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.

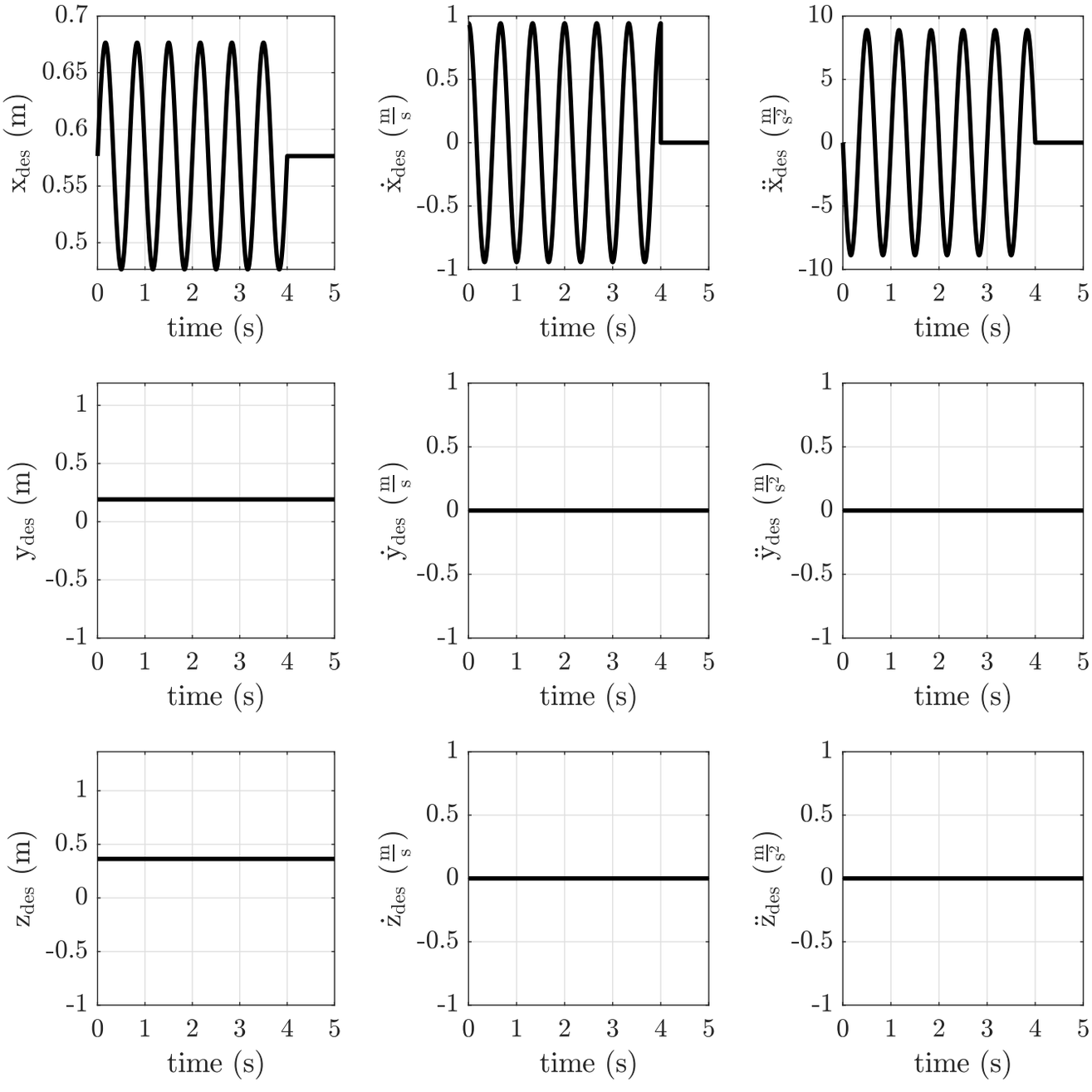

Figure 1: Cartesian reference trajectories for the robot end-effector with Algorithm 1: (a) position, (b) velocity and (c) acceleration.

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.

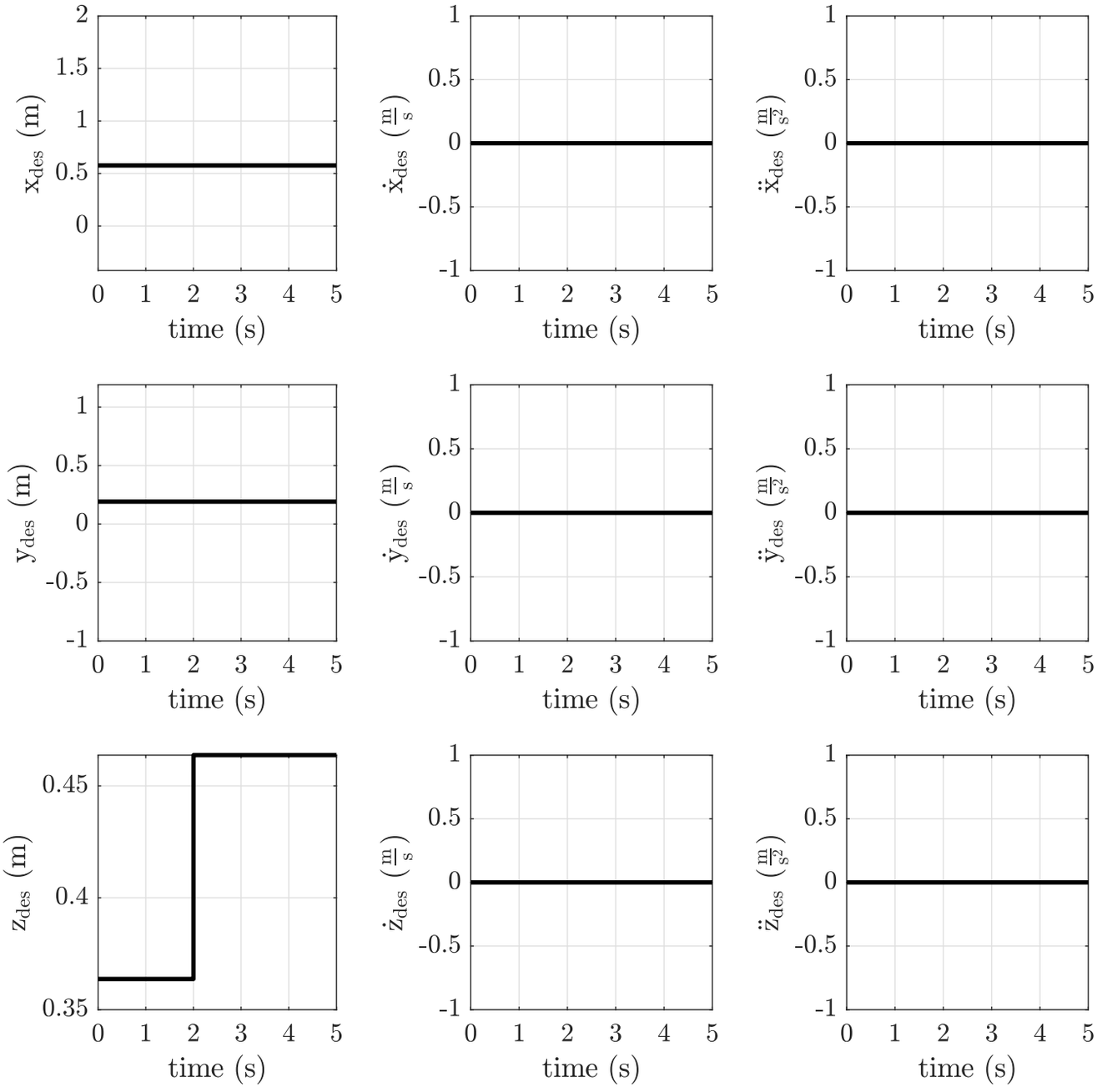

Figure 2: Cartesian reference trajectories for the robot end-effector with Algorithm 2: (a) position, (b) velocity and (c) acceleration.

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). The simulation was stopped after 0.8 seconds due to stability issues.

```
# =========================
#   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 referece points from Cartesian sinusoidal reference trajectory of activity 1.1.

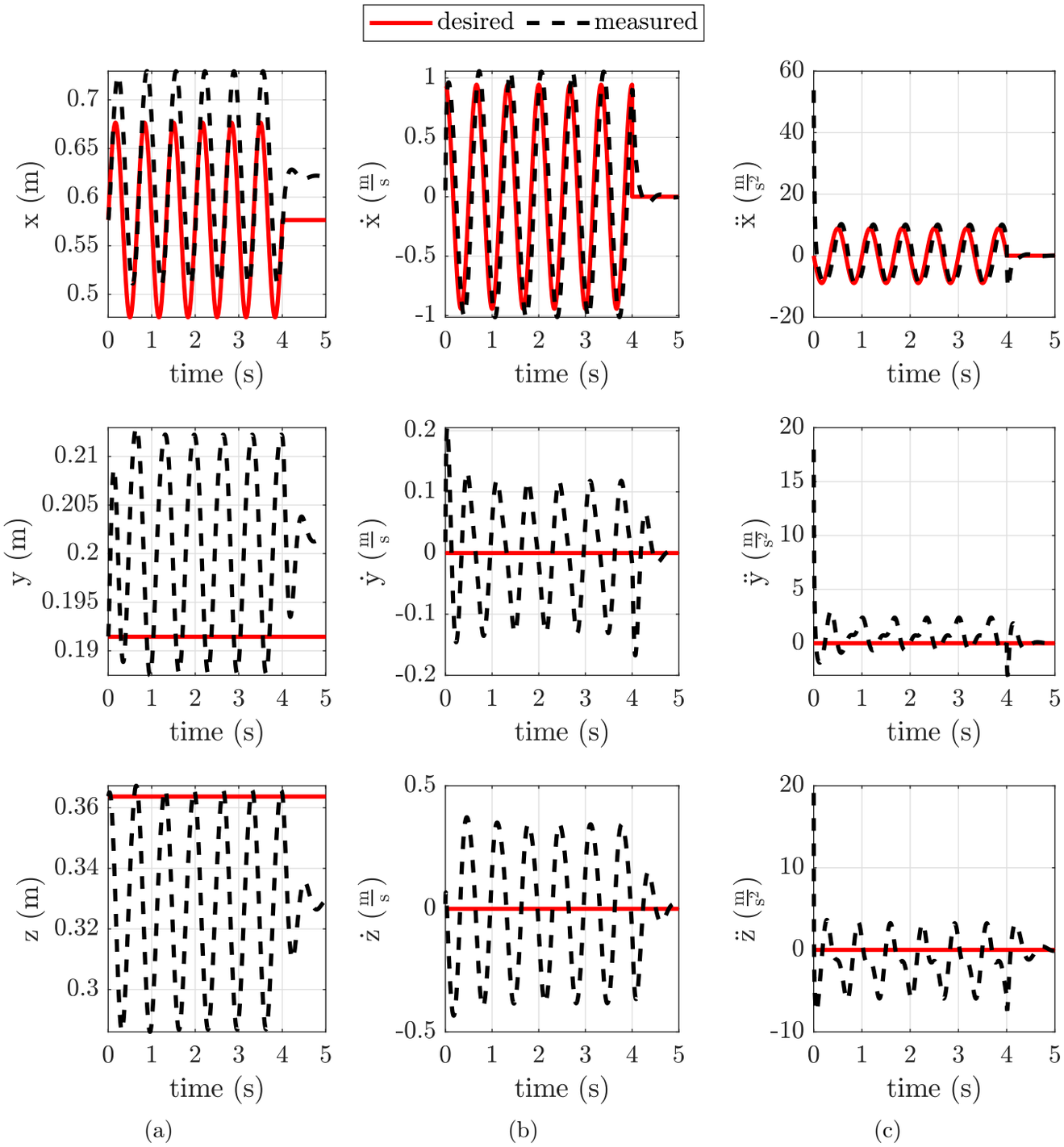

Figure 3: Cartesian trajectory tracking performances using articular proportional-derivative control method, (1), with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$: (a) position, (b) velocity and (c) acceleration.

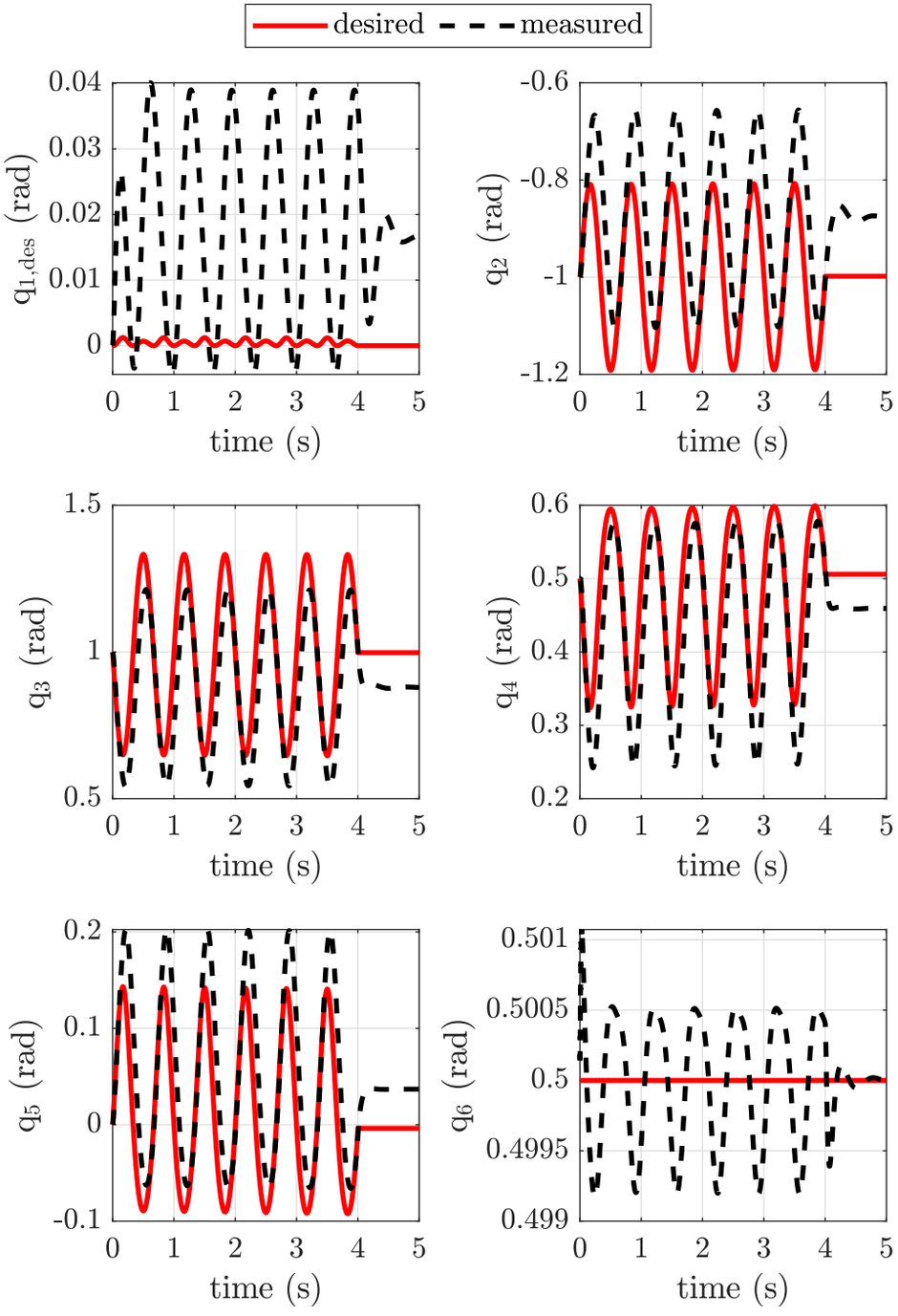

Figure 4: Angular trajectory tracking performances using articular proportional-derivative control method, (1), with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$.

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.

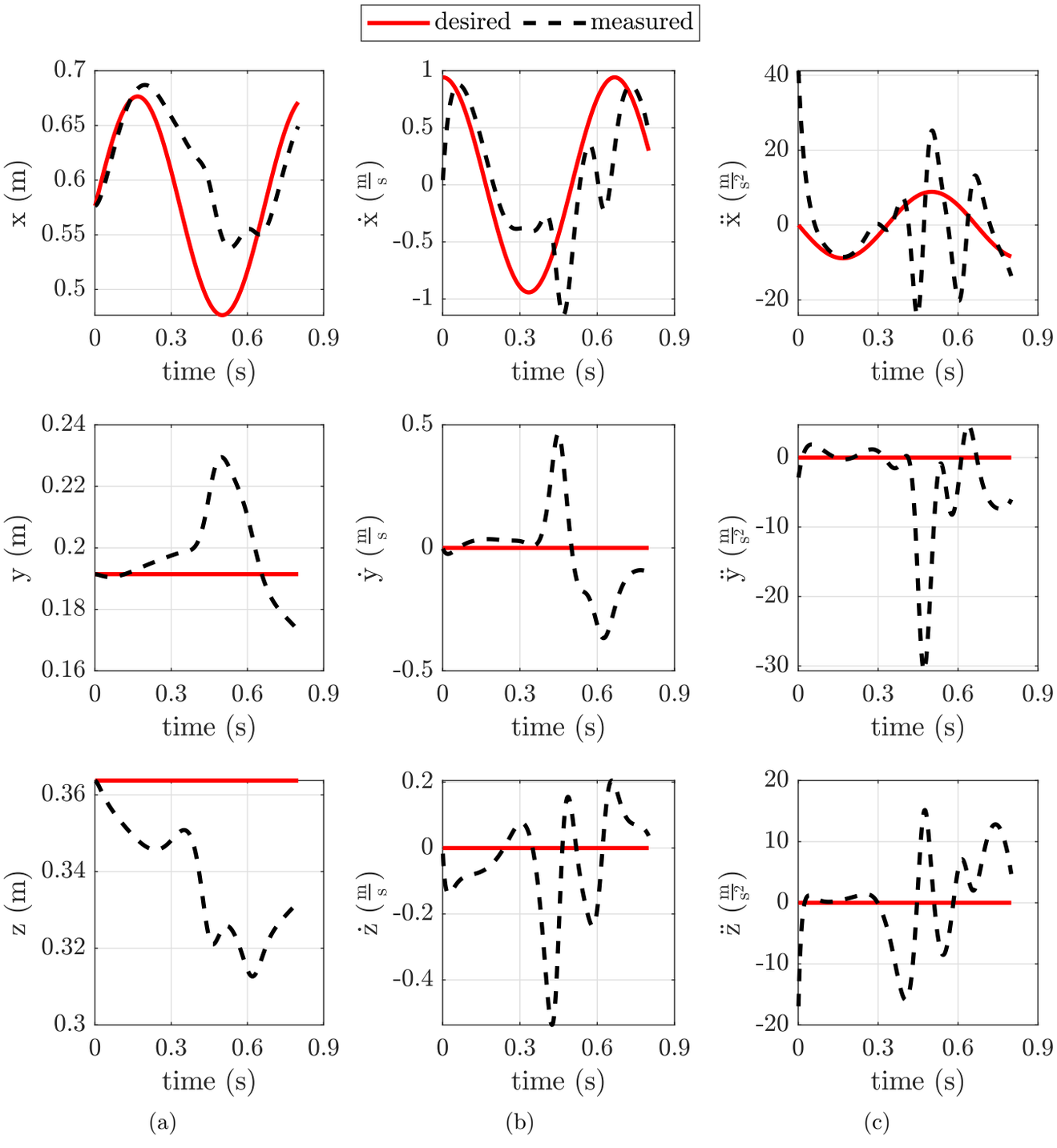

Figure 5: Cartesian trajectory tracking performances using Cartesian proportional-derivative control method, (2), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$ and $K_d = 300\ \frac{\text{N.s}}{\text{m}}$: (a) position, (b) velocity and (c) acceleration.

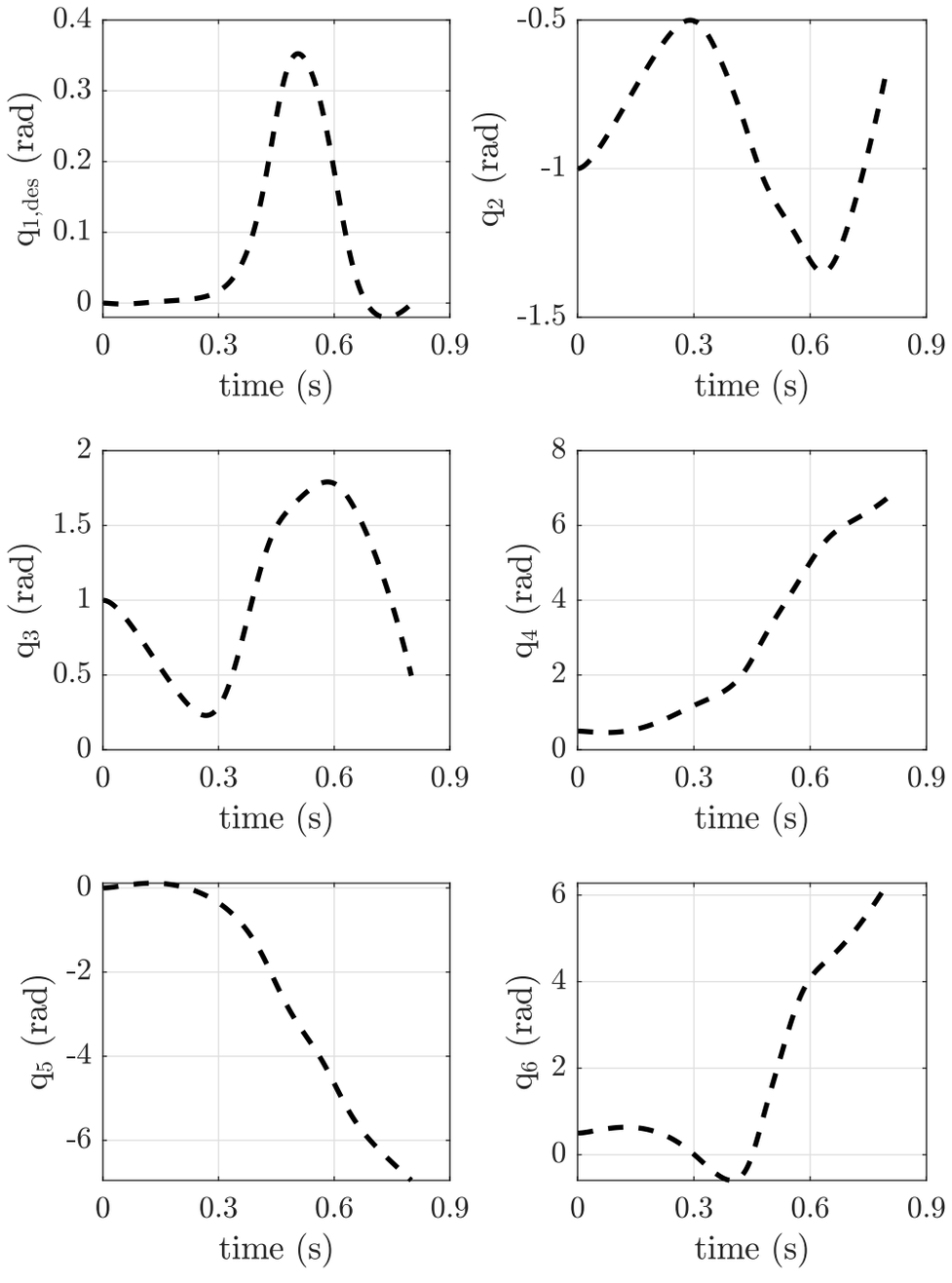

Figure 6: Angular position of each joint of UR5 robot with Algorithm 5.

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.

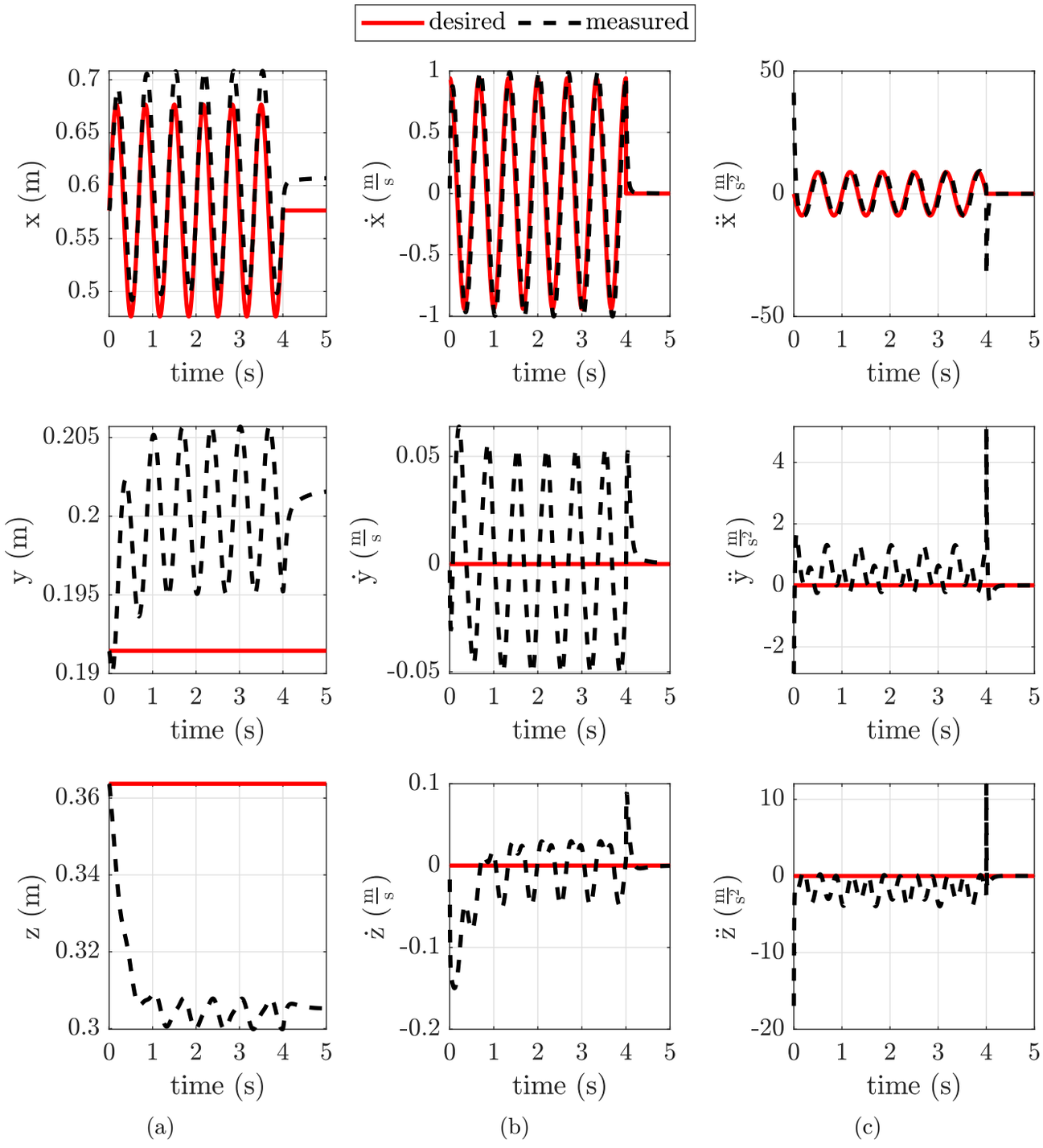

Figure 7: Cartesian trajectory tracking performances using Cartesian proportional-derivative control method and null space projection, (3), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$, $K_d = 10\ \frac{\text{N.m.s}}{\text{rad}}$: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

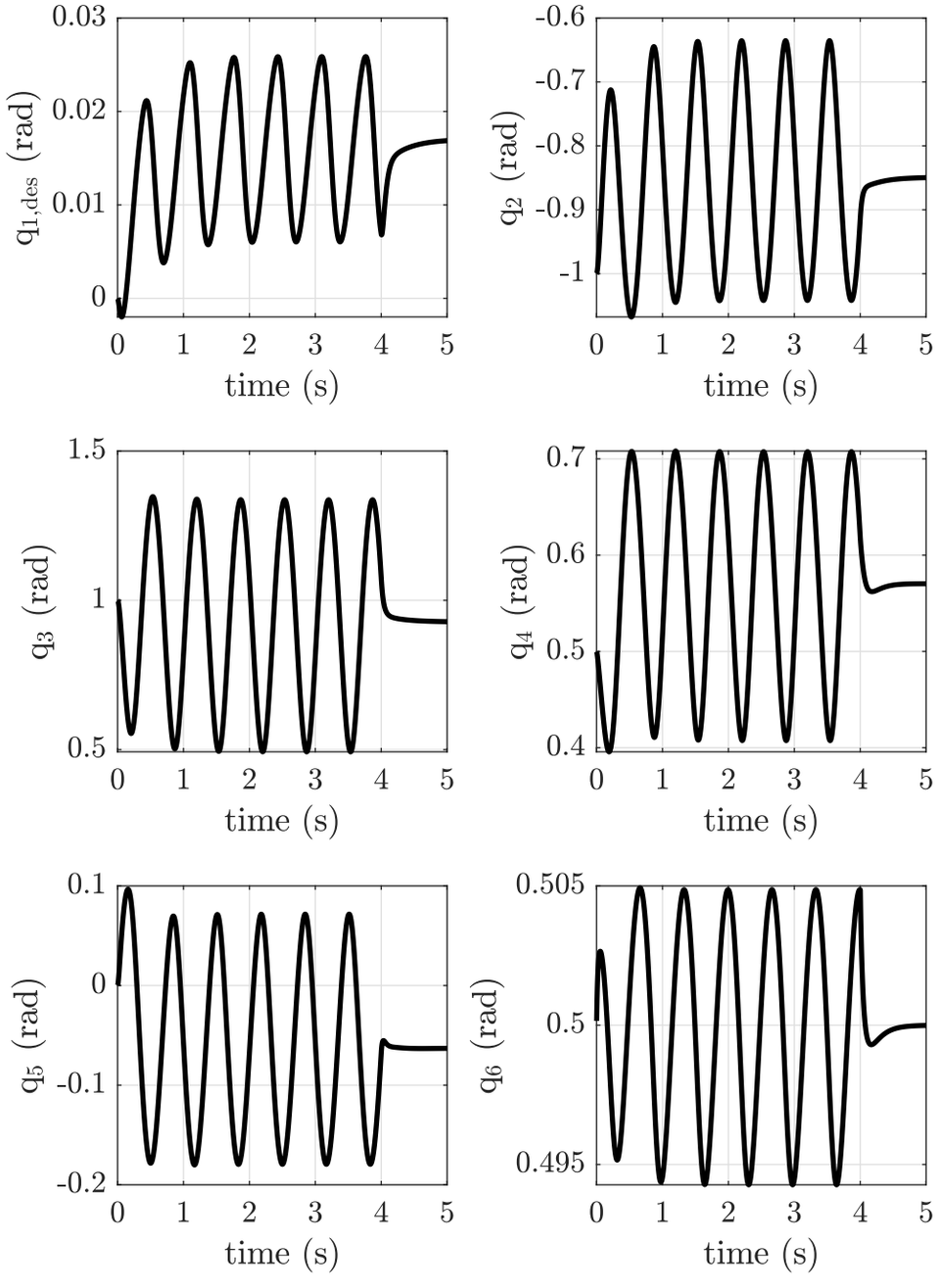

Figure 8: Angular position of each joint of UR5 robot with Algorithm 6.

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.

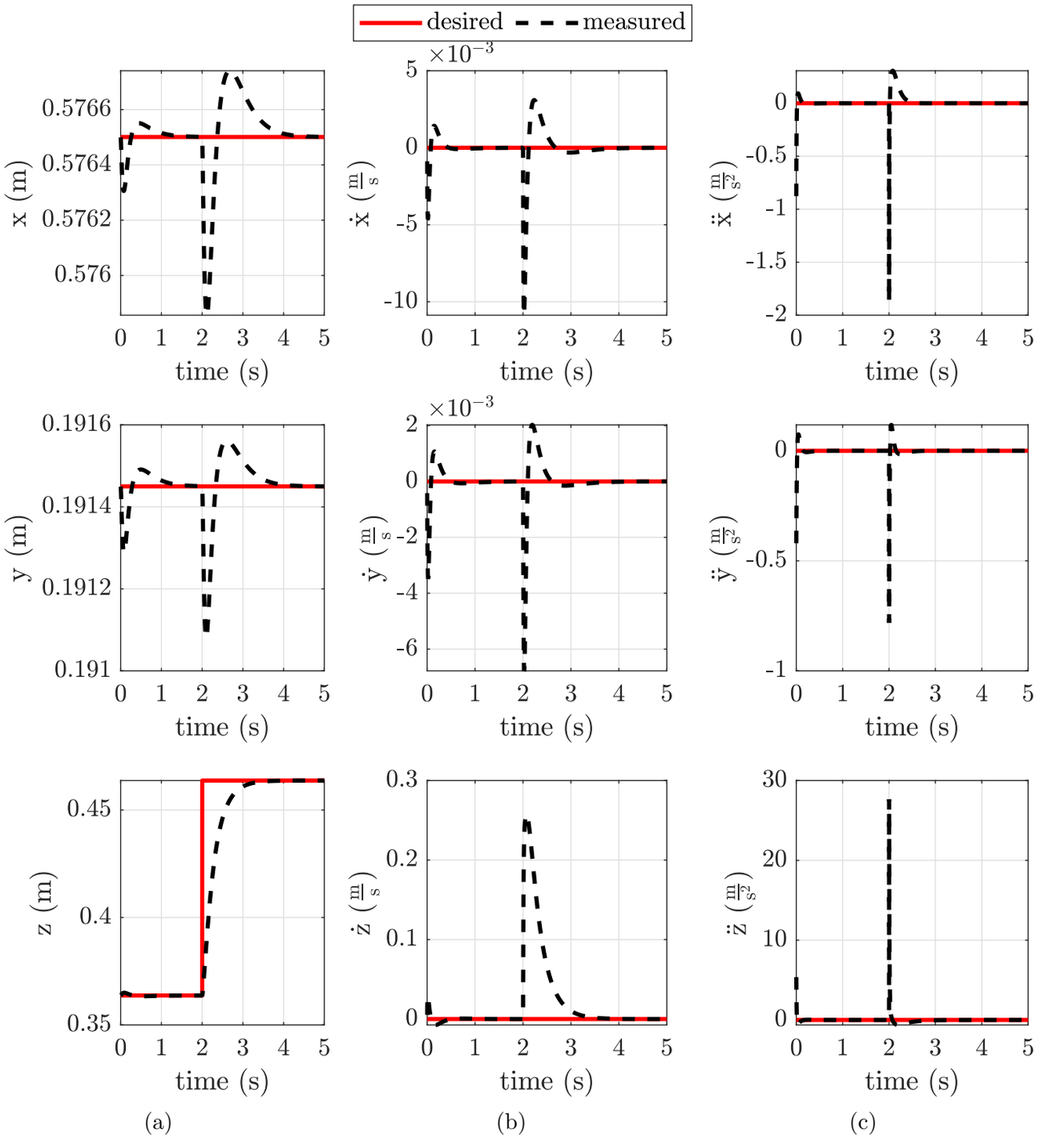

Figure 9: Cartesian trajectory tracking performances using Cartesian proportional-derivative control method with gravity compensation and null space projection, (4), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$, $K_d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

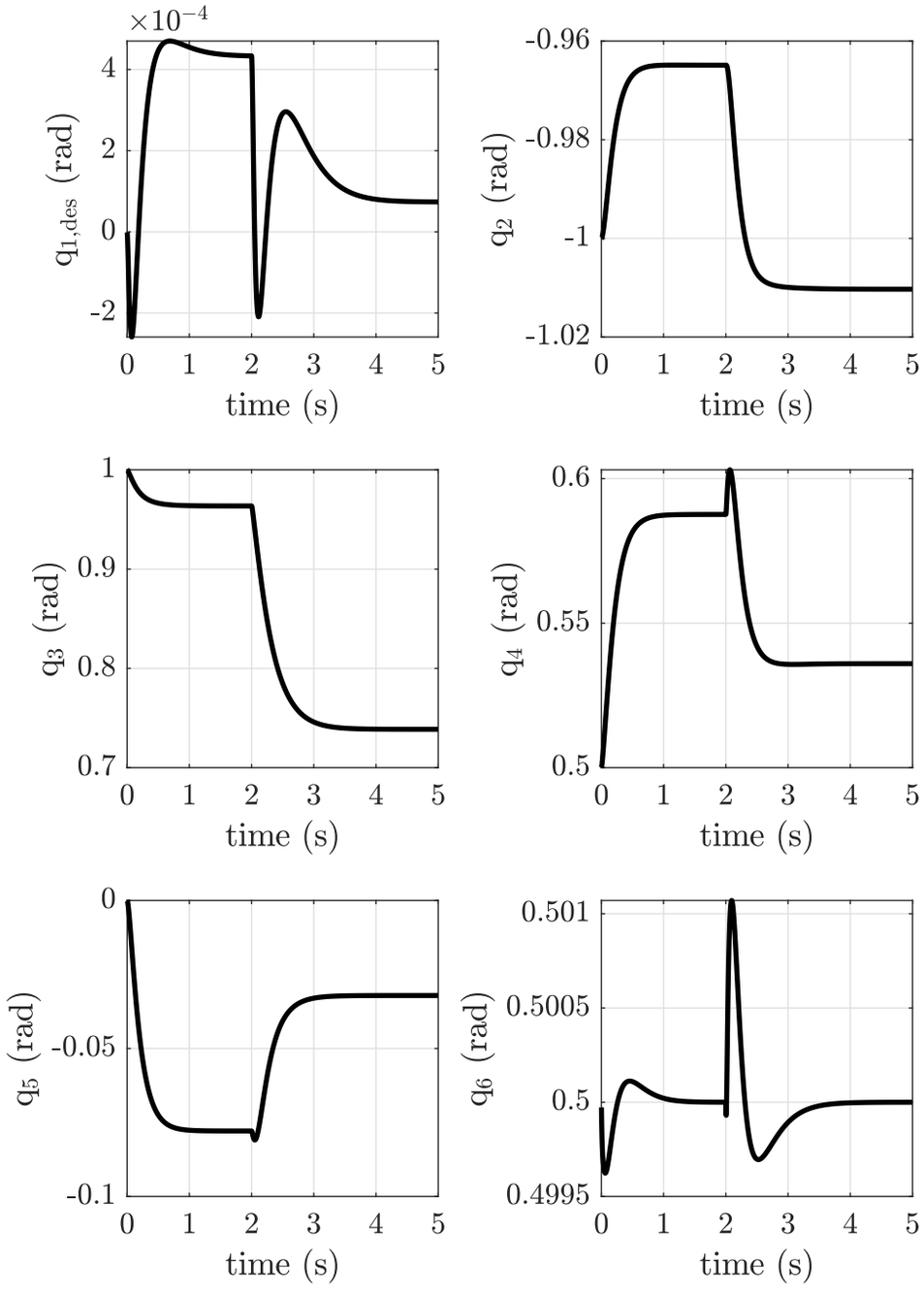

Figure 10: Angular position of each joint of UR5 robot with Algorithm 7.

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.

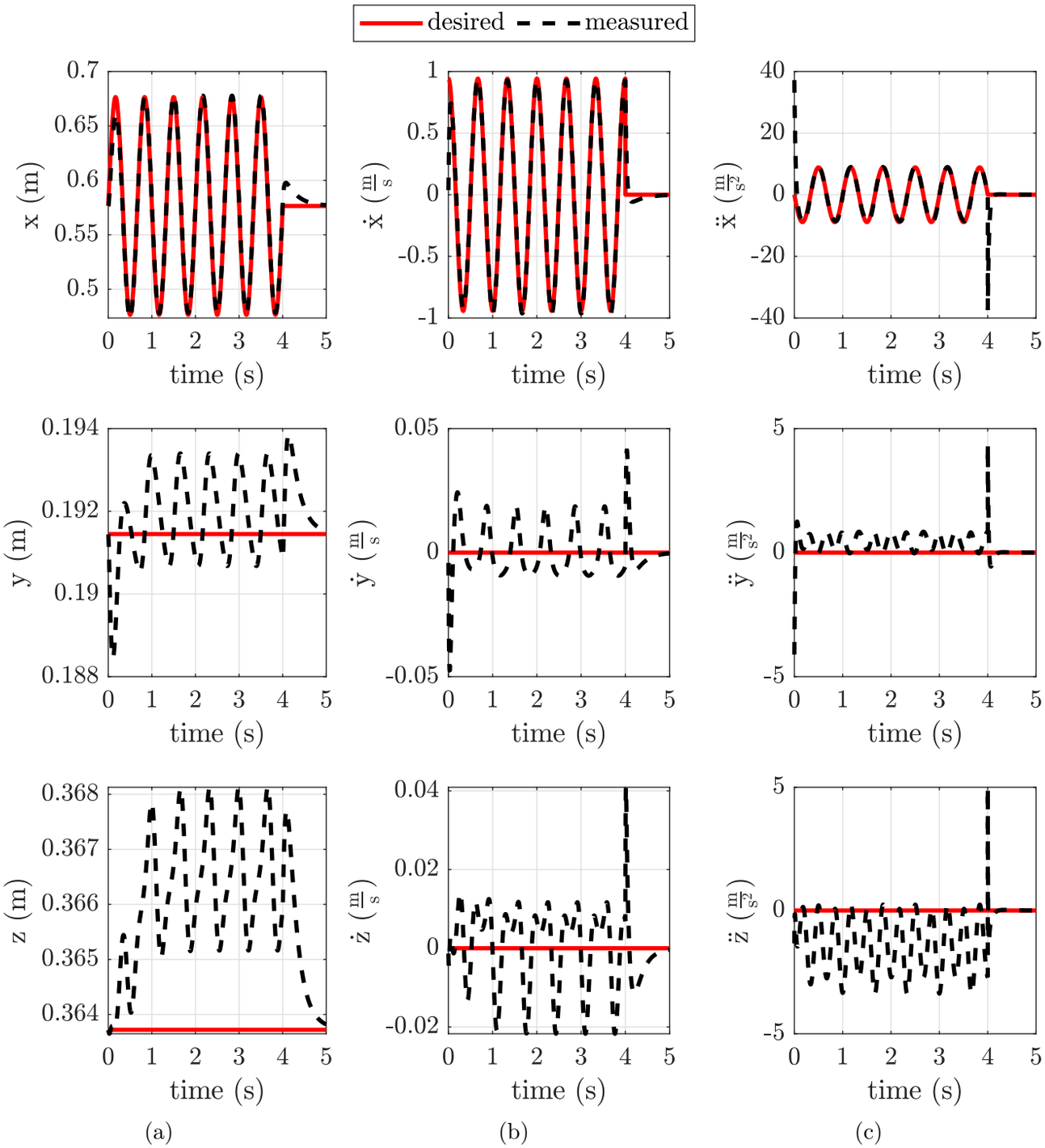

Figure 11: Cartesian trajectory tracking performances using Cartesian proportional-derivative control method with gravity compensation, feed-forward term and null space projection, (5), with $K_p = 1000\ \frac{\text{N}}{\text{m}}$, $K_d = 300\ \frac{\text{N.s}}{\text{m}}$, $K_q = 50\ \frac{\text{N.m}}{\text{rad}}$,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

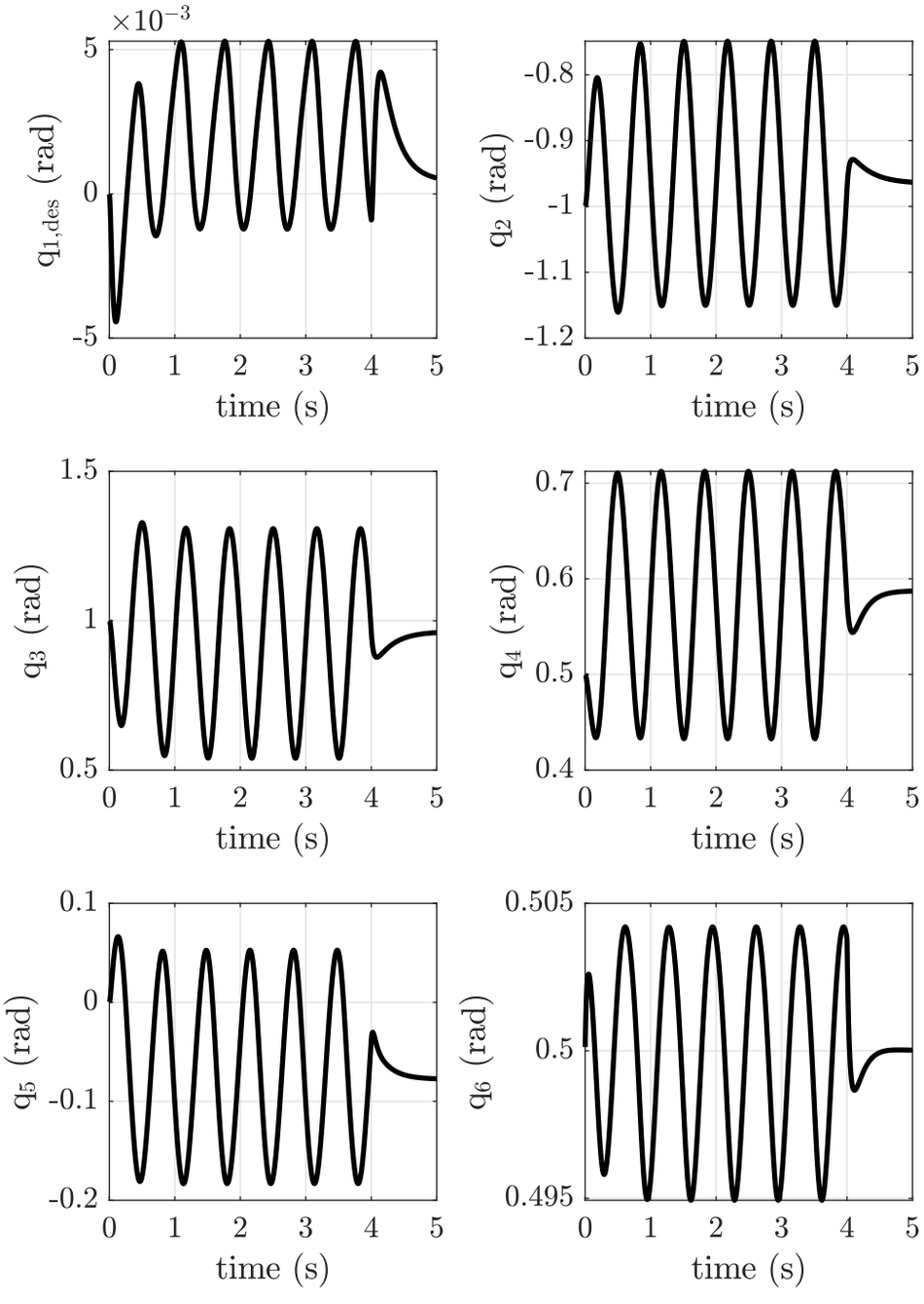

Figure 12: Angular position of each joint of UR5 robot with Algorithm 8.