Robotics Lab: Homework 2

Control a manipulator to follow a trajectory

Mario Selvaggio

This document contains the homwework 2 of the Robotics Lab class.

Control a manipulator to follow a trajectory

The goal of this homework is to develop a ROS package to dynamically control a 7-degrees-of-freedom robotic manipulator arm into the Gazebo environment. The `kdl_ros_control` package (at the following link: https://github.com/mrslvg/kdl_robot) must be used as starting point. The student is requested to address the following points and provide a detailed report of the employed methods. In addition, a personal github repo with all the developed code must be shared with the instuctor. The report is due in one week from the homewerk release.

1. Substitute the current trepezoidal velocity profile with a cubic polinomial linear trajectory

   (a) Modify appropriately the `KDLPlanner` class (files `kdl_planner.h` and `kdl_planner.cpp`) that provides a basic interface for trajectory creation. First, define a new `KDLPlanner::trapezoidal_vel` function that takes the current time t and the acceleration time t_c as `double` arguments and returns three `double` variables s, $\dot{s}$ and $\ddot{s}$ that represent the curvilinear abscissa of your trajectory[1]. Remember: a trapezoidal velocity profile for a curvilinear abscissa $s \in [0, 1]$ is defined as follows

   $$s(t) = \begin{cases} \frac{1}{2}\ddot{s}_c t^2 & 0 \leq t \leq t_c \\ \frac{1}{2}\ddot{s}_c(t - tc/2) & t_c < t < t_f - t_c \\ 1 - \frac{1}{2}\ddot{s}_c(t_f - tc)^2 & t_f - t_c < t \leq t_f \end{cases} \tag{1}$$

   where t_c is the acceleration duration variable while $\dot{s}(t)$ and $\ddot{s}(t)$ can be easily retrieved calculating time derivative of (1).

   (b) Create a function named `KDLPlanner::cubic_polinomial` that creates the cubic polynomial curvilinear abscissa for your trajectory. The function takes as argument a `double t` representing time and returns three `double` s, $\dot{s}$ and $\ddot{s}$ that represent the curvilinear abscissa of your trajectory. Remember, a cubic polinomial is defined as follows

   $$s(t) = a_3t^3 + a_2t^2 + a_1t + a_0 \tag{2}$$

   where coefficients a_3, a_2, a_1, a_0 must be calculated offline imposing boundary conditions, while $\dot{s}(t)$ and $\ddot{s}(t)$ can be easily retrieved calculating time derivative of (2).

2. Create circular trajectories for your robot

   (a) Define a new constructor `KDLPlanner::KDLPlanner` that takes as arguments the time duration `_trajDuration`, the starting point `Eigen::Vector3d␣_trajInit` and the radius `_trajRadius` of your trajectory and store them in the corresponding class variables (to be created in the `kdl_planner.h`).

   (b) The center of the trajectory must be in the vertical plane containing the end-effector. Create the positional path as function of $s(t)$ directly in the function `KDLPlanner::compute_trajectory`: first, call the `cubic_polinomial` function to retrieve s and its derivatives from t; then fill in the `trajectory_point` fields `traj.pos`, `traj.vel`, and `traj.acc`. Remember that a circular path in the $y - z$ plane can be easily defined as follows

   $$x = x_i, \qquad y = y_i - r\cos(2\pi s), \qquad z = z_i - r\sin(2\pi s) \tag{3}$$

   (c) Do the same for the linear trajectory.

[1] Use passage by reference to return multiple arguments: https://www.w3schools.com/cpp/cpp_function_reference.asp

3. Test the four trajectories

   (a) At this point, you can create both linear and circular trajectories, each with trapezoidal velocity of cubic polinomial curvilinear abscissa. Modify your main file `kdl_robot_test.cpp` and test the four trajectories with the provided joint space inverse dynamics controller.

   (b) Plot the torques sent to the manipulator and tune appropriately the control gains `Kp` and `Kd` until you reach a satisfactorily smooth behavior. You can use `rqt_plot` to visualize your torques at each run, save the screenshot.

   (c) **Optional:** Save the joint torque command topics in a bag file and plot it using MATLAB. You can follow the tutorial at the following link https://www.mathworks.com/help/ros/ref/rosbag.html.

4. Develop an inverse dynamics operational space controller

   (a) Into the `kdl_contorl.cpp` file, fill the empty overlayed `KDLController::idCntr` function to implement your inverse dynamics operational space controller. Differently from joint space inverse dynamics controller, the operational space controller computes the errors in Cartesian space. Thus the function takes as arguments the desired `KDL::Frame` pose, the `KDL::Twist` velocity and, the `KDL::Twist` acceleration. Moreover, it takes four gains as arguments: `_Kpp` position error proportional gain, `_Kdp` position error derivative gain and so on for the orientation.

   (b) The logic behind the implementation of your controller is sketched within the function: you must calculate the gain matrices, read the current Cartesian state of your manipulator in terms of end-effector parametrized pose x, velocity $\dot{x}$, and acceleration $\ddot{x}$, retrieve the current joint space inertia matrix M and the Jacobian (compute the analytic Jacobian) and its time derivative, compute the linear e_p and the angular e_o errors (some functions are provided into the `include/utils.h` file), finally compute your inverse dynamics control law following the equation

$$\tau = By + n, \qquad y = J_A^{\dagger}\left(\ddot{x}_d + K_D\ddot{\tilde{x}} + K_P\tilde{x} - \dot{J}_A\dot{q}\right) \tag{4}$$

   (c) Test the controller along the planned trajectories and plot the corresponding joint torque commands.