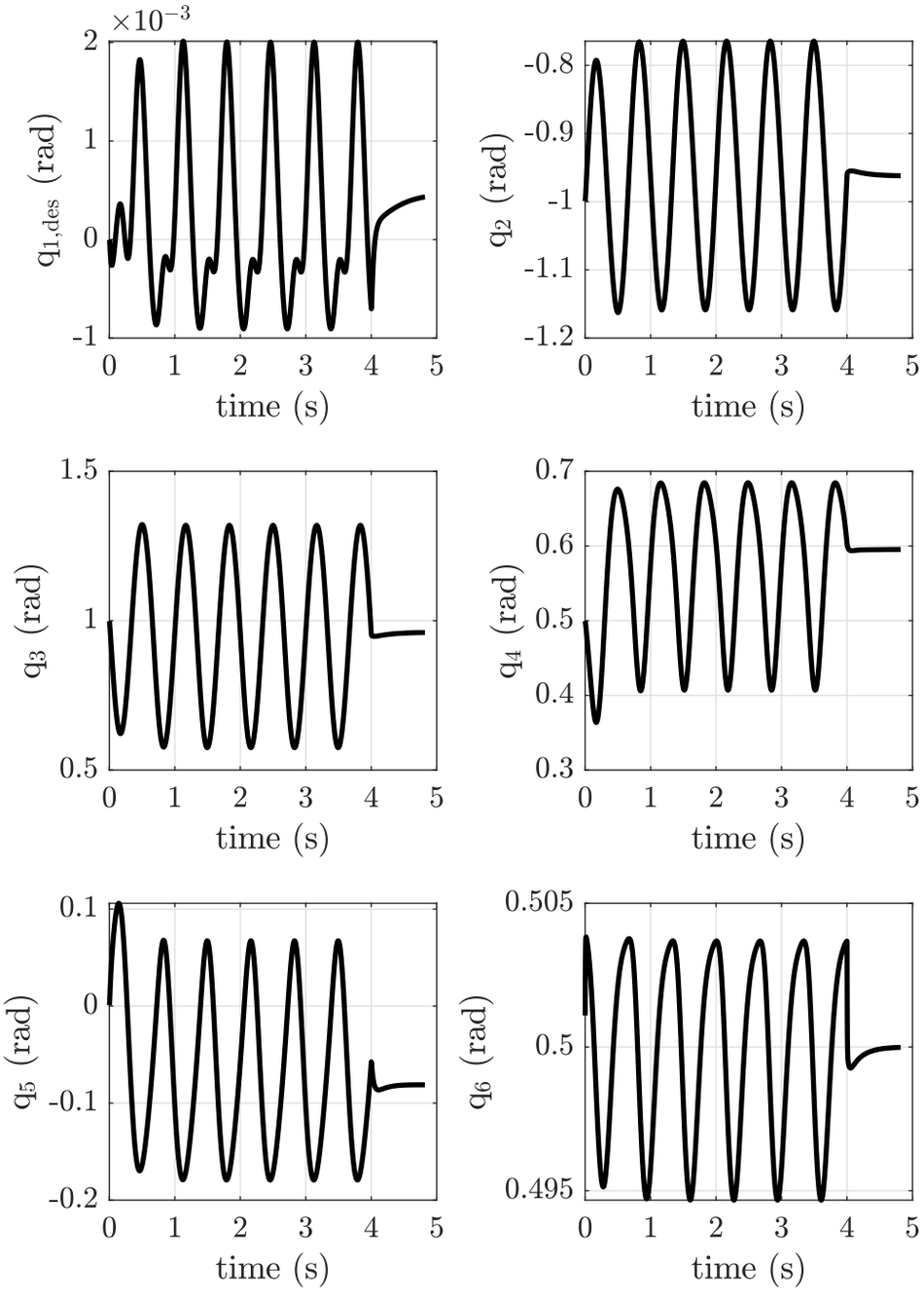


Figure 14: Angular position of each joint of UR5 robot with Algorithm 9.

2.2 Cartesian space inverse dynamics - simplified

The objective of this activity is to control movement of the ur5 robot end-effector so that it follows the Cartesian sinusoidal reference trajectory of activity 1.1. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad and end-effector $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m. Likewise, the Cartesian sinusoidal reference trajectory starts at $\mathbf{p_0}$. Motion control is made up of two approaches: Cartesian inverse dynamics (simplified) and projection of the null space. In this sense, Cartesian inverse dynamics focuses on reducing end-effector position error and the projection of null space maintains the articular position close to $\mathbf{q_0}$. Finally, control law can be computed as

$$\boldsymbol{\tau} = \mathbf{J^T}(\boldsymbol{\Lambda}(\ddot{\mathbf{p}}_{\mathbf{des}} + \mathbf{K_p e} + \mathbf{K_d}\dot{\mathbf{e}})) + \mathbf{b} + \mathbf{N}\left(\mathbf{K_q}(\mathbf{q_0} - \mathbf{q}) - \mathbf{D_q}\dot{\mathbf{q}}\right), \tag{7}$$

$$\boldsymbol{\Lambda} = (\mathbf{J}\mathbf{M}^{-\mathbf{1}}\mathbf{J}^{\mathbf{T}})^{-1},$$
$$\mathbf{N} = (\mathbf{I_{6\times 6}} - \mathbf{J}^{\#}\mathbf{J}),$$

where $\mathbf{J}$ is jacobian matrix, $\boldsymbol{\Lambda}$ is inertia matrix at Cartesian space, $\mathbf{e} = \mathbf{p_{des}} - \mathbf{p}$ is end-effector position error, $\mathbf{K_p}, \mathbf{K_d}$ are the proportional and derivative gains respectively, $\mathbf{J}^{\#}$ is jacobian damped pseudo-inverse, $\mathbf{b}$ is nonlinear effects vector at joint space, $\mathbf{N}$ is the null space projection of $\mathbf{J}^{\#}$, and $\mathbf{K_q}, \mathbf{D_q}$ are the proportional and derivative gains for null space projection.

The Algorithm 10 control the movements of ur5 robot end-effector to track the Cartesian sinusoidal reference trajectory of activity 1.1. In this file, the control law (6) is configured with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$. On one hand, Figure 15 shows that trajectory tracking performance at the Cartesian space is good with mean norm error at each axis $(||e_x||, ||e_y||, ||e_z||)$ of (0.0018, 0.0002, 0.001) cm respectively. Hence, ur5 robot with control law 7 has more position error on the z-axis than activity 2.1. On the other hand, Figure 16 shows angular trajectory of each joint of the ur5 robot. In this figure, the first and fifth joints present small peaks that were not observed in the previous activities. This may indicate that performing linear trajectories in Cartesian space requires small discontinuities in joint space. Finally, null space projection allows the smooth variation of the position of each joint, so the system remained stable for the entire simulation unlike activity 1.4.

```
# ==========================
#   Configuration of node
# ==========================
# create a node:
rospy.init_node("cartesian_space_inverse_dynamics_simplified")
```

```
# public in topic /joint_states to send joint data
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)
# loop rate (in Hz)
rate  = rospy.Rate(1000)  # 1000 [Hz]
dt   = 1e-3     # 1  [ms]
# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
   wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof

# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
```

```
# create gravity vector
g = np.zeros(ndof)

# ================================================
#   set initial cartesian configuration of UR5
# ================================================
# initial cartesian configuration: position, velocity and acceleration
p0 = ur5_robot.get_ee_position()
dp0 = np.array([0.0, 0.0, 0.0])
ddp0 = np.array([0.0, 0.0, 0.0])

# desired cartesian trajectory: position, velocity and acceleration
p_des = copy(p0)
dp_des = np.array([0.0, 0.0, 0.0])
ddp_des = np.array([0.0, 0.0, 0.0])

# measured cartesian trajectory: position, velocity and acceleration
p_med = copy(p0)
dp_med = np.array([0.0, 0.0, 0.0])
ddp_med = np.array([0.0, 0.0, 0.0])

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = np.array([1000, 1000, 1000])   # N/m
# derivative gain
kd = np.array([300, 300, 300])      # N.s/m
# control vector
tau_PD = np.zeros(ndof)

# postural task: gains
Kq = 50*np.eye(ndof)
Dq = 10*np.eye(ndof)
# postural task: control term
tau_0 = np.zeros(ndof)

#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0    # [sec]
```

```
while not rospy.is_shutdown():
    # desired cartesian trajectory
    p_des[0], dp_des[0], ddp_des[0] = sinusoidal_reference_generator(p0[0], 0.1,
        1.5, sine_duration, t)

    # jacobian: position xyz [3x6]
    J = ur5_robot.jacobian(q_med)[0:3, 0:6]
    # jacobian: damped pseudo-inverse [6x3]
    J_inv = ur5_robot.jacobian_damped_pinv(J)
    # jacobian: time-derivative [3x6]
    dJ = ur5_robot.jacobian_time_derivative(q_med, dq_med)[0:3, 0:6]

    # error: position and velocity
    e  =  p_des - p_med
    de  =  dp_des - dp_med

    # dynamics: inertia matrix
    M = ur5_robot.get_M()
    M_x = np.linalg.inv(J.dot(np.linalg.inv(M).dot(J.T)))
    # dynamics: nonlinear effects vector
    b = ur5_robot.get_b()

    # control signal: null space projection
    tau_0 = Kq.dot(q0-q_med) - Dq.dot(dq_med)
    N = np.eye(ndof) - J_inv.dot(J)
    # control signal: Cartesian inverse_dyanmics
    F_d = ddp_des + np.multiply(kp, e) + np.multiply(kd, de)
    tau_PD = J.T.dot( M_x.dot(F_d) ) + b
    # control signal: Cartesian inverse_dyanmics + null space projection
    tau = tau_PD + N.dot(tau_0)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q_med, dq_med, ddq_med = ur5_robot.read_joint_position_velocity_acceleration
        ()
    p_med, dp_med, ddp_med = ur5_robot.
        read_cartesian_position_velocity_acceleration()

    # publish message
    jstate.header.stamp = rospy.Time.now()
    jstate.name   = jnames   # Joints position name
```

```
    jstate.position  = q_med
    jstate.velocity  = dq_med
    pub.publish(jstate)

    # update time
    t = t + dt

    # stop simulation
    if t>=sim_duration:
        print("stopping rviz ...")
        break
    rate.sleep()
```

Algorithm 10: Move the ur5 robot end-effector using the Cartesian space inverse dynamics and null space projection, (7) to follows the Cartesian sinusoidal reference of activity 1.1.

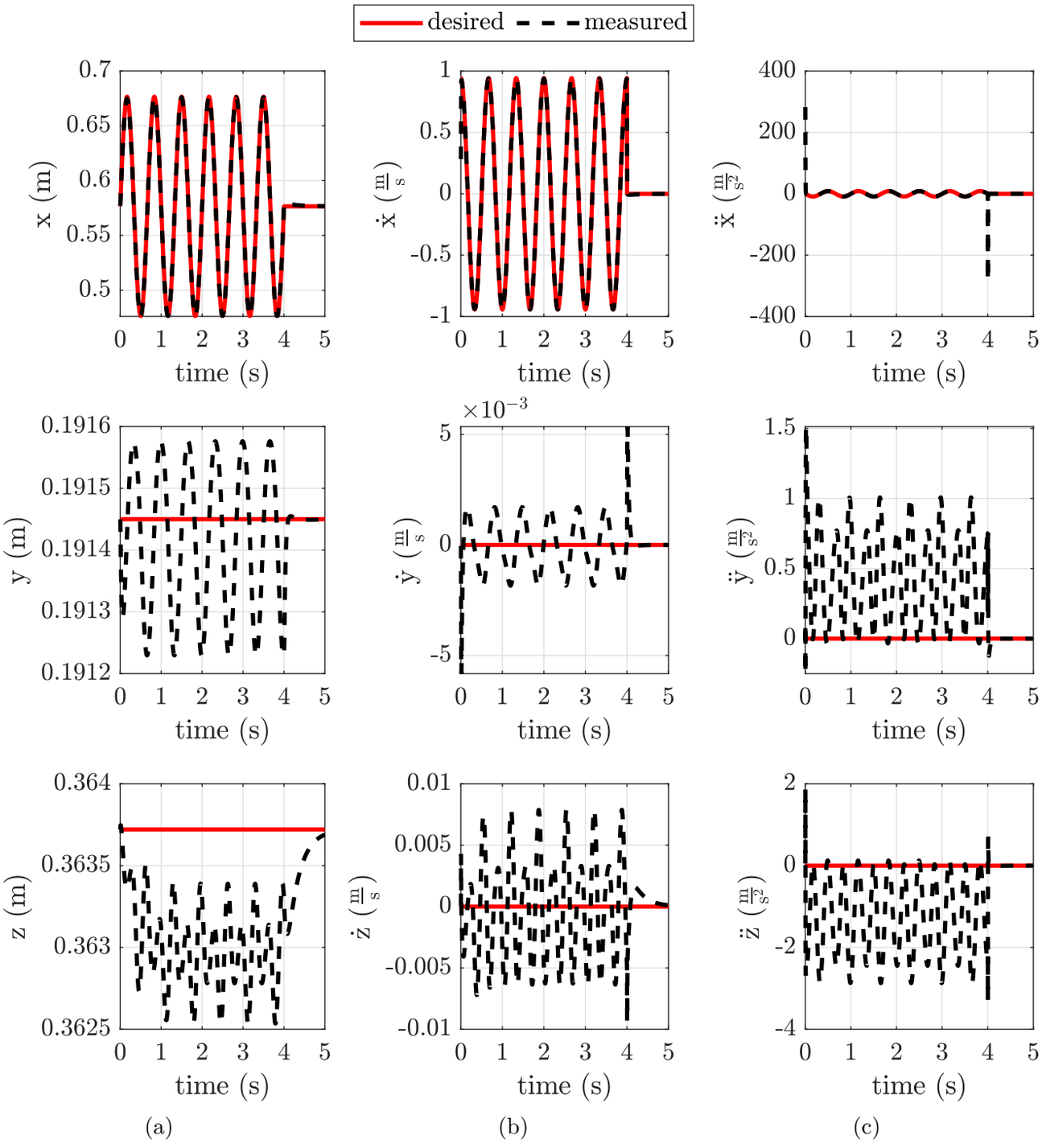


Figure 15: Cartesian trajectory tracking performances using Cartesian space inverse dynamics and null space projection, (7), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$: (a) position, (b) velocity and (c) acceleration.

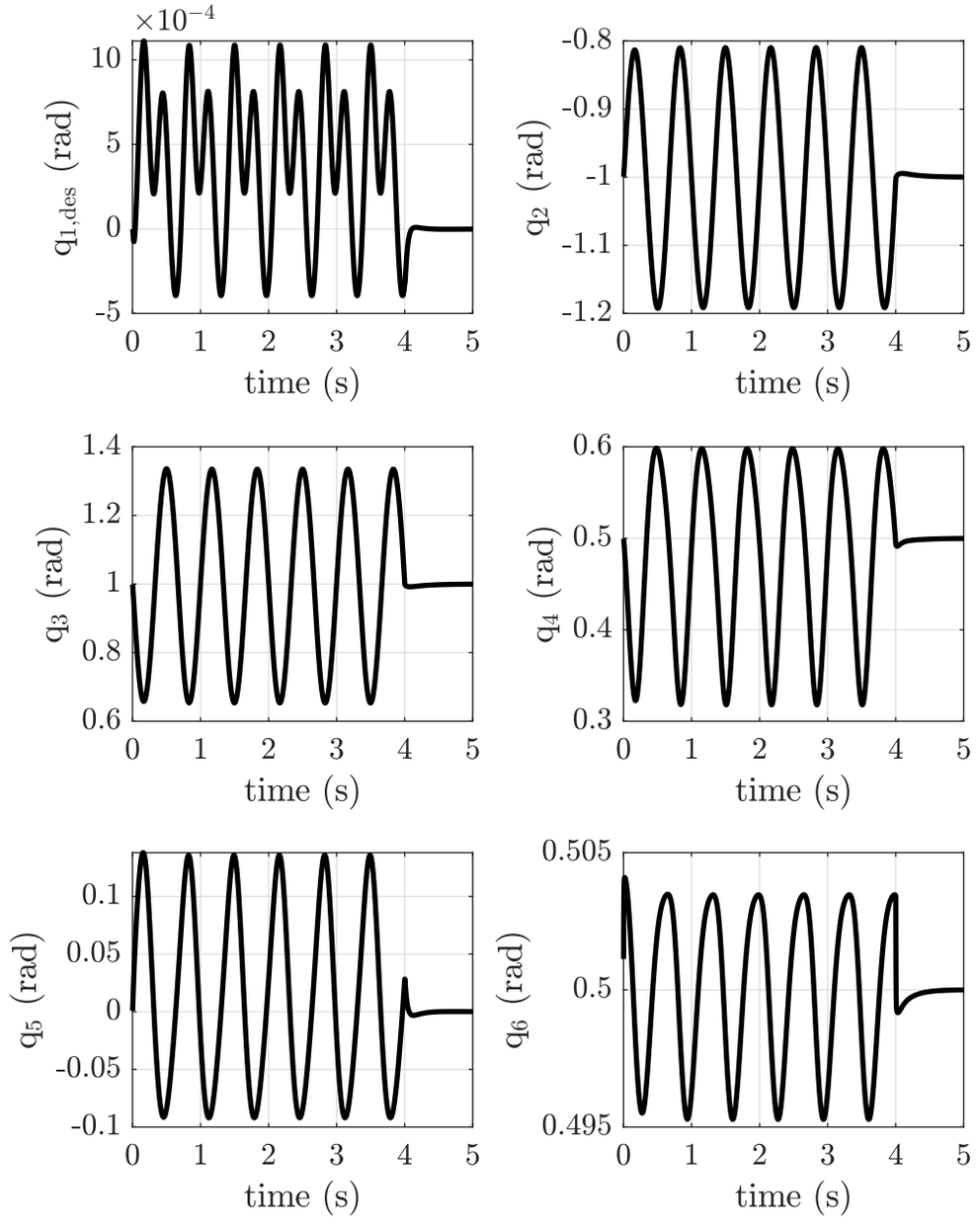


Figure 16: Angular position of each joint of UR5 robot with Algorithm 10.

2.3 Cartesian space inverse dynamics - external force

The objective of this activity is to control movement of the ur5 robot end-effector so that it follows the Cartesian sinusoidal reference trajectory of activity 1.1. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad and end-effector $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m. Likewise, the Cartesian sinusoidal reference trajectory starts at $\mathbf{p_0}$. Then, external force $\mathbf{f_{ext}} = \begin{bmatrix} 0.0 & 0.0 & 200.0 \end{bmatrix}$ N is applied on end-effector after 1 second of simulation. Motion control is made up of two approaches: Cartesian inverse dynamics and projection of the null space. In this sense, Cartesian inverse dynamics focuses on reducing end-effector position error and the projection of null space maintains the articular position close to $\mathbf{q_0}$. Finally, control law can be computed as

$$\boldsymbol{\tau} = \mathbf{J^T}(\boldsymbol{\Lambda}(\ddot{\mathbf{p}}_{\mathbf{des}} + \mathbf{K_p e} + \mathbf{K_d}\dot{\mathbf{e}}) + \boldsymbol{\mu}) + \mathbf{N}\left(\mathbf{K_q}(\mathbf{q_0} - \mathbf{q}) - \mathbf{D_q}\dot{\mathbf{q}}\right) + \mathbf{J^T f_{ext}}, \tag{8}$$

$$\begin{aligned} \boldsymbol{\Lambda} &= (\mathbf{J}\mathbf{M}^{-\mathbf{1}}\mathbf{J^T})^{-1}, \\ \boldsymbol{\mu} &= \mathbf{J}^{\mathbf{T}\#} - \boldsymbol{\Lambda}\dot{\mathbf{J}}\dot{\mathbf{q}}, \\ \mathbf{N} &= (\mathbf{I_{6\times 6}} - \mathbf{J}^{\#}\mathbf{J}), \end{aligned}$$

where $\mathbf{J}$ is jacobian matrix, $\boldsymbol{\Lambda}$ is inertia matrix at Cartesian space, $\mathbf{e} = \mathbf{p_{des}} - \mathbf{p}$ is end-effector position error, $\mathbf{K_p}, \mathbf{K_d}$ are the proportional and derivative gains respectively, $\mathbf{J}^{\#}$ is jacobian damped pseudo-inverse, $\boldsymbol{\mu}$ is nonlinear effects vector at Cartesian space, $\mathbf{N}$ is the null space projection of $\mathbf{J}^{\#}$, $\mathbf{K_q}, \mathbf{D_q}$ are the proportional and derivative gains for null space projection and $\mathbf{f_{ext}}$ is external force.

The Algorithm 11 control the movements of ur5 robot end-effector to track the Cartesian sinusoidal reference trajectory of activity 1.1. In this file, the control law (8) is configured with $K_p = 1000$ $\frac{\text{N}}{\text{m}}$, $K_d = 300$ $\frac{\text{N.s}}{\text{m}}$, $K_q = 50$ $\frac{\text{N.m}}{\text{rad}}$ and $K_d = 10$ $\frac{\text{N.m.s}}{\text{rad}}$. On one hand, Figure 17 shows that trajectory tracking performance at the Cartesian space. In this figure, end-effector position error is close to 0 cm until external force is applied; after that, system presents high position error in z-axis. Then, mean norm error at each axis $(||e_x||, ||e_y||, ||e_z||)$ is (0.0098, 0.0024, 0.0646) cm respectively. On the other hand, Figure 18 shows angular trajectory of each joint of the ur5 robot. Finally, null space projection allows the smooth variation of the position of each joint, so the system remained stable for the entire simulation unlike activity 1.4.

```
# =========================
#   Configuration of node
# =========================
# create a node:
rospy.init_node("cartesian_space_inverse_dynamics")
```

```
# public in topic /joint_states to send joint data
pub = rospy.Publisher('joint_states', JointState, queue_size=1000)
# loop rate (in Hz)
rate  = rospy.Rate(1000)  # 1000 [Hz]
dt   = 1e-3     # 1  [ms]
# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
   wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof

# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
```

```
# create gravity vector
g = np.zeros(ndof)

# ================================================
#   set initial cartesian configuration of UR5
# ================================================
# initial cartesian configuration: position, velocity and acceleration
p0 = ur5_robot.get_ee_position()
dp0 = np.array([0.0, 0.0, 0.0])
ddp0 = np.array([0.0, 0.0, 0.0])

# desired cartesian trajectory: position, velocity and acceleration
p_des = copy(p0)
dp_des = np.array([0.0, 0.0, 0.0])
ddp_des = np.array([0.0, 0.0, 0.0])

# measured cartesian trajectory: position, velocity and acceleration
p_med = copy(p0)
dp_med = np.array([0.0, 0.0, 0.0])
ddp_med = np.array([0.0, 0.0, 0.0])

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = np.array([1000, 1000, 1000])   # N/m
# derivative gain
kd = np.array([300, 300, 300])      # N.s/m
# control vector
tau_PD = np.zeros(ndof)

# postural task: gains
Kq = 50*np.eye(ndof)
Dq = 10*np.eye(ndof)
# postural task: control term
tau_0 = np.zeros(ndof)

#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0    # [sec]
```

```
external_force_start = 1 # [sec]
external_force = np.array([0, 0, 0]) # [N]

while not rospy.is_shutdown():
    # desired cartesian trajectory
    p_des[0], dp_des[0], ddp_des[0] = sinusoidal_reference_generator(p0[0], 0.1,
       1.5, sine_duration, t)

    # jacobian: position xyz [3x6]
    J = ur5_robot.jacobian(q_med)[0:3, 0:6]
    # jacobian: damped pseudo-inverse [6x3]
    J_inv = ur5_robot.jacobian_damped_pinv(J)
    # jacobian: time-derivative [3x6]
    dJ = ur5_robot.jacobian_time_derivative(q_med, dq_med)[0:3, 0:6]

    # external force
    if t>=external_force_start:
        external_force = np.array([0, 0, 200]) # [N]


    # error: position and velocity
    e  =  p_des - p_med
    de  =  dp_des - dp_med

    # dynamics: inertia matrix
    M = ur5_robot.get_M()
    M_x = np.linalg.inv(J.dot(np.linalg.inv(M).dot(J.T)))
    # dynamics: nonlinear effects vector
    b = ur5_robot.get_b()
    b_x = J_inv.T.dot(b) - M_x.dot(dJ.dot(dq_med))

    # control signal: null space projection
    tau_0 = Kq.dot(q0-q_med) - Dq.dot(dq_med)
    N = np.eye(ndof) - J_inv.dot(J)
    # control signal: Cartesian inverse dyanmics
    F_d = ddp_des + np.multiply(kp, e) + np.multiply(kd, de)
    tau_PD = J.T.dot( M_x.dot(F_d) + b_x )
    # Cartesian inverse dyanmics + null space projection + external force
    tau = tau_PD + N.dot(tau_0) + J.T.dot(external_force)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
```

```
    q_med, dq_med, ddq_med = ur5_robot.read_joint_position_velocity_acceleration
        ()
    p_med, dp_med, ddp_med = ur5_robot.
        read_cartesian_position_velocity_acceleration()

    # publish message
    jstate.header.stamp = rospy.Time.now()
    jstate.name   = jnames   # Joints position name
    jstate.position  = q_med
    jstate.velocity  = dq_med
    pub.publish(jstate)

    # update time
    t = t + dt

    # stop simulation
    if t>=sim_duration:
        print("stopping rviz ...")
        break
    rate.sleep()
```

Algorithm 11: Move the ur5 robot end-effector using the Cartesian space inverse dynamics and null space projection, (8) to follows the Cartesian sinusoidal reference of activity 1.1 when external force is applied on end-effector.

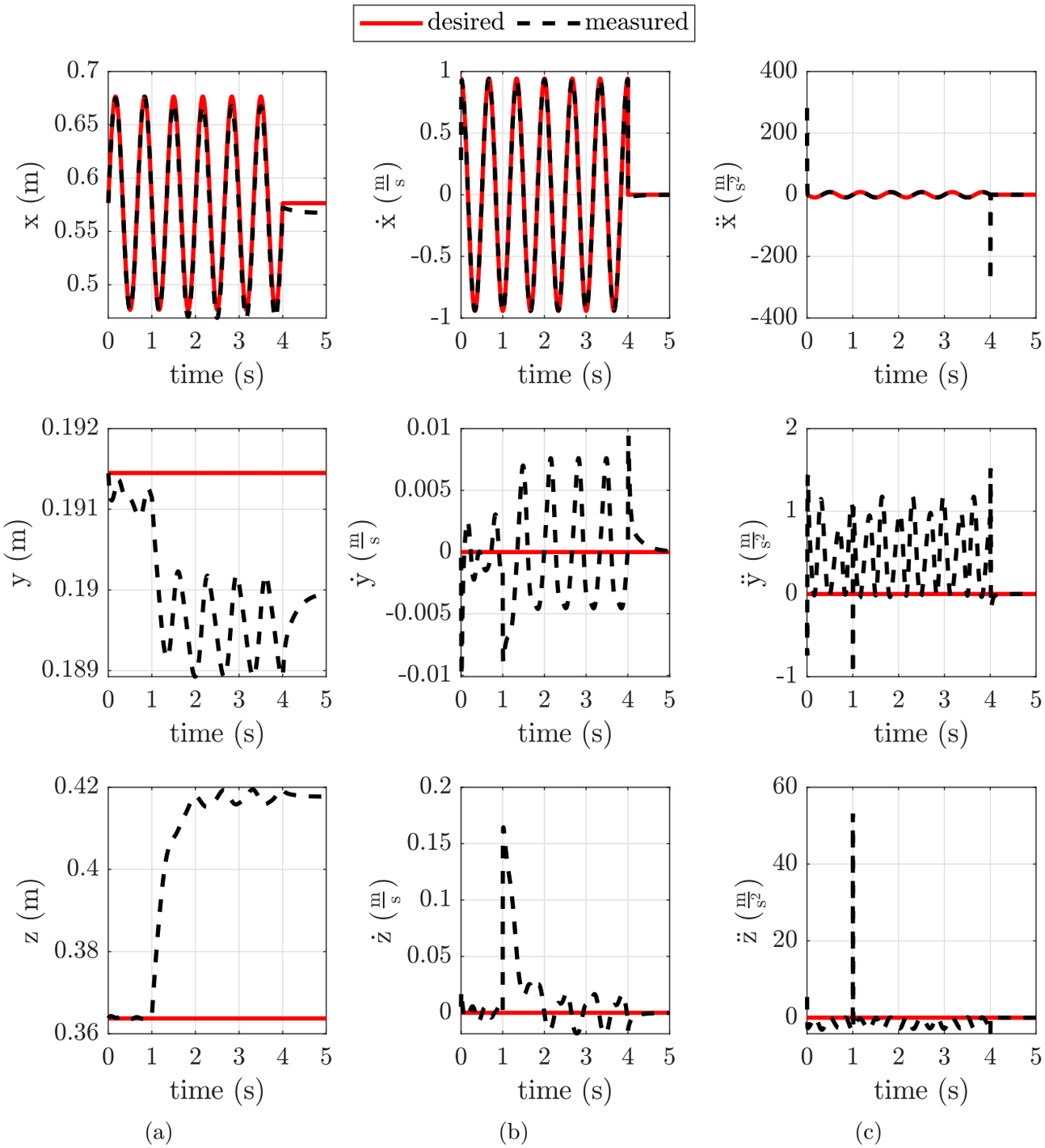


Figure 17: Cartesian trajectory tracking performances using Cartesian space inverse dynamics and null space projection, (8), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$, and when external force ($\mathbf{f_{ext}} = \begin{bmatrix}0.0 & 0.0 & 200.0\end{bmatrix}$ N) is applied on end-effector after 1 second of simulation: (a) position, (b) velocity and (c) acceleration.

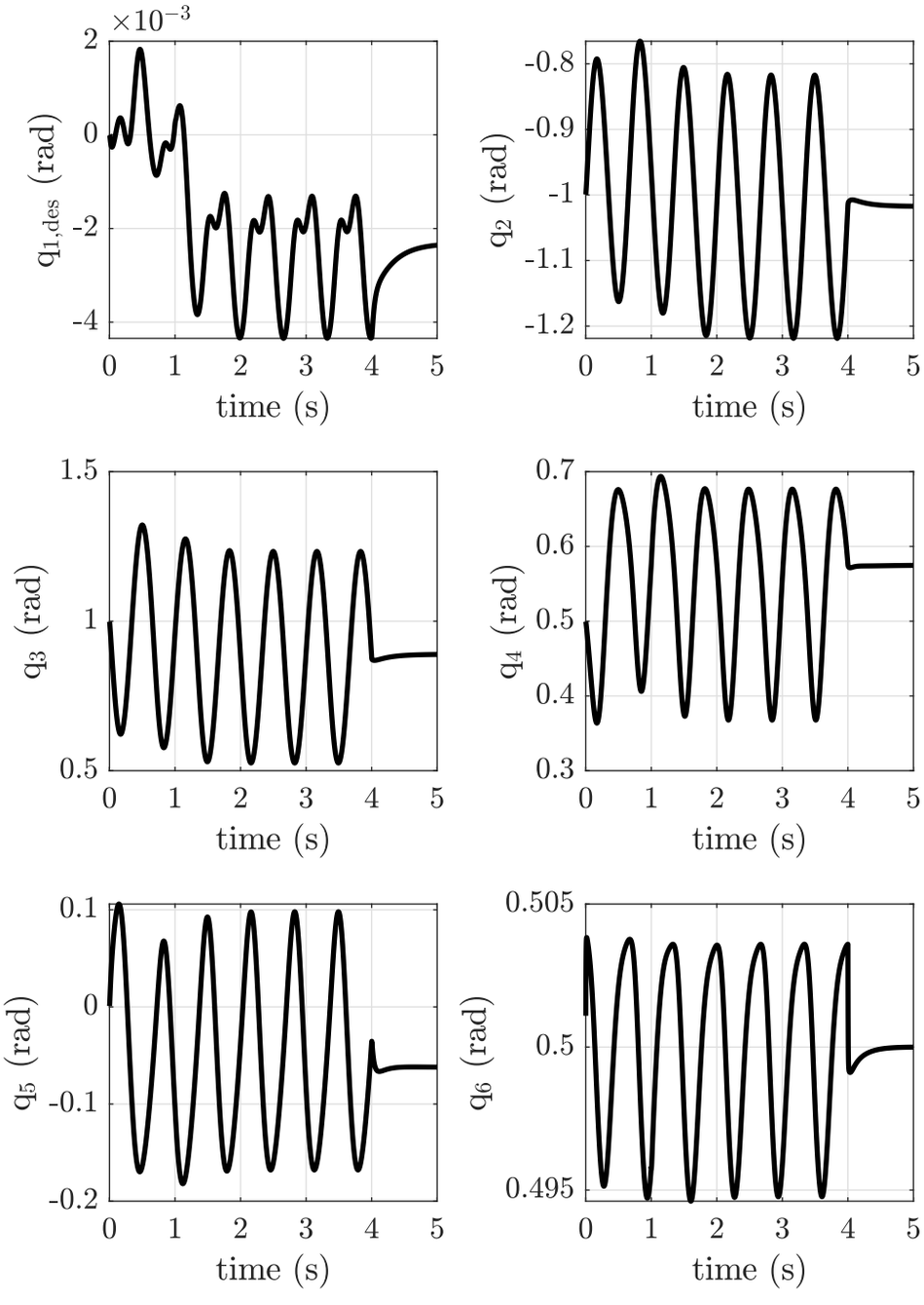


Figure 18: Angular position of each joint of UR5 robot with Algorithm 11.

3 Orientation control

3.1 PD control of end-effector pose

The objective of this activity is to control pose (position and orientation) of the ur5 robot end-effector. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad and end-effector $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m. Therefore, desired Cartesian position is $\mathbf{p_0}$ and angle/axis orientation is $\theta = \pi$ / $\mathbf{r} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$[1]. Motion control is base on pose proportional-derivative method. Finally, control law can be computed as

$$
\begin{aligned}
\boldsymbol{\tau} &= \mathbf{J^T}(\mathbf{W}^d), \\
\mathbf{W}^d &= \begin{bmatrix} \mathbf{F}^d \\ \boldsymbol{\Gamma}^d \end{bmatrix}, \\
\mathbf{F}^d &= \ddot{\mathbf{p}}_{\mathbf{des}} + \mathbf{K_p}(\mathbf{p_{des}} - \mathbf{p}) + \mathbf{K_d}(\dot{\mathbf{p}}_{\mathbf{des}} - \dot{\mathbf{p}}), \\
\boldsymbol{\Gamma}^d &= \mathbf{K_o}\mathbf{e_o} - \mathbf{D_o}(\dot{\mathbf{w}}),
\end{aligned}
\tag{9}
$$

where $\mathbf{J}$ is jacobian matrix, $\mathbf{p_{des}}$ is desired Cartesian position, $\mathbf{K_p}, \mathbf{K_d}$ are Cartesian proportional and derivative gains respectively, $\mathbf{K_o}, \mathbf{D_o}$ are orientation proportional and derivative gains respectively, $\mathbf{w_{des}}$ is desired angular velocity and $\mathbf{e_o}$ is orientation error.

The Algorithm 12 control the movements of ur5 robot end-effector to achieve desired pose. In this file, the control law 9 is configure with $K_p = 1000$ $\frac{\text{N}}{\text{m}}$, $K_d = 300$ $\frac{\text{N.s}}{\text{m}}$, $K_o = 800$ $\frac{\text{N.m}}{\text{rad}}$ and $D_o = 30$ $\frac{\text{N.m.s}}{\text{rad}}$. On one hand, Figure 19 shows that position tracking is acceptable with mean norm error at each axis $(||e_x||, ||e_y||, ||e_z||)$ of (0.0302, 0.0119, 0.0593) cm respectively. On the other hand, Figure 20 shows that orientation tracking is excellent with mean norm error at each axis $(||e_{o,x}||, ||e_{o,y}||, ||e_{o,z}||)$ of (0.0006, 0.0008, 0.0014) rad respectively. Finally, Figure 21 shows final pose ur5 end-effector. In this figure, reference frame of end-effector is rotated π rad around x-axis with respect to reference frame of base-link.

```
# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
```

[1] code to compute angle and axis from rotation matrix in https://github.com/JhonPool4/legged_robots_labs_ws/blob/master/src/labpythonlib/labpythonlib/lab_functions.py

```
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
   wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof

# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ============================================
#   set initial cartesian configuration of UR5
# ============================================
# initial cartesian configuration
p0 = ur5_robot.read_ee_position()    # position
dp0 = np.array([0.0, 0.0, 0.0])     # velocity
ddp0 = np.array([0.0, 0.0, 0.0])    # acceleration
R0 = ur5_robot.read_ee_orientation() # orientation
w0 = ur5_robot.read_ee_angular_velocity() # angular velocity
# desired cartesian trajectory
p_des = copy(p0)                    # position
```

```
dp_des = np.array([0.0, 0.0, 0.0])  # velocity
ddp_des = np.array([0.0, 0.0, 0.0]) # acceleration
R_des = copy(R0)                    # orientation
w_des = np.array([0.0, 0.0, 0.0])   # angular velocity
# measured cartesian trajectory
p_med = copy(p0)                    # position
dp_med = np.array([0.0, 0.0, 0.0])  # velocity
ddp_med = np.array([0.0, 0.0, 0.0]) # acceleration
R_med = copy(R0)                    # orientation
w_med = copy(w0)                    # angular velocity

# ==============================
#   PD controller configuration
# ==============================
# position gains
Kp = 1000*np.eye(3)  # N/m
Kd = 300*np.eye(3)      # N.s/m
# orientation gains
Ko = 800*np.eye(3)
Do = 30*np.eye(3)
# control vector
F_p = np.zeros(3)
F_o = np.zeros(3)
tau = np.zeros(ndof)
#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0    # [sec]

while not rospy.is_shutdown():
    # desired cartesian trajectory
    #p_des[0], dp_des[0], ddp_des[0] = sinusoidal_reference_generator(p0[0], 0.1,
        1.5, sine_duration, t)

    # desired orientation
    R_des = np.array([[1, 0, 0],[0, -1, 0],[0, 0, -1]])

    # jacobian: pose [6x6]
    J = ur5_robot.jacobian(q_med)

    # error: position and velocity
```

```
    e  =  p_des - p_med
    de  =  dp_des - dp_med
    # error: orientation
    R_e = R_med.T.dot(R_des) # required rotation matrix ("error")
    angle_e, axis_e = rot2axisangle(R_e) # angle/axis ("error")
    e_o = R_med.dot(angle_e*axis_e) # w.r.t world frame ("error")

    # control signal: Cartesian PD
    F_p = ddp_des + np.dot(Kp, e) + np.dot(Kd, de)
    # control signal: orientation PD
    F_o = Ko.dot(e_o) + Do.dot(w_des - w_med)

    # control signal: Cartesian inverse_dyanmics + null space projection
    F_pose = np.concatenate((F_p, F_o), axis=0)
    tau = J.T.dot(F_pose)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q_med, dq_med, ddq_med = ur5_robot.read_joint_position_velocity_acceleration
        ()
    p_med, dp_med, ddp_med = ur5_robot.
        read_cartesian_position_velocity_acceleration()
    R_med = ur5_robot.read_ee_orientation()
    w_med = ur5_robot.read_ee_angular_velocity()
    # publish message
    jstate.header.stamp = rospy.Time.now()
    jstate.name   = jnames   # Joints position name
    jstate.position  = q_med
    jstate.velocity  = dq_med
    pub.publish(jstate)
    # update time
    t = t + dt
    # stop simulation
    if t>=sim_duration:
        print("stopping rviz ...")
        break
    rate.sleep()
```

Algorithm 12: Move the ur5 robot end-effector using pose propotional-derivative control method (9) to achieve position $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m and angle/axis orientation $\theta = \pi$ / $\mathbf{r} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$.

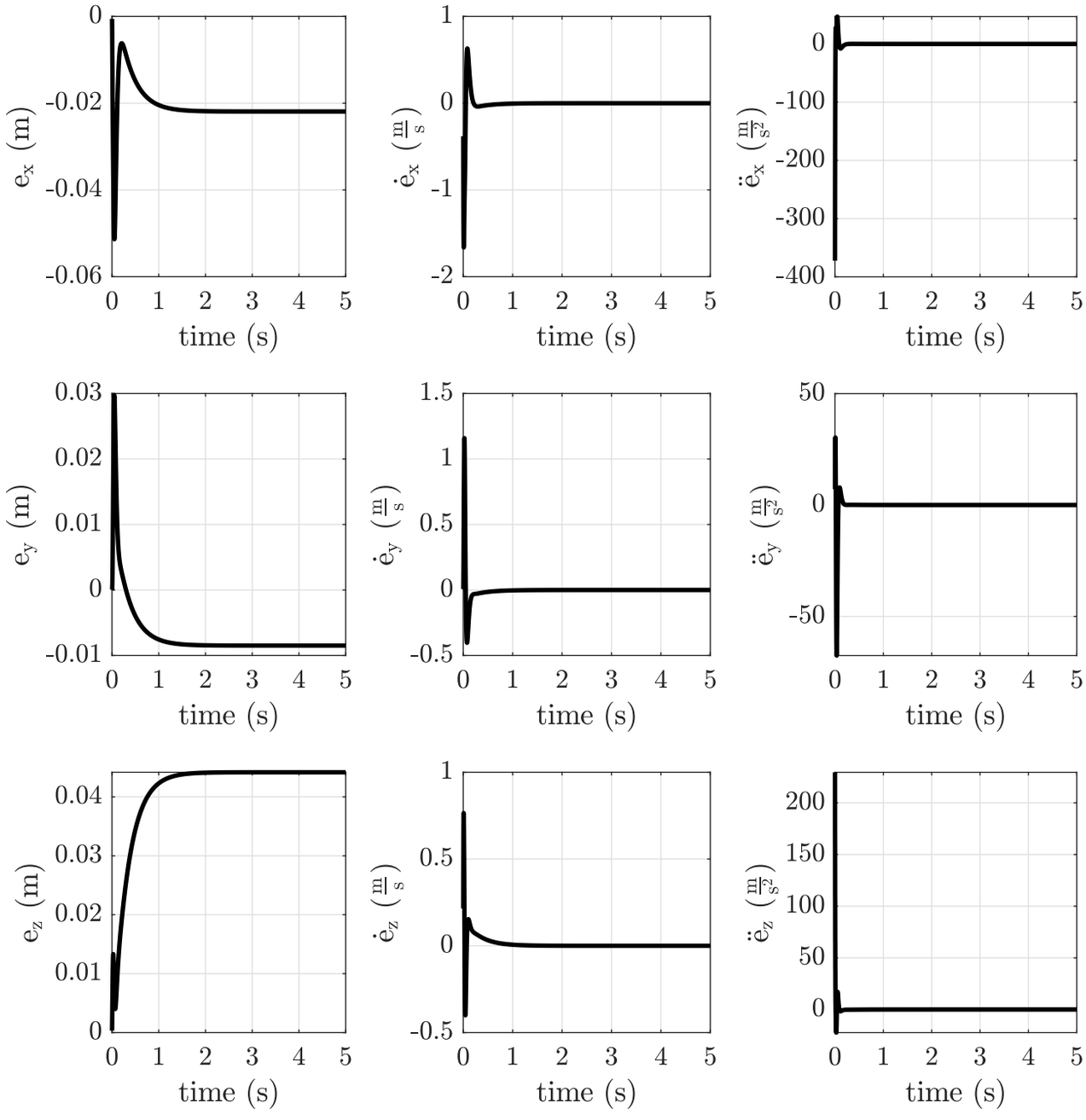


Figure 19: Cartesian trajectory tracking error using pose proportional-derivative control method, (9), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_o = 800\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$: (a) position, (b) velocity and (c) acceleration.

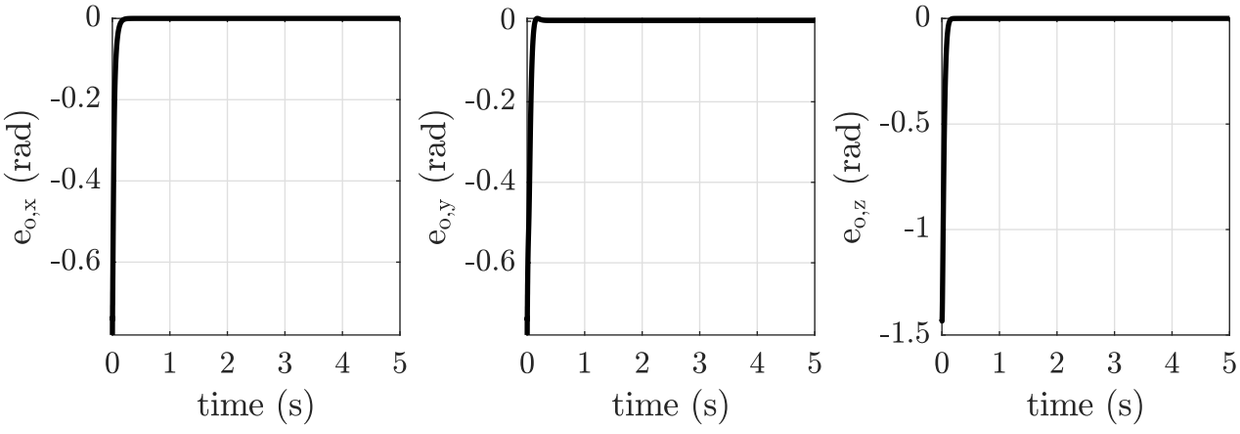
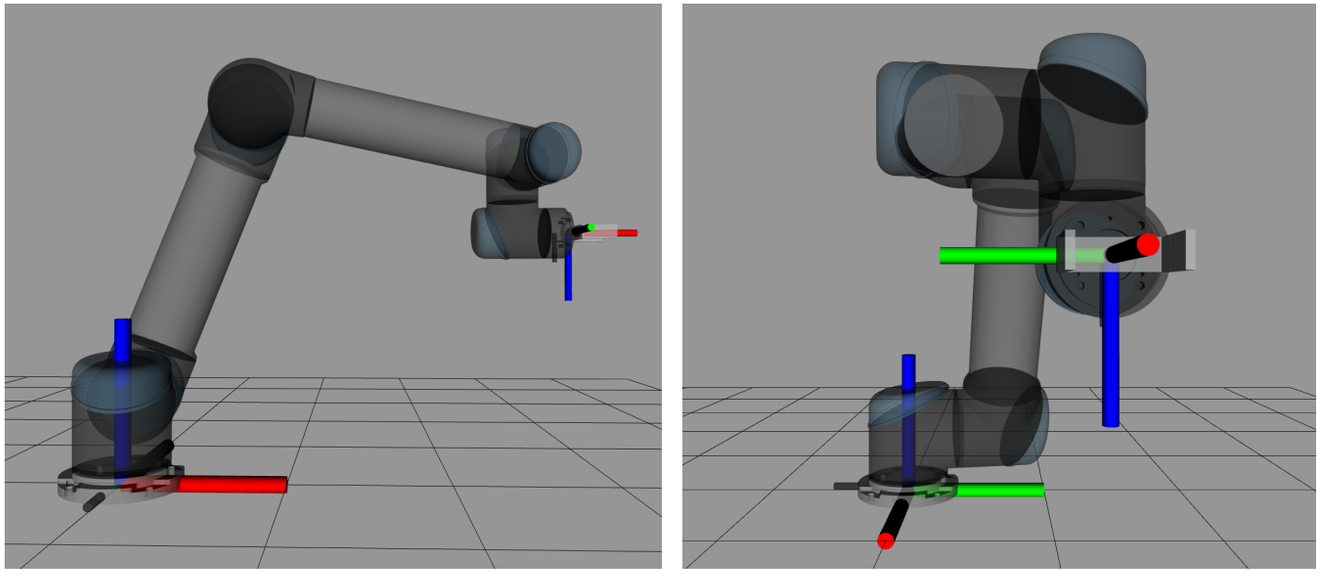


Figure 20: Angular position of each joint of UR5 robot with Algorithm 10.



(a) (b)

Figure 21: Final point of view of UR5 robot with pose proportional-derivative control method: (a) lateral and (b) front.

3.2 PD control of end-effector pose - singularity and sinusoidal reference

The objective of this activity is understand singularity related to Euler angles in ZYX convention. For this purpose, orientation error using angle/axis and Euler representation will be compared. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad that set the end-effector pose as $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m and $\mathbf{o_0}(\alpha, \beta, \gamma) = \begin{bmatrix} 1.57 & 0.0 & -2.14 \end{bmatrix}$ rad; orientation is represented with Euler angles in ZYX convention[2]. Therefore, desired Cartesian position is same as $\mathbf{p_0}$ and desired orientation starts at $\mathbf{o_0}$ then γ increases with $\frac{\pi}{2} \sin 0.4\pi t$. Finally, motion control is pose proportional-derivative method with feed-forward terms and can be computed as

$$
\begin{aligned}
\boldsymbol{\tau} &= \mathbf{J^T}(\mathbf{W}^d), \\
\mathbf{W}^d &= \begin{bmatrix} \mathbf{F}^d \\ \boldsymbol{\Gamma}^d \end{bmatrix}, \\
\mathbf{F}^d &= \ddot{\mathbf{p}}_{\mathbf{des}} + \mathbf{K_p}(\mathbf{p_{des}} - \mathbf{p}) + \mathbf{K_d}(\dot{\mathbf{p}}_{\mathbf{des}} - \dot{\mathbf{p}}), \\
\boldsymbol{\Gamma}^d &= \dot{\mathbf{w}}_{\mathbf{des}} + \mathbf{K_o}\mathbf{e_o} + \mathbf{D_o}(\dot{\mathbf{w}}_{\mathbf{des}} - \dot{\mathbf{w}})
\end{aligned}
\tag{10}
$$

where $\mathbf{J}$ is jacobian matrix, $\mathbf{p_{des}}$ is desired Cartesian position, $\mathbf{K_p}, \mathbf{K_d}$ are Cartesian proportional and derivative gains respectively, $\mathbf{K_o}, \mathbf{D_q o}$ are orientation proportional and derivative gains respectively, $\mathbf{w_{des}}$ is desired angular velocity and $\mathbf{e_o}$ is orientation error.

Algorithm 13 control the movements of ur5 robot end-effector to achieve desired pose. In this file, the control law 9 is configure with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_o = 800\ \frac{\text{N.m}}{\text{rad}}$ and $D_o = 30\ \frac{\text{N.m.s}}{\text{rad}}$. On one hand, Figure 22 shows that position tracking is poor with mean norm error at each axis $(||e_x||, ||e_y||, ||e_z||)$ of (0.0752, 0.0116, 0.0741) cm respectively. On the other hand, Figure 23 shows that orientation tracking (angle/axis representation) is excellent with mean norm error at each axis $(||e_{o,x}||, ||e_{o,y}||, ||e_{o,z}||)$ of (0.00007, 0.00003, 0.0001) rad respectively. Likewise, orientation error represented with Euler angles presents discontinuity unlike the representation with axis/angle. Finally, Figure 24 shows that the discontinuity occurs for rotations greater than $|\pi|$.

```
# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
```

[2]code to compute rotation matrix from Euler angles in https://github.com/JhonPool4/legged_robots_labs_ws/blob/master/src/labpythonlib/labpythonlib/lab_functions.py

```
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
   wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof

# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ============================================
#   set initial cartesian configuration of UR5
# ============================================
# initial cartesian configuration: position
p0 = ur5_robot.read_ee_position()    # position
dp0 = np.array([0.0, 0.0, 0.0])     # velocity
ddp0 = np.array([0.0, 0.0, 0.0])    # acceleration
# initial cartesian configuration: orientation
R0 = ur5_robot.read_ee_orientation() # orientation
w0 = ur5_robot.read_ee_angular_velocity() # angular velocity
rpy0 = rot2rpy(R0)                     # orientation (rpy)
```

```
# desired cartesian trajectory: position
p_des = copy(p0)                    # position
dp_des = np.array([0.0, 0.0, 0.0])  # velocity
ddp_des = np.array([0.0, 0.0, 0.0]) # acceleration
# desired cartesian trajectory: orientation
R_des = copy(R0)
rpy_des = copy(rpy0)                       # orientation (rpy)
drpy_des = np.zeros(3)
ddrpy_des = np.zeros(3)                                  # orientation
w_des = angular_velocity_rpy(rpy_des, drpy_des)     # angular velocity
dw_des = angular_acceleration_rpy(rpy_des, drpy_des, ddrpy_des) # angular
   acceleration
# measured cartesian trajectory: position
p_med = copy(p0)                    # position
dp_med = np.array([0.0, 0.0, 0.0])  # velocity
ddp_med = np.array([0.0, 0.0, 0.0]) # acceleration
# measured cartesian trajectory: orientation
R_med = copy(R0)                    # orientation
rpy_med = copy(rpy0)               # orientation (rpy)
w_med = copy(w0)                    # angular velocity
dw_med = np.zeros(3)                # angular acceleration
# ==============================
#   PD controller configuration
# ==============================
# position gains
Kp = 1000*np.eye(3)  # N/m
Kd = 300*np.eye(3)      # N.s/m
# orientation gains
Ko = 800*np.eye(3)
Do = 30*np.eye(3)
# control vector
F_p = np.zeros(3)
F_o = np.zeros(3)
tau = np.zeros(ndof)
#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0    # [sec]

while not rospy.is_shutdown():
    # desired cartesian trajectory
```

```
    #p_des[0], dp_des[0], ddp_des[0] = sinusoidal_reference_generator(p0[0], 0.1,
        1.5, sine_duration, 0*t)

    # desired orientation: rotation around x axis
    rpy_des[2], drpy_des[2], ddrpy_des[2] = sinusoidal_reference_generator(rpy0
       [2], np.pi/2, 1/5, 6, t)
    R_des   = rpy2rot(rpy_des)
    # desired angular velocity
    w_des = angular_velocity_rpy(rpy_des, drpy_des)
    # desired angular acceleration
    dw_des = angular_acceleration_rpy(rpy_des, drpy_des, ddrpy_des)

    # jacobian: pose [6x6]
    J = ur5_robot.jacobian(q_med)

    # error: position and velocity
    e  =  p_des - p_med
    de  =  dp_des - dp_med
    # error: orientation
    R_e = R_med.T.dot(R_des)
    angle_e, axis_e = rot2axisangle(R_e)
    e_o = R_med.dot(angle_e*axis_e) # w.r.t world frame

    # control signal: Cartesian PD
    F_p = ddp_des + np.dot(Kp, e) + np.dot(Kd, de)
    # control signal: orientation PD
    F_o = dw_des + Ko.dot(e_o) + Do.dot(w_des - w_med)

    # control signal: Cartesian inverse_dyanmics + null space projection
    F_pose = np.concatenate((F_p, F_o), axis=0)
    tau = J.T.dot(F_pose)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q_med, dq_med, ddq_med = ur5_robot.read_joint_position_velocity_acceleration
       ()
    p_med, dp_med, ddp_med = ur5_robot.
       read_cartesian_position_velocity_acceleration()
    R_med = ur5_robot.read_ee_orientation()
    w_med = ur5_robot.read_ee_angular_velocity()
    rpy_med = np.zeros(3)
    rpy_med = rot2rpy(R_med)
```

```
    # publish message
    jstate.header.stamp = rospy.Time.now()
    jstate.name   = jnames   # Joints position name
    jstate.position  = q_med
    jstate.velocity  = dq_med
    pub.publish(jstate)

    # update time
    t = t + dt

    # stop simulation
    if t>=sim_duration:
        print("stopping rviz ...")
        break
    rate.sleep()
```

Algorithm 13: Move the ur5 robot end-effector using pose propotional-derivative control method (10) to achieve position $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m and Euler angles orientation $\gamma = \frac{\pi}{2} \sin 0.4\pi t$.

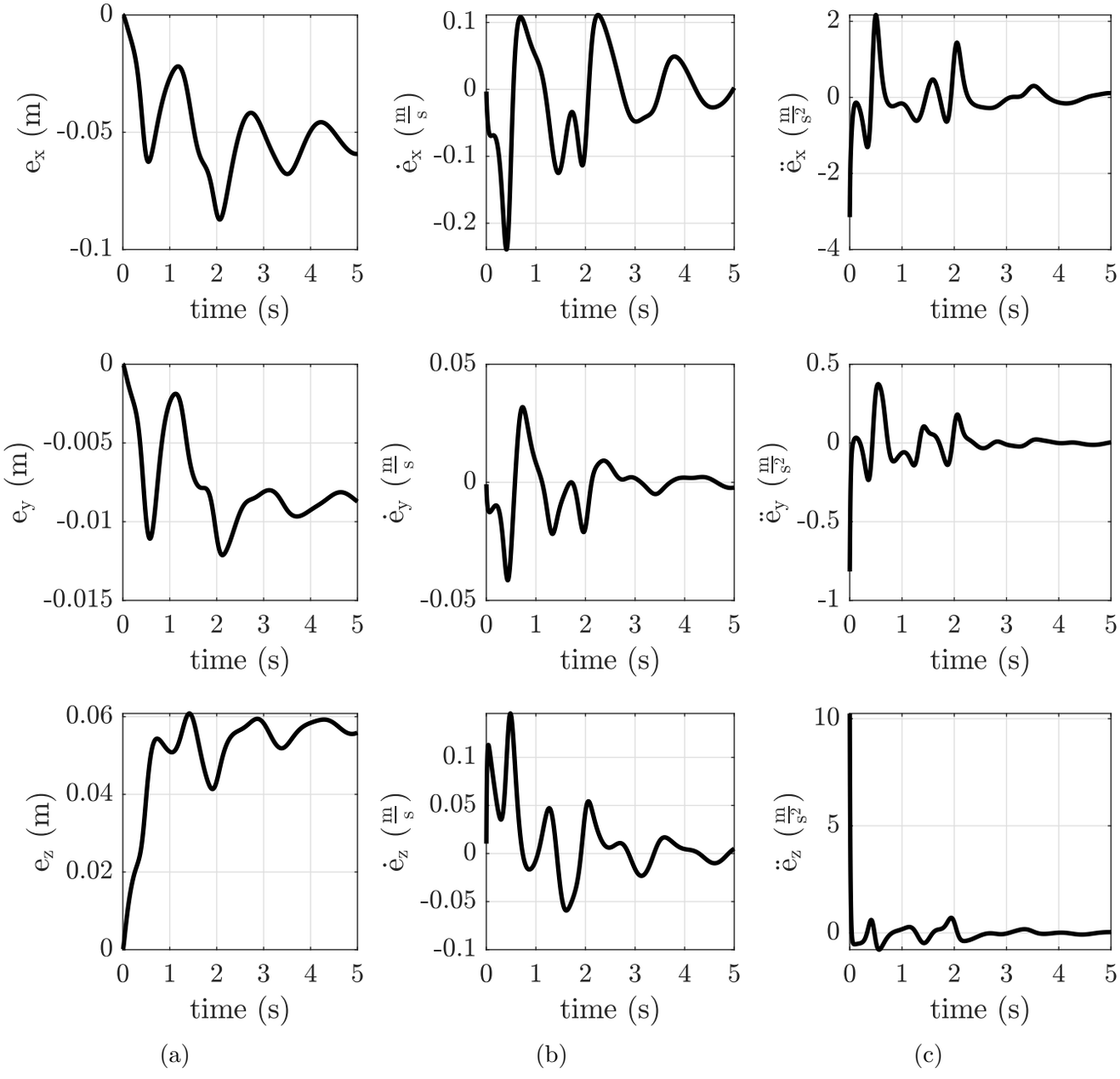


Figure 22: Cartesian trajectory tracking error using pose proportional-derivative control method with feed-forward terms, (10), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_o = 800\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$: (a) position, (b) velocity and (c) acceleration.

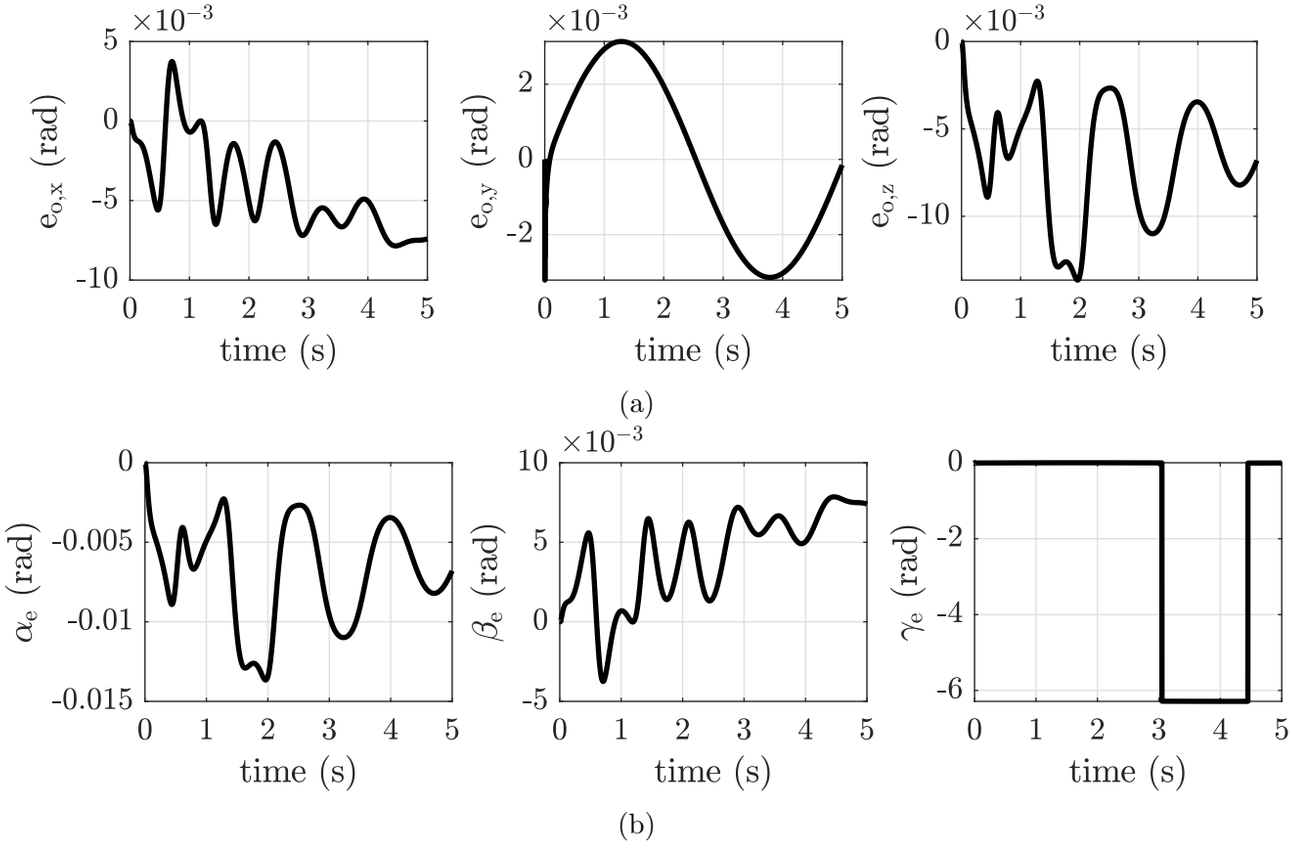


Figure 23: Orientation error using pose proportional-derivative control method with feed-forward terms, (10), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_o = 800\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$: (a) angle/axis and (b) Euler angles in ZYX convention.

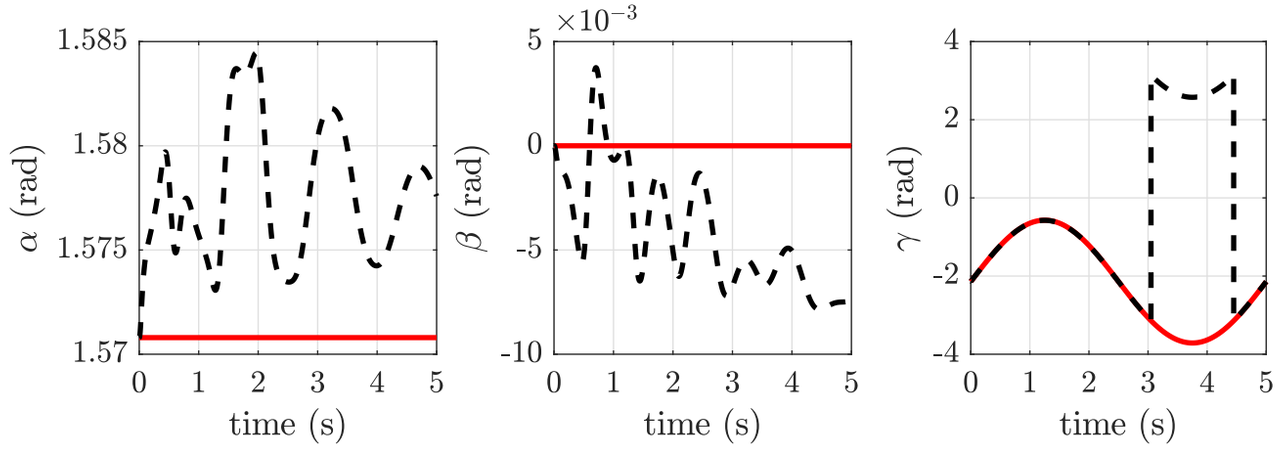


Figure 24: Orientation trajectory tracking represented with Euler angles in ZYX convention.

3.4 PD control of end-effector pose - unwrapping

The objective of this activity is implement an unwrapping algorithm to eliminate the discontinuity of Euler angles[3]. Figure 25 and 26 shows that orientation tracking using Euler angles is continuous despite rotations greater than $|\pi|$.

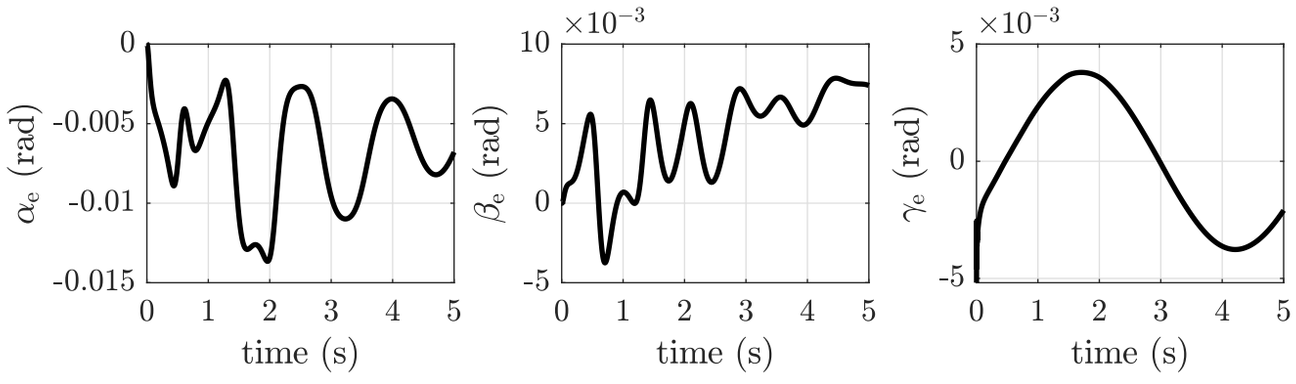


Figure 25: Orientation trajectory tracking represented with Euler angles in ZYX convention.

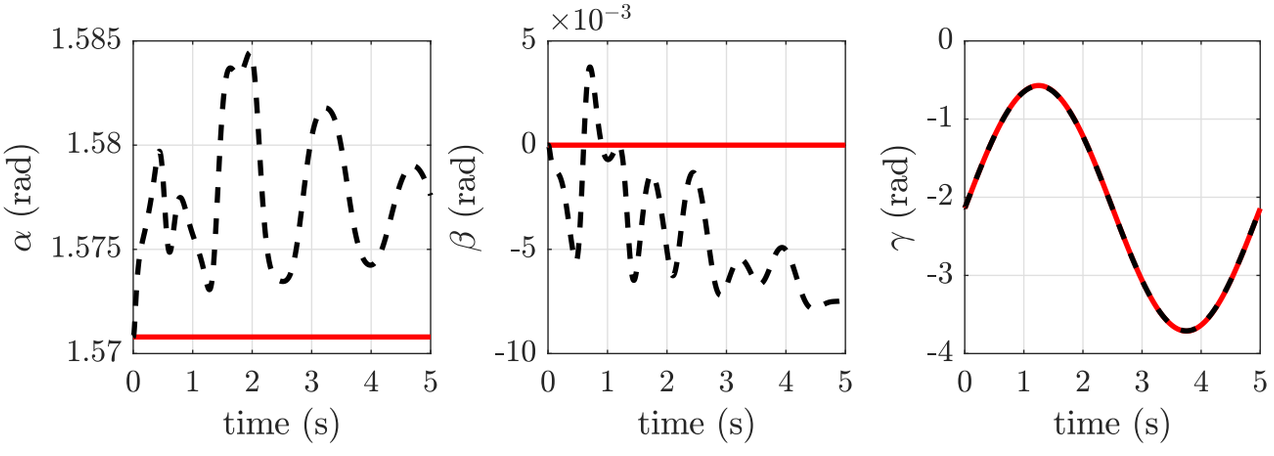


Figure 26: Orientation trajectory tracking represented with Euler angles in ZYX convention.

[3]code to compute Euler angles with unwrapping algorithm form rotation matrix in https://github.com/JhonPool4/legged_robots_labs_ws/blob/master/src/labpythonlib/labpythonlib/lab_functions.py

3.6 PD control of end-effector pose - inverse dynamics

The objective of this activity is to control pose (position and orientation) of the ur5 robot end-effector. The simulation starts with initial joint configuration $\mathbf{q_0} = \begin{bmatrix} 0.0 & -1.0 & 1.0 & 0.5 & 0.0 & 0.5 \end{bmatrix}$ rad that set the end-effector pose as $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m and $\mathbf{o_0}(\alpha, \beta, \gamma) = \begin{bmatrix} 1.57 & 0.0 & -2.14 \end{bmatrix}$ rad; orientation is represented with Euler angles in ZYX convention. Therefore, desired Cartesian position is same as $\mathbf{p_0}$ and desired orientation starts at $\mathbf{o_0}$ then γ increases with $\frac{\pi}{2} \sin 0.4\pi t$. Finally, motion control is pose inverse dynamics and can be computed as

$$
\begin{aligned}
\boldsymbol{\tau} &= \mathbf{J^T}(\boldsymbol{\Lambda}\mathbf{W}^d + \boldsymbol{\mu}), \\
\boldsymbol{\Lambda} &= (\mathbf{J}\mathbf{M^{-1}}\mathbf{J^T})^{\#}, \\
\boldsymbol{\mu} &= \mathbf{J}^{\mathbf{T}\#} - \boldsymbol{\Lambda}\dot{\mathbf{J}}\dot{\mathbf{q}}, \\
\mathbf{W}^d &= \begin{bmatrix} \mathbf{F}^d \\ \boldsymbol{\Gamma}^d \end{bmatrix}, \\
\mathbf{F}^d &= \ddot{\mathbf{p}}_{\mathbf{des}} + \mathbf{K_p}(\mathbf{p_{des}} - \mathbf{p}) + \mathbf{K_d}(\dot{\mathbf{p}}_{\mathbf{des}} - \dot{\mathbf{p}}), \\
\boldsymbol{\Gamma}^d &= \dot{\mathbf{w}}_{\mathbf{des}} + \mathbf{K_o}\mathbf{e_o} + \mathbf{D_o}(\dot{\mathbf{w}}_{\mathbf{des}} - \dot{\mathbf{w}})
\end{aligned}
\tag{11}
$$

where $\mathbf{J}$ is jacobian matrix, $\boldsymbol{\Lambda}$ is inertia matrix at Cartesian space, $\boldsymbol{\mu}$ is nonlinear effects vector at Cartesian space, $\mathbf{p_{des}}$ is desired Cartesian position, $\mathbf{K_p}, \mathbf{K_d}$ are Cartesian proportional and derivative gains respectively, $\mathbf{K_o}, \mathbf{D_o}$ are orientation proportional and derivative gains respectively, $\mathbf{w_{des}}$[4] is desired angular velocity and $\mathbf{e_o}$ is orientation error.

Algorithm 14 control the movements of ur5 robot end-effector to achieve desired pose. In this file, the control law 11 is configure with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_o = 800\ \frac{\text{N.m}}{\text{rad}}$ and $D_o = 30\ \frac{\text{N.m.s}}{\text{rad}}$. On one hand, Figure 27 shows that position tracking is good with mean norm error at each axis $(||e_x||, ||e_y||, ||e_z||)$ of (0.0036, 0.0002, 0.0064) cm respectively. On the other hand, Figure 28 shows that orientation tracking (Euler angles) is good with mean norm error at each axis $(||e_{o,x}||, ||e_{o,y}||, ||e_{o,z}||)$ of (0.000001, 0.0000008, 0.0022) rad respectively. Likewise, desired and measured orientation represented with Euler angles in ZYX convention is shown in Figure 29.

```
# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
```

[4] code to compute desired angular velocity from Euler angles and derivatives in https://github.com/JhonPool4/legged_robots_labs_ws/blob/master/src/labpythonlib/labpythonlib/lab_functions.py

```
q0 =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq0 =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq0 = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# desired trajectory: position, velocity and acceleration
q_des =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_des =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_des = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# measured trajectory: position, velocity and acceleration
q_med =   np.array([ 0.0, -1.0, 1.0, 0.5, 0.0, 0.5])
dq_med =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq_med = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

# ============================
#   UR5 robot configuration
# ============================
# joints name of UR5 robot
jnames = ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint','
   wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
# path of labs_ur5.urdf
urdf_path = os.path.join(pwd, "../../ur5_description/urdf/labs_ur5.urdf")
# the class robot load labs_ur5.urdf
ur5_robot = Robot(q0, dq0, dt, urdf_path)
# number of degress of freedom
ndof = ur5_robot.ndof

# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ============================================
#   set initial cartesian configuration of UR5
# ============================================
# initial cartesian configuration: position
p0 = ur5_robot.read_ee_position()    # position
dp0 = np.array([0.0, 0.0, 0.0])     # velocity
ddp0 = np.array([0.0, 0.0, 0.0])    # acceleration
# initial cartesian configuration: orientation
R0 = ur5_robot.read_ee_orientation() # orientation
```

```
w0 = ur5_robot.read_ee_angular_velocity() # angular velocity
rpy0 = rot2rpy(R0)                        # orientation (rpy)
# desired cartesian trajectory: position
p_des = copy(p0)                      # position
dp_des = np.array([0.0, 0.0, 0.0])  # velocity
ddp_des = np.array([0.0, 0.0, 0.0]) # acceleration
# desired cartesian trajectory: orientation
R_des = copy(R0)
rpy_des = copy(rpy0)                          # orientation (rpy)
drpy_des = np.zeros(3)
ddrpy_des = np.zeros(3)                                    # orientation
w_des = angular_velocity_rpy(rpy_des, drpy_des)     # angular velocity
dw_des = angular_acceleration_rpy(rpy_des, drpy_des, ddrpy_des) # angular
    acceleration
# measured cartesian trajectory: position
p_med = copy(p0)                      # position
dp_med = np.array([0.0, 0.0, 0.0])  # velocity
ddp_med = np.array([0.0, 0.0, 0.0]) # acceleration
# measured cartesian trajectory: orientation
R_med = copy(R0)                      # orientation
rpy_med = copy(rpy0)                 # orientation (rpy)
w_med = copy(w0)                      # angular velocity
dw_med = np.zeros(3)                  # angular acceleration
# ================================
#   PD controller configuration
# ================================
# position gains
Kp = 400*np.eye(3)  # N/m
Kd = 40*np.eye(3)      # N.s/m
# orientation gains
Ko = 400*np.eye(3)
Do = 40*np.eye(3)
# control vector
F_p = np.zeros(3)
F_o = np.zeros(3)
tau = np.zeros(ndof)
#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0    # [sec]
```

```
while not rospy.is_shutdown():
    # desired cartesian trajectory
    #p_des[0], dp_des[0], ddp_des[0] = sinusoidal_reference_generator(p0[0], 0.1,
        1.5, sine_duration, 0*t)

    # desired orientation: rotation around x axis
    rpy_des[2], drpy_des[2], ddrpy_des[2] = sinusoidal_reference_generator(rpy0
       [2], np.pi/2, 1/5, 6, t)
    R_des   = rpy2rot(rpy_des)
    # desired angular velocity
    w_des = angular_velocity_rpy(rpy_des, drpy_des)
    # desired angular acceleration
    dw_des = angular_acceleration_rpy(rpy_des, drpy_des, ddrpy_des)

    # jacobian: pose [6x6]
    J = ur5_robot.jacobian(q_med)
    # jacobian: damped pseudo-inverse [6x6]
    J_inv = ur5_robot.jacobian_damped_pinv(J)
    # jacobian: time-derivative [6x6]
    dJ = ur5_robot.jacobian_time_derivative(q_med, dq_med)

    # error: position and velocity
    e  =  p_des - p_med
    de  =  dp_des - dp_med
    # error: orientation
    R_e = R_med.T.dot(R_des)
    angle_e, axis_e = rot2axisangle(R_e)
    e_o = R_med.dot(angle_e*axis_e) # w.r.t world frame

    # dynamics: inertia matrix
    M = ur5_robot.get_M()
    M_x = damped_pinv(J.dot(np.linalg.inv(M).dot(J.T)))
    # dynamics: nonlinear effects vector
    b = ur5_robot.get_b()
    b_x = J_inv.T.dot(b) - M_x.dot(dJ.dot(dq_med))

    # control signal: Cartesian PD
    F_p = ddp_des + np.dot(Kp, e) + np.dot(Kd, de)
    # control signal: orientation PD
    F_o = dw_des + Ko.dot(e_o) + Do.dot(w_des - w_med)

    # control signal: Cartesian inverse_dyanmics + null space projection
    F_pose = np.concatenate((F_p, F_o), axis=0)
```

```
    tau = J.T.dot(M_x.dot(F_pose) + b_x)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q_med, dq_med, ddq_med = ur5_robot.read_joint_position_velocity_acceleration
        ()
    p_med, dp_med, ddp_med = ur5_robot.
        read_cartesian_position_velocity_acceleration()
    R_med = ur5_robot.read_ee_orientation()
    w_med = ur5_robot.read_ee_angular_velocity()
    rpy_med = rot2rpy_unwrapping(R_med, rpy_med)

    # publish message
    jstate.header.stamp = rospy.Time.now()
    jstate.name   = jnames   # Joints position name
    jstate.position  = q_med
    jstate.velocity  = dq_med
    pub.publish(jstate)

    # update time
    t = t + dt

    # stop simulation
    if t>=sim_duration:
        print("stopping rviz ...")
        break
    rate.sleep()
```

Algorithm 14: Move the ur5 robot end-effector using pose inverse dynamics (11) to achieve position $\mathbf{p_0} = \begin{bmatrix} 0.577 & 0.192 & 0.364 \end{bmatrix}$ m and Euler angles orientation $\gamma = \frac{\pi}{2} \sin 0.4\pi t$.

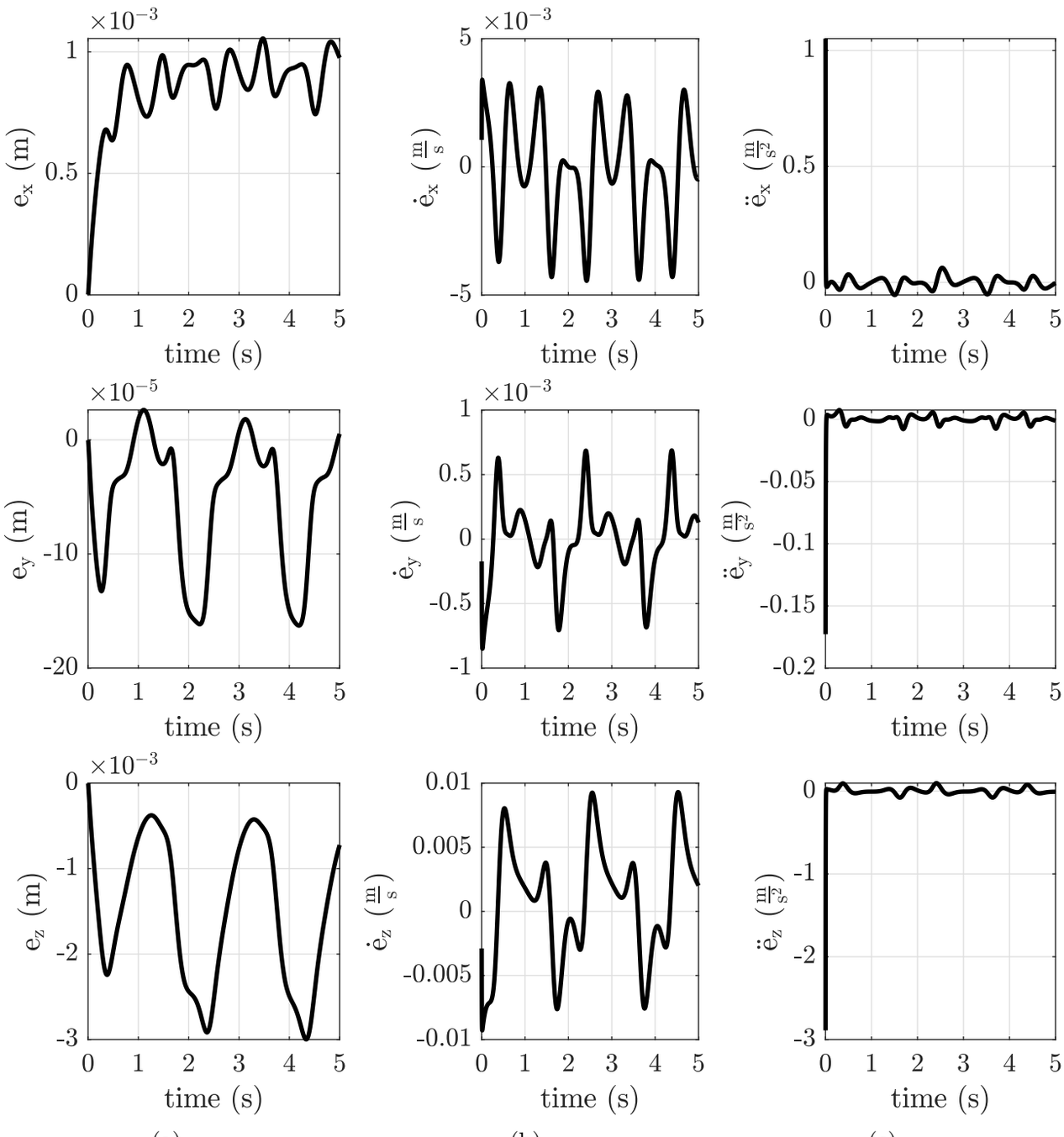


Figure 27: Cartesian trajectory tracking error using inverse dynamics, (11), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_o = 800\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$: (a) position, (b) velocity and (c) acceleration.

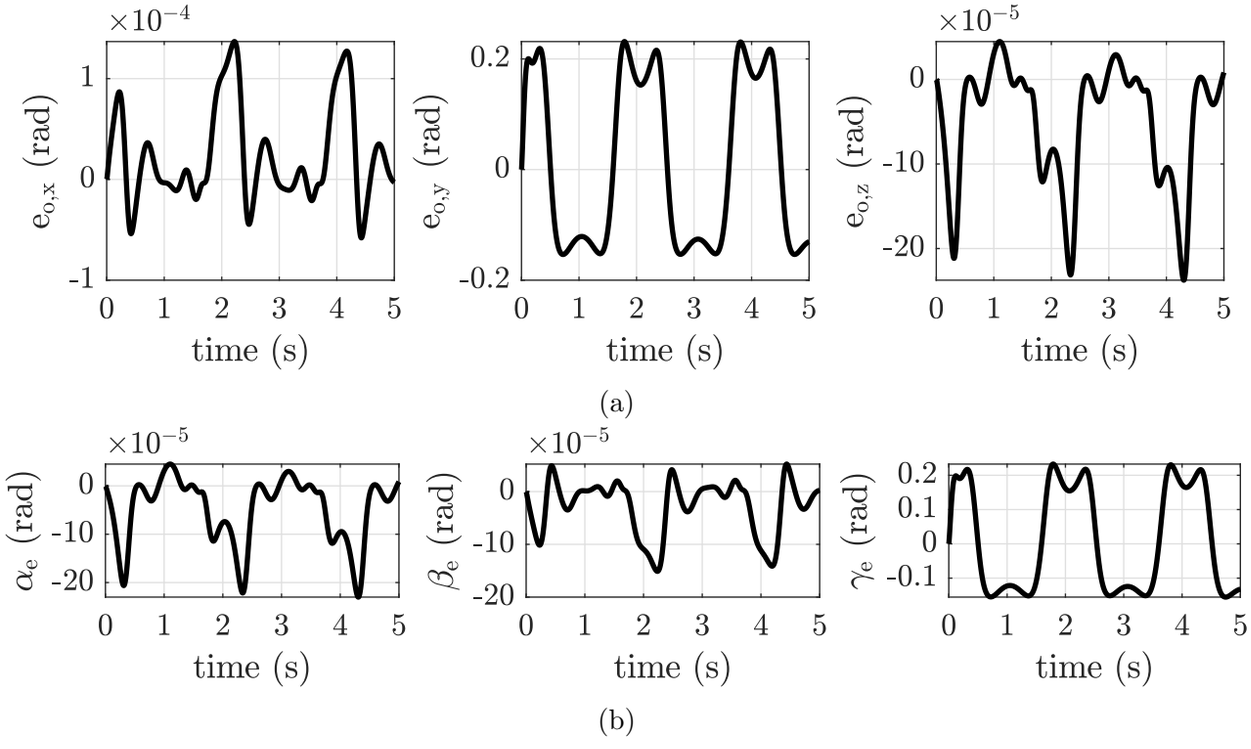


Figure 28: Orientation error using pose proportional-derivative control method with feed-forward terms, (10), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_o = 800\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$: (a) angle/axis and (b) Euler angles in ZYX convention.

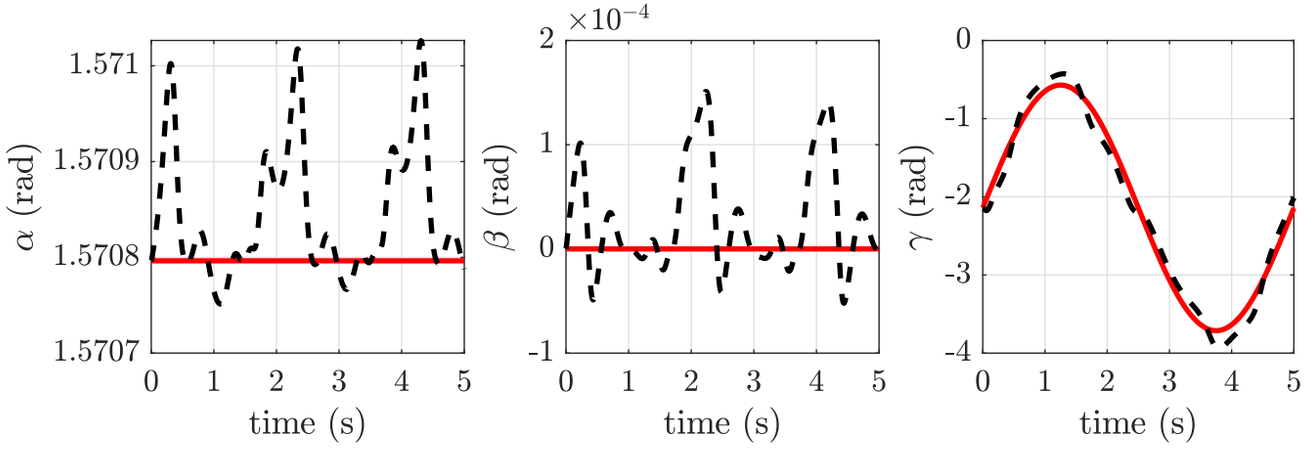


Figure 29: Orientation trajectory tracking represented with Euler angles in ZYX convention.