Name: Jhon Charaja
Laboratory number: 1
Course: Legged robots
Date: October 8, 2021

1 Decentralized control

1.1 Sinusoidal reference generation

The objective of this activity is display the UR5 robot on rviz and move the second and fifth joints with a sinusoidal reference position. The simulation starts with the initial joint configuration $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad and then move the second and fifth joints of the UR5 robot. The two joints will move following a sinusoidal trajectory during the first 4 seconds and maintain a constant joint position during the last second. The function to generate the sinusoidal reference is described in Algorithm 1 and the rosnode file to move the second and fifth joints of UR5 robot with the required movement is described in Algorithm 2. Finally, Figure 1-3 show the joint position, velocity and acceleration of each joint of the UR5 robot.

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

Inputs:
------
    - q0: initial joint position
    - dq0: initial joint velocity
    - ddq0: initial joint acceleration
    - a: amplitude [rad]
    - f: frecuency [hz]
    - t: simulation time [sec]
Outputs:
-------
    - q: angular position
"""
w = 2*np.pi*f                # [rad/s]
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]
```

```
return q, dq, ddq
```

Algorithm 1: Function to generate sinusoidal reference.

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

# 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)  # [Hz]
dt    = 1e-3      # [ms]

# object(message) type JointState
jstate = JointState()

# ============================
#   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']
# number of degress of freedom
ndof = 6

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0   = np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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])

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

while not rospy.is_shutdown():
 # generate sinusoidal joint reference
 if t<=sine_duration:
     # second link
     q_des[1], dq_des[1], ddq_des[1] =
        sinusoidal_reference_generator(q0[1], 0.2, 1, t)
     last_q_des_1 = q_des[1]
```

```
    # fifth link
    q_des[4], dq_des[4], ddq_des[4] =
       sinusoidal_reference_generator(q0[4], 0.4, 1.5, t)
    last_q_des_4 = q_des[4]
else:
    # second link
    q_des[1], dq_des[1], ddq_des[1] = step_reference_generator(0,
       last_q_des_1)
    # fifth link
    q_des[4], dq_des[4], ddq_des[4] = step_reference_generator(0 ,
        last_q_des_4)

  # publish message
  jstate.header.stamp = rospy.Time.now()
  jstate.name   = jnames    # Joints position name
  jstate.position  = q_des     # joint position
  jstate.velocity  = dq_des    # joint velocity
  pub.publish(jstate)

  # update time
  t = t + dt

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

Algorithm 2: Move the second and fifth joint of UR5 robot with the required movement of activity 1.1.

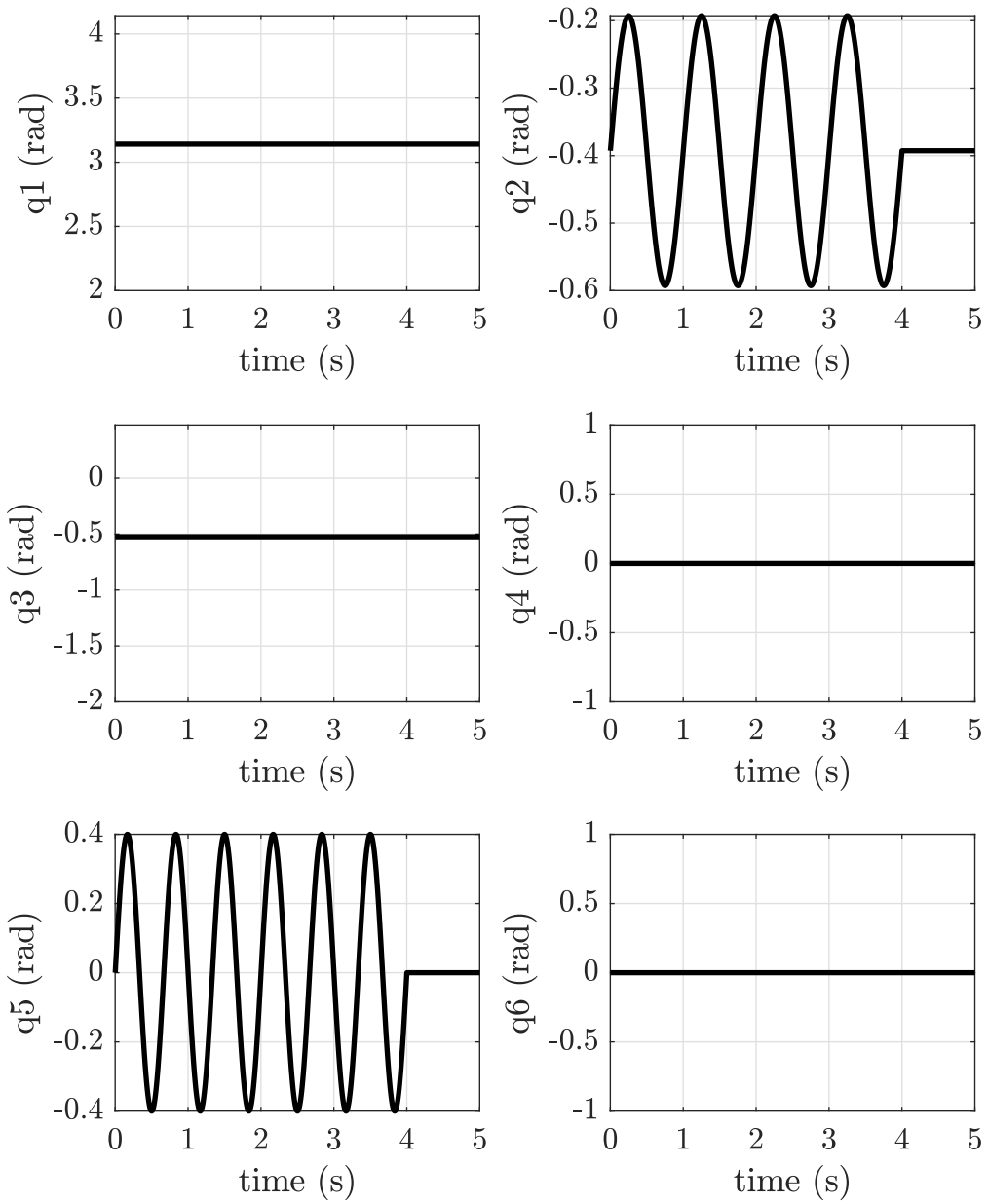


Figure 1: Angular position of each joint of UR5 robot with Algorithm 2.

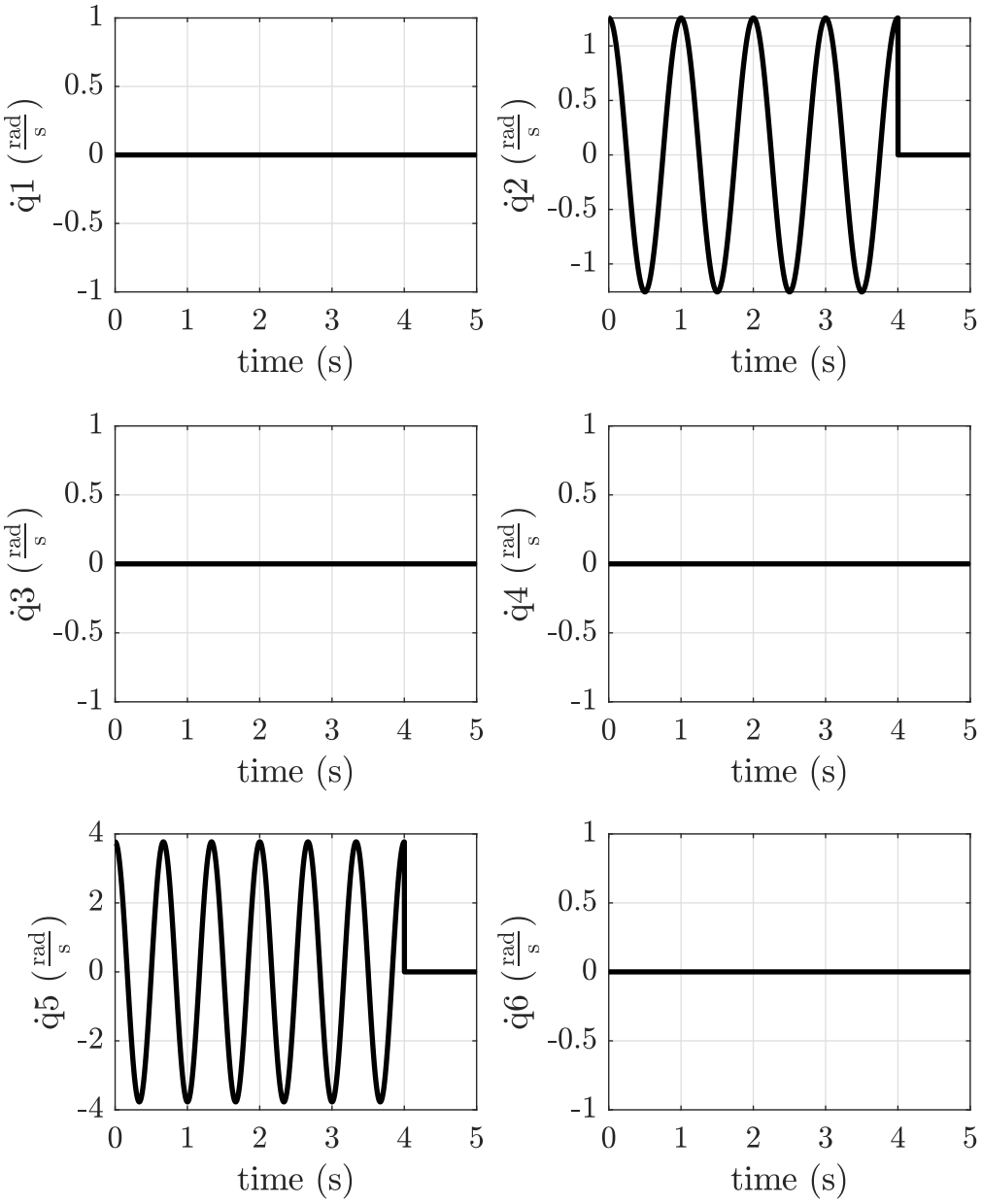


Figure 2: Angular velocity of each joint of UR5 robot with Algorithm 2.

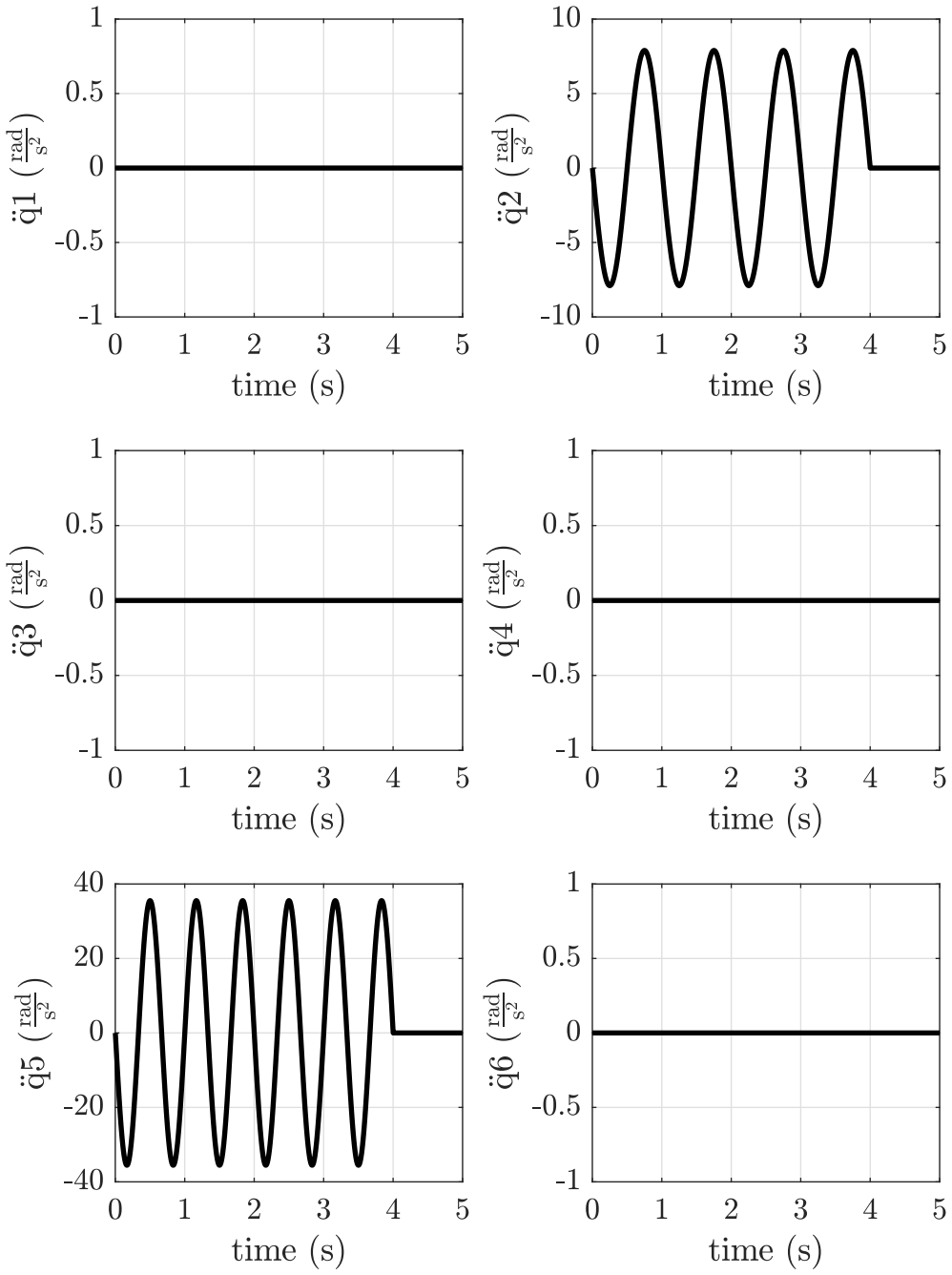


Figure 3: Angular acceleration of each joint of UR5 robot with Algorithm 2.

1.2 Step reference generation

The objective of this activity is display the UR5 robot on rviz and move the second and fifth joints of the UR5 robot with a constant reference position. The simulation starts with the initial joint configuration $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad and then move the second and fifth joints of the UR5 robot. The two joints will maintain the initial configuration during the first 2 seconds and follow a step reference during the last 3 seconds. The function to generate the step reference is described in Algorithm 3 and the rosnode file to move the second and fifth joints of UR5 robot is described in Algorithm 4. Finally, Figure 4-6 show the joint position, velocity and acceleration of each joint of the UR5 robot.

```
def step_reference_generator(q0, a):
    """
    Info: generate a constant reference.

    Inputs:
    ------
        - q0: initial joint position
        - a: constant reference
    Outputs:
        - q: angular position
        - dq: angular velocity
        - ddq: angular acceleration
    """
    q = q0 + a  # [rad]
    dq = 0      # [rad/s]
    ddq = 0     # [rad/s^2]
    return q, dq, ddq
```

Algorithm 3: Function to generate step reference.

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

# 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)  # [Hz]
dt    = 1e-3      # [ms]

# object(message) type JointState
jstate = JointState()

# ============================
#   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']
# number of degress of freedom
```

```
ndof = 6

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0   = np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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])

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

while not rospy.is_shutdown():
    # generate step reference after 2 seconds
    if t>=step_start:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator(
            q0[1],-0.4)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator(
            q0[4], 0.5)

    # publish message
    jstate.header.stamp = rospy.Time.now()
    jstate.name   = jnames    # Joints position name
    jstate.position  = q_des     # joint position
    jstate.velocity  = dq_des    # joint velocity
    pub.publish(jstate)

    # update time
    t = t + dt

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

    rate.sleep()
```

Algorithm 4: Move the second and fifth joint of UR5 robot with the required movement of activity 1.2.

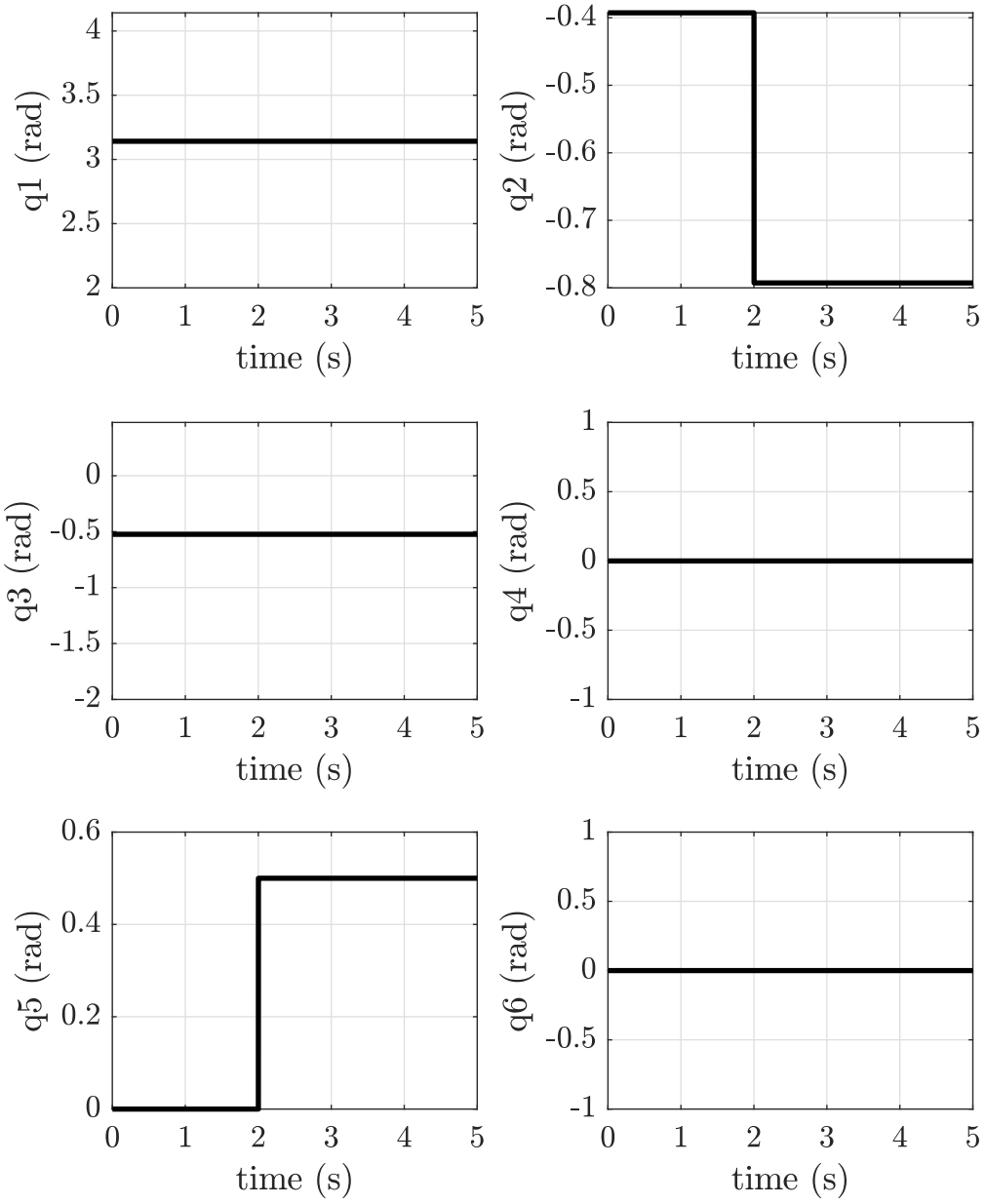


Figure 4: Angular position of each joint of UR5 robot with Algorithm 2.

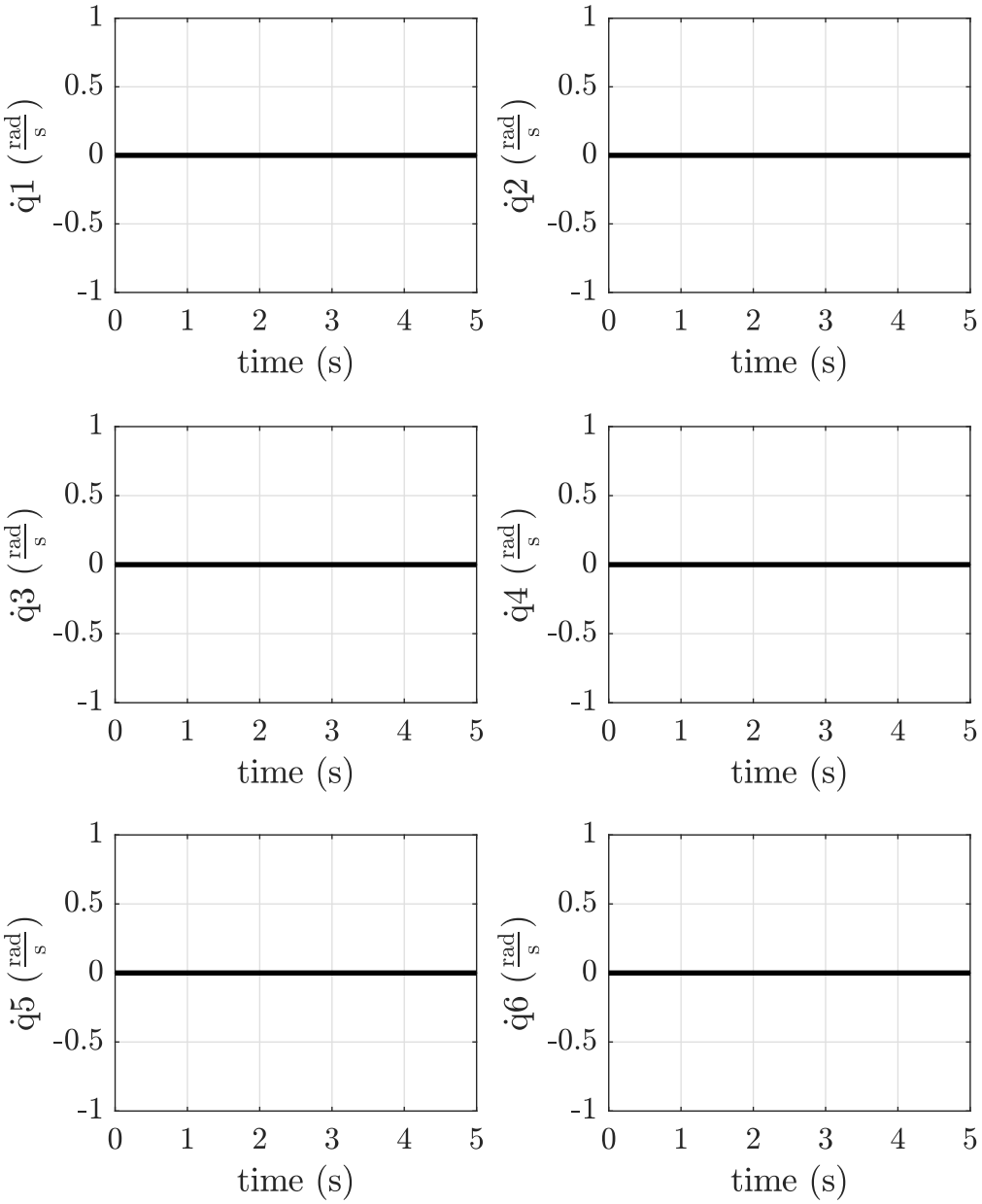


Figure 5: Angular velocity of each joint of UR5 robot with Algorithm 2.

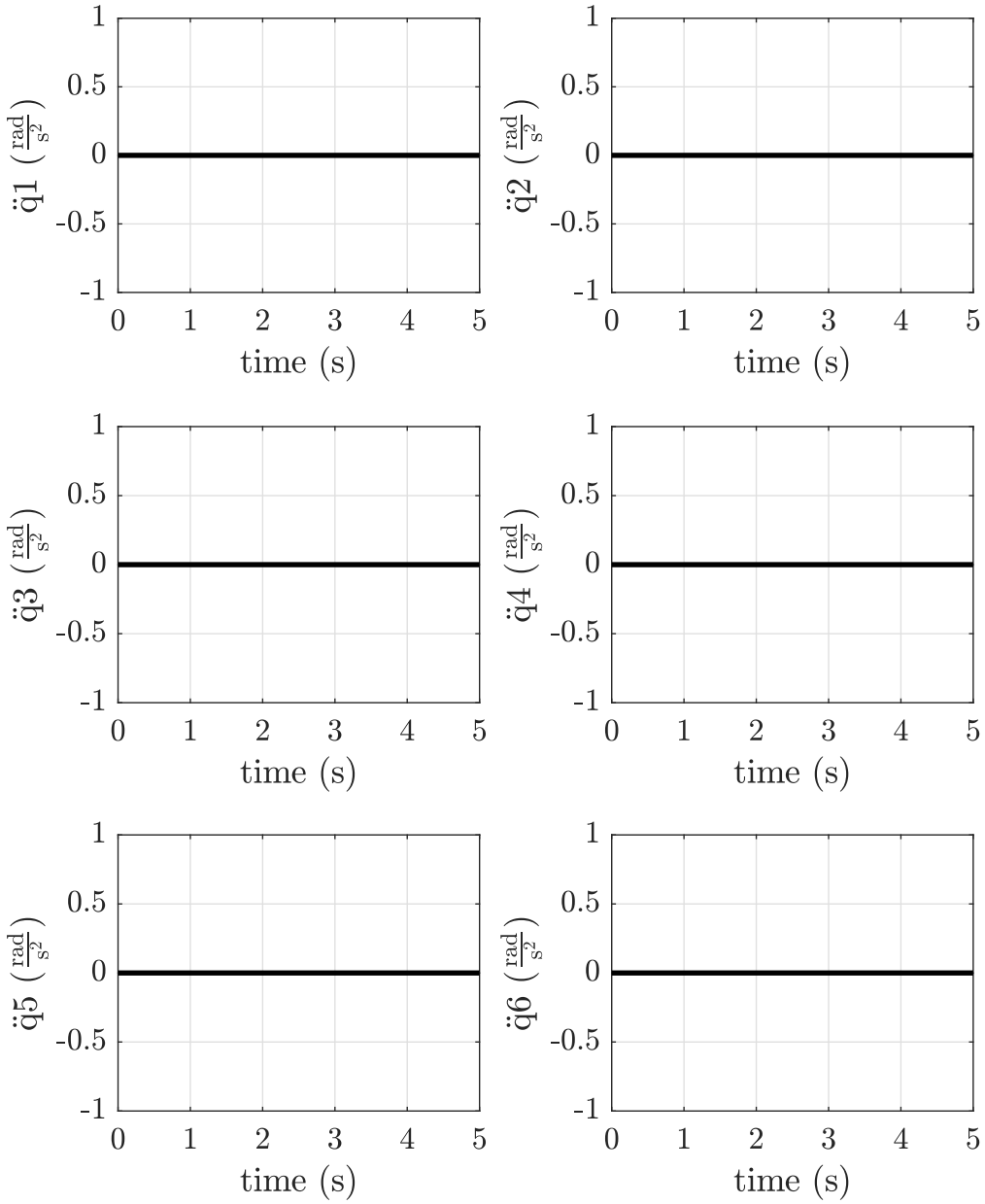


Figure 6: Angular acceleration of each joint of UR5 robot with Algorithm 2.

1.3 Joint PD control

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with a PD control law. The simulation starts with the initial joint configuration $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad and then move the second and fifth joints of the UR5 robot. The two joints will maintain the initial configuration during the first 2 seconds and follow a step reference during the last 3 seconds. Finally, the rosnode file that control the movement of the six joints of UR5 robot is described in Algorithm 5. In this file, the PD control method is configured with $K_p = 300\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 20\ \frac{\text{N.m.s}}{\text{rad}}$.

Figure 7 shows the tracking performance of each joint of the UR5 robot. On one hand, the second joint (q_2) presents overshoot close to 25% and steady state error close to -0.2 rad. On the other hand, the fifth joint (q_5) presents overshoot close to 0% and steady state error close to 0 rad. The variation of the temporal parameters of each joint is due to control gains and physical characteristics of the system. On one hand, the second joint must lift more mass than the fifth joint; and in the same way, the end effector of the robot is further from the second joint than from the fifth joint. In this context, the torque generated by robot's weight is greater in the second joint. On the other hand, control gains do not generate enough torque to compensate for the robot's weight. The torque generated by the robot's weight on the second joint can be calculated as $mgl\sin(\frac{\pi}{2} + \frac{\pi}{8}) \approx -70$ N.m and the torque applied by the control method, at steady state, can be calculated as $K_p\theta_{\text{error}}$. Hence, equating both equations it is obtained that the error in steady state is ≈ -0.2 rad. Finally, the overshoot is due to the value of the derivative gain (k_d), to eliminate the oscillations it is recommended to use $k_d = 2\sqrt{k_p}$.

```
# ==========================
#   Configuration of node
# ==========================
# create a node:
rospy.init_node("node_joint_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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 300*np.ones(ndof)
# derivative gain
kd = 20*np.ones(ndof)
# control vector
tau = np.zeros(ndof)

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

while not rospy.is_shutdown():
    # generate step reference after 2 seconds
    if t>=step_start:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator(
            q0[1], -0.4)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator(
            q0[4], 0.5)

    # error: position and velocity
    e  =  q_des - q
```

```
    de  =  dq_des - dq

    # proportional-derivative control method
    tau = np.multiply(kp, e) + np.multiply(kd, de)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

Algorithm 5: Move the second and fifth joint of UR5 robot with the required movement of activity 1.3.

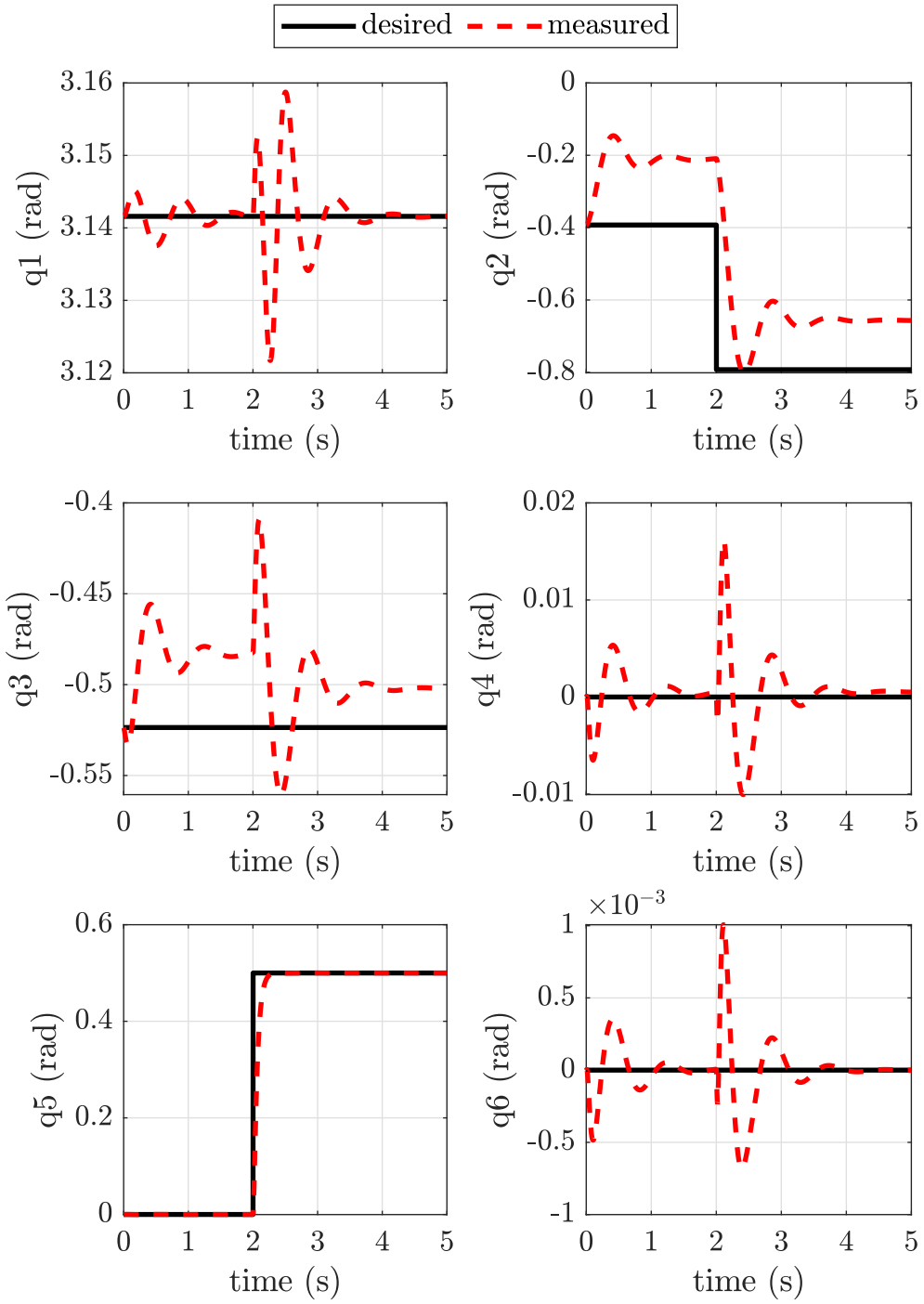


Figure 7: Angular position of each joint of UR5 robot with Algorithm 5.

1.4 Joint PD control - high gains

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with a PD control law. The simulation starts with the initial joint configuration $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad and then move the second and fifth joints of the UR5 robot. The two joints will maintain the initial configuration during the first 2 seconds and follow a step reference during the last 3 seconds. Finally, the rosnode file that control the movement of the six joints of UR5 robot is described in Algorithm 6. In this file, the PD control method is configured with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$.

The Figure 8 shows the tracking performance of each joint of the UR5 robot. On one hand, the second joint (q_2) presents overshoot close to 25% and steady state error close to -0.1 rad. On the other hand, the fifth joint (q_5) does not present overshoot and steady state error close to 0 rad. The variation of the temporal parameters of each joint is due to control gains and physical characteristics of the system. On one hand, the second joint must lift more mass than the fifth joint; and in the same way, the end effector of the robot is further from the second joint than from the fifth joint. In this context, the torque generated by robot's weight is greater in the second joint. On the other hand, control gains do not generate enough torque to compensate for the robot's weight. The torque generated by the robot's weight on the second joint can be calculated as $mgl\sin(\frac{\pi}{2} + \frac{\pi}{8}) \approx -70$ N.m and the torque applied by the control method, at steady state, can be calculated as $K_p\theta_{\text{error}}$. Hence, equating both equations it is obtained that the error in steady state is ≈ -0.1 rad. Finally, the overshoot is due to the value of the derivative gain (k_d), to eliminate the oscillations it is recommended to use $k_d = 2\sqrt{k_p}$.

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

# 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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ==========================================
#   Set initial joint configuration of UR5
# ==========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ================================
#   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]
step_start = 2.0 # [sec]

while not rospy.is_shutdown():
    # generate step reference after 2 seconds
    if t>=step_start:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator(
            q0[1], -0.4)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator(
            q0[4], 0.5)

    # error: position and velocity
    e  =  q_des - q
```

```
    de  =  dq_des - dq

    # proportional-derivative control method
    tau = np.multiply(kp, e) + np.multiply(kd, de)

    # send control signal
    ur5_robot.send_control_command(tau)

    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

Algorithm 6: Move the second and fifth joint of UR5 robot with the required movement of activity 1.4.

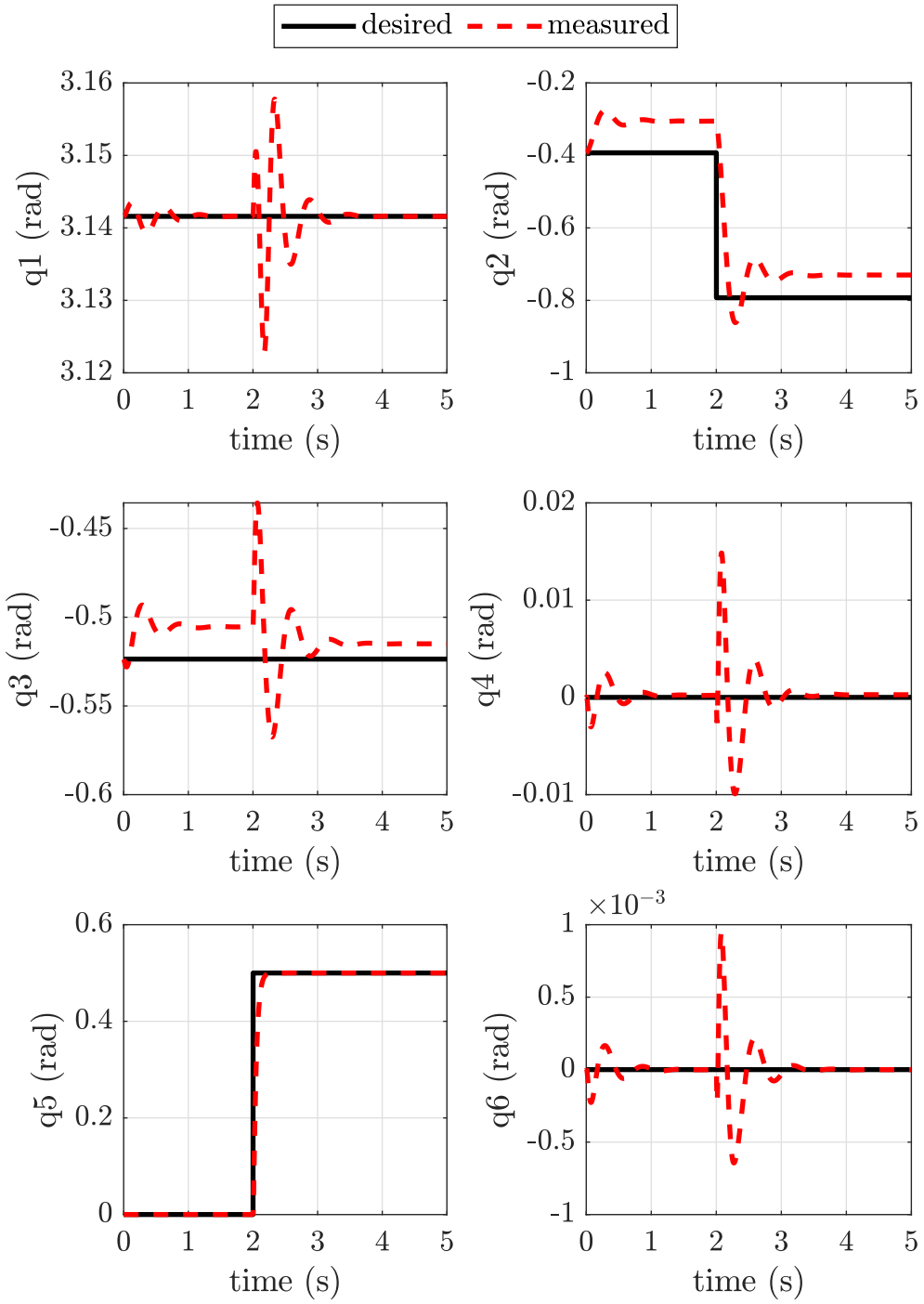


Figure 8: Angular position of each joint of UR5 robot with Algorithm 6.

1.5 Joint PD control - critical damping

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with a PD control law. The simulation starts with the initial joint configuration $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad and then move the second and fifth joints of the UR5 robot. On one hand, the two joints will maintain the initial configuration during first 2 seconds and follow a step reference during last 3 seconds. On the other hand, the two joints will move following a sinusoidal trajectory during first 4 seconds and maintain a constant joint position during last second. The performance of the control method will be evaluated in next subsections.

1.5.1 Step reference

The Algorithm 7control the movements of second and fifth joints of the UR5 robot to follow a step reference trajectory. In this file, the PD control method is configured with $K_p = 300\ \frac{\text{N.m}}{\text{rad}}$ and $K_{d,i} = 2\sqrt{k_{p,i}m_{i,i}}\ \frac{\text{N.m.s}}{\text{rad}}$. Figure 9 shows the tracking performance of each joint of the UR5 robot. On one hand, the second joint (q_2) presents overshoot close to 0% and steady state error close to -0.18 rad. On the other hand, the fifth joint (q_5) presents overshoot close to 0% and steady state error close to 0 rad. The variation of the temporal parameters of each joint is due to the control gains and the physical characteristics of the system. On one hand, the second joint must lift more mass than the fifth joint; and in the same way, the end effector of the robot is further from the second joint than from the fifth joint. In this context, the torque generated by robot's weight is greater in the second joint. On the other hand, control gains do not generate enough torque to compensate for the robot's weight. The torque generated by the robot's weight on the second joint can be calculated as $mgl\sin(\frac{\pi}{2}+\frac{\pi}{8}) \approx -70$ N.m and the torque applied by the control method, at steady state, can be calculated as $K_p\theta_{\text{error}}$. Hence, equating both equations it is obtained that the error in steady state is ≈ -0.2 rad. Finally, the Figure 10 show the variation of the derivative gain (k_d) of PD control method with critical damping approach when the reference of each joint is a constant value. The new value of derivative gain of the second joint is 67.5 $\frac{\text{N.m.s}}{\text{rad}}$ and of the fifth joint is 17.5 $\frac{\text{N.m.s}}{\text{rad}}$.

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

# 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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

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

```
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 300*np.ones(ndof)
# control vector
tau = np.zeros(ndof)

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

while not rospy.is_shutdown():
    # generate step reference after 2 seconds
    if t>=step_start:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator(
            q0[1], -0.4)
```

```
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator(
            q0[4], 0.5)

    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    M = ur5_robot.get_M()

    # update value of kd with critical damping approach
    # equation: kd_{i,i} = M_{i,i}*2*sqrt(kd_{i,i})
    kd = 2*np.sqrt(np.multiply(kp, M.diagonal()))


    # proportional-derivative control method
    tau = np.multiply(kp, e) + np.multiply(kd, de)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

Algorithm 7: Move the second and fifth joint of UR5 robot with the requirement motion of activity 1.5.1

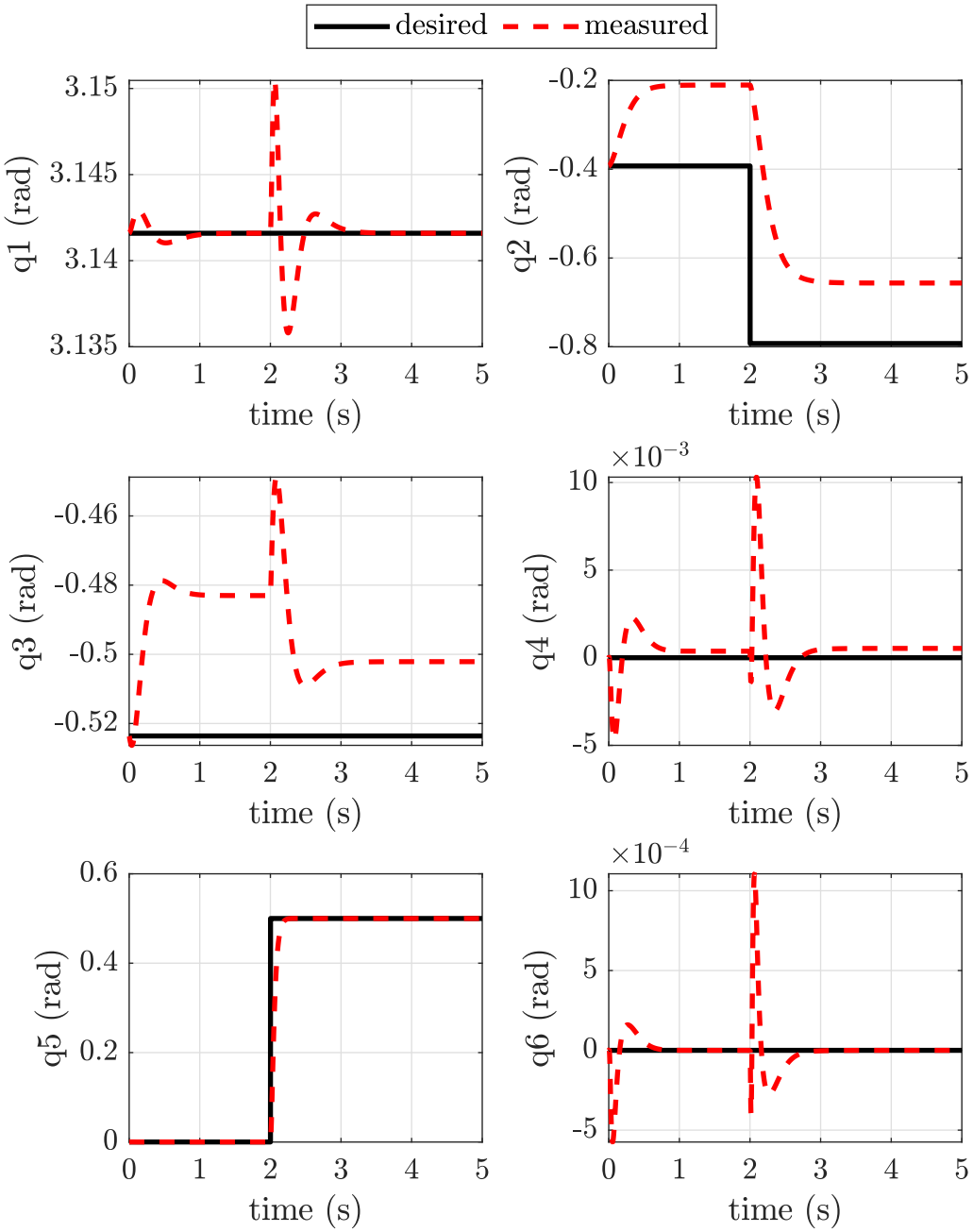


Figure 9: Angular position of each joint of UR5 robot with Algorithm 7.

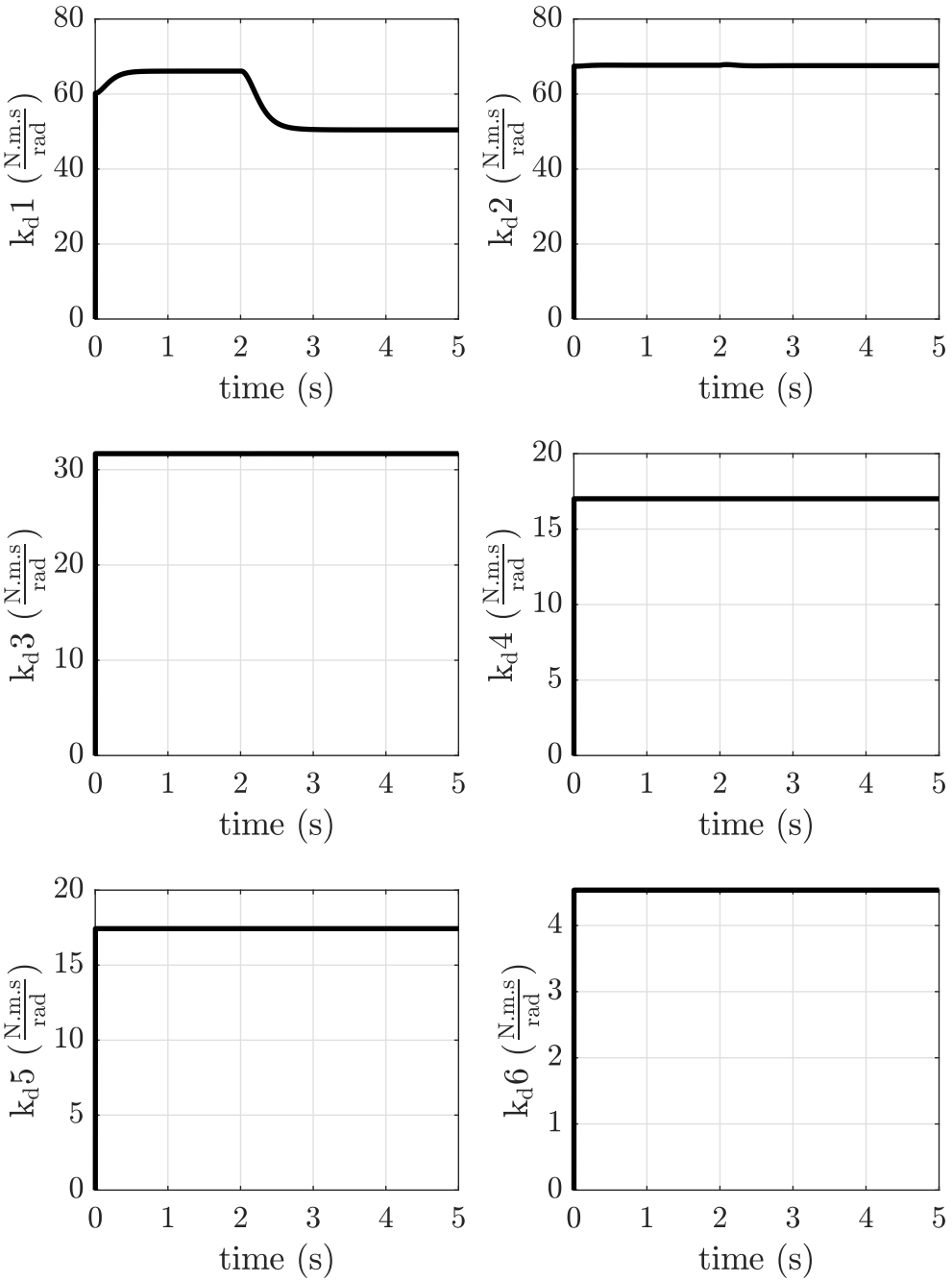


Figure 10: The derivative gain (k_d) of the PD control method with critical damping formulation when the reference of each joint is a constant value .

1.5.2 Sinusoidal reference

The Algorithm 8 control the movements of second and fifth joints of the UR5 robot to follow a sinusoidal reference trajectory. In this file, the PD control method is configured with $K_p = 300 \frac{\text{N.m}}{\text{rad}}$ and $K_{d,i} = 2\sqrt{k_{p,i} m_{i,i}} \frac{\text{N.m.s}}{\text{rad}}$. Figure 9 shows the tracking performance of each joint of the UR5 robot.

Figure 11 shows the tracking performance of each joint of the UR5 robot. On one hand, the second joint (q_2) presents overshoot close to 0% and steady state error close to -0.2 rad. On the other hand, the fifth joint (q_5) presents overshoot close to 0% and steady state error close to 0 rad. The variation of the temporal parameters of each joint is due to the control gains and the physical characteristics of the system. On one hand, the second joint must lift more mass than the fifth joint; and in the same way, the end effector of the robot is further from the second joint than from the fifth joint. In this context, the torque generated by robot's weight is greater in the second joint. On the other hand, control gains do not generate enough torque to compensate for the robot's weight. The torque generated by the robot's weight on the second joint can be calculated as $mgl \sin(\frac{\pi}{2} + \frac{\pi}{8}) \approx -70$ N.m and the torque applied by the control method, at steady state, can be calculated as $K_p \theta_{\text{error}}$. Hence, equating both equations it is obtained that the error in steady state is ≈ -0.2 rad. Finally, the Figure 12 show the variation of the derivative gain (k_d) of PD control method with critical damping approach when the reference of second and fifth joint is a sinusoidal. The new value of derivative gain of the second joint is 67.5 $\frac{\text{N.m.s}}{\text{rad}}$ with oscillations of $\pm$ 0.5 $\frac{\text{N.m.s}}{\text{rad}}$ and of the fifth joint is 17.5 $\frac{\text{N.m.s}}{\text{rad}}$.

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

# 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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 300*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():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
           sinusoidal_reference_generator(q0[1], dq0[1], ddq0[1],
           0.2, 1, t)
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
           sinusoidal_reference_generator(q0[4], dq0[4], ddq0[4],
           0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    # maintain final angular position
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
```

```
            (0, last_q_des_1)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
            (0 , last_q_des_4)

    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    M = ur5_robot.get_M()

    # update value of kd with critical damping approach
    # equation: kd_{i,i} = 2*sqrt(kd_{i,i} * M_{i,i})
    kd = 2*np.sqrt(np.multiply(kp, M.diagonal()))

    # proportional-derivative control method
    tau = np.multiply(kp, e) + np.multiply(kd, de)

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

Algorithm 8: Move the second and fifth joint of UR5 robot with the requirement motion of activity 1.5.2

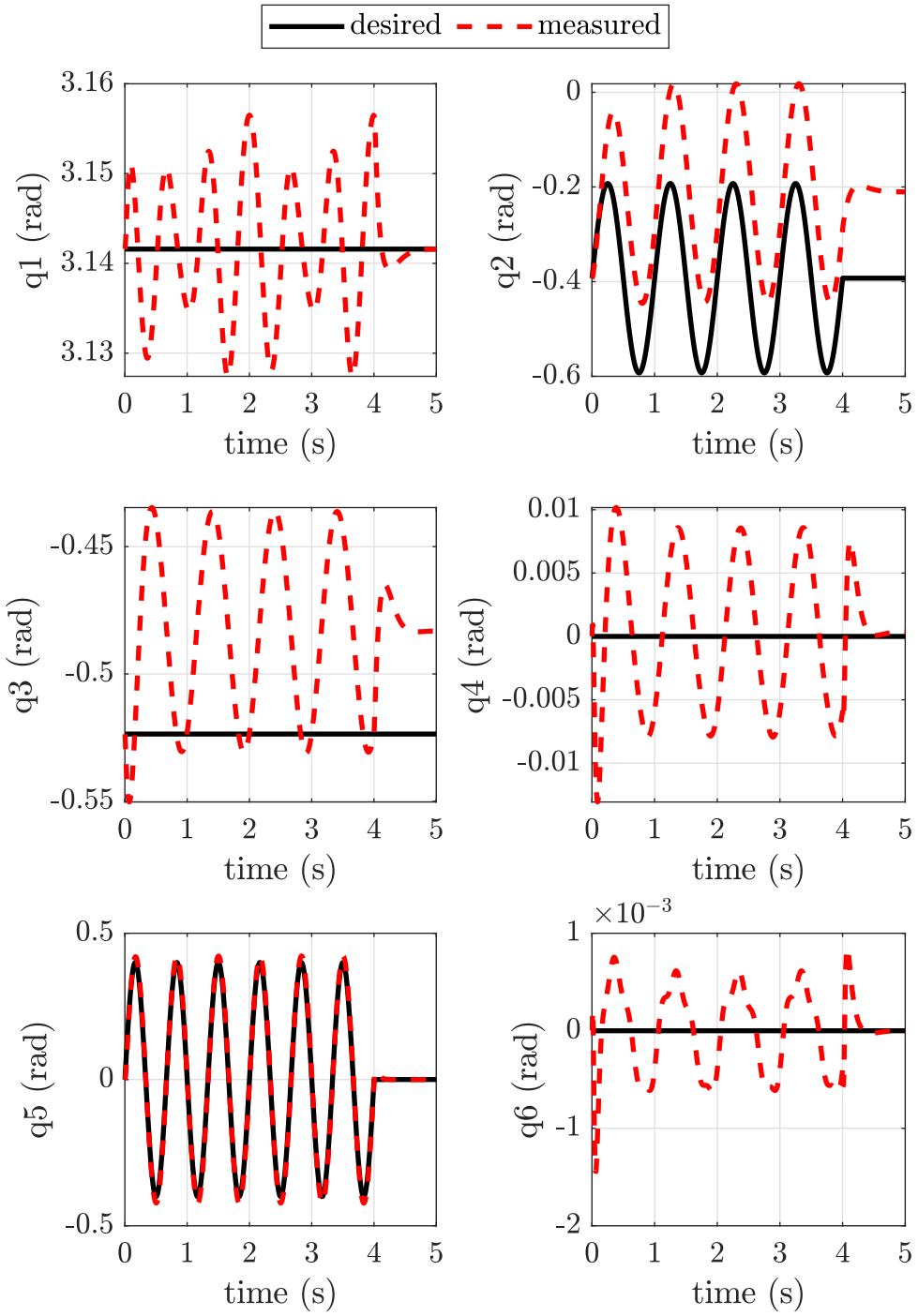


Figure 11: Angular position of each joint of UR5 robot with Algorithm 8.

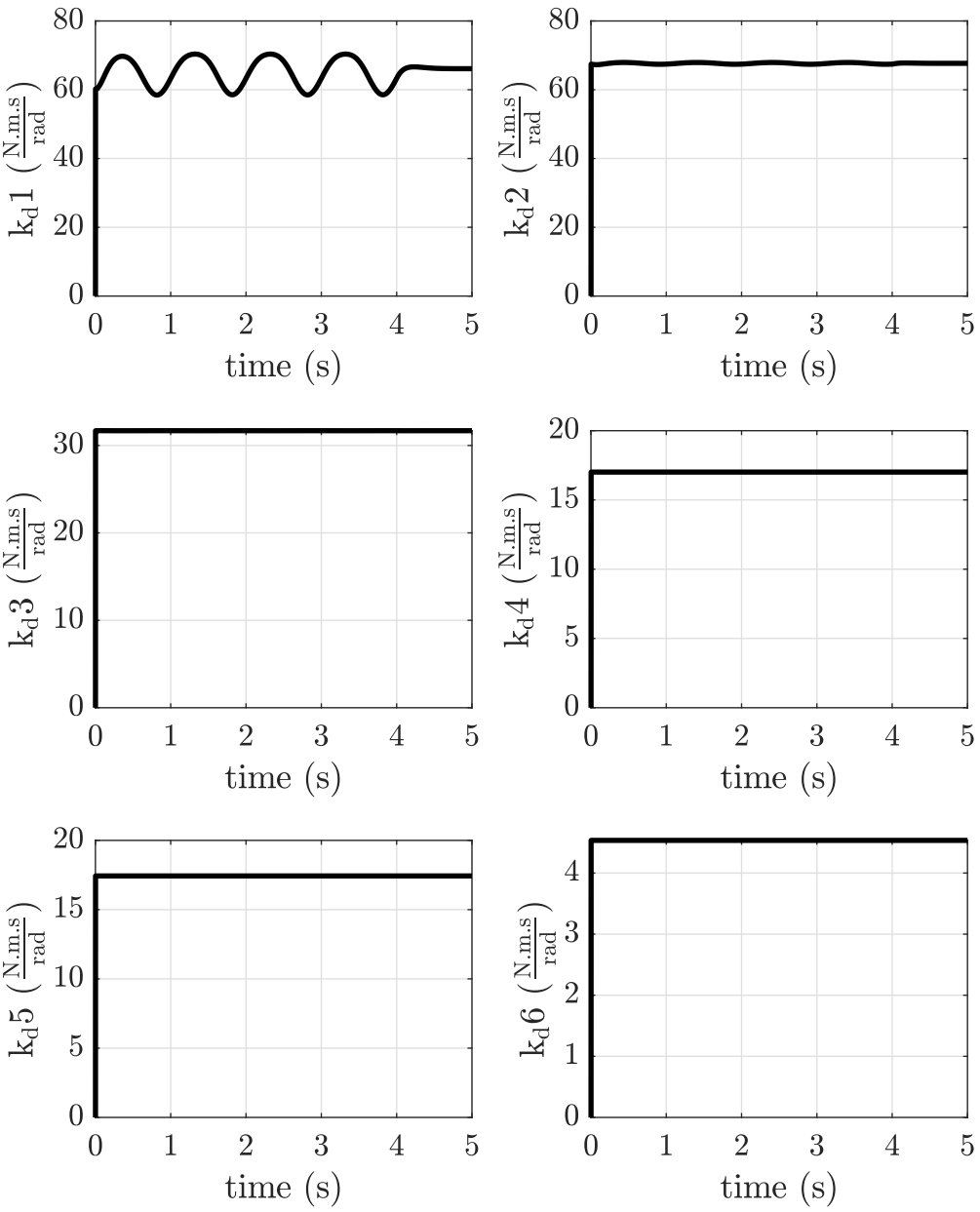


Figure 12: The derivative gain (k_d) of the PD control method with critical damping formulation when the reference of second and fifth joint is sinusoidal.

1.6 Joint PD control + gravity compensation

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with a PD with gravity compensation control method. The simulation starts with the initial joint configuration $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad and then move the second and fifth joints of the UR5 robot. On one hand, the two joints will maintain the initial configuration during first 2 seconds and follow a step reference during last 3 seconds. On the other hand, the two joints will move following a sinusoidal trajectory during first 4 seconds and maintain a constant joint position during last second. The performance of the control method will be evaluated in next subsections.

1.6.1 Step reference

The Algorithm 9 control the movements of second and fifth joints of the UR5 robot to follow a step reference trajectory. In this file, the PD with gravity compensation control method is configured with $K_p = 300\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 20\ \frac{\text{N.m.s}}{\text{rad}}$. Figure 13 shows the tracking performance of each joint of the UR5 robot. On one hand, the second joint (q_2) presents overshoot (15%) and steady state error close to 0 rad. On the other hand, the fifth joint (q_5)presents overshoot close to 0% and steady state error close to 0 rad. The variation of the temporal parameters of each joint is due to the control gains and the physical characteristics of the system. On one hand, the second joint must lift more mass than the fifth joint; and in the same way, the end effector of the robot is further from the second joint than from the fifth joint. In this context, the torque generated by robot's weight is greater in the second joint. On the other hand, the tracking error at transient state is due to the effect of Coriolis and centripetal forces. On the other hand, the tracking error at steady state is close to 0 rad due to gravity term in control law is compensating the forces generated by UR5 robot weight. Finally, the Figure 14 show the variation of the gravity term (g) of the control method with a step reference.

```
# ==========================
#   Configuration of node
# ==========================
# create a node:
rospy.init_node("node_joint_PD_control_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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

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

```
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ==============================
#   PD controller configuration
# ==============================
# proportional gain
kp = 300*np.ones(ndof)
# derivative gain
kd = 20*np.ones(ndof)
# control vector
tau = np.zeros(ndof)

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

while not rospy.is_shutdown():
    # generate step reference after 2 seconds
    if t>=step_start:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator(
           q0[1], -0.4)
        # fifth link
```

```
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator(
            q0[4], 0.5)

    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute gravitational effects vector
    g = ur5_robot.get_g()

    # PD control method + gravity compensation
    tau = np.multiply(kp, e) + np.multiply(kd, de) + g

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

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

Algorithm 9: Move the second and fifth joint of UR5 robot with the requirement motion of activity 1.6.1

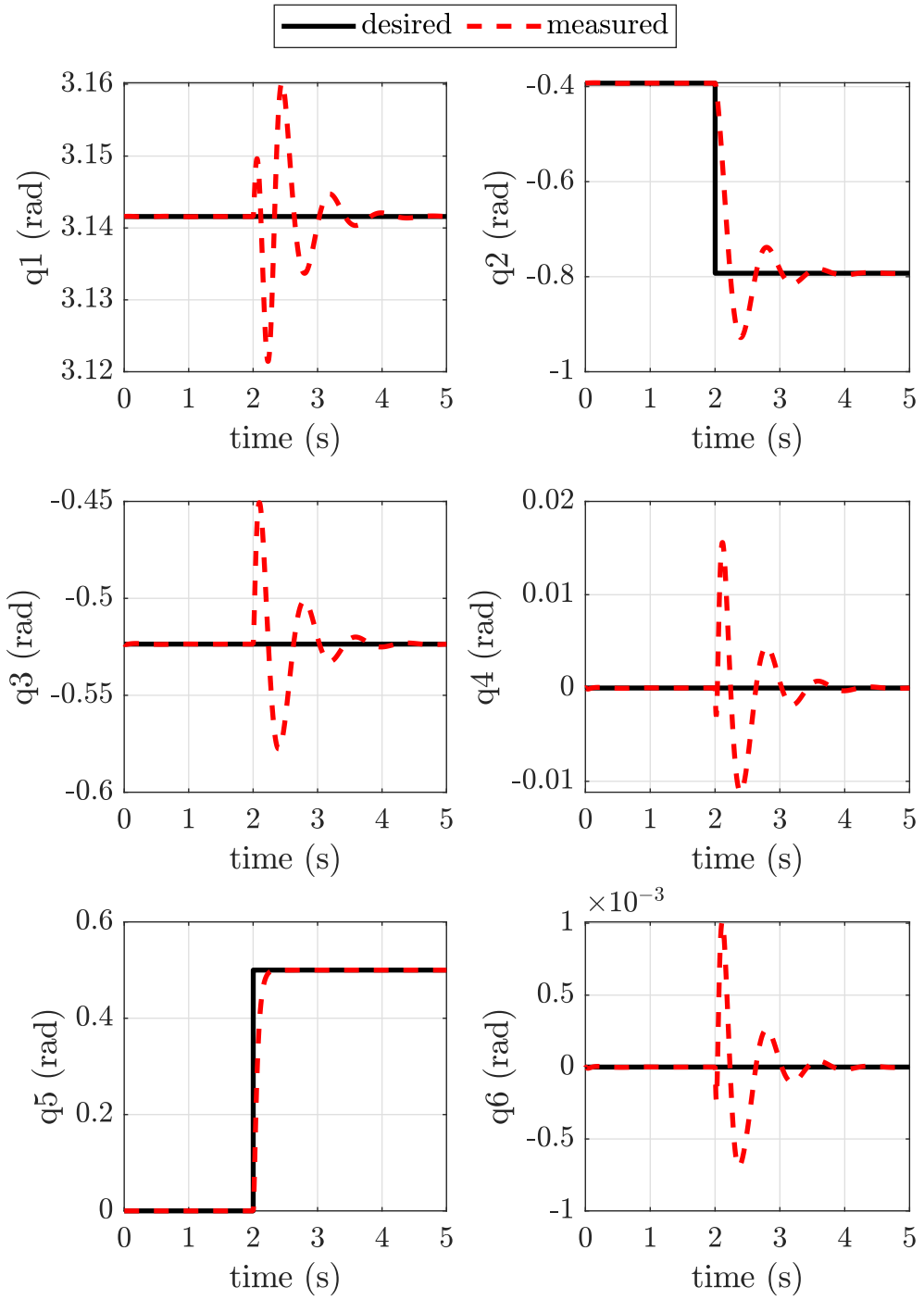


Figure 13: Angular position of each joint of UR5 robot with Algorithm 9.

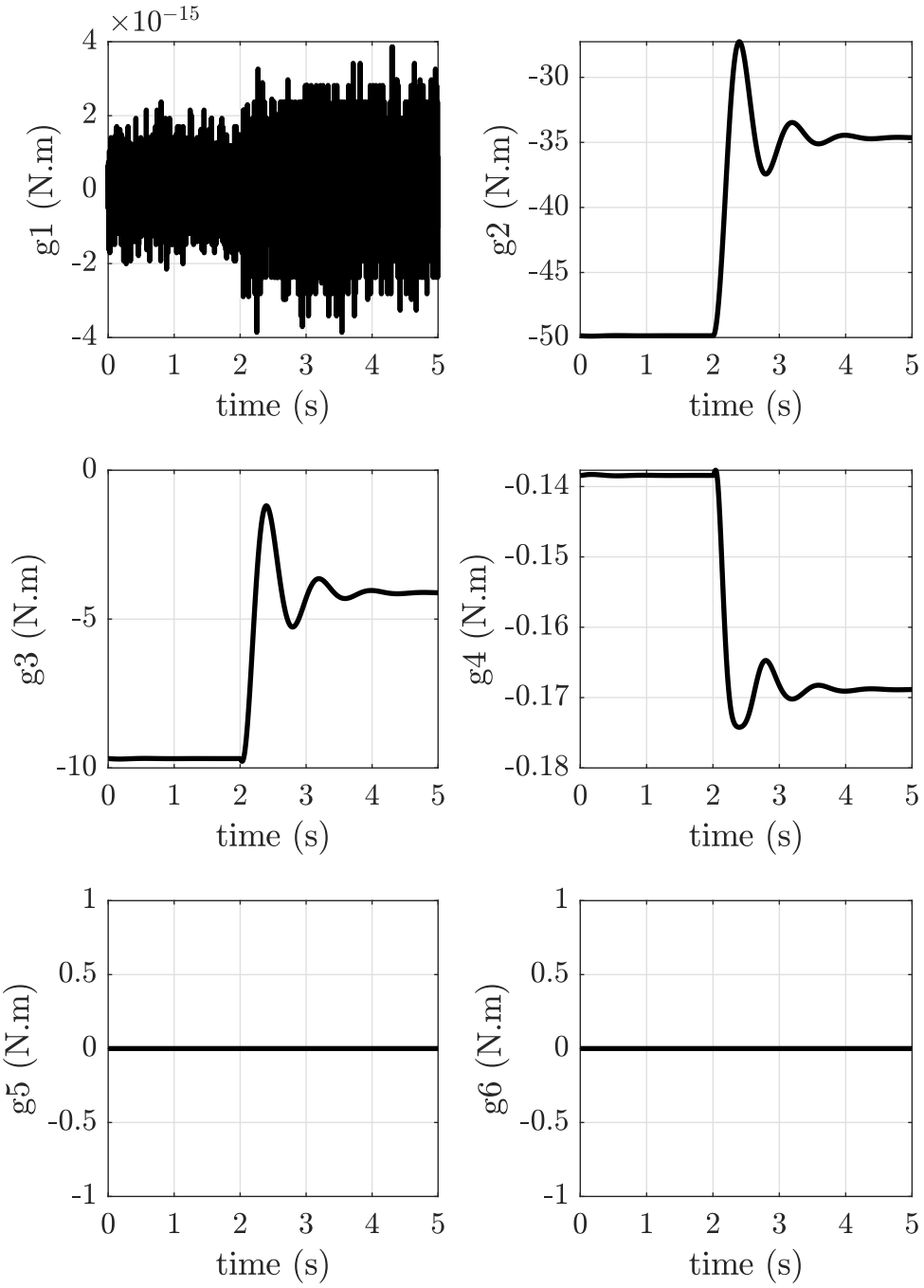


Figure 14: The gravity term (g) of the PD with gravity compensation control method when the reference of each joint is a constant value.

1.6.2 Sinusoidal reference

The Algorithm 10 control the movements of second and fifth joints of the UR5 robot to follow a sinusoidal reference trajectory. In this file, the PD with gravity compensation control method is configured with $K_p = 300\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 20\ \frac{\text{N.m.s}}{\text{rad}}$. Figure 15 shows the tracking performance of each joint of the UR5 robot. On one hand, the second joint (q_2) presents overshoot (15%) and steady state error close to 0 rad. On the other hand, the fifth joint (q_5)presents overshoot close to 0% and steady state error close to 0 rad. The variation of the temporal parameters of each joint is due to the control gains and the physical characteristics of the system. On one hand, the second joint must lift more mass than the fifth joint; and in the same way, the end effector of the robot is further from the second joint than from the fifth joint. In this context, the torque generated by robot's weight is greater in the second joint. On the other hand, the tracking error at transient state is due to the effect of Coriolis and centripetal forces. On the other hand, the tracking error at steady state is close to 0 rad due to gravity term in control law is compensating the forces generated by UR5 robot weight. Finally, the Figure 16 show the variation of the gravity term (g) of the control method when the reference of second and fifth joint is a sinusoidal trajectory.

```
# ==========================
#   Configuration of node
# ==========================
# create a node:
rospy.init_node("node_joint_PD_control_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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 300*np.ones(ndof)
# derivative gain
kd = 20*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():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
           sinusoidal_reference_generator(q0[1], dq0[1], ddq0[1],
           0.2, 1, t)
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
           sinusoidal_reference_generator(q0[4], dq0[4], ddq0[4],
           0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
           (0, last_q_des_1)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
           (0 , last_q_des_4)
```

```
    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute gravitational effects vector
    g = ur5_robot.get_g()

    # PD control method + gravity compensation
    tau = np.multiply(kp, e) + np.multiply(kd, de) + g

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

Algorithm 10: Move the second and fifth joint of UR5 robot with the requirement motion of activity 1.6.2

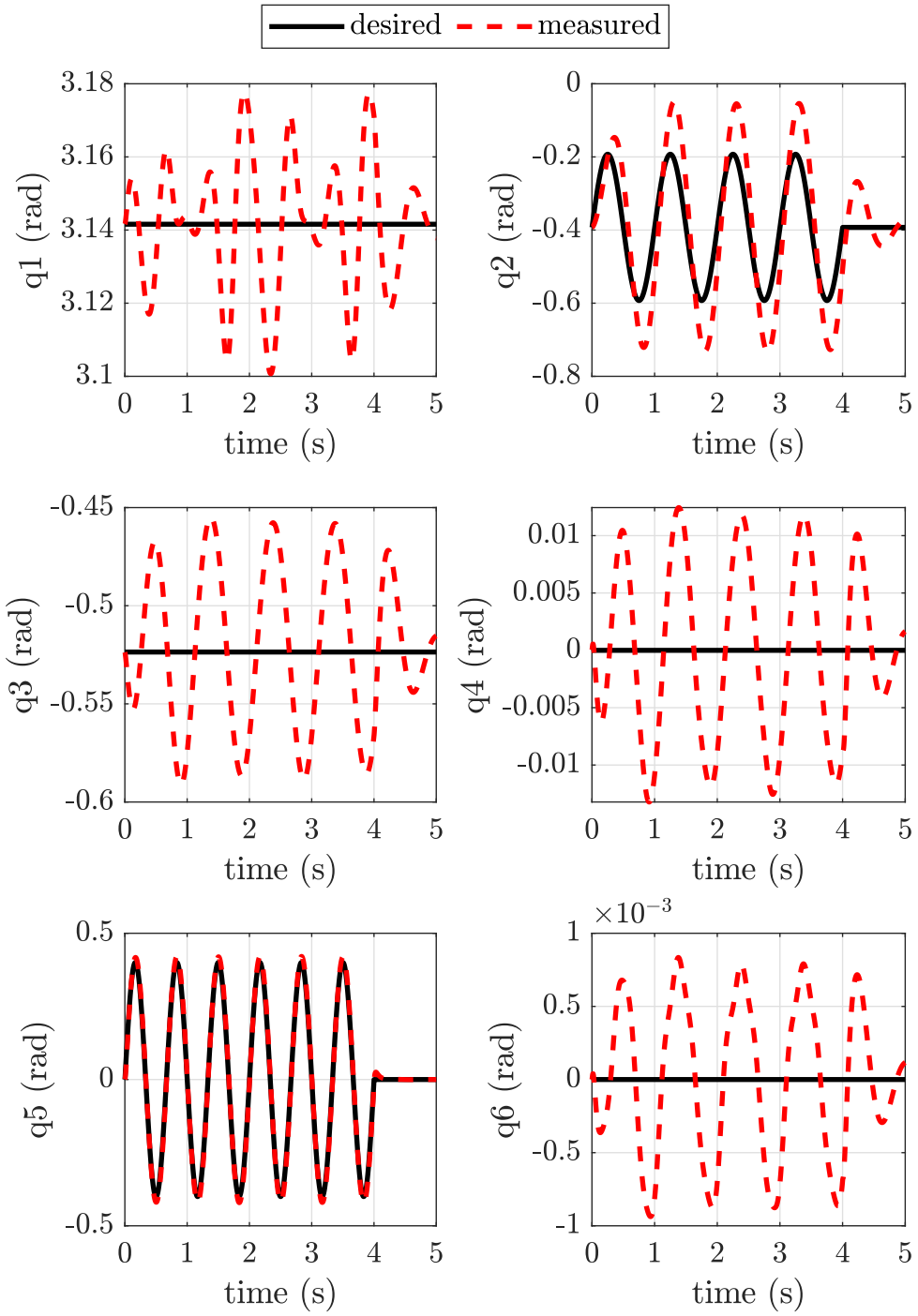


Figure 15: Angular position of each joint of UR5 robot with Algorithm 10.

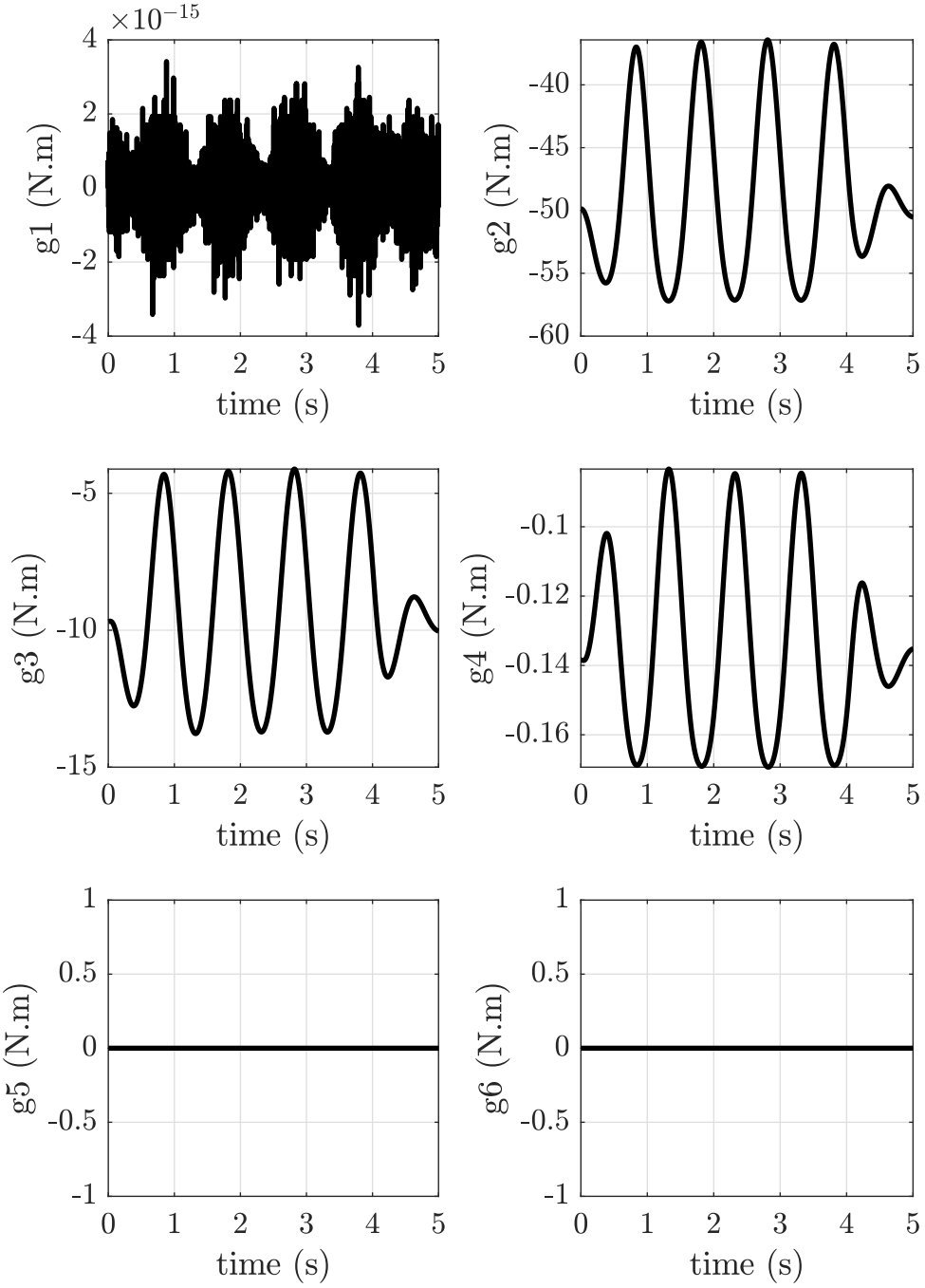


Figure 16: The gravity term (g) of the PD with gravity compensation control method when the reference of second and fifth joints is a sinusoidal trajectory.

1.7 Joint PD control + gravity + feed-forward term

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with a PD with gravity compensation and a feed-forward term control method. The control is control method is given by $\tau = M\ddot{q}_{\text{des}} + k_p e + k_d \dot{e} + g$ where M is the inertia matrix and g is the gravitational effects vector. The simulation starts with the initial joint configuration $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad and then move the second and fifth joints of the UR5 robot. On one hand, the two joints will maintain the initial configuration during first 2 seconds and follow a step reference during last 3 seconds. On the other hand, the two joints will move following a sinusoidal trajectory during first 4 seconds and maintain a constant joint position during last second. Finally, the performance of the control method will be evaluated in next subsections.

1.7.1 Sinusoidal reference

The Algorithm 11 control the movements of second and fifth joints of the UR5 robot to follow a sinusoidal reference trajectory. In this file, the PD with gravity compensation and a feed-forward term control method is configured with $K_p = 300\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 20\ \frac{\text{N.m.s}}{\text{rad}}$. Figure 17 shows the tracking performance of each joint of the UR5 robot. On one hand, the second joint (q_2) presents position error at the beginning and final of the simulation but a good trajectory tracking from 1 to 4 seconds. The position error at the beginning is due to the reference speed starting with a value other than 0. Figure 18 shows the reference and measured velocity of each joint of UR5 robot. On the other hand, the fifth joint (q_5) presents an excellent trajectory tracking except when reference trajectory change from sinusoidal to constant. The position error of q_5 could be generated by the Coriolis and centripetal forces that has a high value when the reference trajectory change from sinusoidal to constant. Figure 19 shows the value of Coriolis and centripetal forces for each joint and a red dot indicating the peak value.

```
# ==========================
#   Configuration of node
# ==========================
# create a node:
rospy.init_node("node_joint_PD_control_gravity_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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

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

```
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ==============================
#   PD controller configuration
# ==============================
# proportional gain
kp = 300*np.ones(ndof)
# derivative gain
kd = 20*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():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
           sinusoidal_reference_generator(q0[1], dq0[1], ddq0[1],
           0.2, 1, t)
```

```
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
            sinusoidal_reference_generator(q0[4], dq0[4], ddq0[4],
            0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
            (0, last_q_des_1)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
            (0 , last_q_des_4)

    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    M = ur5_robot.get_M()

    # compute gravitational effects vector
    g = ur5_robot.get_g()

    # PD control method + gravity compensation + feedforward term
    tau_ff = M.dot(ddq_des)
    tau = np.multiply(kp, e) + np.multiply(kd, de) + tau_ff + g

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

Algorithm 11: Move the second and fifth joint of UR5 robot with the requirement motion of activity 1.7.1

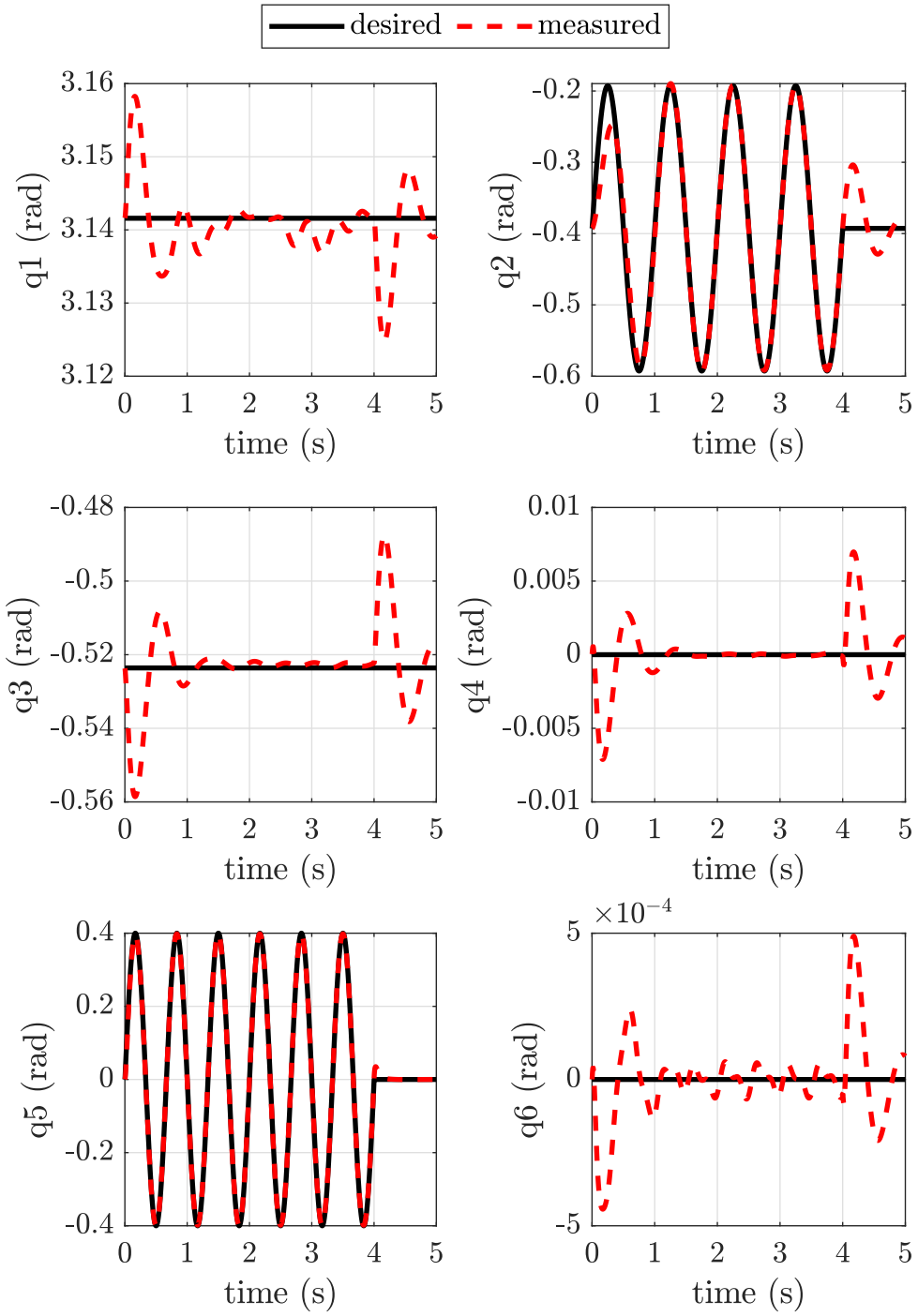


Figure 17: Angular position of each joint of UR5 robot with Algorithm 11.

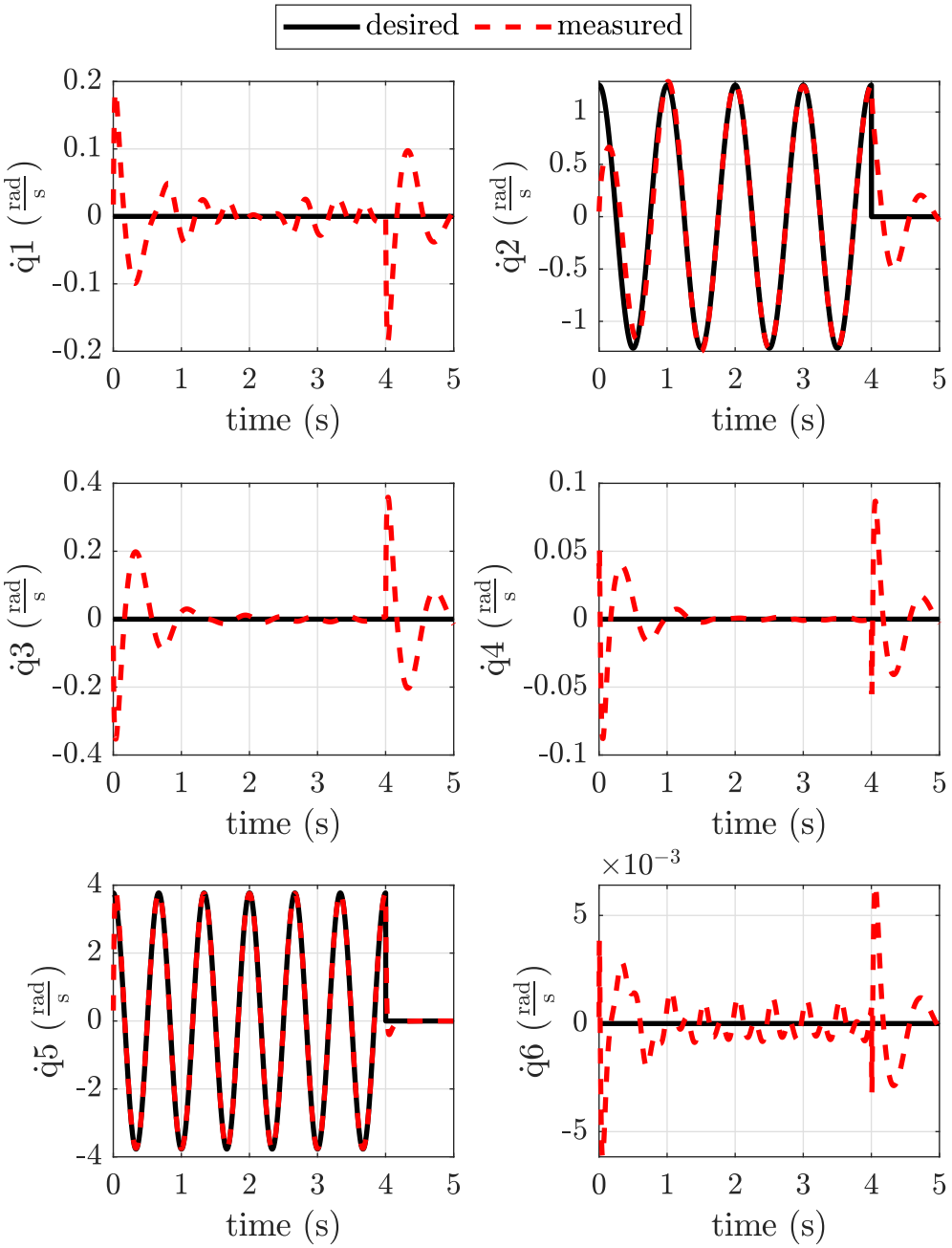


Figure 18: Angular velocity of each joint of UR5 robot with Algorithm 11.

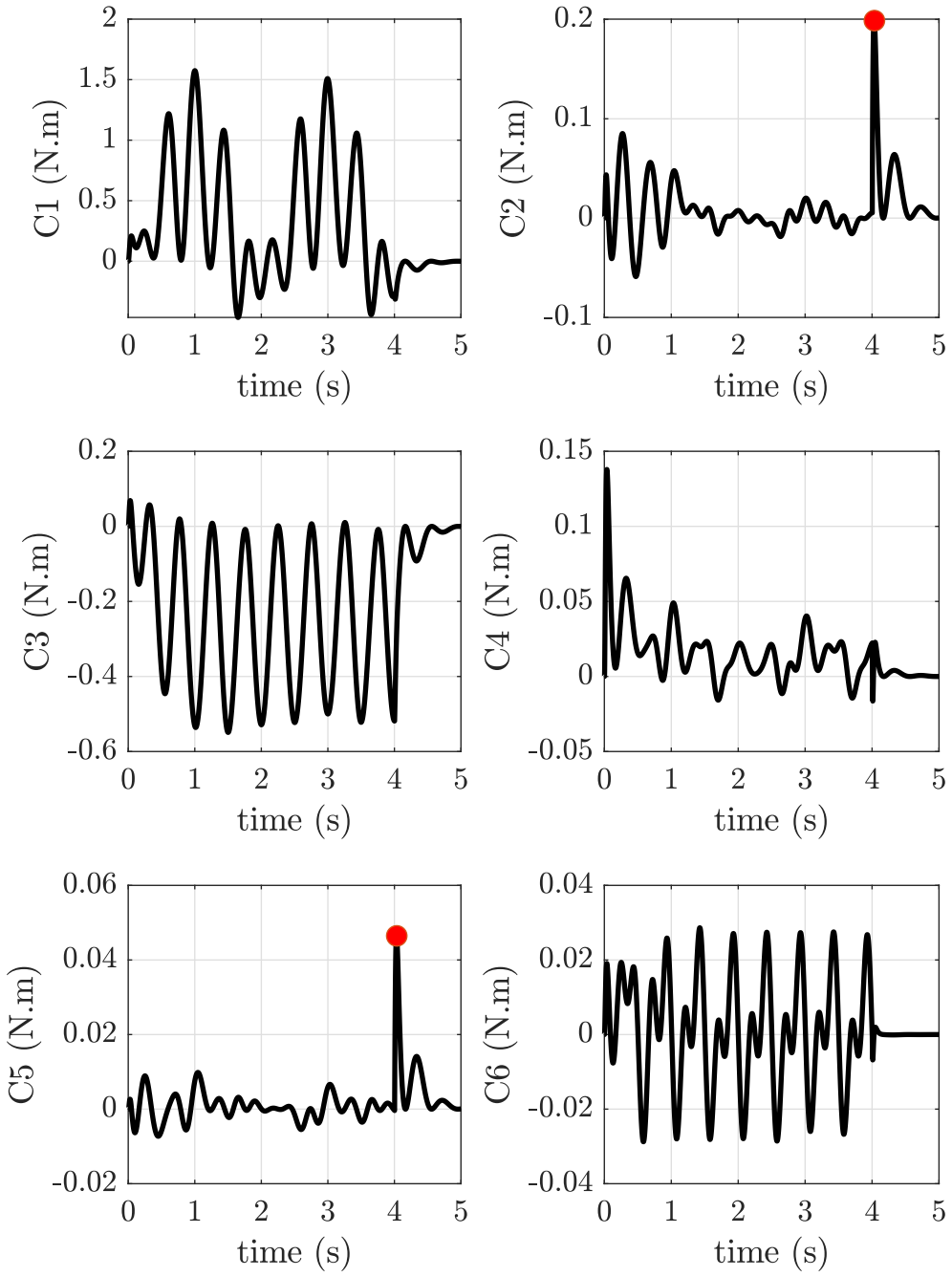


Figure 19: Coriolis and centripetal forces (C). The red dot indicates the peak value of C when the reference trajectory change form sinusoidal to constant.

1.7.2 Step reference

The Algorithm 12 control the movements of second and fifth joints of the UR5 robot to follow a step reference trajectory. In this file, the PD with gravity compensation and a feed-forward term control method is configured with $K_p = 300\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 20\ \frac{\text{N.m.s}}{\text{rad}}$. Figure 20 shows the tracking performance of each joint of the UR5 robot. In this case, the performance of the new control method is same as PD with gravity compensation due to desired acceleration is $0\ \frac{\text{rad}}{\text{s}^2}$, so feed-forward term is 0.

```
# ========================
#   Configuration of node
# ========================
# create a node:
rospy.init_node("
   node_joint_PD_control_gravity_compensation_feed_forward_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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
```

```
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)
# create gravity vector
g = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 300*np.ones(ndof)
# derivative gain
kd = 20*np.ones(ndof)
# control vector
tau = np.zeros(ndof)

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

while not rospy.is_shutdown():
    # generate step reference after 2 seconds
    if t>=step_start:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator(
           q0[1], -0.4)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator(
           q0[4], 0.5)

    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    M = ur5_robot.get_M()

    # compute gravitational effects vector
    g = ur5_robot.get_g()

    # Coriolis and centripetal forces
    C = ur5_robot.get_b() - g

    # PD control method + gravity compensation + feedforward term
    tau_ff = M.dot(ddq_des)
    tau = np.multiply(kp, e) + np.multiply(kd, de) + tau_ff + g

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

```
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

Algorithm 12: Move the second and fifth joint of UR5 robot with the requirement motion of activity 1.7.2

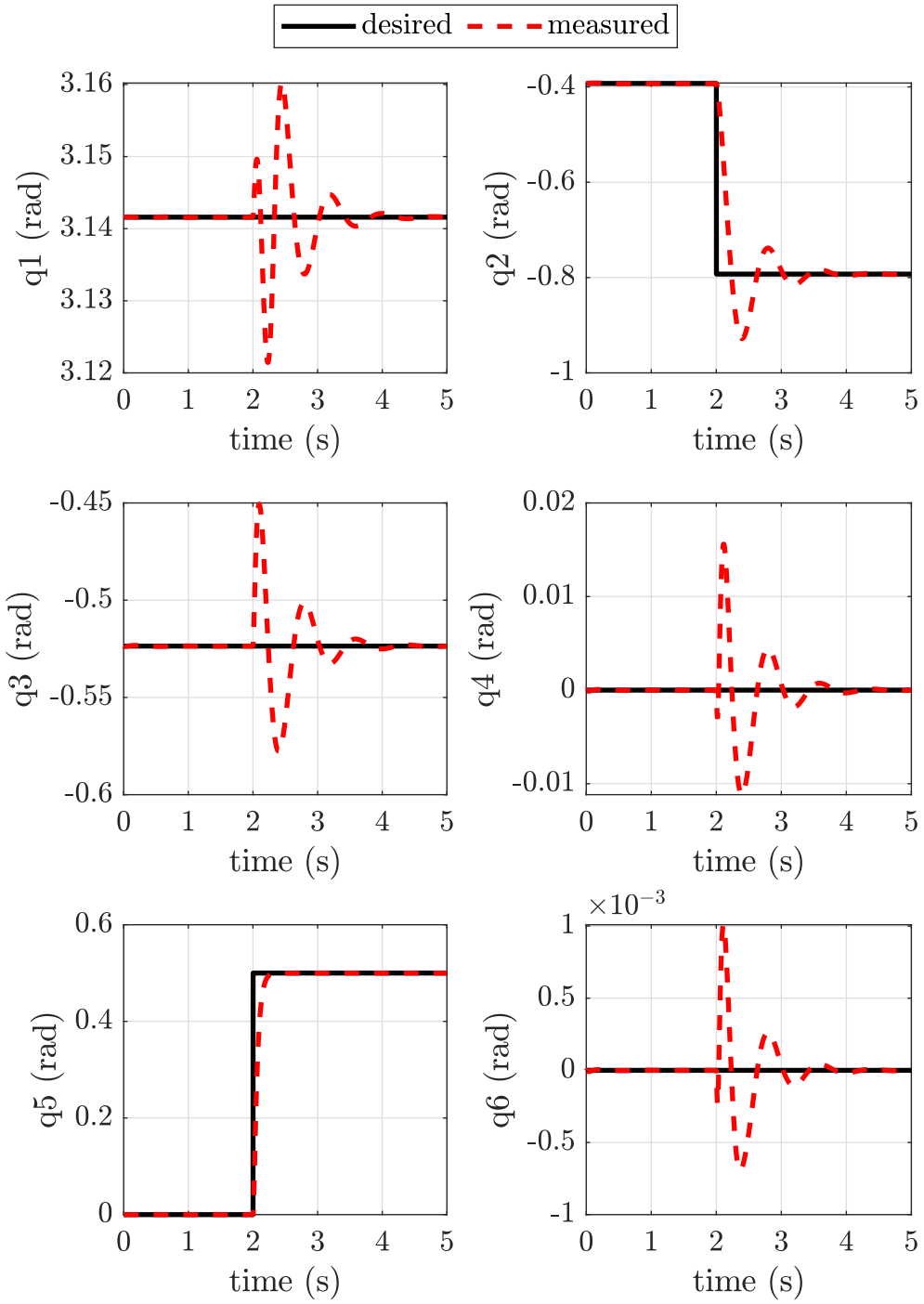


Figure 20: Angular position of each joint of UR5 robot with Algorithm 12.

2 Centralized control

2.1 Inverse dynamics

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with inverse dynamics control method. The simulation starts with the initial joint configuration $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad and then move the second and fifth joints of the UR5 robot. The two joints will move following a sinusoidal trajectory during the first 4 seconds and maintain a constant joint position during the last second. Finally, the rosnode file that control the movement of the six joints of UR5 robot is described in Algorithm 13. In this file, the inverse dynamics is configure with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_d = 30\ \frac{\text{N.m.s}}{\text{rad}}$.

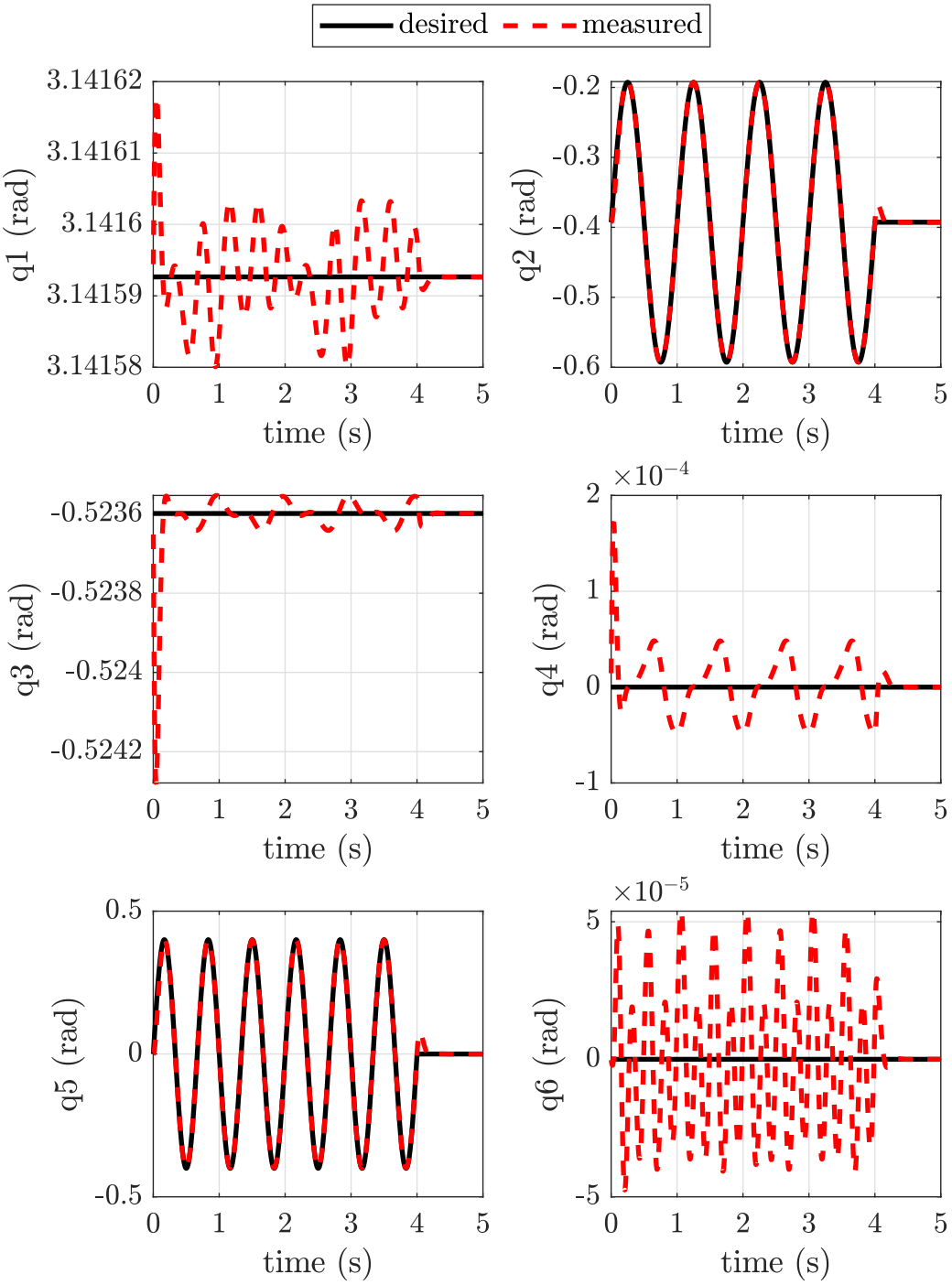
Figure 21 and 22 show trajectory tracking performance of each joint of the UR5 robot. On one hand, second and fifth joints presents and excellent position tracking except when the reference trajectory change from sinusoidal to constant value. On the other hand, second and fifth joints presents velocity error at the beginning due to reference speed starting with a value other than 0.

```
# ==========================
#   Configuration of node
# ==========================
# create a node:
rospy.init_node("node_inverse_dynamics_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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)

# ==============================
#   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():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
            sinusoidal_reference_generator(q0[1], dq0[1], ddq0[1],
            0.2, 1, t)
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
            sinusoidal_reference_generator(q0[4], dq0[4], ddq0[4],
            0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
            (0, last_q_des_1)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
            (0 , last_q_des_4)

    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq
```

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

    # compute nonlinear effects vector (b)
    b = ur5_robot.get_b()

    # control law: PD control + Feedback linearization
    tau = M.dot(ddq_des+np.multiply(kp, e)+np.multiply(kd, de)) + b

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

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

Algorithm 13: Move the second and fifth joint of UR5 robot with the required movement of activity 2.1.



H

Figure 21: Angular position of each joint of UR5 robot with Algorithm 13.

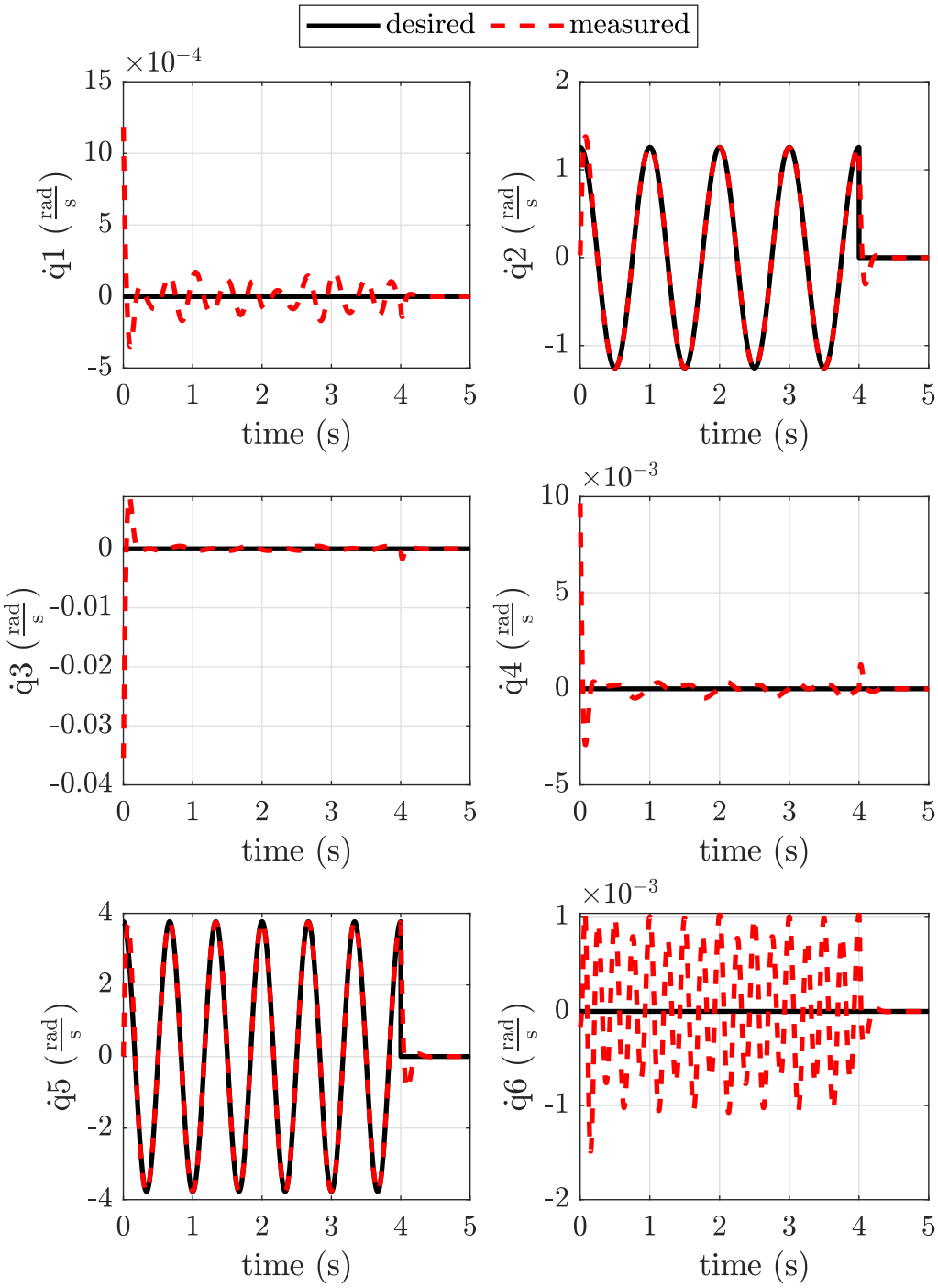


Figure 22: Angular velocity of each joint of UR5 robot with Algorithm 13.

2.2 Inverse dynamics - initial velocity

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with inverse dynamics control method. The simulation starts with the initial joint position $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad and joint velocity $\begin{bmatrix}0.0 & 0.4\pi & 0.0 & 0.0 & 1.2\pi & 0.0\end{bmatrix}$. The second and fifth joints will move following a sinusoidal trajectory during the first 4 seconds and maintain a constant joint position during the last second. Finally, the rosnode file that control the movement of the six joints of UR5 robot is the same as Algorithm 13 except that the initial velocity of the robot matches desired initial velocity. Figure 23 shows the joint velocity error of each joint. In this figure, the velocity of second and fifth joints are equal to 0 $\frac{\text{rad}}{\text{s}}$ expect when the reference trajectory change from sinusoidal to constant value.

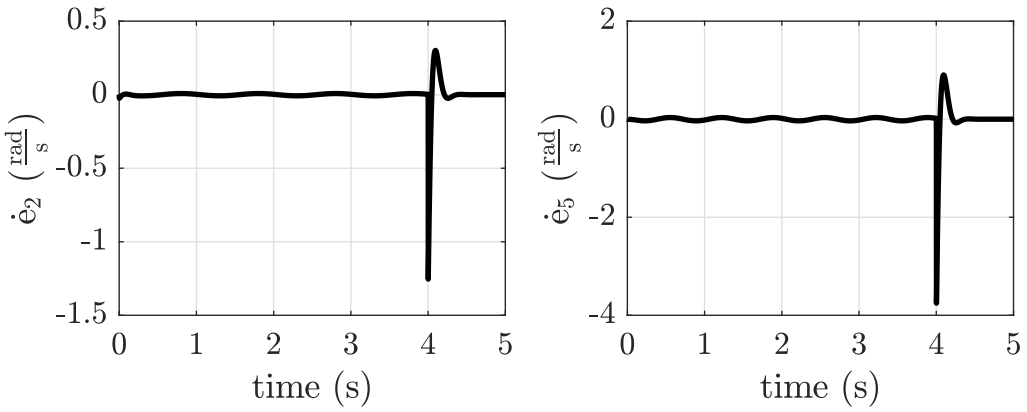


Figure 23: Angular velocity error of each joint of UR5 robot with Algorithm 13 and equal initial velocity for robot and reference.

2.3 Inverse dynamics - low gains

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with inverse dynamics control method. The simulation starts with the initial joint position $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad and joint velocity $\begin{bmatrix}0.0 & 0.4\pi & 0.0 & 0.0 & 1.2\pi & 0.0\end{bmatrix}$. The second and fifth joints will move following a sinusoidal trajectory during the first 4 seconds and maintain a constant joint position during the last second. Finally, the rosnode file that control the movement of the six joints of UR5 robot is described in Algorithm 14. In this file, the inverse dynamics is configure with $K_p = 60\ \frac{\text{N.m}}{\text{rad}}$ and $K_{d,i} = 2\sqrt{k_{p,i}}\ \frac{\text{N.m.s}}{\text{rad}}$.

Figure 21 and 22 show trajectory tracking performance of each joint of the UR5 robot. The second and fifth joints presents and excellent position tracking except at the end when reference trajectory change from sinusoidal to constant value.

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

# 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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq0 =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 0.0])
ddq = 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']
```

```
# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 60*np.ones(ndof)
# derivative gain
kd = 2*np.sqrt(kp[0])*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():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
           sinusoidal_reference_generator(q0[1], 0.2, 1, t)
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
           sinusoidal_reference_generator(q0[4], 0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
           (0, last_q_des_1)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
           (0 , last_q_des_4)

    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    M = ur5_robot.get_M()

    # compute nonlinear effects vector
    b = ur5_robot.get_b()
```

```
    # control law: PD control + Feedback linearization
    tau = M.dot(ddq_des +  np.multiply(kp, e) + np.multiply(kd, de)
        ) + b

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

Algorithm 14: Move the second and fifth joint of UR5 robot with the required movement of activity 2.3.

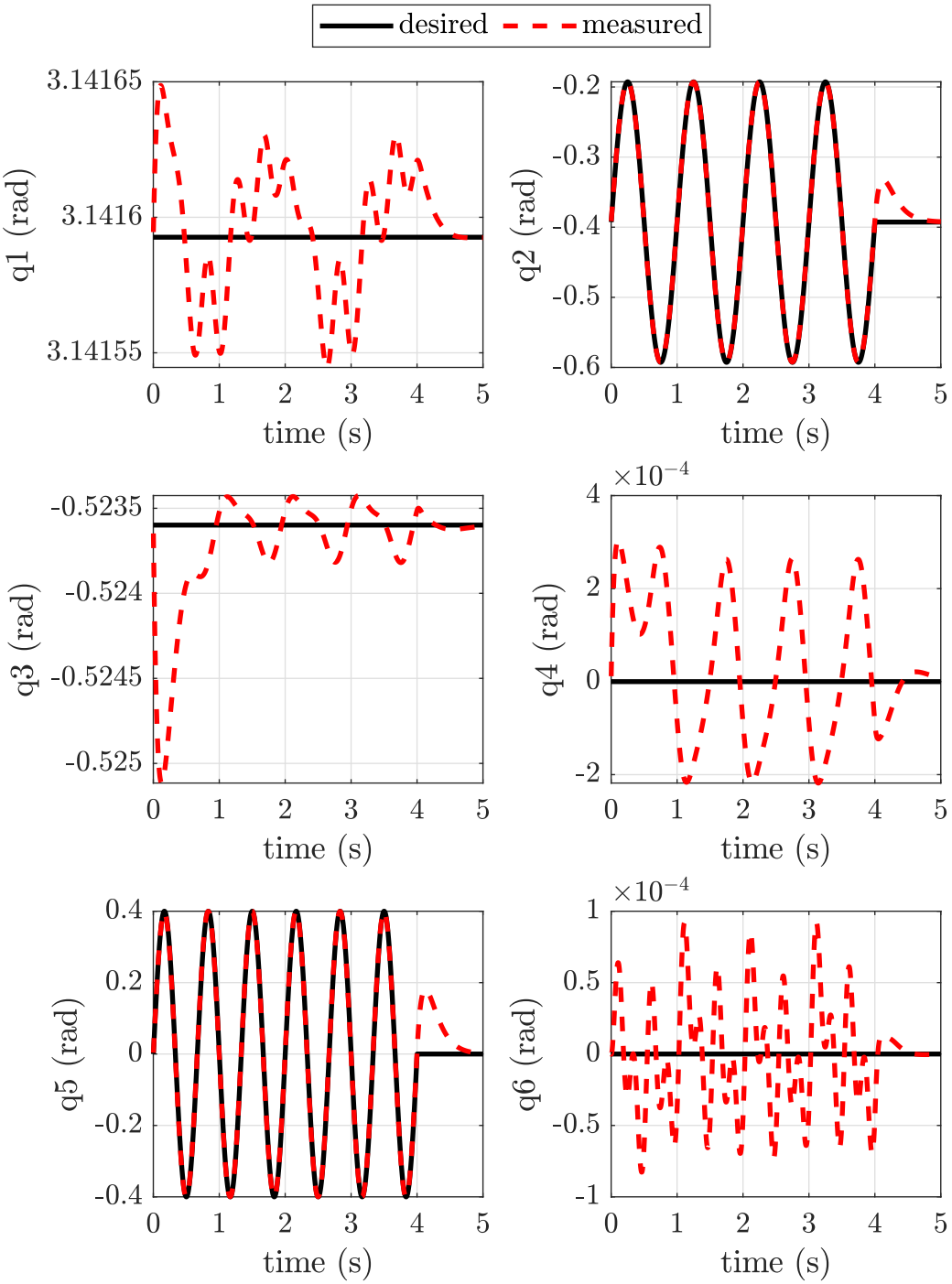


Figure 24: Angular position of each joint of UR5 robot with Algorithm 14

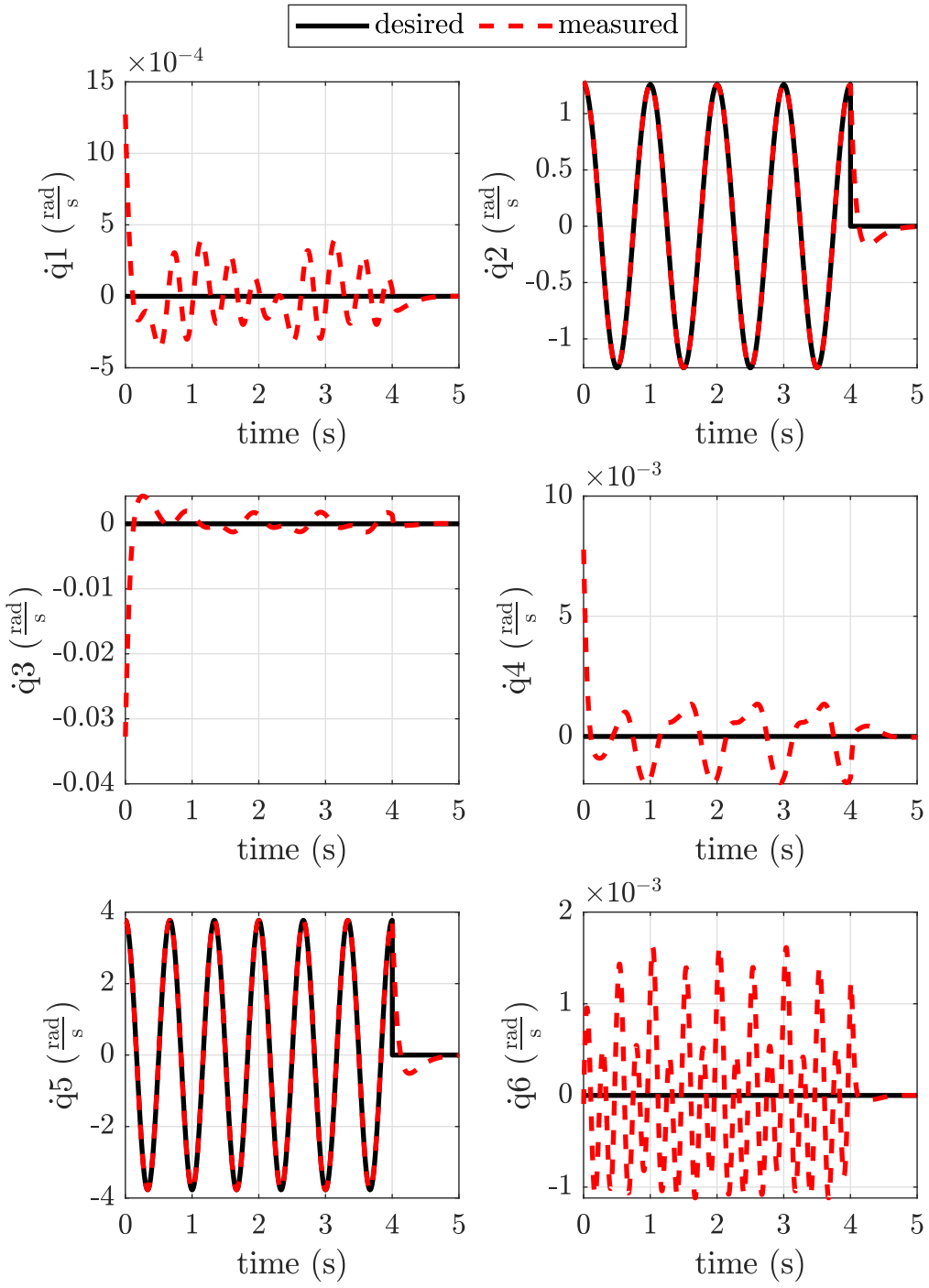


Figure 25: Angular velocity of each joint of UR5 robot with Algorithm 14

2.4 Inverse dynamics - external force

The objective of this activity is display the UR5 robot on rviz, control the motion of its joints with inverse dynamics control method and apply a force vector on the robot end-effector. The simulation starts with the initial joint position $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad, joint velocity $\begin{bmatrix}0.0 & 0.4\pi & 0.0 & 0.0 & 1.2\pi & 0.0\end{bmatrix}$ and external force $\begin{bmatrix}0.0 & 0.0 & 0.0\end{bmatrix}$ N. On one hand, second and fifth joints will move following a sinusoidal trajectory during the first 4 seconds and maintain a constant joint position during the last second. On the other hand, external force will change to $\begin{bmatrix}0.0 & 0.0 & 50\end{bmatrix}$ N after 3 seconds. Finally, the rosnode file that control the movement of the six joints of UR5 robot and apply an external force is described in Algorithm 15. In this file, the inverse dynamics is configure with $K_p = 60\ \frac{\text{N.m}}{\text{rad}}$ and $K_{d,i} = 2\sqrt{k_{p,i}}\ \frac{\text{N.m.s}}{\text{rad}}$.

Figure 26 shows trajectory tracking performance of each joint of the UR5 robot. In this figure, second and fifth joints presents and excellent position tracking except when the external force is activated. Figure 27 shows the trajectory tracking error of each joint. In this figure, position error is close to 0 rad in all the joints except when the external force is applied. Finally, the trajectory tracking error of the six joints with three values of proportional gain is shown in Figure 28. This figure indicates that increasing the value of the proportional gain decreases the error in position generated by the external force.

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

# 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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq0 =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 60*np.ones(ndof)
# derivative gain
kd = 2*np.sqrt(kp[0])*np.ones(ndof)
# control vector
tau = np.zeros(ndof)

#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 4.0 # [sec]
force_start   = 3.0 # [sec]

while not rospy.is_shutdown():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
           sinusoidal_reference_generator(q0[1], 0.2, 1, t)
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
           sinusoidal_reference_generator(q0[4], 0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
           (0, last_q_des_1)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
           (0 , last_q_des_4)
```

```
    if t>=force_start:
        # external force
        f_ext = np.array([0.0, 0.0, 50]) # N
        # position jacobian
        J = jacobian_xyz_ur5(q)
        # external torque
        tau_ext = np.dot (J.transpose(), f_ext) # N.m
    else:
        tau_ext = np.zeros(6) # N.m

    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    M = ur5_robot.get_M()

    # compute nonlinear effects vector
    b = ur5_robot.get_b()

    # control law: PD control + Feedback linearization
    tau = M.dot(ddq_des +  np.multiply(kp, e) + np.multiply(kd, de)
       ) + b + tau_ext

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
       read_joint_position_velocity_acceleration()

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

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

Algorithm 15: Move the second and fifth joint of UR5 robot with the required movement of activity 2.4.

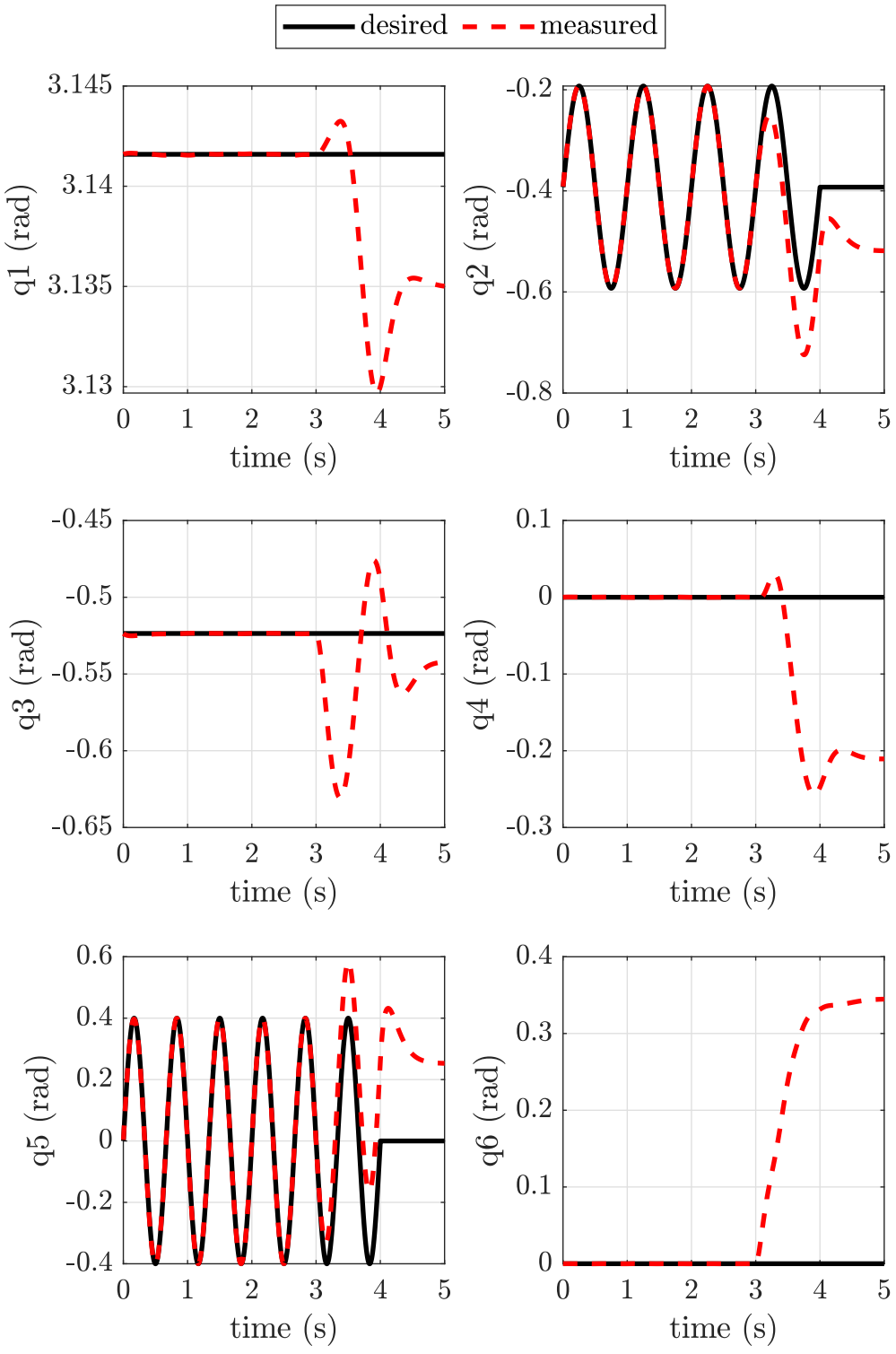


Figure 26: Angular position of each joint of UR5 robot with Algorithm 15

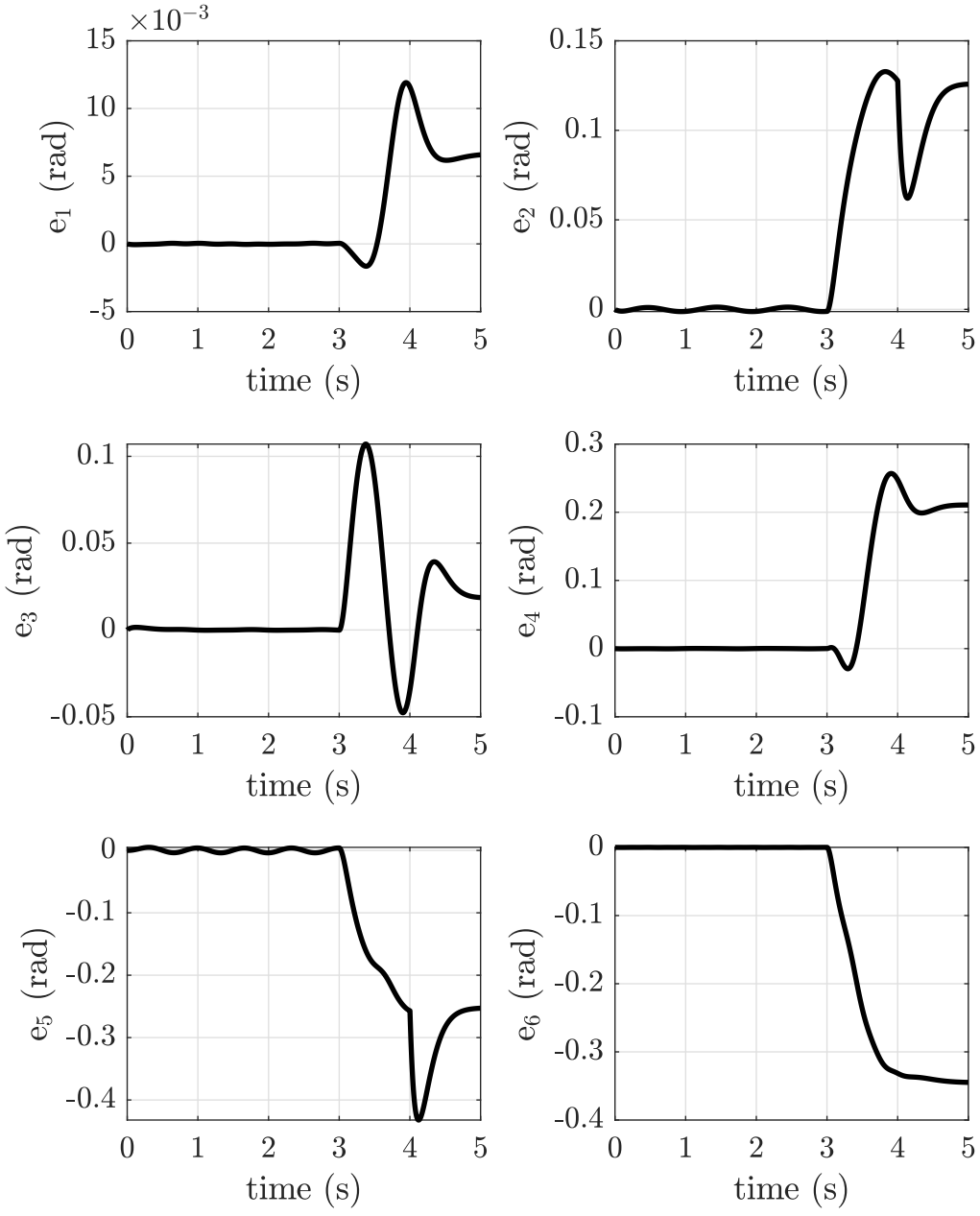


Figure 27: Angular position error of each joint of UR5 robot with Algorithm 15.

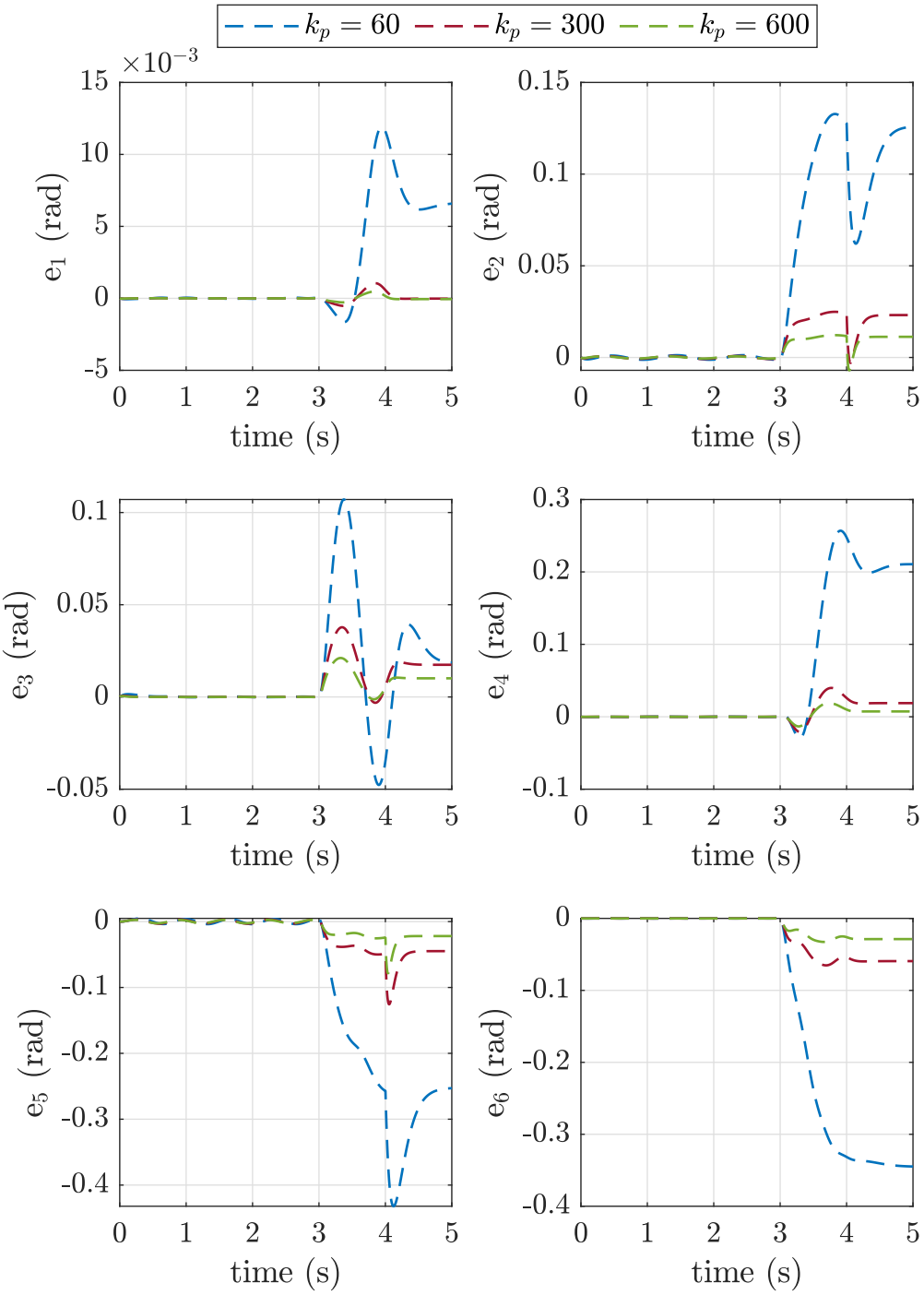


Figure 28: Angular position error of each joint of UR5 robot with three different values of proportional gain.

2.5 Inverse dynamics - uncertainty cancellation

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with inverse dynamics control method when the model has uncertainties. The simulation starts with the initial joint position $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad, joint velocity $\begin{bmatrix}0.0 & 0.4\pi & 0.0 & 0.0 & 1.2\pi & 0.0\end{bmatrix}$ and external force $\begin{bmatrix}0.0 & 0.0 & 0.0\end{bmatrix}$ N. The second and fifth joints will move following a sinusoidal trajectory during the first 4 seconds and maintain a constant joint position during the last second. Finally, the rosnode file that control the movement of the six joints of UR5 robot is Algorithm 16. In this file, the inverse dynamics is configure with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_{d,i} = 2\sqrt{k_{p,i}}\ \frac{\text{N.m.s}}{\text{rad}}$.

Figure 29 and 30 show trajectory tracking performance of each joint of the UR5 robot. The two joints present an acceptable position error generated by the uncertainty in inertia matrix and nonlinear effects vector.

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

# 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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq0 =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 600*np.ones(ndof)
# derivative gain
kd = 2*np.sqrt(kp[0])*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():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
            sinusoidal_reference_generator(q0[1], 0.2, 1, t)
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
            sinusoidal_reference_generator(q0[4], 0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
            (0, last_q_des_1)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
            (0 , last_q_des_4)

    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    M = ur5_robot.get_M()
    M_hat = 0.9*M

    # compute nonlinear effects vector
```

```
    b = ur5_robot.get_b()
    b_hat = 0.9*b

    # control law: PD control + Feedback linearization
    tau = M_hat.dot(ddq_des +  np.multiply(kp, e) + np.multiply(kd,
        de)) + b_hat

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
       read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

Algorithm 16: Move the second and fifth joint of UR5 robot with the required movement of activity 2.5.

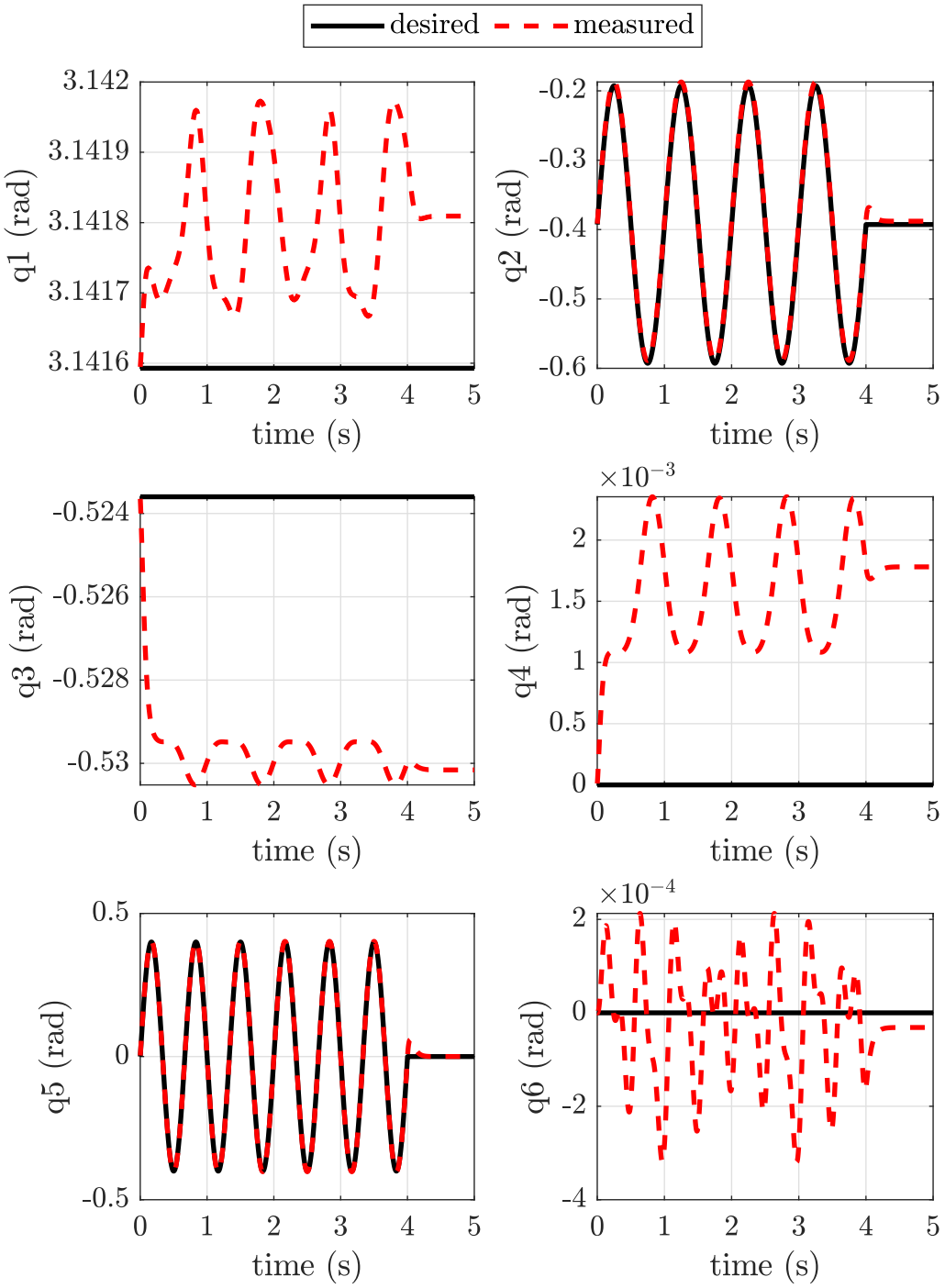


Figure 29: Angular position of each joint of UR5 robot with Algorithm 16

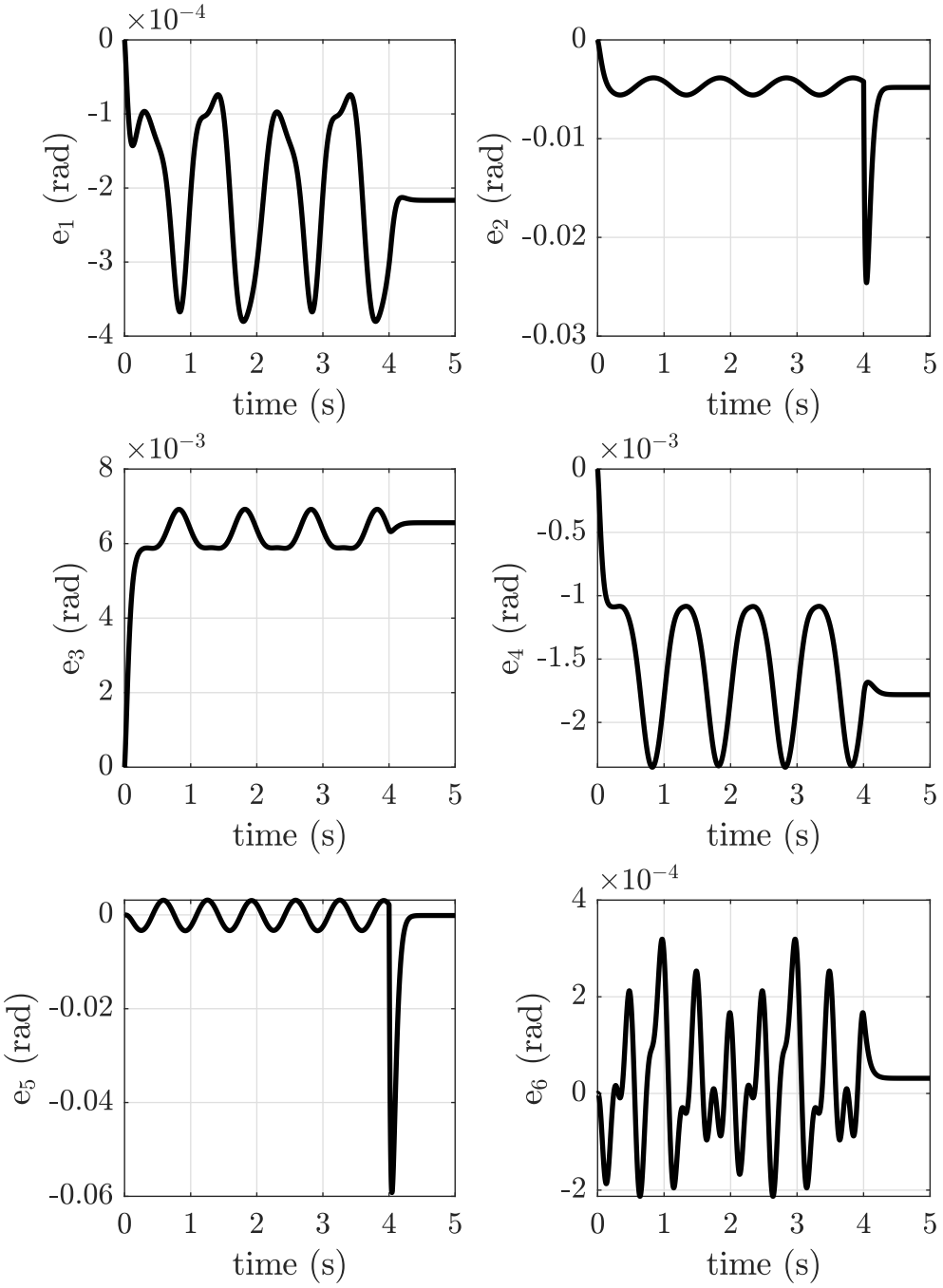


Figure 30: Angular position error of each joint of UR5 robot with Algorithm 16

2.6 Inverse dynamics - desired states

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with inverse dynamics control. The simulation starts with the initial joint position $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad, joint velocity $\begin{bmatrix}0.0 & 0.4\pi & 0.0 & 0.0 & 1.2\pi & 0.0\end{bmatrix}$ and external force $\begin{bmatrix}0.0 & 0.0 & 0.0\end{bmatrix}$ N. The second and fifth joints will move following a sinusoidal trajectory during the first 4 seconds and maintain a constant joint position during the last second. Finally, the rosnode file that control the movement of the six joints of UR5 robot is Algorithm 17. In this file, the inverse dynamics is configure with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_{d,i} = 2\sqrt{k_{p,i}}\ \frac{\text{N.m.s}}{\text{rad}}$.

Figure 31 and 32 show trajectory tracking performance of each joint of the UR5 robot. The joints have an excellent trajectory tracking except when the reference change from sinusoidal to constant value.

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

# 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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq0 =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 0.0])
ddq = 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']
```

```
# number of degress of freedom
ndof = 6
# load the ur5.urdf
ur5_robot_model = rbdl.loadModel(os.path.join(pwd,'../../
   ur5_description/urdf/v2_ur5.urdf'))
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 600*np.ones(ndof)
# derivative gain
kd = 2*np.sqrt(kp[0])*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():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
           sinusoidal_reference_generator(q0[1], 0.2, 1, t)
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
           sinusoidal_reference_generator(q0[4], 0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
           (0, last_q_des_1)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
           (0 , last_q_des_4)

    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    rbdl.CompositeRigidBodyAlgorithm(ur5_robot_model, q_des, M)

    # compute nonlinear effects vector
    rbdl.NonlinearEffects(ur5_robot_model, q_des, dq_des, b)
```

```
# control law: PD control + Feedback linearization
tau = M.dot(ddq_des +  np.multiply(kp, e) + np.multiply(kd, de)
    ) + b

# update states
ddq = np.linalg.inv(M).dot(tau-b)
dq  =  dq + dt*ddq
q   =  q + dt*dq + 0.5*dt*dt*ddq

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

# update time
t = t + dt

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

Algorithm 17: Move the second and fifth joint of UR5 robot with the required movement of activity 2.6.

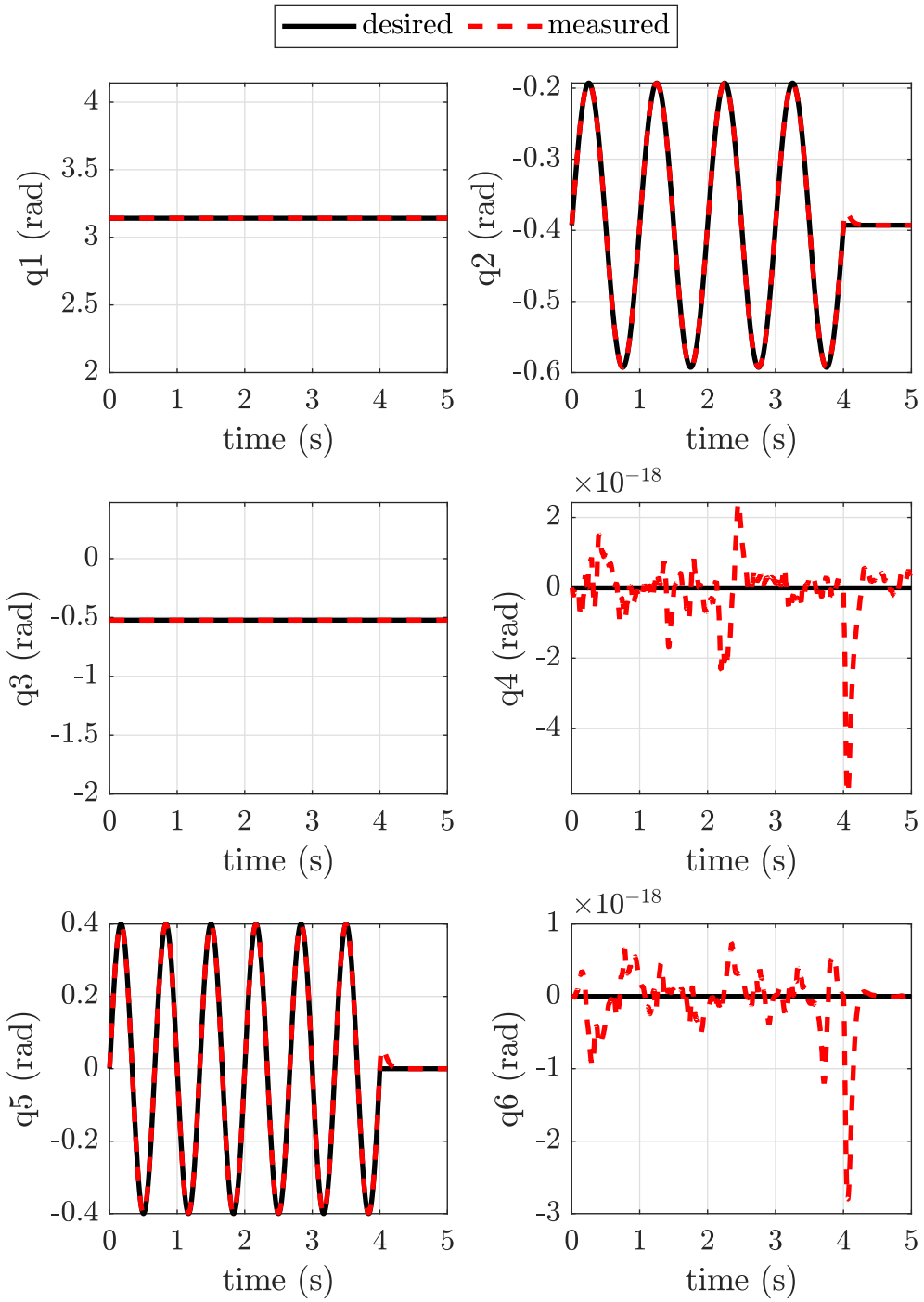


Figure 31: Angular position of each joint of UR5 robot with Algorithm 17.

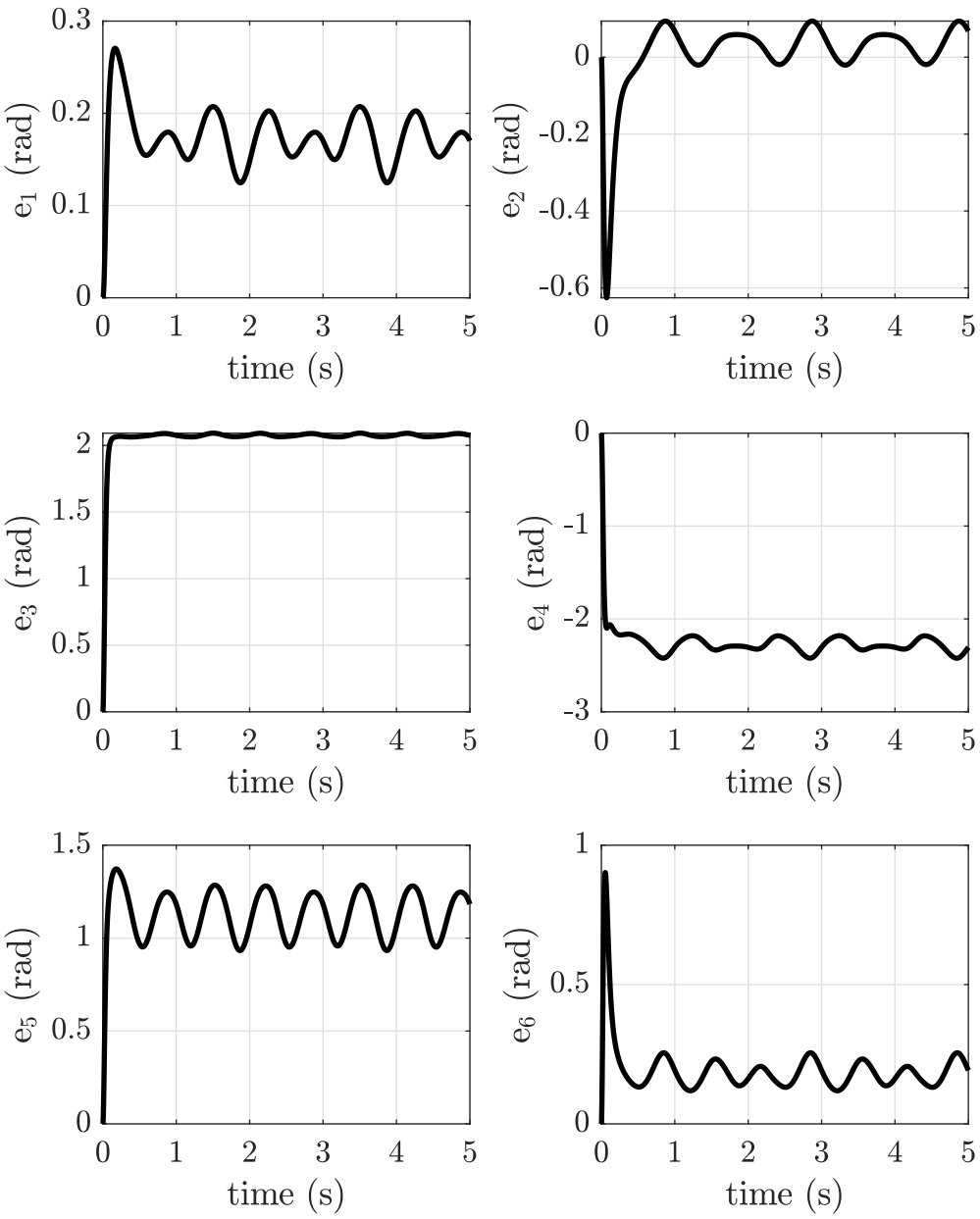


Figure 32: Angular position error of each joint of UR5 robot with Algorithm 17.

2.7 Inverse dynamics - unilateral compliant contact

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with inverse dynamics control method with compliant contact. The simulation starts with the initial joint position $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad, joint velocity $\begin{bmatrix}0.0 & 0.4\pi & 0.0 & 0.0 & 1.2\pi & 0.0\end{bmatrix}$ and external force $\begin{bmatrix}0.0 & 0.0 & 0.0\end{bmatrix}$ N. The second and fifth joints will move following a sinusoidal trajectory during the first 4 seconds and maintain a constant joint position during the last second. Finally, in this activity, the sinusoidal trajectory will be modified from 4 seconds to 5 seconds.

2.7.1 One axis: z

The rosnode file that control the movement of the six joints of UR5 robot is Algorithm 18. In this file, the inverse dynamics is configure with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_{d,i} = 2\sqrt{k_{p,i}}\ \frac{\text{N.m.s}}{\text{rad}}$. Figure 33 and 34 show trajectory tracking performance of each joint of the UR5 robot. The joint with more error is third joint (q_3).

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

# 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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq0 =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)

# ==============================
#   PD controller configuration
# ==============================
# proportional gain
kp = 600*np.ones(ndof)
# derivative gain
kd = 2*np.sqrt(kp[0])*np.ones(ndof)
# control vector
tau = np.zeros(ndof)

# ============================
#   Enviornment configuration
# ============================
p0 = np.array([0.0, 0.0, 0.0]) # m
k_env = 10000*np.ones(3) # N/m
d_env = 2*np.sqrt(k_env[0])*np.ones(3) # N.s/m
n = np.array([0, 0, 1])
#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 5.0 # [sec]
force_start   = 1.0 # [sec]


while not rospy.is_shutdown():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
            sinusoidal_reference_generator(q0[1], 0.2, 1, t)
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
            sinusoidal_reference_generator(q0[4], 0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
            (0, last_q_des_1)
        # fifth link
```

```
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
            (0 , last_q_des_4)

    # position of end-effector
    p_ee = fkine_ur5(q)[0:3,3]
    # position jacobian
    J = jacobian_xyz_ur5(q)
    # velocity of end-effector
    dp_ee = np.dot(J, dq)

    if (np.dot(n, p0-p_ee))<0:
        # external force
        f_ext = np.multiply(k_env, p0-p_ee) - np.multiply(d_env,
            dp_ee) # N
        # external force: z axis
        f_ext = np.multiply(n, f_ext) # N

        # external torque
        tau_ext = np.dot (J.transpose(), f_ext) # N.m
    else:
        # external torque
        tau_ext = np.zeros(6) # N.m
    print("f_ext: ", f_ext)
    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    M = ur5_robot.get_M()

    # compute nonlinear effects vector
    b = ur5_robot.get_b()

    # control law: PD control + Feedback linearization
    tau = M.dot(ddq_des +  np.multiply(kp, e) + np.multiply(kd, de)
        ) + b + tau_ext

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

```
        break
    rate.sleep()
```

Algorithm 18: Move the second and fifth joint of UR5 robot with the required movement of activity 2.7.1.

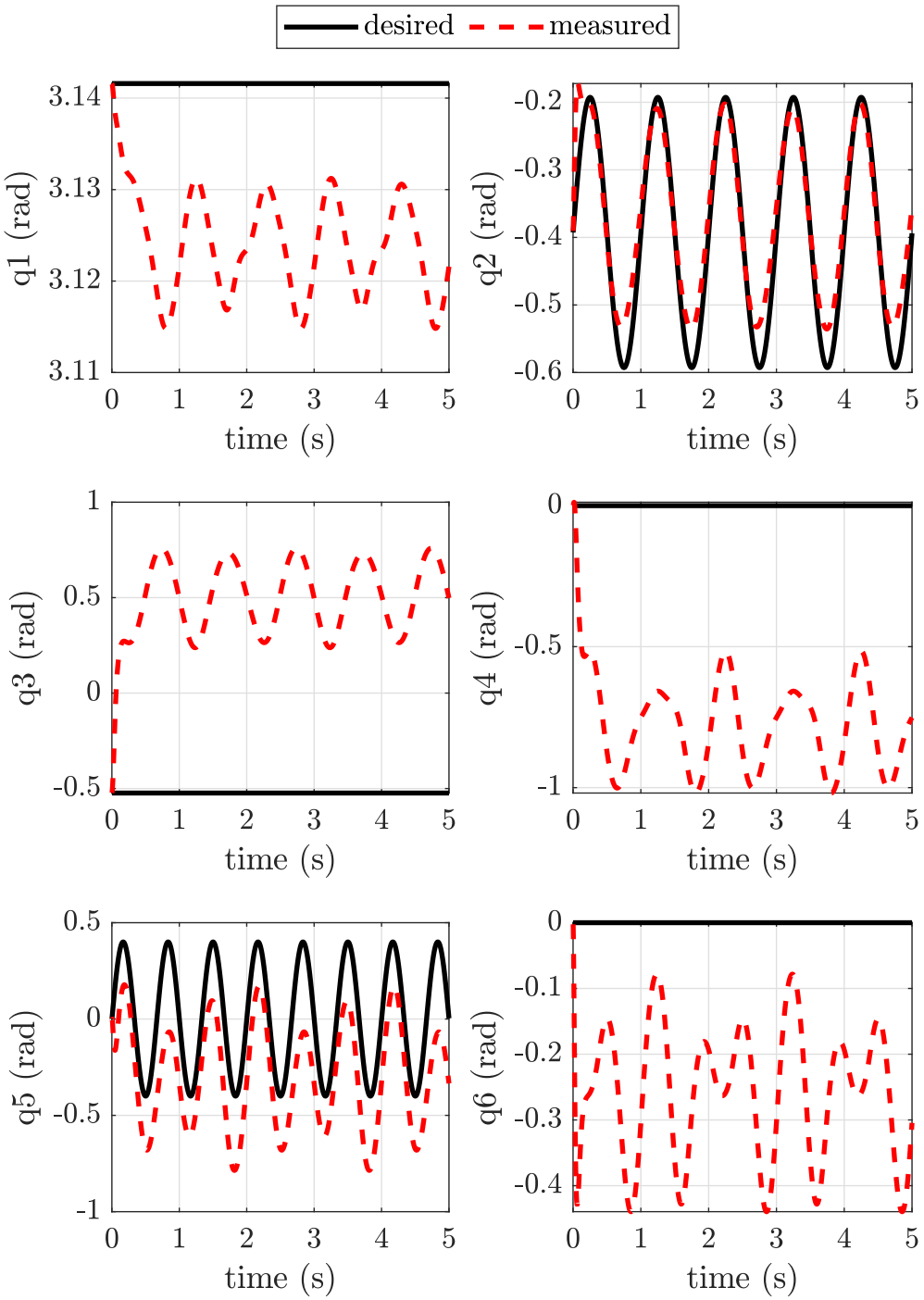


Figure 33: Angular position of each joint of UR5 robot with Algorithm 18

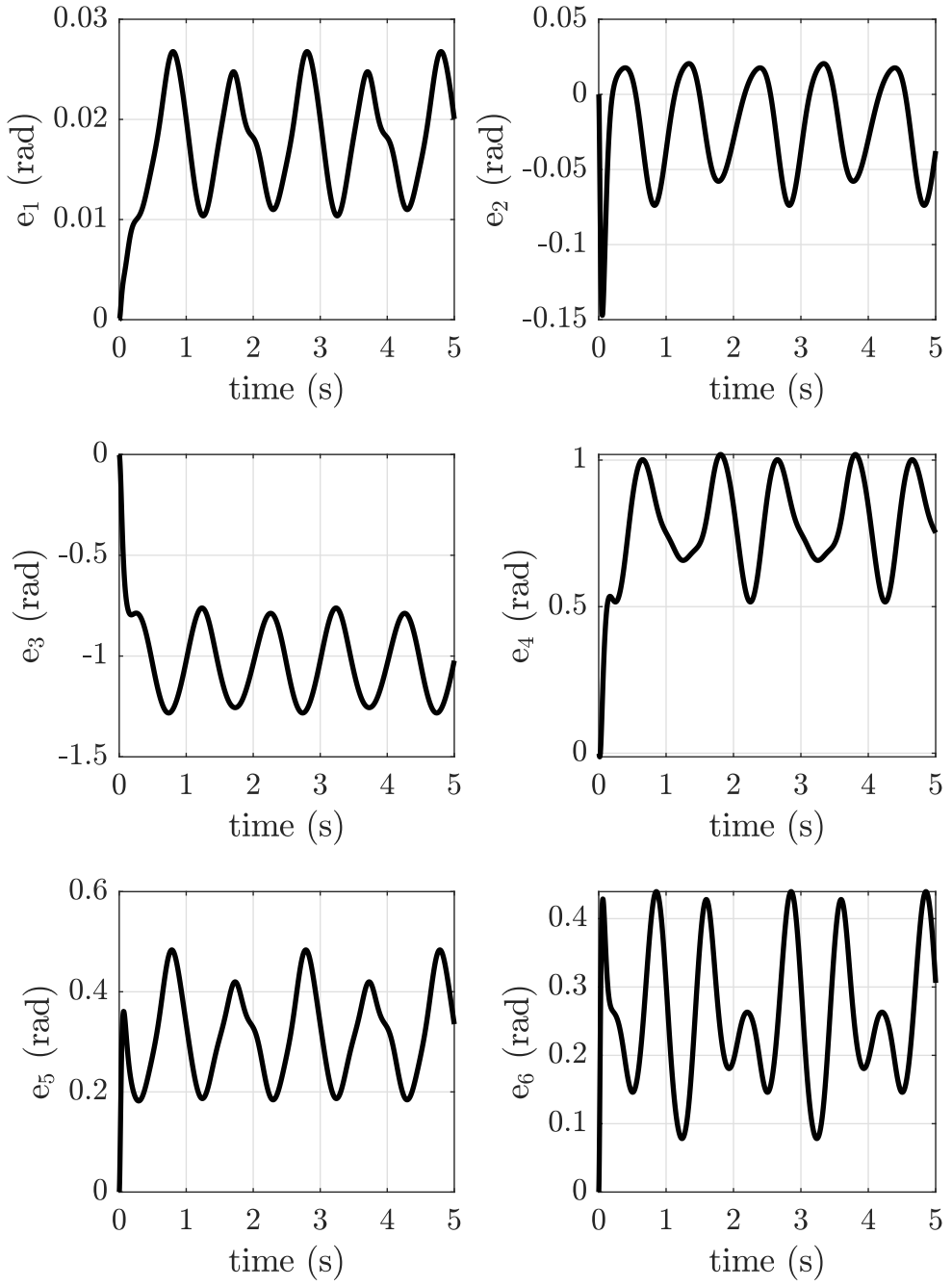


Figure 34: Angular position error of each joint of UR5 robot with Algorithm 18.

2.7.2 Three axis

The rosnode file that control the movement of the six joints of UR5 robot is Algorithm 19. In this file, the inverse dynamics is configure with $K_p = 600\ \frac{\text{N.m}}{\text{rad}}$ and $K_{d,i} = 2\sqrt{k_{p,i}}\ \frac{\text{N.m.s}}{\text{rad}}$. Figure 35 and 36 show trajectory tracking performance of each joint of the UR5 robot. In this case, all the joints present position error due to environment force.

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

```
# ==========================
# create a node:
rospy.init_node("node_inverse_dynamics_unilateral_compliant_contact
    ")

# 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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq0 =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 600*np.ones(ndof)
# derivative gain
kd = 2*np.sqrt(kp[0])*np.ones(ndof)
```

```
# control vector
tau = np.zeros(ndof)

# ============================
#   Enviornment configuration
# ============================
p0 = np.array([0.0, 0.0, 0.0]) # m
k_env = 10000*np.ones(3) # N/m
d_env = 2*np.sqrt(k_env[0])*np.ones(3) # N.s/m
n = np.array([0, 0, 1])
#===============
#   Simulation
#===============
t = 0.0              # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 5.0 # [sec]
force_start   = 1.0 # [sec]


while not rospy.is_shutdown():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
           sinusoidal_reference_generator(q0[1], 0.2, 1, t)
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
           sinusoidal_reference_generator(q0[4], 0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
           (0, last_q_des_1)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
           (0 , last_q_des_4)

    # position of end-effector
    p_ee = fkine_ur5(q)[0:3,3]
    # position jacobian
    J = jacobian_xyz_ur5(q)
    # velocity of end-effector
    dp_ee = np.dot(J, dq)

    if (np.dot(n, p0-p_ee))<0:
        # external force
        f_ext = np.multiply(k_env, p0-p_ee) - np.multiply(d_env,
           dp_ee) # N

        # external torque
        tau_ext = np.dot (J.transpose(), f_ext) # N.m
    else:
        # external torque
        tau_ext = np.zeros(6) # N.m
    print("f_ext: ", f_ext)
```

```
    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    M = ur5_robot.get_M()

    # compute nonlinear effects vector
    b = ur5_robot.get_b()

    # control law: PD control + Feedback linearization
    tau = M.dot(ddq_des +  np.multiply(kp, e) + np.multiply(kd, de)
        ) + b + tau_ext

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

Algorithm 19: Move the second and fifth joint of UR5 robot with the required movement of activity 2.7.2.

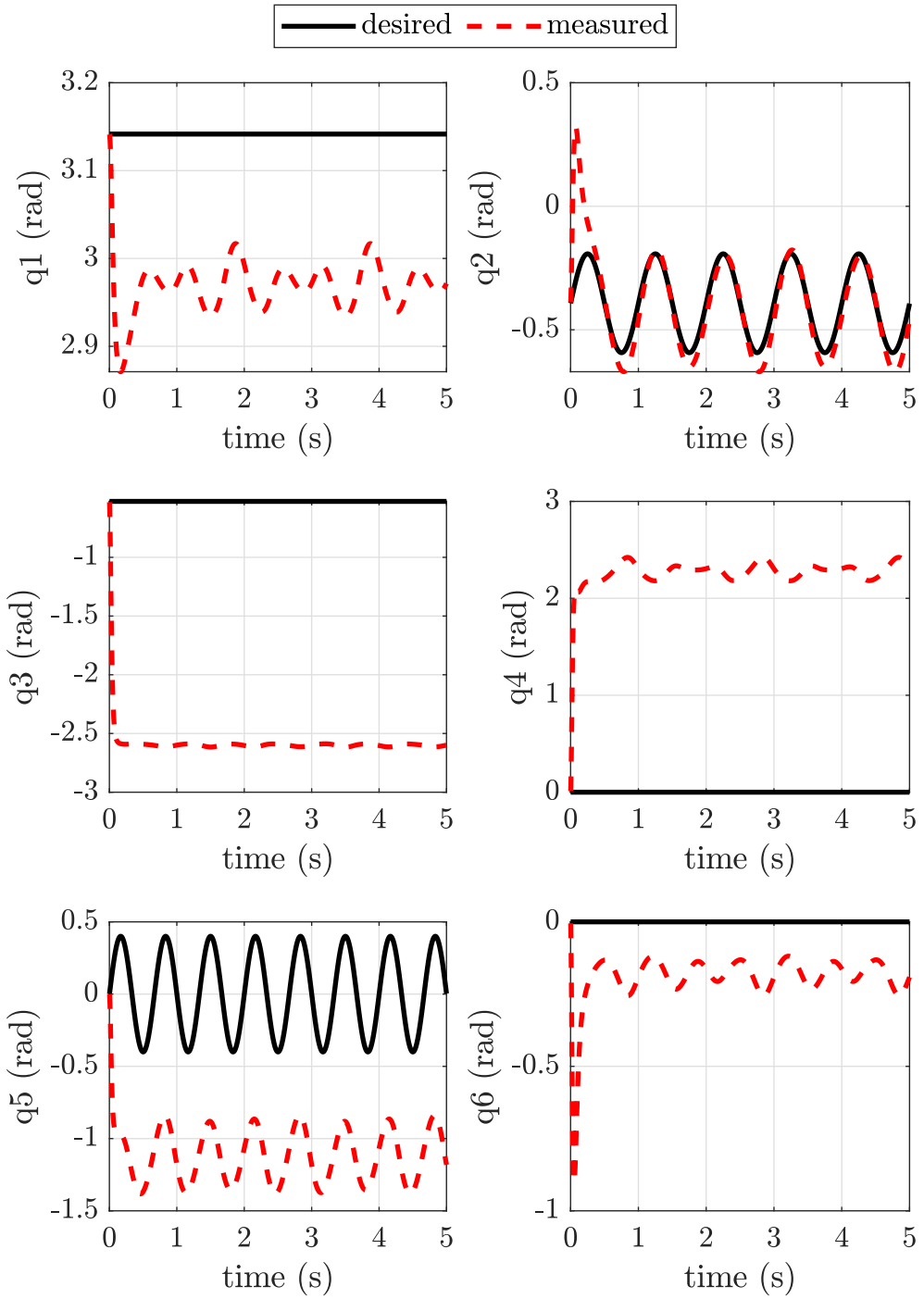


Figure 35: Angular position of each joint of UR5 robot with Algorithm 19.

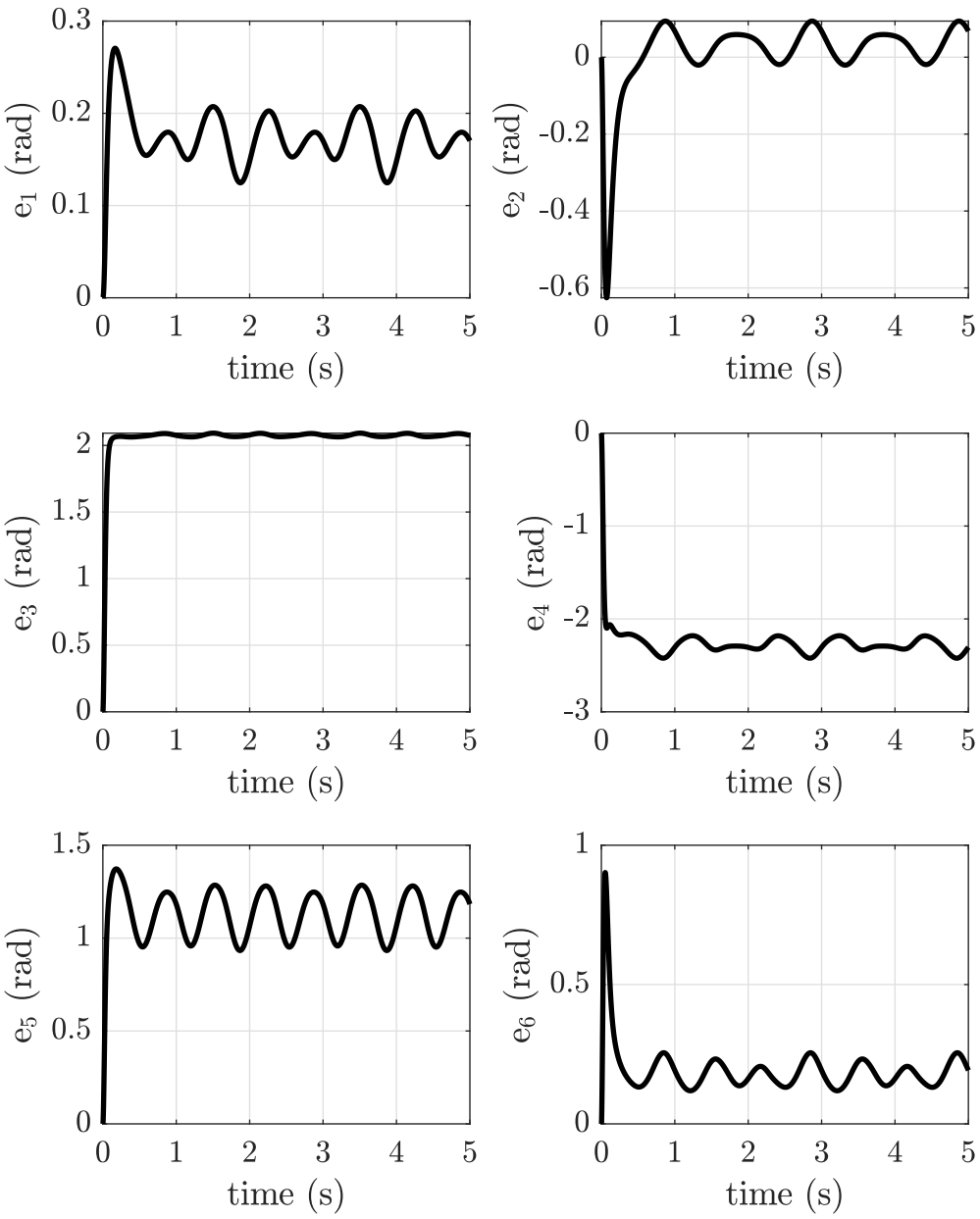


Figure 36: Angular position error of each joint of UR5 robot with Algorithm 19.

2.8 Coulomb friction

The objective of this activity is display the UR5 robot on rviz and control the motion of its joints with inverse dynamics control method with compliant contact. The simulation starts with the initial joint position $\begin{bmatrix}\pi & -\frac{\pi}{8} & -\frac{\pi}{6} & 0.0 & 0.0 & 0.0\end{bmatrix}$ rad, joint velocity $\begin{bmatrix}0.0 & 0.4\pi & 0.0 & 0.0 & 1.2\pi & 0.0\end{bmatrix}$ and external force $\begin{bmatrix}0.0 & 0.0 & 0.0\end{bmatrix}$ N. The second and fifth joints will move following a sinusoidal trajectory during the first 4 seconds and maintain a constant joint position during the last second. Finally, in this activity, the sinusoidal trajectory will be modified from 4 seconds to 5 seconds.

Figure 37 and 38 show the trajectory tracking of all joints of UR5 robot with $\mu = 0.2$ and $\mu = 0.8$.

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

# 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)  # 100 [Hz]
dt    = 1e-3      # 10  [ms]

# object(message) type JointState
jstate = JointState()

# ========================================
#   Set initial joint configuration of UR5
# ========================================
# initial configuration: position, velocity and acceleration
q0 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq0 =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 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([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
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 =   np.array([np.pi, -np.pi/8,  -np.pi/6, 0.0, 0.0, 0.0])
dq =  np.array([0.0, 0.4*np.pi, 0.0, 0.0, 1.2*np.pi, 0.0])
ddq = 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']

# number of degress of freedom
ndof = 6
```

```
# the class robot load the ur5.urdf
ur5_robot = Robot(ndof,q0, dq0, dt)
# create inertia matrix
M = np.zeros([ndof,ndof])
# create nonlinear effects vector
b = np.zeros(ndof)

# ================================
#   PD controller configuration
# ================================
# proportional gain
kp = 600*np.ones(ndof)
# derivative gain
kd = 2*np.sqrt(kp[0])*np.ones(ndof)
# control vector
tau = np.zeros(ndof)

# ============================
#   Enviornment configuration
# ============================
p0 = np.array([0.0, 0.0, 0.0]) # m
k_env = 10000*np.ones(3) # N/m
d_env = 1000*np.ones(3) # N.s/m
n = np.array([0, 0, 1])
mu = 0.2
#===============
#   Simulation
#===============
t = 0.0             # [sec]
sim_duration = 5.0  # [sec]
sine_duration = 5.0 # [sec]
force_start   = 1.0 # [sec]


while not rospy.is_shutdown():
    # generate sinusoidal joint reference
    if t<=sine_duration:
        # second link
        q_des[1], dq_des[1], ddq_des[1] =
           sinusoidal_reference_generator(q0[1], 0.2, 1, t)
        last_q_des_1 = q_des[1]
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] =
           sinusoidal_reference_generator(q0[4], 0.4, 1.5, t)
        last_q_des_4 = q_des[4]
    else:
        # second link
        q_des[1], dq_des[1], ddq_des[1] = step_reference_generator
           (0, last_q_des_1)
        # fifth link
        q_des[4], dq_des[4], ddq_des[4] = step_reference_generator
           (0 , last_q_des_4)

    # position of end-effector
    p_ee = fkine_ur5(q)[0:3,3]
    # position jacobian
```

```
    J = jacobian_xyz_ur5(q)
    # velocity of end-effector
    dp_ee = np.dot(J, dq)

    if (np.dot(n, p0-p_ee))<0:
        # external force
        f_ext = np.multiply(k_env, p0-p_ee) - np.multiply(d_env,
            dp_ee) # N
        if f_ext[0]>= mu*f_ext[2]:
            f_ext[0] = mu*f_ext[2]
        if f_ext[0]<= -mu*f_ext[2]:
            f_ext[0] = -mu*f_ext[2]

        if f_ext[1]>= mu*f_ext[2]:
            f_ext[1] = mu*f_ext[2]
        if f_ext[1]<= -mu*f_ext[2]:
            f_ext[1] = -mu*f_ext[2]
        # external torque
        tau_ext = np.dot (J.transpose(), f_ext) # N.m
    else:
        # external torque
        tau_ext = np.zeros(6) # N.m
    print("f_ext: ", f_ext)
    # error: position and velocity
    e  =  q_des - q
    de  =  dq_des - dq

    # compute inertia matrix
    M = ur5_robot.get_M()

    # compute nonlinear effects vector
    b = ur5_robot.get_b()

    # control law: PD control + Feedback linearization
    tau = M.dot(ddq_des +  np.multiply(kp, e) + np.multiply(kd, de)
        ) + b + tau_ext

    # send control signal
    ur5_robot.send_control_command(tau)
    # update states
    q, dq, ddq = ur5_robot.
        read_joint_position_velocity_acceleration()

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

    # update time
    t = t + dt

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

```
        break
    rate.sleep()
```

Algorithm 20: Move the second and fifth joint of UR5 robot with the required movement of activity 2.8.

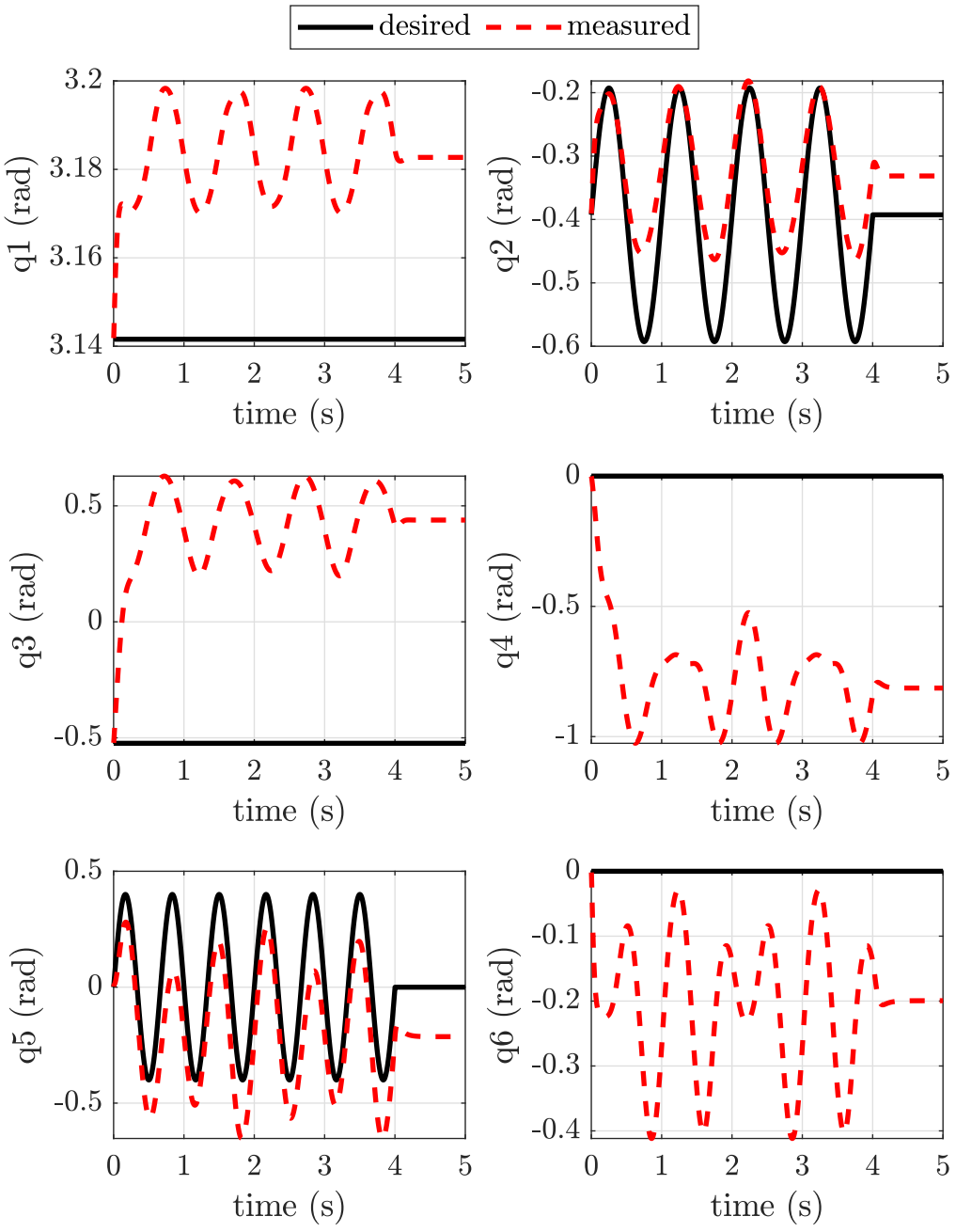


Figure 37: Angular position of each joint of UR5 robot with Algorithm 20

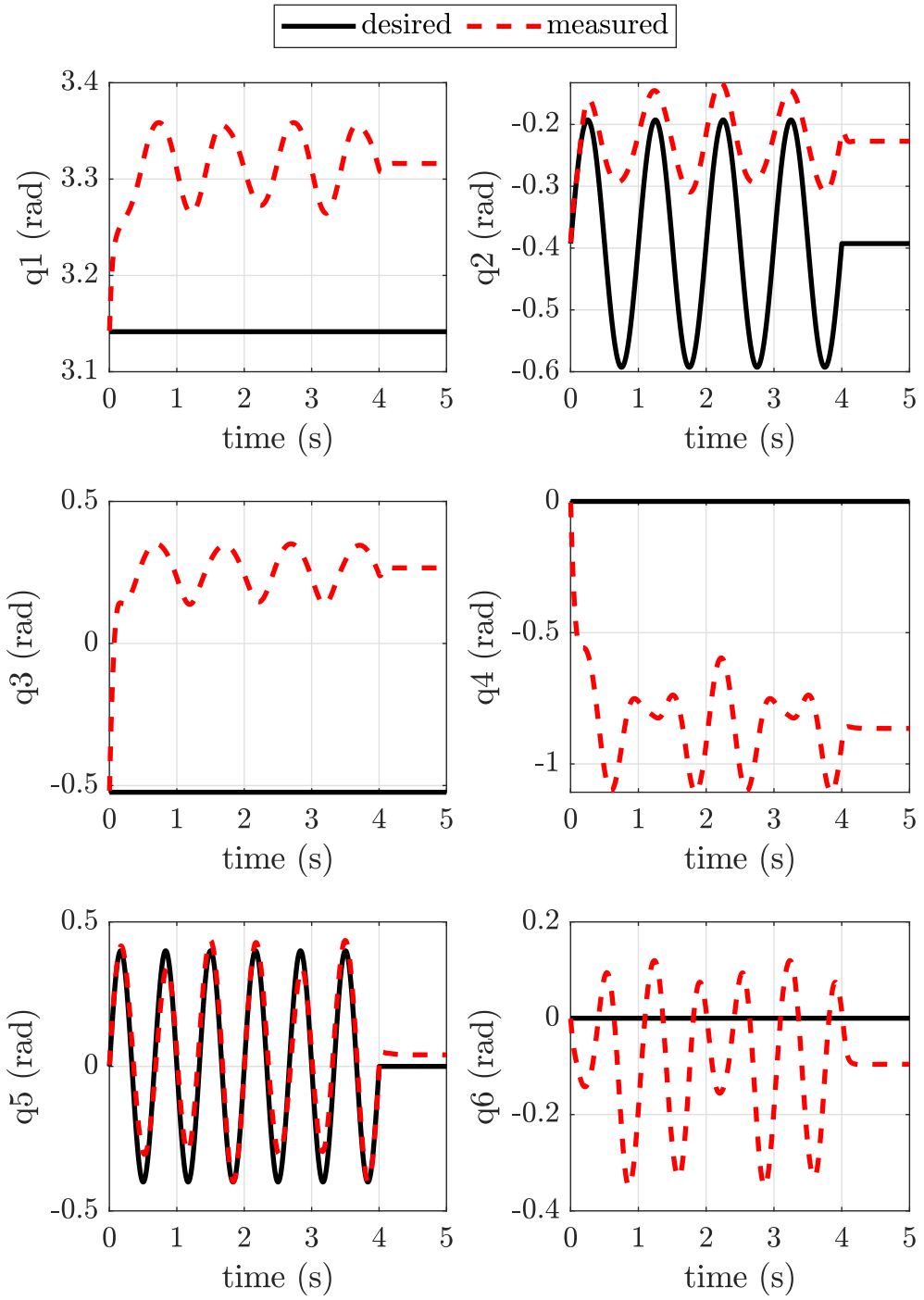


Figure 38: Angular position of each joint of UR5 robot with Algorithm 20