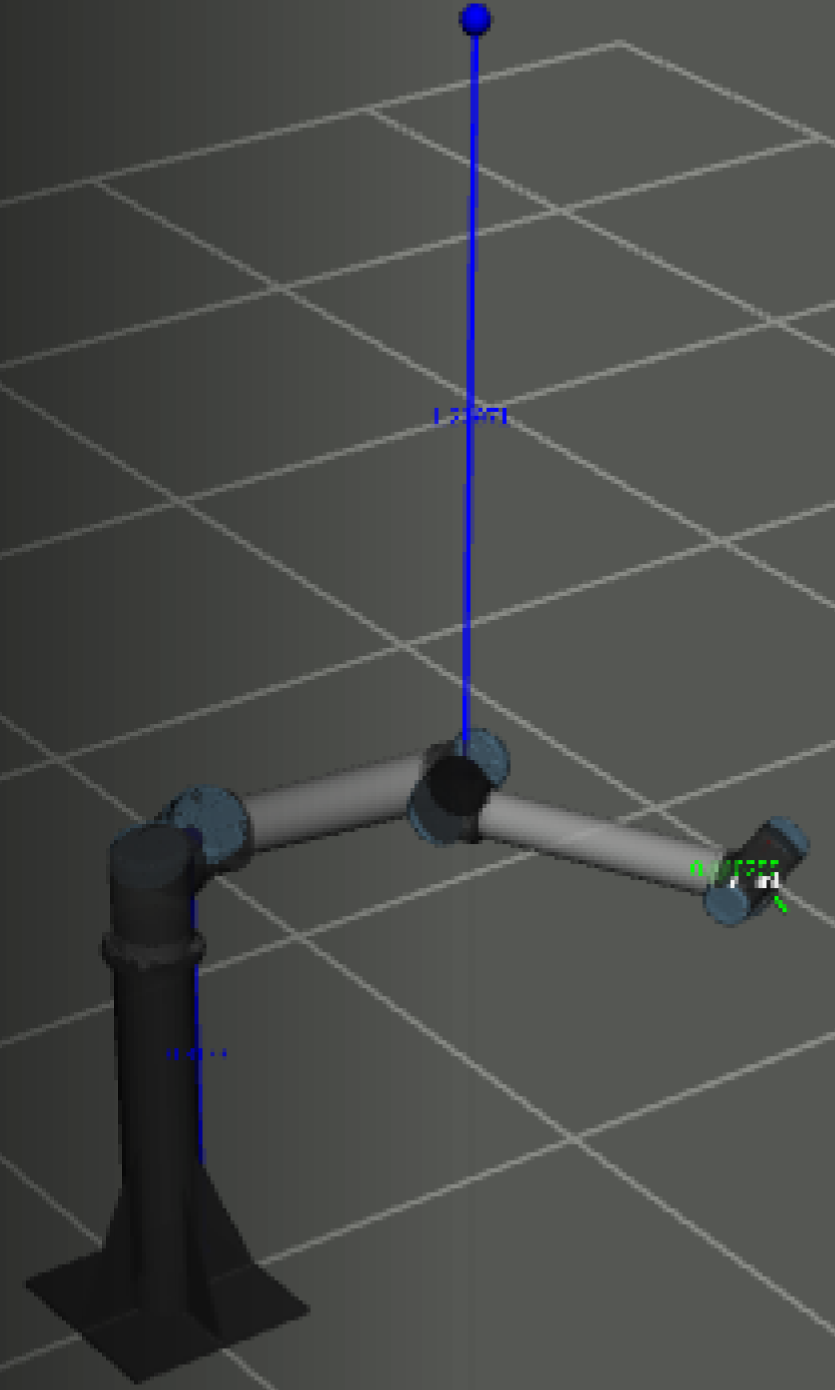
Modelling and Control of Robot Manipulators



Mandatory Assignment: Robot Motion Control in ROS

Objectives

- To simulate the robot motion control of Universal Robot 10 (UR 10) within the Robot Operating System (ROS).
- To utilize inverse kinematics and a trajectory following interface of the Universal Robot (UR10) and simulate the robot in Gazebo.

Inverse Kinematics

- Given desired end-effector position, we need to determine the robot joint configuration.
- To do so, we import the urdf file into MATLAB.
- Then, we generate an inverse kinematics solver iksolver for the rigid body tree object

```
% Import urdf file
ur10 = importrobot('ur10.urdf');

%Generate a inverse kinematics solver object for the rigid body tree ur

ikSolver = robotics.InverseKinematics('RigidBodyTree',ur10);
```

```
ikSolver =

  inverseKinematics with properties:

        RigidBodyTree: [1×1 rigidBodyTree]
      SolverAlgorithm: 'BFGSGradientProjection'
     SolverParameters: [1×1 struct]
```

Define a Target Pose

- The target pose is a result from a translation and a series of rotation by Euler angles.

Translation: $[d_x \;\; d_y \;\; d_z] = [0.6 \;\; 0 \;\; 1.0]$

Rotation in Euler angles representation: $[\phi \;\; \theta \;\; \psi] = \left[\frac{\pi}{4} \;\; \frac{\pi}{4} \;\; -\frac{\pi}{4}\right]$

```
%targetPosition=[0.9 2 1.0];
targetPosition=[0.6 0 1.0];
targetOrientation=[pi/4, pi/4, -pi/4];

% get tform of target pose
tformTrans=trvec2tform(targetPosition);
tformRot=eul2tform(targetOrientation);
tformTargetPose=tformTrans*tformRot;
```

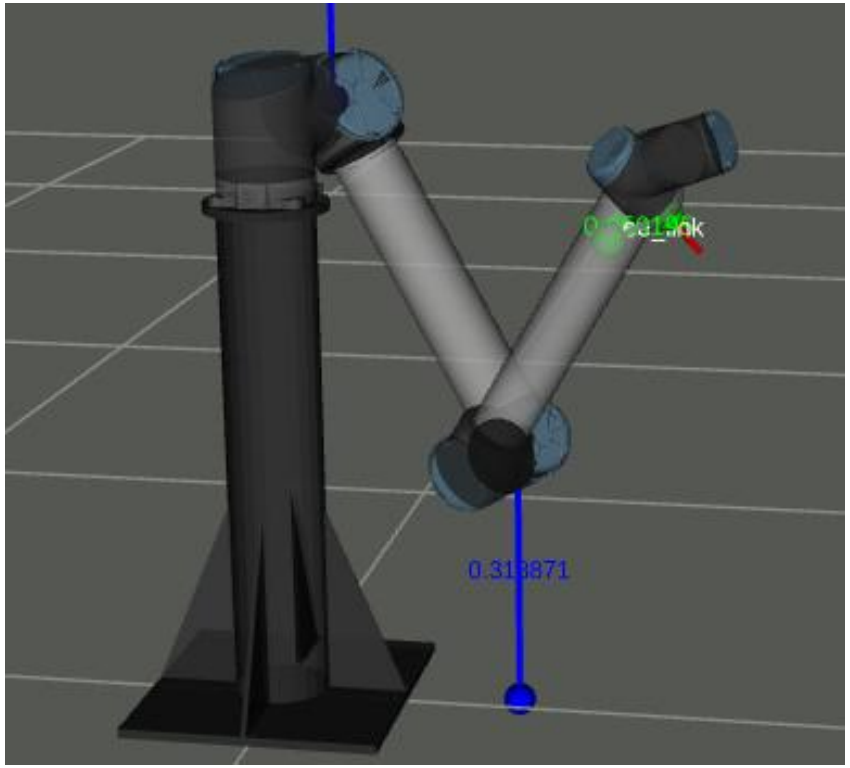
- tformTrans = $\begin{bmatrix} 1 & 0 & 0 & 0.6 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$,
- tformRot = $\begin{bmatrix} 0.5 & -0.8536 & -0.1464 & 0 \\ 0.5 & 0.1464 & 0.8536 & 0 \\ -0.7071 & -0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- tformTargetPose = $tformTrans * tformRot = \begin{bmatrix} 0.5 & -0.8536 & -0.1464 & 0.6 \\ 0.5 & 0.1464 & 0.8536 & 0 \\ -0.7071 & -0.5 & 0.5 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Determine the targetConf

- Unit Error Weight Vector = $[1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1]$
- Initial Guess = $[0\ \ 1\ \ -2\ \ 2\ \ 1\ \ 1]$

```
weights = ones(6,1);
initial_guess = JointVec2JointConf(ur10,[0 1 -2 2 1 1]);

[targetConf, solnInfo] = ikSolver('ee_link',tformTargetPose,weights,initial_guess);
```

- The target configuration looks like this:



Joint Name	Joint Position
shoulder_pan_joint	-0.3824
shoulder_lift_joint	0.9804
elbow_joint	-2.0033
wrist_1_joint	2.22
wrist_2_joint	0.8626
wrist_3_joint	1.3279

Connect to ROS, Show topics, Subscriber

- Connect to ROS in MATLAB:

```
rosinit('http://ubuntu:11311/')

Initializing global node /matlab_global_node_80327 with NodeURI http://192.168.153.1:61888/ and MasterURI http://ubuntu:11311/.
```

- Show topics from /ur10/joint_states and ur10/vel_based_pos_traj_controller/command:

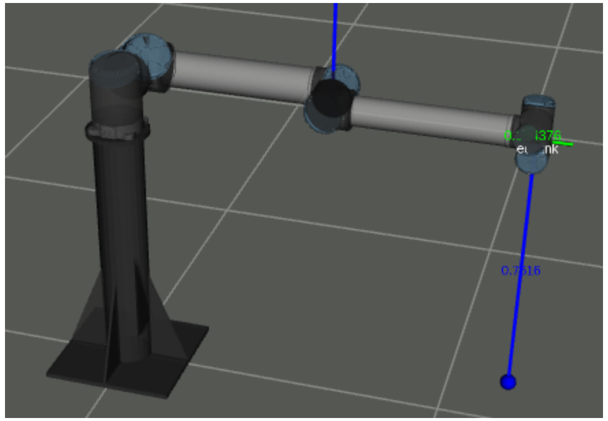
```
rostopic info /ur10/joint_states

Type: sensor_msgs/JointState

Publishers:
* /gazebo (http://192.168.153.128:42209/)

Subscribers:
* /robot_state_publisher (http://192.168.153.128:37685/)
* /ur_distance_publisher (http://192.168.153.128:33147/)
```

```
rostopic info ur10/vel_based_pos_traj_controller/command

Type: trajectory_msgs/JointTrajectory

Publishers:

Subscribers:
* /gazebo (http://192.168.153.128:42209/)
```

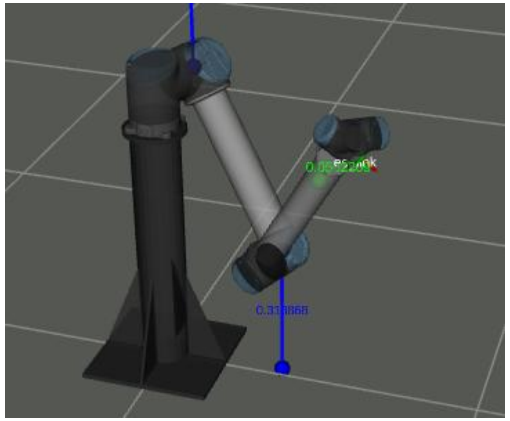
- Instantiate a subscriber: jointStateSub = rossubscriber('/ur10/joint_states');

Point-to-Point Motion

- Perform point-to-point motion from current configuration to the target configuration from task 3.



Current Configuration



Target Configuration

```
%create publisher
jointTrajectoryPub = rospublisher('/ur10/vel_based_pos_traj_controller/command');

%convert target configuration
q_target = JointConf2JointVec(targetConf);
t_target = 2.5;

% time offset to let robot finish motion
t_offset = 5; % increase if robot does not reach it's configuration in time

%send target configuration to the robot
jointTrajectoryPub.send(JointVec2JointTrajectoryMsg(ur10,q_target,t_target))
pause(t_target + t_offset)
```

Creates a publisher for the topic **/ur10/vel_based_pos_traj_controller/command**

Joint configuration of UR10 robot (denoted by $\dot{q}_1$ to $\dot{q}_6$)

Call the function JointVec2JointTrajectoryMsg to send the target configuration to the UR10 robot.

Topic Interface and Action Interface

- **Topic Interface**

❑Uses the **trajectory_msgs/JointTrajectory** message

❑Fire-and-forget alternative, meaning the execution command is not monitored.

❑No mechanism to notify the publisher of the command about tolerance violations.

- **Action Interface (Used in this case)**

❑Uses **control_msgs/FollowJointTrajectoryGoal** too specify trajectories.

❑Typically used for tasks that require a sequence of steps.

❑In this assignment, a node sends a goal to an action server to move the UR10 robot to the target configuration, receiving periodic feedback on the progress of the movement.

Execute the rated loop

- Rated loop is used instead of the pause command to monitor the temporal evaluation of the joint state vector.

```
% Move to start configuration
q_home = [0 0 0 0 0 0];
t_home = 2.5;
jointTrajectoryPub.send(JointVec2JointTrajectoryMsg(ur10,q_home,t_home));
pause(t_home+t_offset)

% Define joint velocity
qvel=[0 0 0 0 0 0];

% Create rate object
rate = 50; % Hz
rateObj=robotics.Rate(rate);
tf = t_target + 0.5;
rateObj.reset; % reset time of rate object

% Preallocation
N = tf*rate;
timeStamp=zeros(N,1);
jointStateStamped=zeros(N,6); % array to record joint pose
jointVelStamped=zeros(N,6); % array to record joint velocity
```

Call the function JointVec2JointTrajectoryMsg to send the **HOME configuration** to the UR10 robot.

Set rate = 50 as the loop rate in Hz or the number of iterations per second. Then create the rate object. With rateObj.reset, the clock of the rate object rateObj is reset.

N=7*50 = 350

```
% Store start time
jointStateMsg=jointStateSub.receive();
t0=double(jointStateMsg.Header.Stamp.Sec)+ double(jointStateMsg.Header.Stamp.Nsec)*10^-9;

% Move and monitor
jointTrajectoryPub.send(JointVec2JointTrajectoryMsg(ur10,q_target,t_target));

for i=1:N
    % Receive and convert
     jointStateMsg=jointStateSub.receive();
     [jointState, jointVel]=JointStateMsg2JointState(ur10,jointStateMsg);

    % Store signals
     jointStateStamped(i,:)=jointState;
     jointVelStamped(i,:)=jointVel;
     timeStamp(i)=double(jointStateMsg.Header.Stamp.Sec) + ...
         double(jointStateMsg.Header.Stamp.Nsec)*10^-9-t0;
%
     waitfor(rateObj);
end
```

timeStamp

350x1 double

	1	2
1	0.0200	
2	0.0300	
3	0.0400	
4	0.0700	
5	0.0900	
6	0.1000	
7	0.1100	
8	0.1300	
9	0.1400	
10	0.1500	
11	0.1800	
12	0.1900	
13	0.2100	
14	0.2200	
15	0.2300	

Based on the for-loop above, the jointStateMsg is received and converted for 350 times (Because N = 350). The jointState and jointVel are determined from the function **JointStateMsg2JointState** by converting the received message to the 2 parameters. For each iteration, the time stamps are captured and is appended into the timestamp(i), as well as their corresponding jointState and jointVel. The waitfor(rateObj) means synchorinize the loop execution time of MATLAB and the ROS interface.

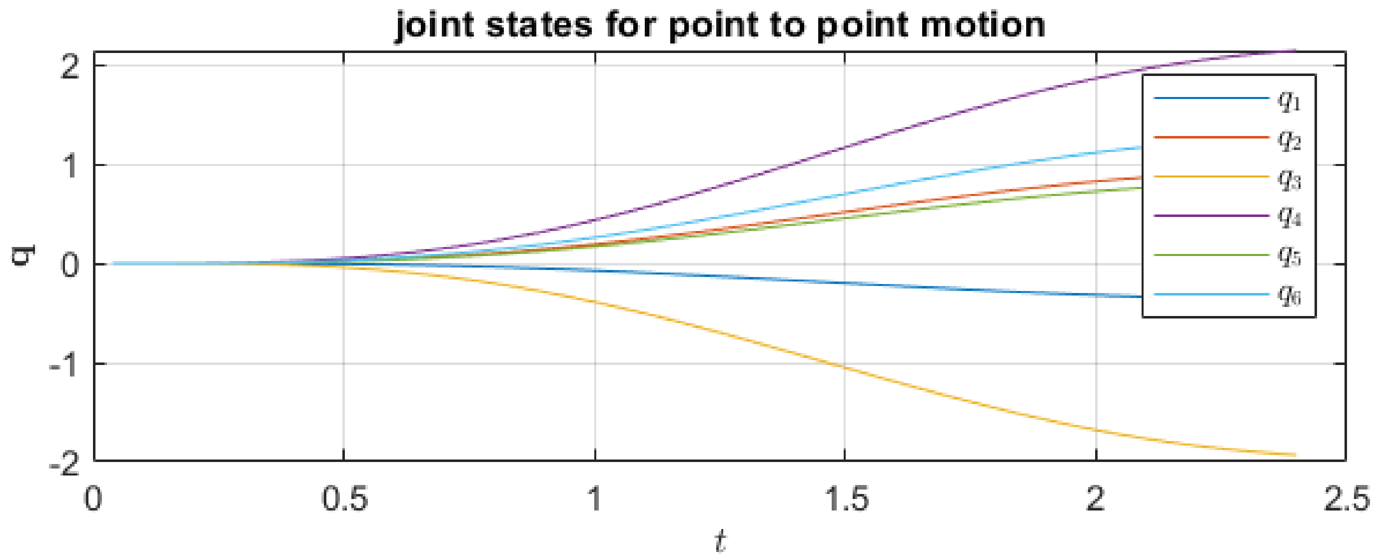
jointStateMsg

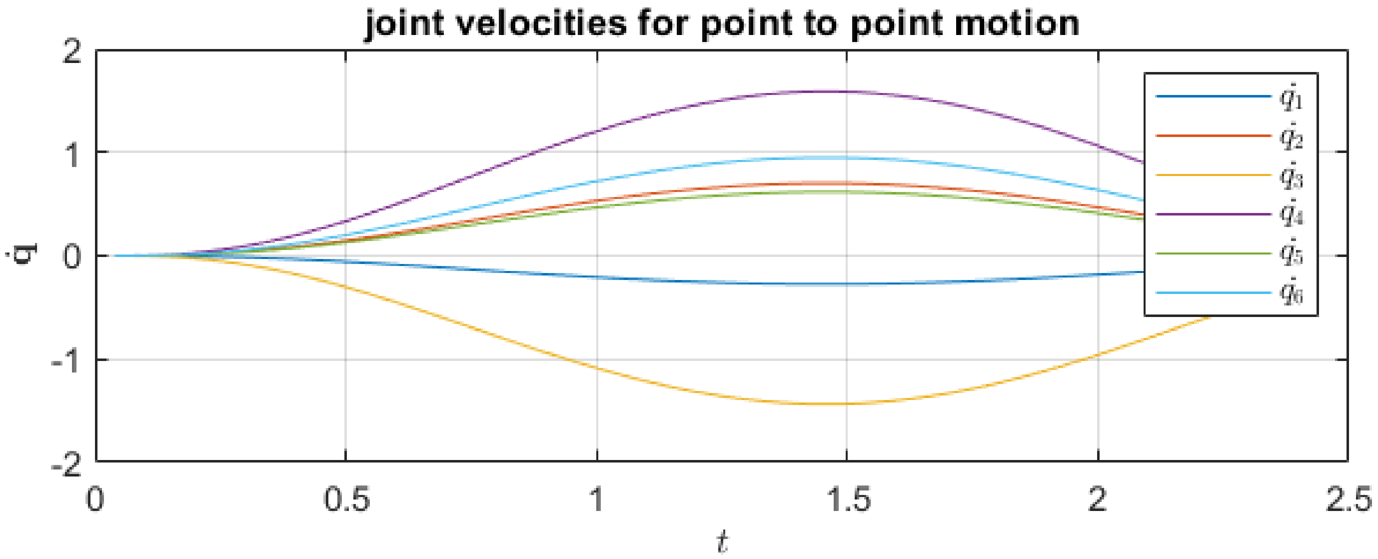
1x1 JointState

Property	Value
MessageType	'sensor_msgs/JointState'
Header	*1x1 Header*
Name	*6x1 cell*
Position	[-1.0015;0.0015;-0.9999;1.0013;0.0013;1.9990]
Velocity	[0.0071;-0.0068;-6.5984e-04;-0.0058;-0.0061;0.0047]
Effort	[0;0;0;0;0;0]

The **JointState** consists of the Position, Velocity, and the Effort of each Joint of UR10 robot. When the program is executed for **N** iterations, the Position and Velocity will be updated based on the real-time trajectory of UR10 robot from current configuration to the target configuration.

Graphs of Joint State and Joint Velocity

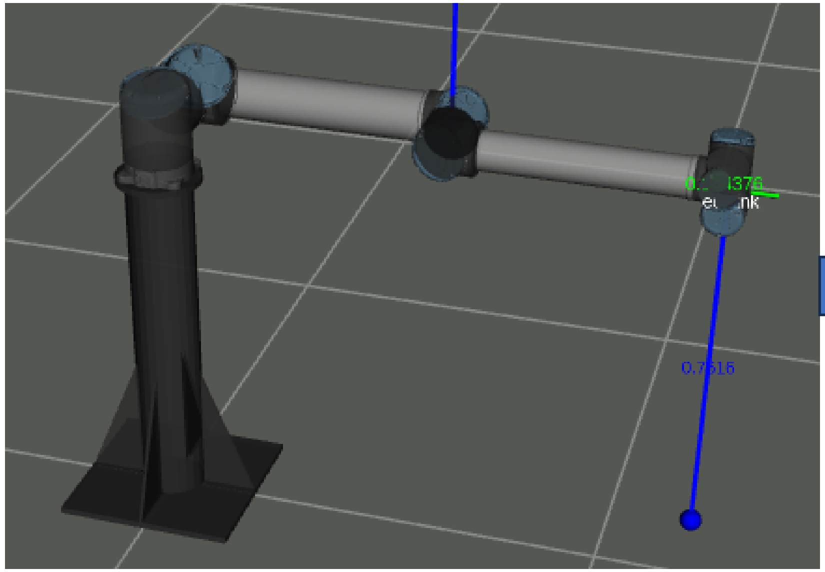




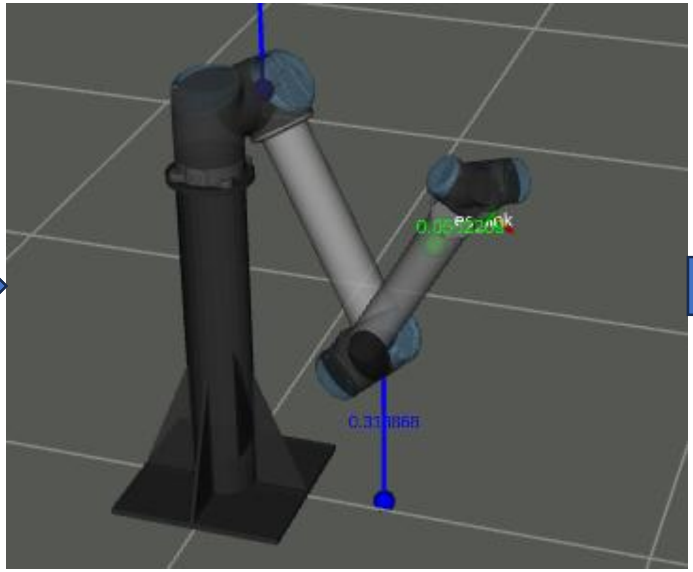
Motion Trajectory via Two-Way Points (Task 12)

```
% Back to home
jointTrajectoryPub.send(JointVec2JointTrajectoryMsg(ur10,q_home, t_home));
pause(t_home+t_offset)

% Create second waypoint
t_targets=[2.5; 6.0];
qf = [-1 0 -1 1 0 2];
q_targets=[q_target; qf];

task='12';
% task='14';

switch task
    case '12'
        % stop at intermediate way-point
         qvel=zeros(2,6); % stop at intermediate way-point


    case '14'
%          % stop at intermediate way-point
         qvel1 = [-0.4 0.0 0.0 -0.3 0.0 0.0];
         qvelf = [0 0 0 0 0 0];
         qvel=[qvel1; qvelf];

end
```

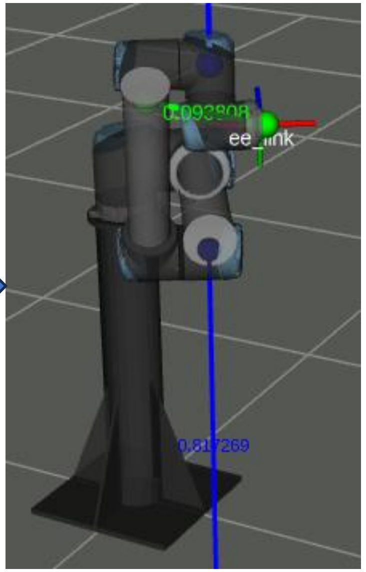
Set the time at the points at $t_1 = 2.5s, t_2 = 6s$
qf = The final joint position at $t = 6s$
Basically, the UR10 robot will stop at the target configuration at $t = 2s$, and move to the final configuration at $t = 6s$



At $t = 0s$



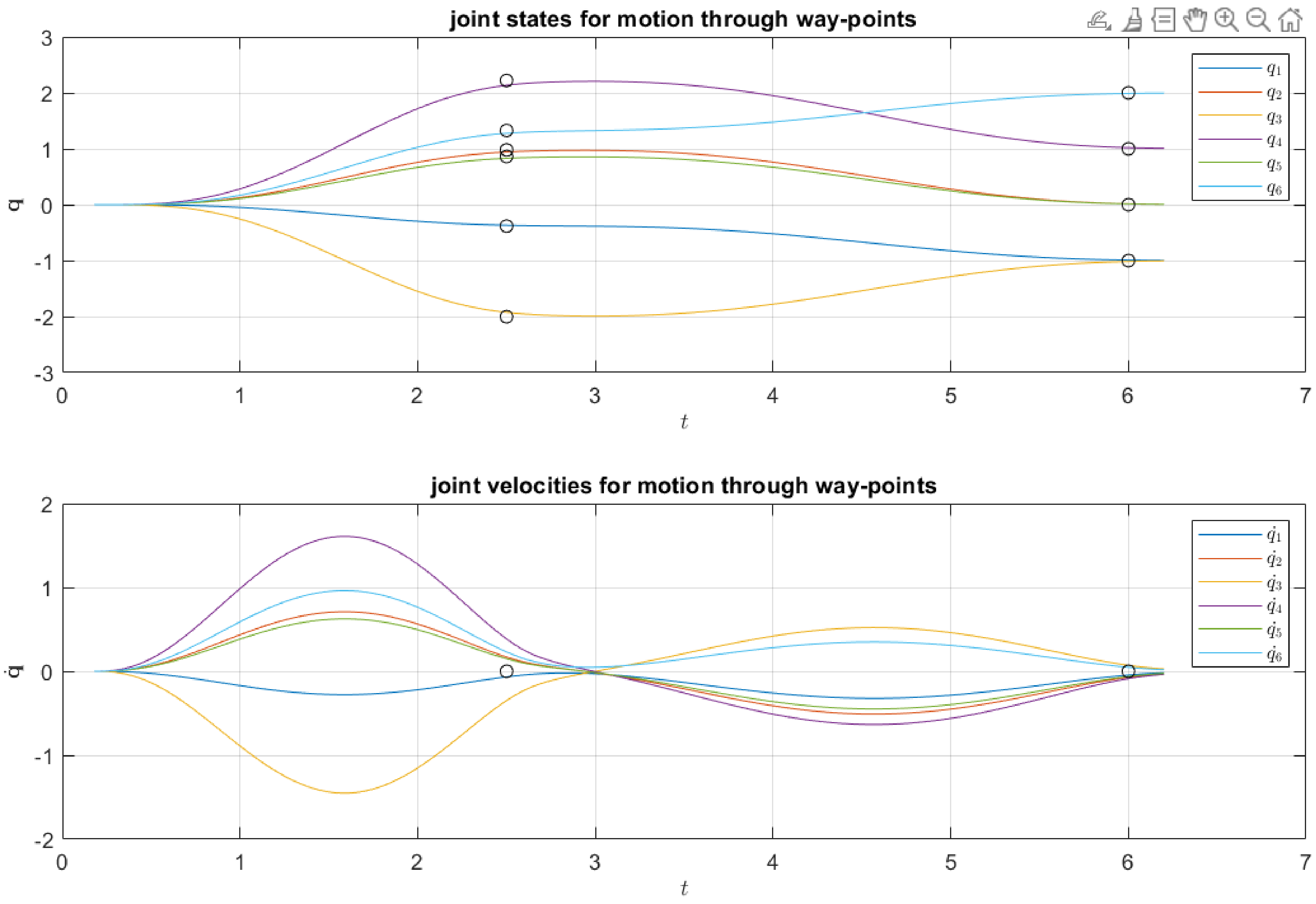
At $t = 2s$ (1st way-point)



At $t = 6s$ (2nd way-point)

- At the first way-joint, q_2 and q_5 either stop or undergo a motion reversal.
- However, it seems awkward that q_1 and q_4 come to a stop before continue to the 2nd way-point.

Graphs of Joint State and Joint Velocity



Motion Trajectory via Two-Way Points (Task 14)

- Compared to Task 12, now a velocity vector is introduced in the 1st way-point.

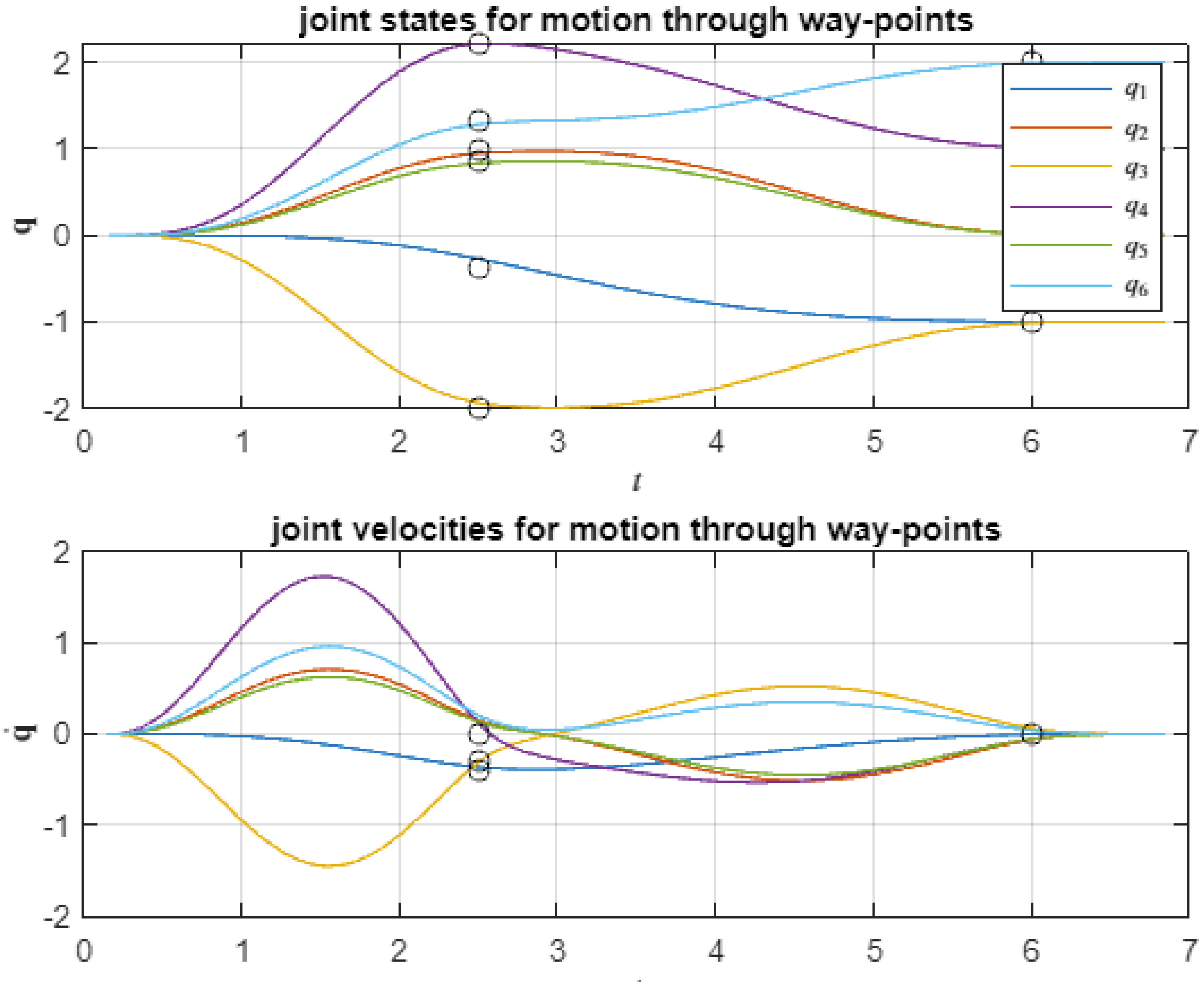
$\dot{\boldsymbol{q}} = [-\mathbf{0.4}\ \ \mathbf{0}\ \ \mathbf{0}\ \ -\mathbf{0.3}\ \ \mathbf{0}\ \ \mathbf{0}]$

- Which means, q_1 *and* q_4 won't stop when the UR10 robot stops at the 1st-way point. This serves as a transition from first way-point to the second way-point.

```
    case '14'
%         % stop at intermediate way-point, q1 and q4 will continue moving
        qvel1 = [-0.4 0.0 0.0 -0.3 0.0 0.0];
        qvelf = [0 0 0 0 0 0];
        qvel=[qvel1; qvelf];

end
```

- qvel1 = 1st way-point, where $\dot{q_1}$ and $\dot{q_4}$ will continue moving.
- qvelf = 2nd way-point, where all joints will stop.

Graphs of Joint State and Joint Velocity



Display list of actions on the ROS network (Task 15)

```
rosaction list;
```

```
/ur10/vel_based_pos_traj_controller/follow_joint_trajectory
```

List all ros actions -> we only have follow_joint_trajectory

```
rosaction info 'ur10/vel_based_pos_traj_controller/follow_joint_trajectory';
```

```
Action Type: control_msgs/FollowJointTrajectory

Goal Message Type: control_msgs/FollowJointTrajectoryGoal
Feedback Message Type: control_msgs/FollowJointTrajectoryFeedback
Result Message Type: control_msgs/FollowJointTrajectoryResult
```

Check control message types contained in follow_joint_trajectory

```
Action Server:
* /gazebo (http://192.168.52.129:46783/)

Action Clients: None
```

Gazebo is running a http server
There are no clients connected yet (we will change that)

Instantiate an action client and an empty goal message (Task 16)

Set up a client for receiving follow_joint_trajectory

```
followJointTrajectoryTopicName=['ur10/vel_based_pos_traj_controller/follow_joint_trajectory'];
[followJointTrajectoryActClient]=rosactionclient(followJointTrajectoryTopicName);
waitForServer(followJointTrajectoryActClient,15);

followJointTrajectoryMsg=rosmessage('control_msgs/FollowJointTrajectoryGoal');
```

Wait until the server is ready (blocking)

Set up message for asking the robot to start following the trajectory

Inspect the structure of the goal message (Task 17)

```
followJointTrajectoryMsg
```

```
followJointTrajectoryMsg =
  ROS FollowJointTrajectoryGoal message with properties:

          MessageType: 'control_msgs/FollowJointTrajectoryGoal'
           Trajectory: [1×1 JointTrajectory]
        PathTolerance: [0×1 JointTolerance]
        GoalTolerance: [0×1 JointTolerance]
    GoalTimeTolerance: [1×1 Duration]

  Use showdetails to show the contents of the message
```

Inspect the helper function (Task 18)

```
jointConf=homeConfiguration(robot);
followJointTrajectoryMsg=rosmessage('control_msgs/FollowJointTrajectoryGoal');
followJointTrajectoryMsg.Trajectory=JointVec2JointTrajectoryMsg(robot, q, t, qvel, qacc);
for j=1:size(q,2)
    followJointTrajectoryMsg.GoalTolerance(j)=rosmessage('control_msgs/JointTolerance');
    followJointTrajectoryMsg.PathTolerance(j)=rosmessage('control_msgs/JointTolerance');
    %followJointTrajectoryMsg.GoalTimeTolerance(j)=rosmessage('std_msgs/Duration');
    followJointTrajectoryMsg.GoalTimeTolerance.Sec=0;
    followJointTrajectoryMsg.GoalTimeTolerance.Nsec= int32(1e7);  % 10 ms
    followJointTrajectoryMsg.GoalTolerance(j).Name=jointConf(j).JointName;
    followJointTrajectoryMsg.PathTolerance(j).Name=jointConf(j).JointName;
    followJointTrajectoryMsg.GoalTolerance(j).Position=qtol;
    followJointTrajectoryMsg.GoalTolerance(j).Velocity=qveltol;
    followJointTrajectoryMsg.GoalTolerance(j).Acceleration=qacctol;
    followJointTrajectoryMsg.PathTolerance(j).Position=qtol;
    followJointTrajectoryMsg.PathTolerance(j).Velocity=qveltol;
    followJointTrajectoryMsg.PathTolerance(j).Acceleration=qacctol;
```

Set up FollowJoinTrajectoryMessage and set the desired Trajectory (like before)

Set up tolerances for reaching the trajectory goal points

Follow a trajectory via the action client (blocking) (Task 19)

Home robot (like before)

```
% Back to home
jointTrajectoryPub.send(JointVec2JointTrajectoryMsg(ur10,q_home,t_home));
pause(t_home+t_offset)

% Create action msg
followJointTrajectoryMsg = JointVec2FollowJointTrajectoryMsg(ur10,q_targets,t_targets,qvel);

% % Send message
[resultMsg, resultState]=followJointTrajectoryActClient.sendGoalAndWait(followJointTrajectoryMsg);

% % Evaluate result
if (resultMsg.ErrorCode)
     disp('Arm motion error');
     showdetails(resultMsg);
 else
     disp(['UR arm motion completed with state ', resultState, '.']);
end
```

Set up FollowJointTrajectoryGoal And send it through our client

Wait for the goal to complete (blocking call)

Check for errors

Follow a trajectory via the action client (non-blocking + monitoring)

- Home robot (like before)
- Use Join State Subscriber to receive robot trajectory (like before)

- Use our previously instantiate client to start the motion

```
% Move and monitor
followJointTrajectoryMsg = JointVec2FollowJointTrajectoryMsg(ur10,q_targets,t_targets,qvel);
followJointTrajectoryActClient.sendGoal(followJointTrajectoryMsg);
```

Non-blocking call, so we can continue running our code while the robot is moving

Plot the joint states and joint velocities

- We plot the robot motion (like before) and plot (like before)