This assignment is mandatory for students in the master robotics and automation in order to fulfill the requirements for **six** course credits. Once you completed the assignment you are supposed to upload your code, a PDF of your code including the results and a PDF (max. 1 DIN A4 page) with explanation what you have done to the Moodle workspace of this course. The submissions are individually and will be checked for plagiarism.

This lab has the main objective to familiarize students with the simulation and control of robotic manipulators within the Robot Operating System 2 (ROS2). In particular you are going to utilize inverse kinematics and a trajectory following interface of the Universal Robot 10 (UR10) and simulate the robot in Gazebo.

Further explanations regarding the code required for this task are visible in the Matlab Livescript or the Jupyter Notebook provided in the Moodle workspace of this course.

Inverse Kinematics

The inverse kinematics problem is the antagonist to the forward kinematics problem: Given the desired end effector pose determine the joint configuration to achieve that pose. Inverse kinematics either rely on a closed-form solution or a numerical solution. Analytical solutions provide a set of equations that fully describe the connection between the end effector position and the joint angles. For standard serial manipulators, such as robot arms with a spherical wrist, closed form solutions of the inverse kinematics exist.

Numerical solutions are universal as they rely on numerical algorithms, and provide solutions even if no closed-form solution is available. For a given pose there might be multiple solutions, e.g. elbow-up and elbow-down posture, or no solution at all, e.g. if the end effector pose is outside the manipulators workspace.

For a numerical solution the inverse kinematics problem is formulated as an optimization problem. In fact the objective is to minimize the error between the target transform $\mathbf{H}_t$ for the end effector and the forward kinematics of a joint configuration $\mathbf{H}_e(\mathbf{q})$. For that purpose we decompose the pose error into a position and a rotation part. The position error is simply the distance of the origin of both transforms

$$\mathbf{e_p}(\mathbf{q}) = [e_x \;\; e_y \;\; e_z]^T = [p_{xt} - p_{xe} \;\; p_{yt} - p_{ye} \;\; p_{zt} - p_{ze}]^T \tag{1}$$

with
$[p_{xt} \;\; p_{yt} \;\; p_{zt}]^T = [\mathbf{H}_t(1,4) \; \mathbf{H}_t(2,4) \; \mathbf{H}_t(3,4)]^T$
and
$[p_{xe} \;\; p_{ye} \;\; p_{ze}]^T = [\mathbf{H}_e(\mathbf{q})(1,4) \; \mathbf{H}_e(\mathbf{q})(2,4) \; \mathbf{H}_e(\mathbf{q})(3,4)]^T$

For the rotation part the relative orientation matrix $\mathbf{R}_d = \mathbf{R}_t\mathbf{R}_e(\mathbf{q})'$ is converted to an axis angle representation $[\alpha \;\; w_x \;\; w_y \;\; w_z]^T$. The orientation error is given by

$$\mathbf{e_w}(\mathbf{q}) = [e_{wx} \;\; e_{wy} \;\; e_{wz}]^T = [\alpha w_x \;\; \alpha w_y \;\; \alpha w_z]^T \tag{2}$$

The overall error is a six-dimensional vector $\mathbf{e}(\mathbf{q}) = [e_x \; e_y \; e_z \; e_{wx} \; e_{wy} \; e_{wz}]^T$.

The Robotics system toolbox provides a helper function to calculate the error vector $\mathbf{e}$ between two transforms

```
robotics.manip.internal.IKHelpers.poseError(tformt, tformq)
```

The objective is to minimize the error norm in the least squares sense.

$$\min_{\mathbf{q}} \frac{1}{2}||\mathbf{e}(\mathbf{q})|| = \min_{\mathbf{q}} \frac{1}{2}(e_x^2 + e_y^2 + e_z^2 + e_{wx}^2 + e_{wy}^2 + e_{wz}^2) \tag{3}$$

A more general error is obtained by weighting the individual errors

$$\min_{\mathbf{q}} \frac{1}{2}\mathbf{e}_W(\mathbf{q}) = \min_{\mathbf{q}} \frac{1}{2}\mathbf{e}'\mathbf{W}\mathbf{e} \tag{4}$$

in which $\mathbf{W}$ is a positive definite matrix, often diagonal. In fact if a feasible solution $\mathbf{q}^*$ of the inverse kinematics problem exists, namely the target pose is within the robot workspace then the pose error becomes zero

$$\mathbf{e}(\mathbf{q}^*) = \mathbf{0} \tag{5}$$

The problem (4) constitutes a non-linear least squares problem for which efficient optimization algorithms exist. The Jacobian is the matrix of first order derivatives of a vector valued function. In our case we are interested in partial derivatives of the error vector w.r.t. joint angles $J_{ij} = \frac{\partial e_i}{\partial q_j}$ that form the Jacobian $\mathbf{J}$. The current solution $\mathbf{q}$ is improved with a Levenberg-Marquardt (damped least squares) step $\mathbf{q}' = \mathbf{q} + \Delta\mathbf{q}$ with $\Delta\mathbf{q}$ obtained from the algebraic solution of

$$(\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})\Delta\mathbf{q} = \mathbf{J}^T\mathbf{e} \tag{6}$$

The `InverseKinematics` class creates an inverse kinematics (IK) solver to calculate joint configurations for a desired end effector pose based on a specified rigid body tree model. This code generates an inverse kinematics solver object for the `robotics.RigidBodyTree` object and determines the inverse kinematics solution as a configuration object for the target pose `tform`.

```
ik = robotics.InverseKinematics('RigidBodyTree',robot);
weights = ones(6,1);
initialpose = robot.homeConfiguration;
randconf=robot.randomConfiguration;
targetpose = robot.getTransform(randconf,'ee_link');
[targetsol, solnInfo] = ik('ee_link',targetpose,weights,initialpose);
robot.show(targetsol);
```

The Robotics System Toolbox provides a non documented helper function `robotics.manip.internal.IKHelpers.poseError` to calculate the pose error according to to equations (1) and (2) between two transforms. This code determines the pose (task space) and joint space error between the commanded pose and the numerical solution of inverse kinematics.

```
% pose error in task space
poseerror=robotics.manip.internal.IKHelpers.poseError(targetpose,...
robot.getTransform(targetsoln,'ee_link'));
% joint space error
jointerror=JointConf2JointVec(randconf)-JointConf2JointVec(targetsol);
```

The helper function

```
function [q] = JointConf2JointVec( configuration )
```

extracts the joint vector from the fields `JointPosition` in the configuration structure array.

Now, please complete tasks 1 to 3.

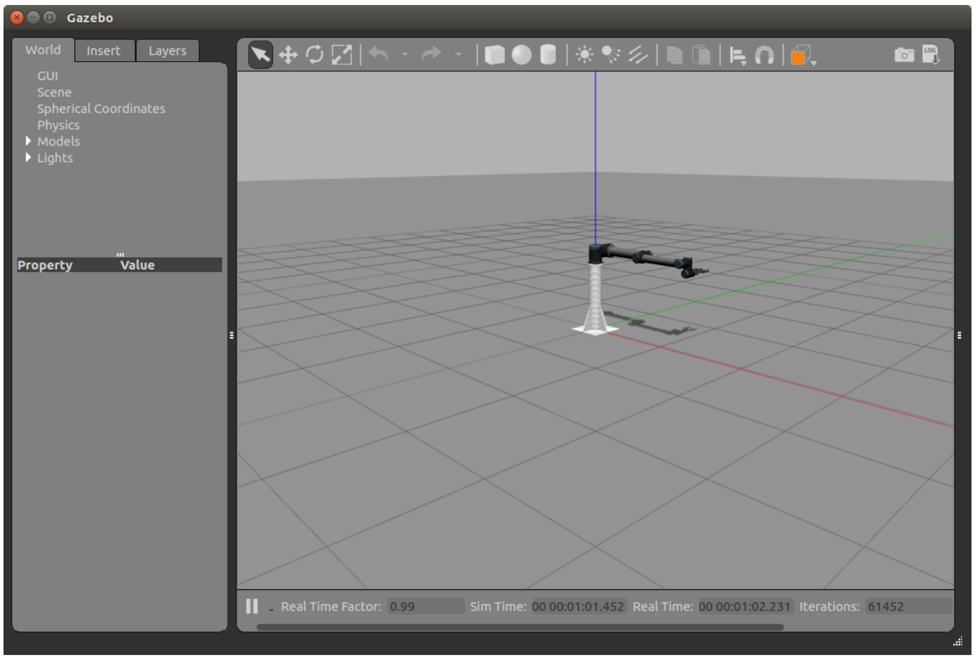

Figure 1: UR10 in Gazebo environment.

Rviz

RViz is the standard ROS tool for visualization of robot related data. RViz provides a number of different camera perspectives so called views to visualize the environment. A display in RViz is something that draws data in the 3D world such as a point cloud or the robot state. The RViz User Guide explains the GUI and the functionalities that RViz provides. The Built-In Display `RobotModel` shows a visual representation of the robot according to current joint and link TF transforms.

Gazebo

Gazebo is a robot simulator for indoor and outdoor environments. Gazebo as a physics engine simulates multiple robots, sensors and objects (see figure 1). Gazebo resides in a three-dimensional world and supports the simulation of rigid-body physics, namely robots that push things around or pick up objects. In Gazebo robots interact with the world in a plausible physical manner, for example a mobile robot starts to spin if centrifugal forces on a curved trajectory exceed a threshold. In Gazebo robots possess inertia and move according to forces and torques imposed on their bodies. Gazebo is suitable to realistically simulate the dynamic behavior of robotic arms and manipulators. Gazebo is less suited to capture interactions that involve compliance such as grasping objects with a two finger gripper.

UR Package

The UR repository contains metapackages and files for installation/usage of the Universal

Robot. Refer to the Moodle workspace to find instructions on how to install this package and its dependencies. The package supports pure UR visualization (RViz), UR simulation (Gazebo) or control of the real UR robot:

- Universal Robot Visualization only

  Visualization assumes that some external node publishes the UR joint state, for example a GUI in ROS, a publisher in Matlab or a node that publishes the actual UR robots joint encoder readings.

- Universal Robot Simulation (Gazebo)

  Gazebo simulates the robot dynamics and interactions with the environments. The motion of the UR robot is governed by the joint torques applied by the actuators. By default a `joint_trajectory_controller` is launched in conjunction with the simulation that is ready to receive joint trajectories.

- Real Universal Robot

  The UR package employs a modified version of the `ur_modern_driver` package. The control interface via `joint_trajectory_controller` is identical to the Gazebo simulation mode, which facilitates the code transfer from simulation onto the real robot.

These modes are accessible by their corresponding `.launch`-files in the `ur_description` or `ur_simulation_gazebo` package, e.g.:

```
ros2 launch ur_description view_ur.launch.py ur_type:=ur10
```

for just visualizing the UR 10 robot in certain joint configurations. These launch files start RViz with a predefined setup and graphical helper tools to operate the robot and gripper depending on the mode of operation. Please also inspect the topics which are made available by the package.

ROS Subscriber

ROS shares information among nodes by sending and receiving messages via publisher and subscribers. Messages are a simple data structure for sharing data.

Your Matlab code subscribes to a topic with `use ros2subscriber` and receives messages on this topic with `receive`. First you need to create a ROS2 node in Matlab that connects to the ROS2 network with

```
node = ros2node(nodename);
```

You create the subscriber with

```
sub = ros2subscriber(node,topicname,msgtype);
```

The subscriber either receives the most recent message in the past with

```
data = sub.LatestMessage;
```

or waits for the next message with

```
data = sub.receive();
```

In the first case the program control flow immediately returns to Matlab, in the second case the Matlab program waits for the next message to be published.

The task is to start the robot simulation in ROS and inspect the available topics within Matlab in order to create a subscriber for the robot's joint stats.

Now, please complete tasks 4 to 7.

ROS Publisher

The Publisher object in Matlab assumes the role of a publisher on the ROS network. The object publishes a specific message type on a given topic. Messages published by the publisher are send to all subscribers of the topic. The same topic might have multiple publishers (not so common) and subscribers.

```
pub = ros2publisher(node,topicname);
```

creates a publisher object `pub`, for a topic, `topicname`, that already exists on the ROS master topic list. The publisher gets the topic message type from the topic list on the ROS master. Whenever the Matlab node publishes messages on that topic, ROS nodes that subscribe to that topic receive those messages.

```
pub = ros2publisher(node,topicname,msgname);
```

creates a publisher, `pub`, for a topic, `topicname`, that already exists on the ROS master topic list. If the ROS master topic list already contains a matching topic, the ROS master adds the MATLAB global node to the list of publishers for that topic.

```
msg = ros2message(pub);
```

creates an empty message determined by the topic published by pub.

```
msg = ros2message(messagetype)
```

creates an empty ROS message object with message type.

The command

```
pub.send(msg);
```

publishes a message to the topic specified by the publisher pub. This message is received by all subscribers in the ROS network that subscribe to the topic.

Joint Trajectory Controller

In order to simulate the UR robot motion in Gazebo invoke the following launch files:

```
ros2 launch ur_simulation_gazebo ur_sim_control.launch.py ur_type:=ur10
 gazebo_gui:=false
```

The launch file starts Gazebo (hidden), RViz with a predefined setup, and ROS Controllers with interfaces.

Attention: Due to some ROS2/Gazebo changes you have to change one line in the downloaded package `ur_description` before starting the simulation. In the file `ur.urdf.xacro` in the folder `urdf` change the z coordinate of the robot base position from 0 to 2 in line 126 (Earlier: `<origin xyz="0 0 0" rpy="0 0 0" />` to `<origin xyz="0 0 2" rpy="0`

`0 0" />`). The robot will fly above the ground but avoids any collisions that interfere with the task.

The controller tracks joint-space reference trajectories on a group of joints. Trajectories are specified as a set of waypoints to be reached at specific time instants, which the controller attempts to execute assuming that the joint velocities and accelerations are compliant with the robots mechanical limits. Waypoints consist of positions, and optionally velocities and accelerations. The joint trajectory profile is a fifth order polynomial

$$q(t) = a_5 t^5 + a_4 t^4 + a_3 t^3 + a_2 t^1 + a_1 t + a_0 \tag{7}$$

subject to boundary conditions. In case of a point to point motion from an initial joint state q_i to a final joint state q_f within t_f seconds the boundary conditions impose zero velocity and acceleration at start and end

$$\begin{aligned} q(0) &= q_i \\ q(t_f) &= q_f \\ \dot{q}(0) &= 0 \\ \dot{q}(t_f) &= 0 \\ \ddot{q}(0) &= 0 \\ \ddot{q}(t_f) &= 0 \end{aligned}$$

providing six constraints for the six parameters a_i. Figure 2 shows the joint state, velocity and acceleration profiles.

In the following, Matlab will publish joint reference waypoints in order to let the robot's joint trajectory controller create the profile and perform tracking.

There are two mechanisms for sending trajectories to the controller:

- topic interface
- action interface

The former uses the `trajectory_msgs/JointTrajectory` message and the latter uses `control_msgs/FollowJointTrajectoryGoal` to specify trajectories, and specify reference values for all the controller joints.

The recommended way to command trajectories is through the action interface, and is favored when execution monitoring is desired. We will later get back to the action interface.

The topic interface is a fire-and-forget alternative, which means that the the execution of the command is not monitored. There is no mechanism to notify the publisher of the

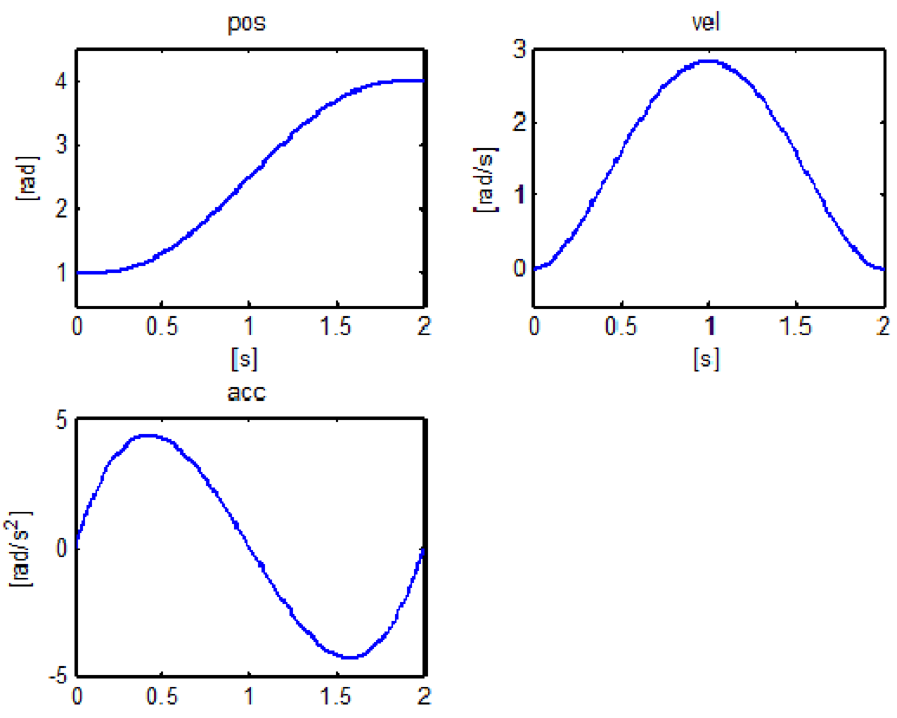

Figure 2: Fifth order polynomial joint trajectory profile

command about tolerance violations. Some degree of monitoring is available if your code continuously queries the joint states via the joint state topic. However, explicit monitoring is much more cumbersome to realize than with the action interface.

Trajectory Control via Topic Interface

In the following Matlab sends trajectory commands to the ROS controllers.

Now, please complete tasks 8 to 14.

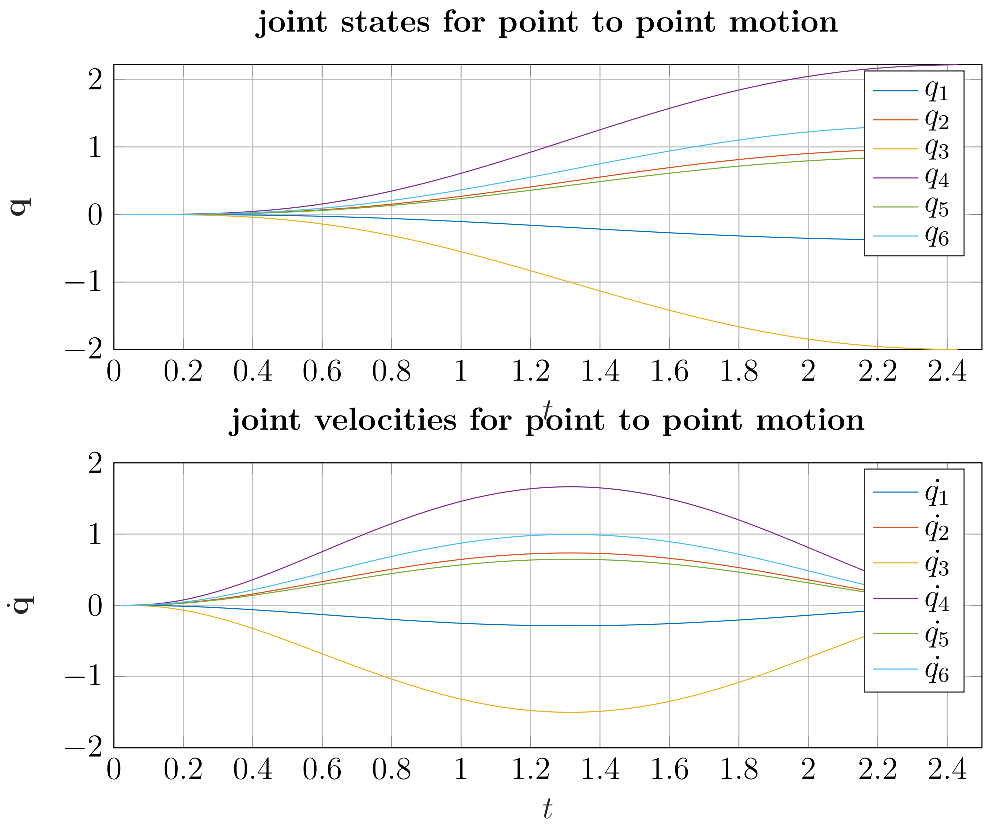

Figure 3: Joint state vector $\mathbf{q}$ and joint velocity $\dot{\mathbf{q}}$ for point to point motion. Note that velocity results with ROS2 are currently not working as you can see here and you don't need to search for a solution.

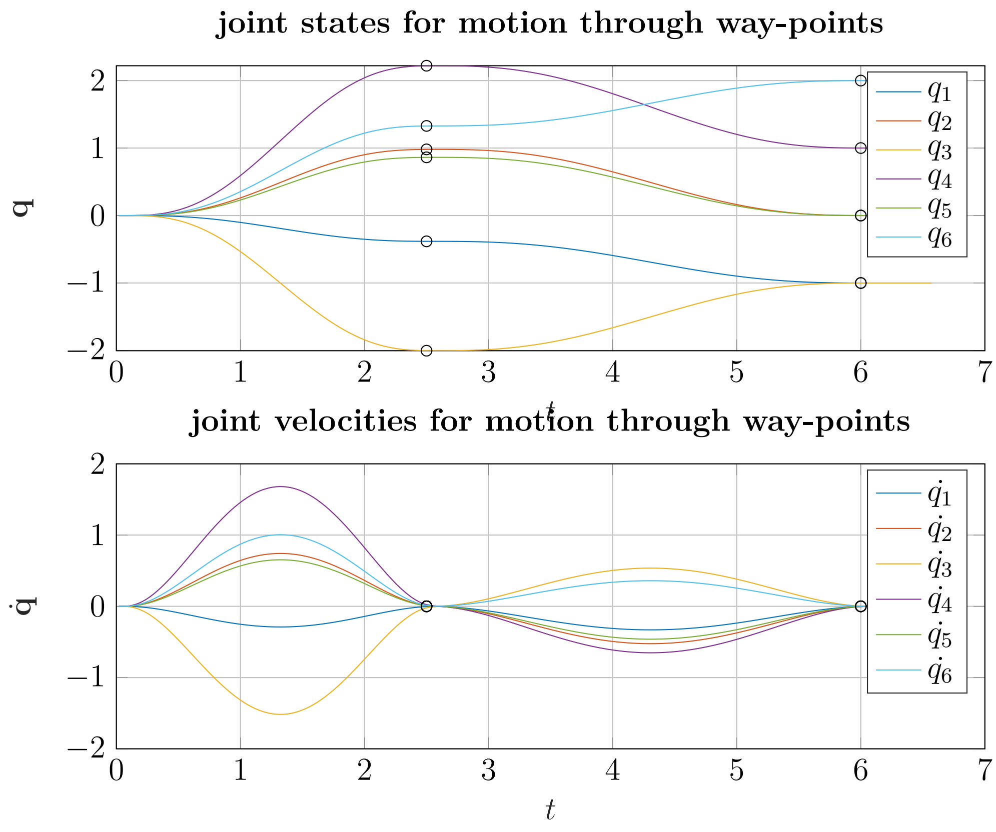

Figure 4: Joint state vector $\mathbf{q}$ and joint velocity $\dot{\mathbf{q}}$ for motion through way-points. Note that velocity results with ROS2 are currently not working as you can see here and you don't need to search for a solution.

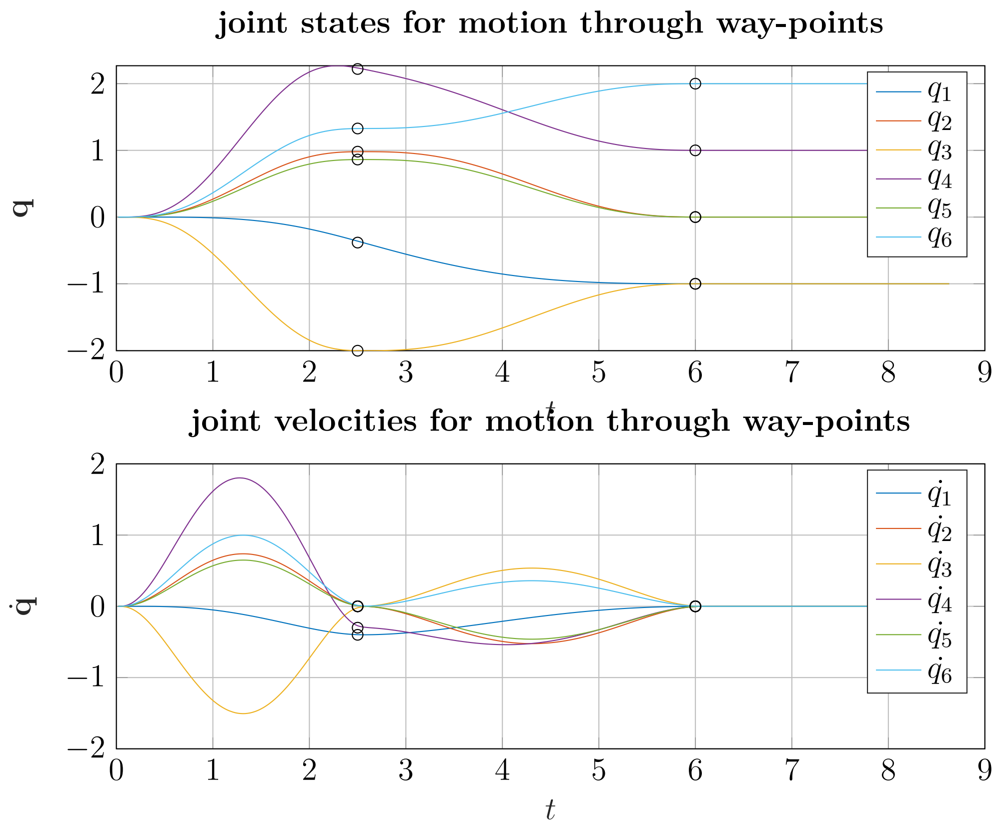

Figure 5: Joint state vector $\mathbf{q}$ and joint velocity $\dot{\mathbf{q}}$ for motion through way-points. Note that velocity results with ROS2 are currently not working as you can see here and you don't need to search for a solution.

Action Server and Client

The approach of controlling the robot motion via the topic interface has the obvious drawback that the program in Matlab has to wait until the movement has completed. That blocks Matlab from processing other information and it does not allow the program to abort or interrupt the motion command.

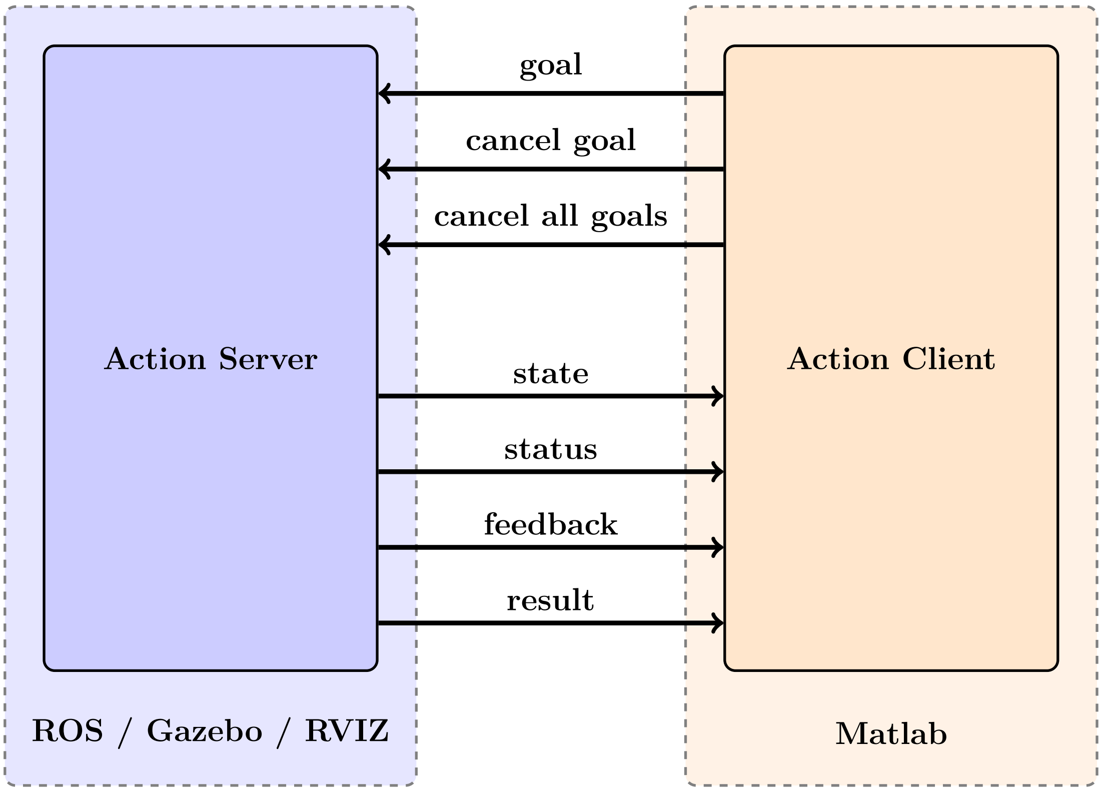

Figure 6: ROS action client server communication structure [1]

ROS Actions implement a client to server communication with adhering to a specific protocol. The actions utilize ROS topics to send goal messages from a client to the server as shown in figure 6. It is possible to cancel an ongoing goal using the action client. After receiving a goal, the server processes it and returns information about the progress back to the client. This information includes the status of the server, the state of the current goal, feedback on that goal during operation, and eventually a result message when the goal is complete.

The `sendGoal` function sends goals to the server and immediately returns. While the server is executing a goal, the callback function `FeedbackFcn`, is called to provide data relevant to that goal. Cancel the current goal using `cancelGoal` or all goals on server using `cancelAllGoals`.

[1]source: RST
[2]source: RST

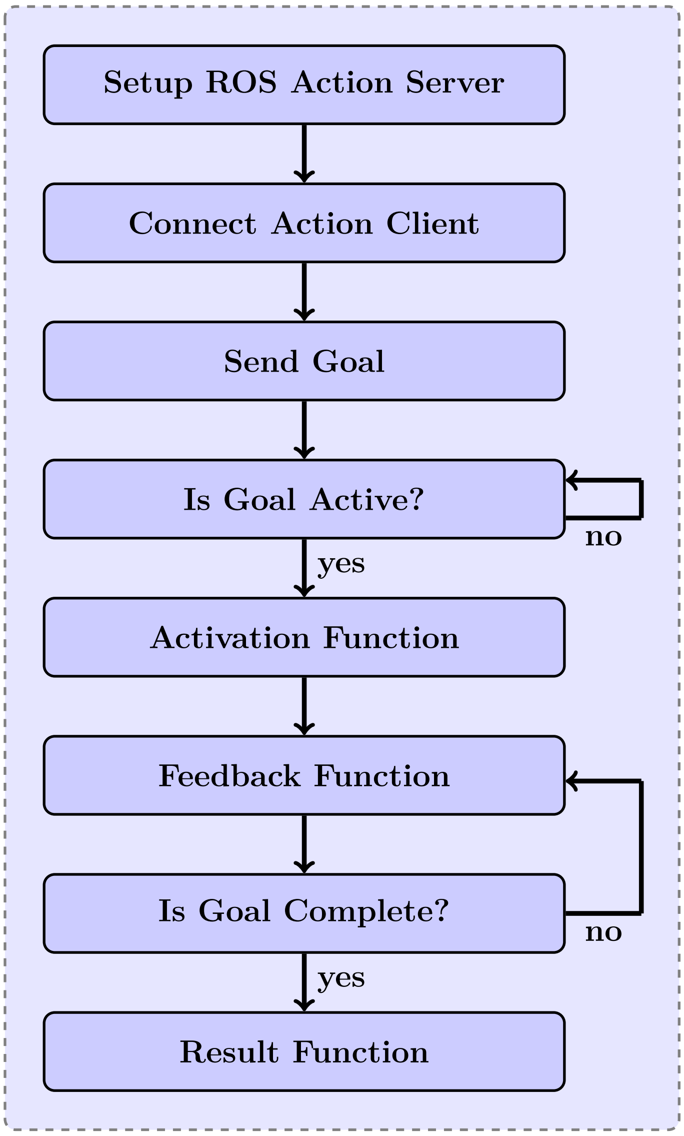

Figure 7: ROS action setup and control flow [2]

The control flow of an action client is illustrated in figure 7 composed of the following steps

- Setup ROS action server. With `ros2 action list` inspect which actions are available on the ROS network.
- Create an action client and connect it to the server with `ros2actionclient` with an action type available on the ROS network. Retrieve a blank `goalMsg` from `ros2actionclient`. Use `waitForServer` to wait for the action client to connect to the server.

- Send a goal using `sendGoal`. Specify the `goalMsg` that corresponds to the action type. Modify the blank message `goalMsg` with your desired parameters.
- When a goal status becomes `'active'`, the goal begins execution and the `ActivationFcn` callback function is called.
- While the goal status remains `'active'`, the server continues to execute the goal. The feedback callback function processes information about this goals execution periodically whenever a new feedback message is received. Use the `FeedbackFcn` to access or process the message data sent from the ROS server.
- When the goal is achieved, the server returns a result message and status. Use the `ResultFcn` callback to access or process the result message and status.

```
[actClient,goalMsg] = ros2actionclient(node,actionname,actiontype);
```

returns a goal message `goalMsg` to send the action client. The goal message is initialized with default values for that message.

The command

```
waitForServer(actClient);
```

waits for the action client to connect to server upon which it can send action to the server with

```
[resultMsg,resultState]=sendGoal(actClient,goalMsg,timeout)
```

The specified action client tracks this goal. The function does not wait for the goal to be executed and returns immediately. If the `ActionFcn`, `FeedbackFcn`, and `ResultFcn` callbacks of the client are defined, they are called when the goal is processing on the action server. All callbacks associated with a previously sent goal are disabled, but the previous goal is not canceled.

Trajectory Control via Action Server and Client

This section is concerned with the action interface for the joint trajectory controller. Action goals allow to specify not only the trajectory to execute, but also (optionally) path and goal tolerances. When no tolerances are specified, the defaults given in the parameter server are used. If tolerances are violated during trajectory execution, the action goal is aborted and the client is notified. Position tolerances for a particular joint cause the trajectory to succeed with the next way-point if the joint is within goal position plus, minus the goal tolerance.

Now, please complete tasks 15 to 21.

References

[1] ROS Tutorials, [3], 2018

[2] Jason M. O'Kane, A Gentle Introduction to ROS, [4], 2015

[3] Springer Handbook of Robotics, Springer, [5], 2015

[3] `http://wiki.ros.org/ROS/Tutorials`
[4] `https://cse.sc.edu/~jokane/agitr/`
[5] `http://link.springer.com/referenceworkentry/10.1007/978-3-540-30301-5_2`