

Robotics Lab: Homework 2

Students: Anzalone Claudio, Maisto Paolo, Manzoni Antonio

Here is the link to my public repo on github:

https://github.com/Claudio564/HW2_RL_ANZALONE

We want to specify that all the participants of the group worked at each stage of the development of the project. In order to simplify the drafting of the report (as recommended by the professor) we have fairly divided the writing of the development of the various points.

Control a manipulator to follow a trajectory

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.

Remember: a trapezoidal velocity profile for a curvilinear abscissa $s \in [0, 1]$ is defined as follows

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

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).

This function `void KDLPlanner::trapezoidal_vel`, named *trapezoidal_vel*, belongs to the `KDLPlanner` class. It returns `void`, indicating that it doesn't return a value. Instead, it uses reference parameters ('`double &s`', '`double &dot_s`', '`double &ddot_s`') to modify variables outside the function. In this reference parameters the function stores the curvilinear abscissa, velocity, and acceleration, respectively. Other input parameter of this function is '`double time`' that represents the current time. The function calculates the second derivative of the trajectory's curvilinear abscissa, denoted as *ddot_traj_s*.

It then uses conditional statements to determine the phase of the trajectory (acceleration, constant velocity, or deceleration) based on the current time (time).

The variables *s*, *dot_s*, and *ddot_s* are updated accordingly in each phase.

Overall, this function generates a trapezoidal velocity profile for a given time, updating the relevant variables based on the current phase of the trajectory.

It is noted that the initial curvilinear abscissa s_i and the final curvilinear abscissa s_f are 0 and 1, respectively.

```

70  /*TRAPEZOIDAL VELOCITY PROFILE*/
71  void KDLPlanner::trapezoidal_vel(double time, double &s, double &dot_s, double &ddot_s)
72  {
73      double ddot_traj_s = -1.0/(std::pow(accDuration_,2)-trajDuration_*accDuration_);
74
75      if(time <= accDuration_)
76      {
77          s = 0.5*ddot_traj_s*std::pow(time,2);
78          dot_s = ddot_traj_s*time;
79          ddot_s = ddot_traj_s;
80      }
81      else if(time <= trajDuration_-accDuration_)
82      {
83          s = ddot_traj_s*accDuration_*(time-accDuration_/2);
84          dot_s = ddot_traj_s*accDuration_;
85          ddot_s = 0.0;
86      }
87      else
88      {
89          s = 1 - 0.5*ddot_traj_s*std::pow(trajDuration_-time,2);
90          dot_s = ddot_traj_s*(trajDuration_-time);
91          ddot_s = -ddot_traj_s;
92      }
93  }
94

```

- (b) Create a function named `KDLPlanner::cubic_polynomial` 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`, `ṡ` and `ṡ̈` that represent the curvilinear abscissa of your trajectory.

Remember, a cubic polinomial is defined as follows

$$s(t) = a_3t^3 + a_2t^2 + a_1t + a_0 \quad (2)$$

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).

This function, named *cubic_polynomial*, is part of the `KDLPlanner` class. It returns void, indicating that it doesn't return a value. Similar to the *trapezoidal_vel* function, it uses reference parameters (*'double &s, double &dot_s, double &ddot_s'*) to modify variables outside the function. These parameters represent the curvilinear abscissa, velocity, and acceleration, respectively. The function calculates the position (s) velocity (\dot{s}) and acceleration (\ddot{s}) at a given time based on a cubic polynomial velocity profile. The coefficients of the polynomial are pre-computed using this equations shown in the following picture (it is retrieved from slides of Foundation of Robotics course so in this format it has been used this kind of notation with the q instead of s):

$$\begin{aligned}
 a_0 &= q_i \\
 a_1 &= \dot{q}_i \\
 a_3t_f^3 + a_2t_f^2 + a_1t_f + a_0 &= q_f \\
 3a_3t_f^2 + 2a_2t_f + a_1 &= \dot{q}_f
 \end{aligned}$$

The coefficients are computed considering s_i and s_f are 0 and 1, respectively.

The cubic polynomial represents a smooth velocity profile, and the function updates the variables accordingly.

```

95  /*CUBIC POLYNOMIAL VELOCITY PROFILE*/
96  void KDLPlanner::cubic_polynomial(double time, double &s, double &dot_s, double &ddot_s)
97  {
98      double a_0 = 0.0;
99      double a_1 = 0.0;
100     double a_2 = 3/std::pow(trajDuration_,2);
101     double a_3 = -2/(std::pow(trajDuration_,3));
102
103     s = a_3*std::pow(time,3)+a_2*std::pow(time,2)+a_1*time+a_0;           //s -> position
104     dot_s = 3*a_3*std::pow(time,2)+2*a_2*time+a_1;                       //dot_s -> velocity
105     ddot_s = 6*a_3*time+2*a_2;                                             //ddot_s -> acceleration
106 }

```

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`).

In these steps a new constructor has been declared with the following command: "`KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double _trajRadius);`" in the file "`kdl_planner.h`". With this command the following parameters are passed: duration of the trajectory, initial point of the trajectory and radius of the circular trajectory to be completed. Then it is also defined in the file "`kdl_planner.cpp`".

- (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_polynomial` 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

$$x = x_i, \quad y = y_i + r \cos(2\pi s), \quad z = z_i + r \sin(2\pi s) \quad (3)$$

At this point it remains only to define a void function as follows: "`trajectory_point circular_trajectory(double &s, double &dot_s, double &ddot_s);`". which is declared in the file "`kdl_planner.h`" as `KDLPlanner` defined above, to be implemented in the file "`kdl_planner.cpp`"; in which it defines the circular trajectory to be followed (or rather formulas 3).

Then the "`compute_trajectory`" function was used to define the trajectory, which first invokes the function "`cubic_polynomial`" which returns s and its derivatives, in order to define a polynomial cubic velocity profile and then calls the `circular_trajectory` function with s and its derivatives as input, which returns the desired trajectory so the circular trajectory with a cubic polynomial velocity profile.

```
167     double s_time, dot_s_time, ddot_s_time;
168     cubic_polynomial(time, s_time, dot_s_time, ddot_s_time);
169     return circular_trajectory(s_time, dot_s_time, ddot_s_time);
```

- (c) Do the same for the linear trajectory.

As in the previous point also in this case the trajectory is defined in the "`compute_trajectory`" function, which function just like the previous case. At first it invokes the "`cubic_polynomial`" function to define the velocity profile and then the "`linear_trajectory`" function which based on s and its derivatives returns a linear trajectory with a polynomial cubic velocity profile.

```
153     double s_time, dot_s_time, ddot_s_time;
154     cubic_polynomial(time, s_time, dot_s_time, ddot_s_time);
155     return linear_trajectory(s_time, dot_s_time, ddot_s_time);
```

3. Test the four trajectories

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

In the folder Simulations within github repository there are loaded the videos of the tests of four trajectories that have been implemented.

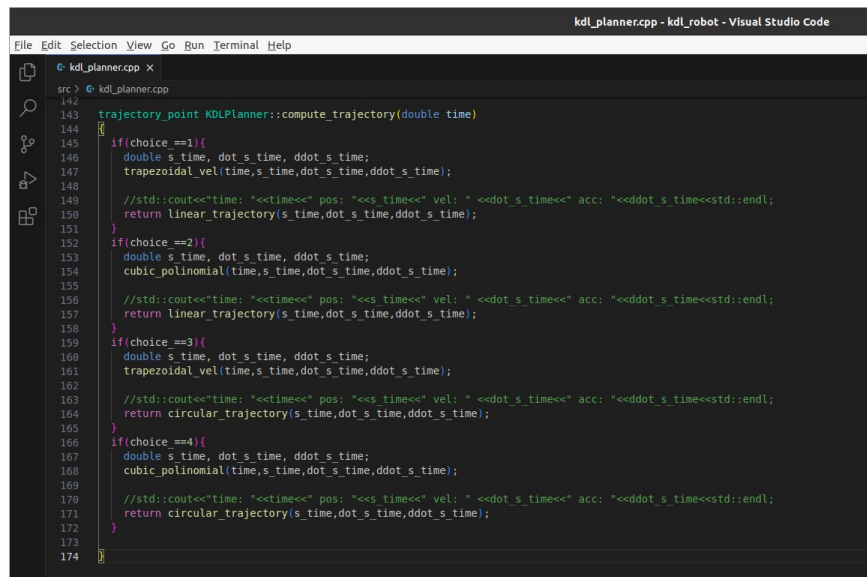
Used commands to make simulations

```
$ cd catkin_ws
$ roscore
$ roslaunch iiwa_gazebo iiwa_gazebo_effort.launch
$ rosrn kdl_ros_control kdl_robot_test ./src/iiwa_stack/iiwa_description/urdf/iiwa14.urdf
```

You notice that the trajectories to be performed are four, that are:

- 1 → linear trajectory with trapezoidal velocity profile;
- 2 → linear trajectory with cubic polynomial velocity profile;
- 3 → circular trajectory with trapezoidal velocity profile;
- 4 → circular trajectory with cubic polynomial velocity profile.

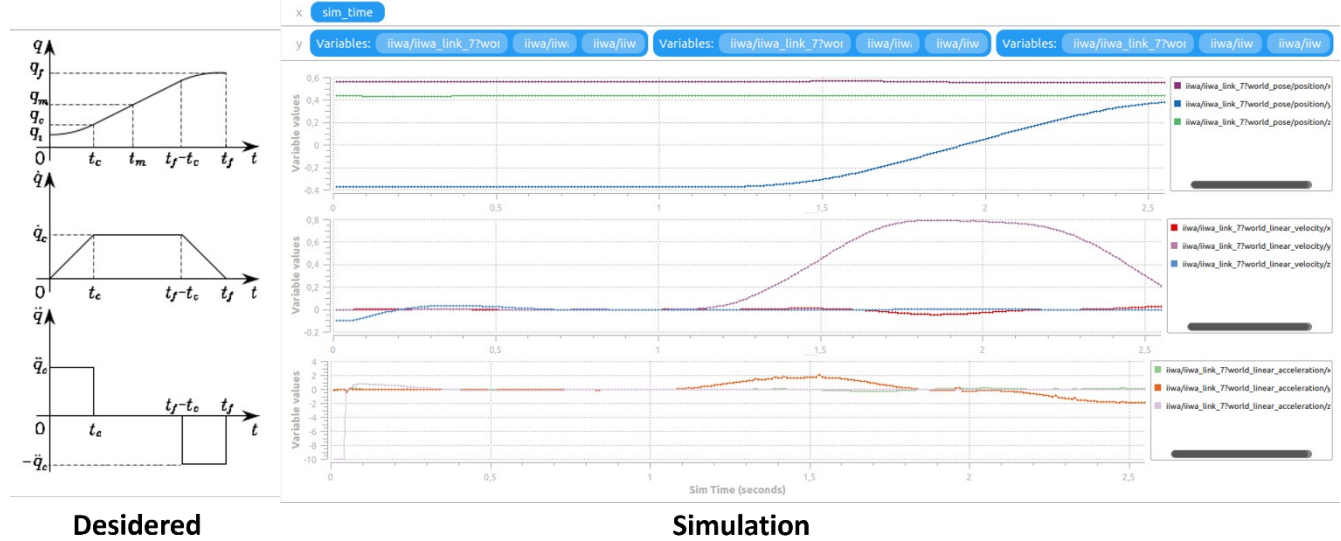
These trajectories are all implemented in the function called "`compute_trajectory`". In this project it is given the possibility to choose from terminal which trajectory to run the robot. So based on the choice this function will return the desired trajectory. The choice must be made by entering a number that associates the trajectory just as written in the lines above.



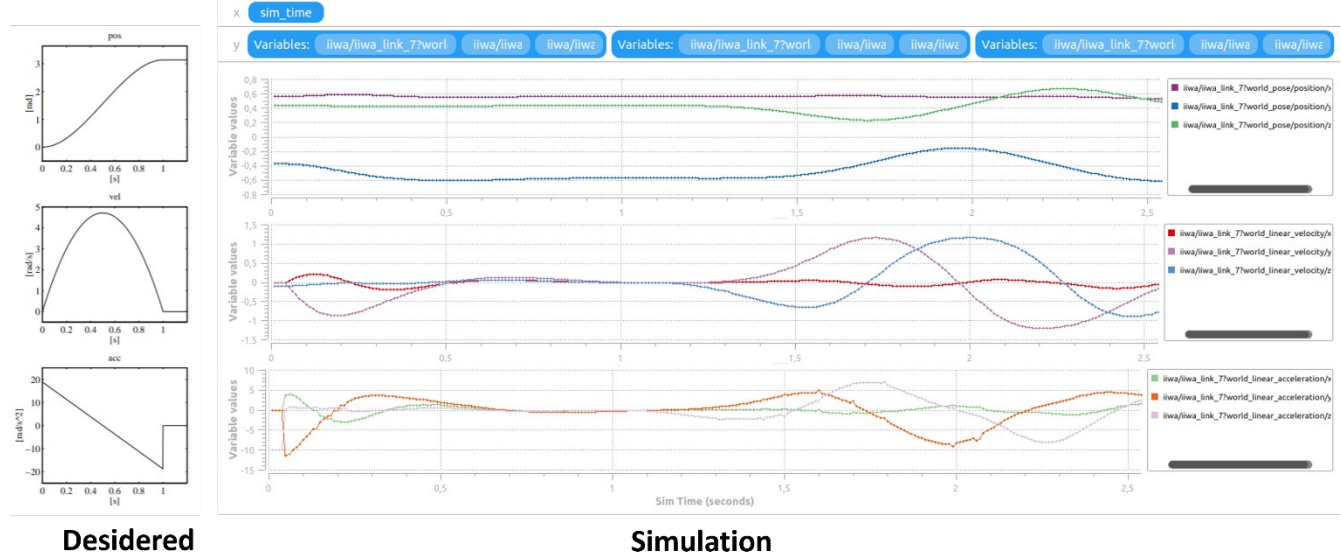
```
kdl_planner.cpp - kdl_robot - Visual Studio Code
File Edit Selection View Go Run Terminal Help
kdl_planner.cpp x
src > kdl_planner.cpp
142
143 trajectory_point KDLPlanner::compute_trajectory(double time)
144 {
145     if(choice ==1){
146         double s_time, dot_s_time, ddot_s_time;
147         trapezoidal_vel(time,s_time,dot_s_time,ddot_s_time);
148         //std::cout<<"time: "<<time<<" pos: "<<s_time<<" vel: " <<dot_s_time<<" acc: "<<ddot_s_time<<std::endl;
149         return linear_trajectory(s_time,dot_s_time,ddot_s_time);
150     }
151     if(choice ==2){
152         double s_time, dot_s_time, ddot_s_time;
153         cubic_polynomial(time,s_time,dot_s_time,ddot_s_time);
154         //std::cout<<"time: "<<time<<" pos: "<<s_time<<" vel: " <<dot_s_time<<" acc: "<<ddot_s_time<<std::endl;
155         return linear_trajectory(s_time,dot_s_time,ddot_s_time);
156     }
157     if(choice ==3){
158         double s_time, dot_s_time, ddot_s_time;
159         trapezoidal_vel(time,s_time,dot_s_time,ddot_s_time);
160         //std::cout<<"time: "<<time<<" pos: "<<s_time<<" vel: " <<dot_s_time<<" acc: "<<ddot_s_time<<std::endl;
161         return circular_trajectory(s_time,dot_s_time,ddot_s_time);
162     }
163     if(choice ==4){
164         double s_time, dot_s_time, ddot_s_time;
165         cubic_polynomial(time,s_time,dot_s_time,ddot_s_time);
166         //std::cout<<"time: "<<time<<" pos: "<<s_time<<" vel: " <<dot_s_time<<" acc: "<<ddot_s_time<<std::endl;
167         return circular_trajectory(s_time,dot_s_time,ddot_s_time);
168     }
169 }
170
171
172
173
174 }
```

As follow we are showing the plots of position, velocity and acceleration of the link 7 of the iiwa robot in different cases:

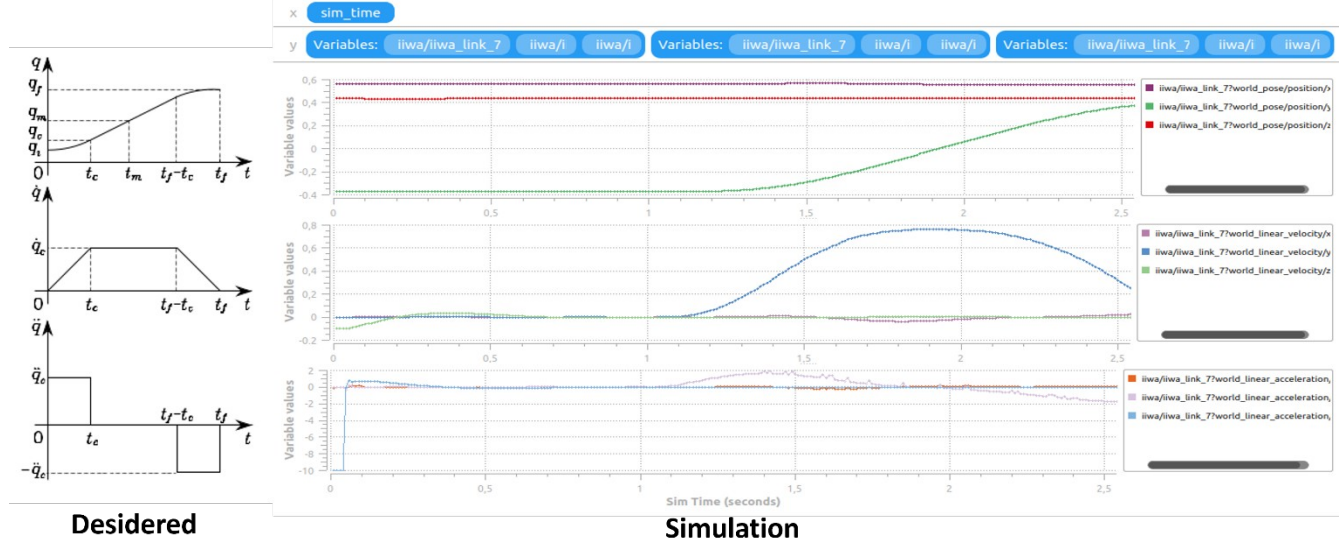
Linear trajectory with trapezoidal velocity profile



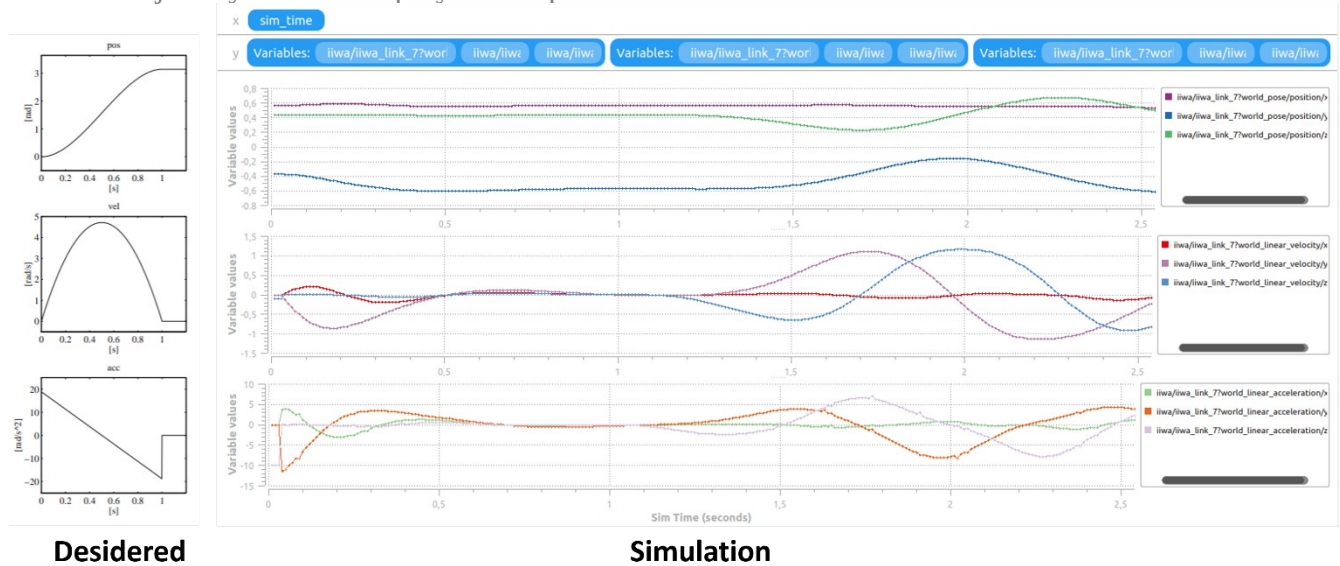
Linear trajectory with cubic polynomial profile



Circular trajectory with trapezoidal profile



Circular trajectory with cubic polynomial profile



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

To tune appropriately the control gains K_p and K_d , and in order to fulfill our request to reach a satisfactorily smooth behavior, we carried out several tests using the "trial and error" method and obtained the following results:

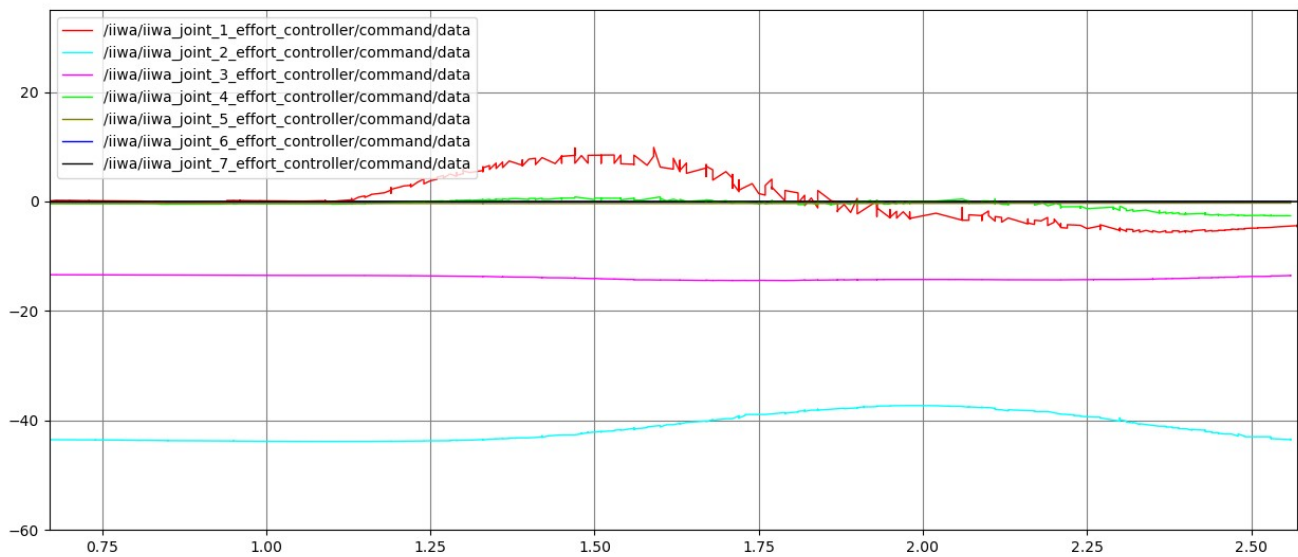
- $K_p=120$ and $K_d=15$; in the first case we set high control gains and we obtain a fluid movement of the robot and it completes the trajectory until the end but the torque plots are not very smooth;

- $K_p=30$ and $K_d=6$; to obtain a smoother behavior of the torques we tried to use low control gains but the required trajectory is not followed correctly and the movement is not completed;
- $K_p=50$ and $K_d=\sqrt{50}$; therefore we chose to use as control gains those that were already present initially, because with these we obtain both a smooth plots of the torques and a good trajectory tracking so with these conditions we have a worth trade off.

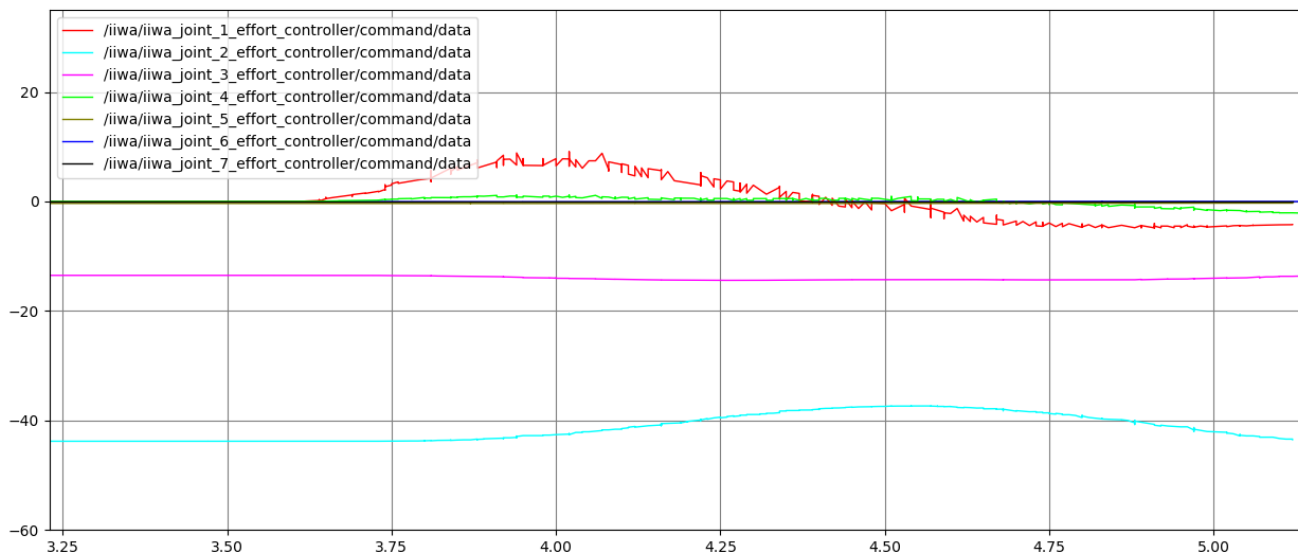
Used commands to visualize torques plots

```
$ cd catkin_ws
$ roscore
$ roslaunch iiwa_gazebo iiwa_gazebo_effort.launch
$ rqt_plot
$ rosrn kdl_ros_control kdl_robot_test ./src/iiwa_stack/iiwa_description/urdf/iiwa14.urdf
```

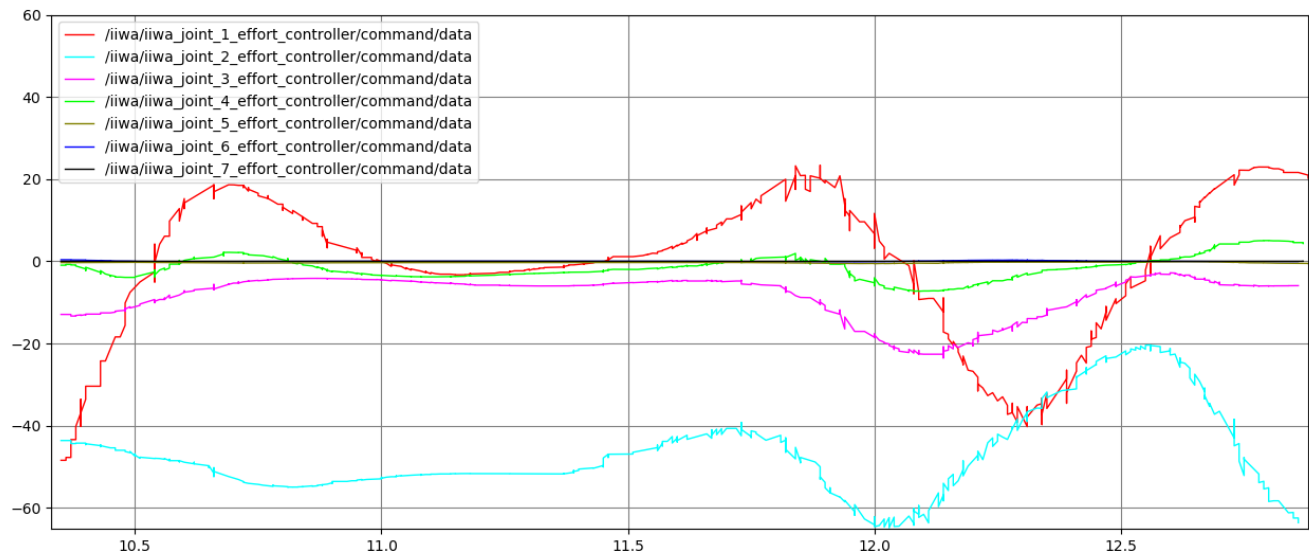
Linear trajectory with trapezoidal velocity profile with $K_p=50$ and $K_d=\sqrt{50}$:



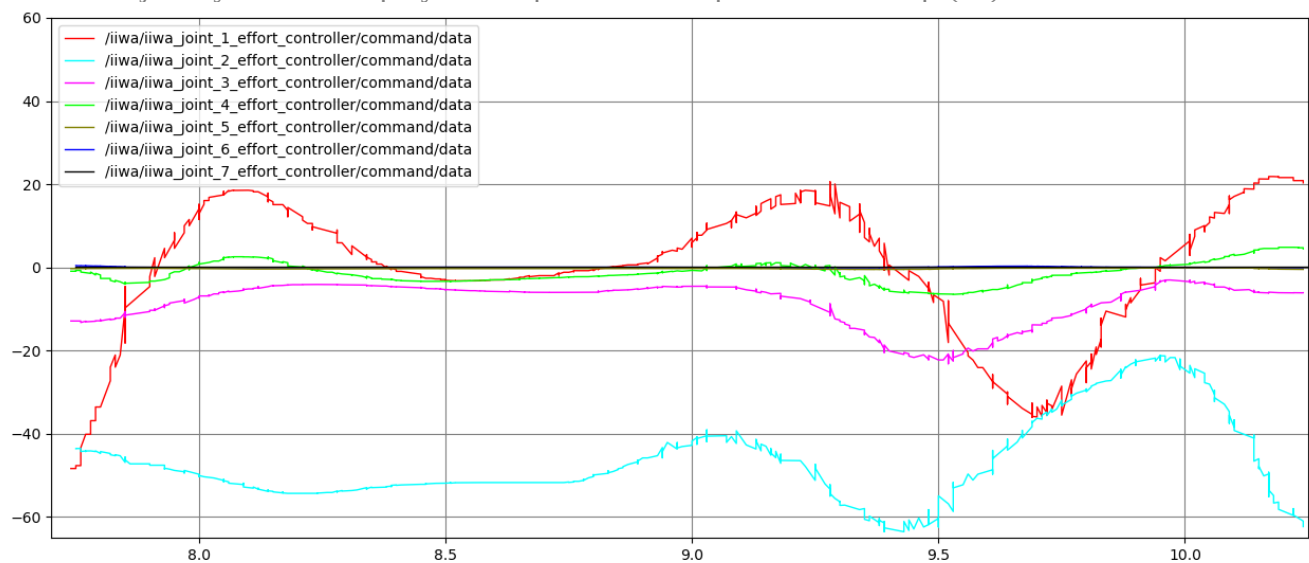
Linear trajectory with cubic polynomial velocity profile with $K_p=50$ and $K_d=\sqrt{50}$:



Circular trajectory with trapezoidal velocity profile with $K_p=50$ and $K_d=\sqrt{50}$:

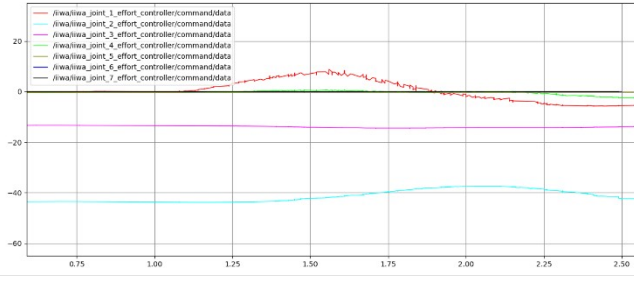


Circular trajectory with cubic polynomial profile with $K_p=50$ and $K_d=\sqrt{50}$:

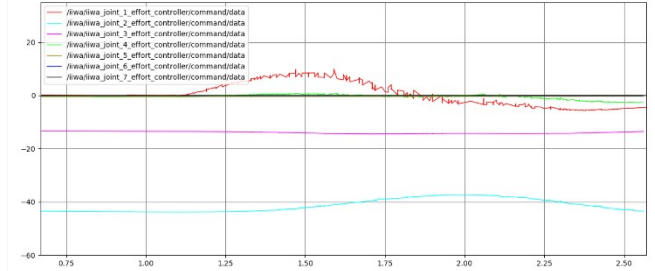


Comparison between plots of three cases of linear trajectory and trapezoidal velocity profile with different control gains:

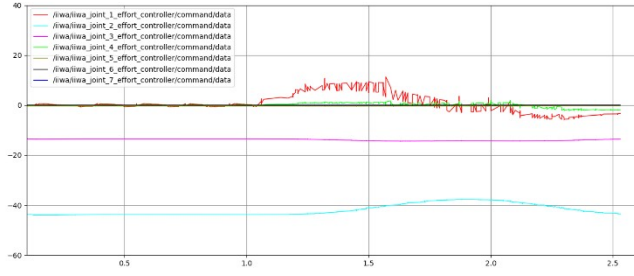
$$K_P = 30 \quad K_D = 6$$



$$K_P = 50 \quad K_D = \sqrt{50}$$

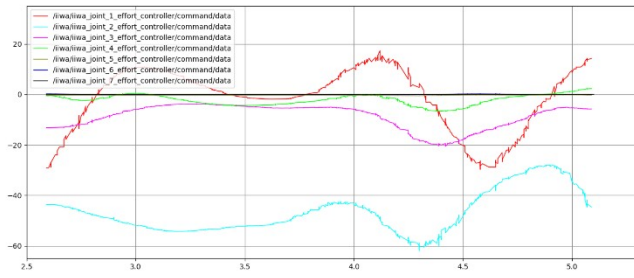


$$K_P = 120 \quad K_D = 15$$

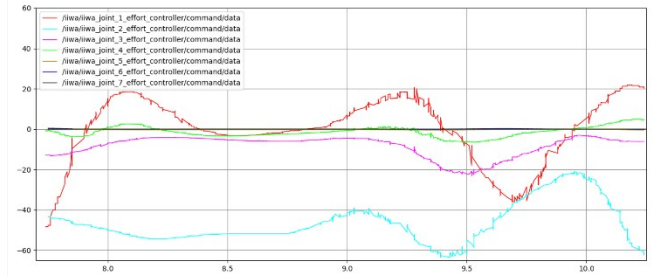


Comparison between plots of three cases of circular trajectory and cubic polynomial velocity profile with different control gains:

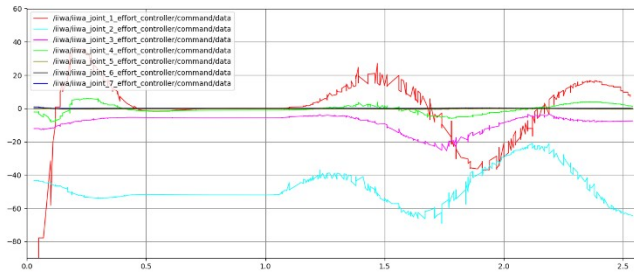
$$K_P = 30 \quad K_D = 6$$



$$K_P = 50 \quad K_D = \sqrt{50}$$



$$K_P = 120 \quad K_D = 15$$



- (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>.

We saved the joint torque command topics in a bag file and plot it using MATLAB.

Used commands in LINUX

```
$ cd catkin_ws  
$ roscore  
$ roslaunch iiwa_gazebo iiwa_gazebo_effort.launch  
$ rosrun kdl_ros_control kdl_robot_test ./src/iiwa_stack/iiwa_description/urdf/iiwa14.urdf  
$ rosbag record -a
```

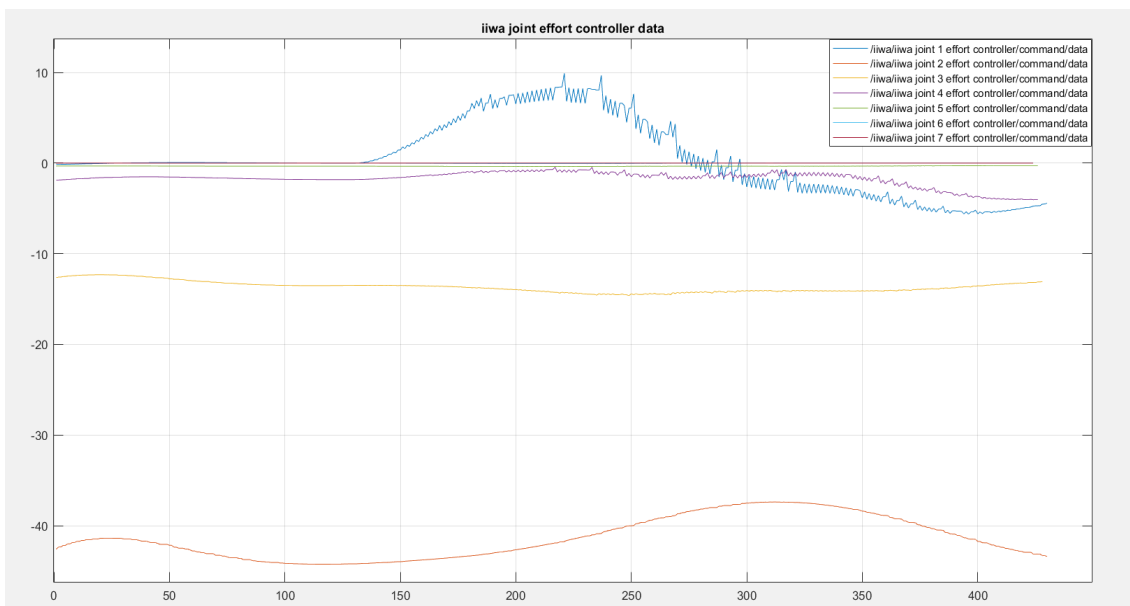
Used commands in MATLAB

```
bag = rosbag('lin_trap.bag')  
bSel_1 = select(bag,'Topic','/iiwa/iiwa_joint_1_effort_controller/command')  
msgStructs_1 = readMessages(bSel_1,'DataFormat','struct');  
msgStructs_1{1}  
datadata_1 = cellfun(@(m) double(m.Data),msgStructs_1);  
plot(datadata_1)
```

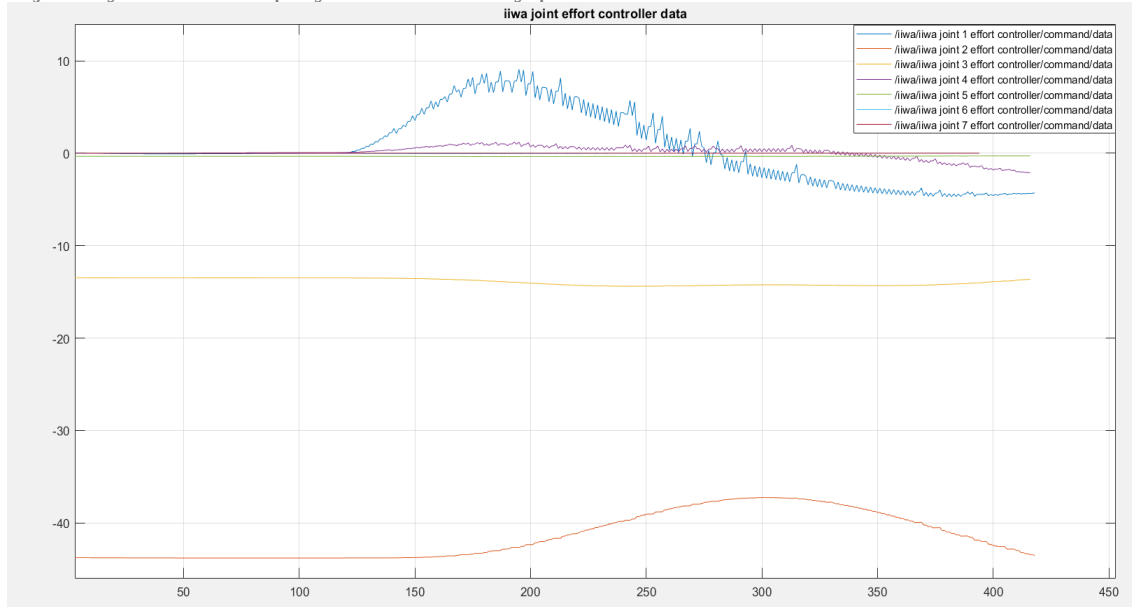
These commands are repeated for all topics “/iiwa/iiwa_joint_X_effort_controller/command” and so it is used these commands to plot all in only one plot:

```
plot(datadata_1)  
hold on  
plot(datadata_2)  
hold on  
[...]  
plot(datadata_7)
```

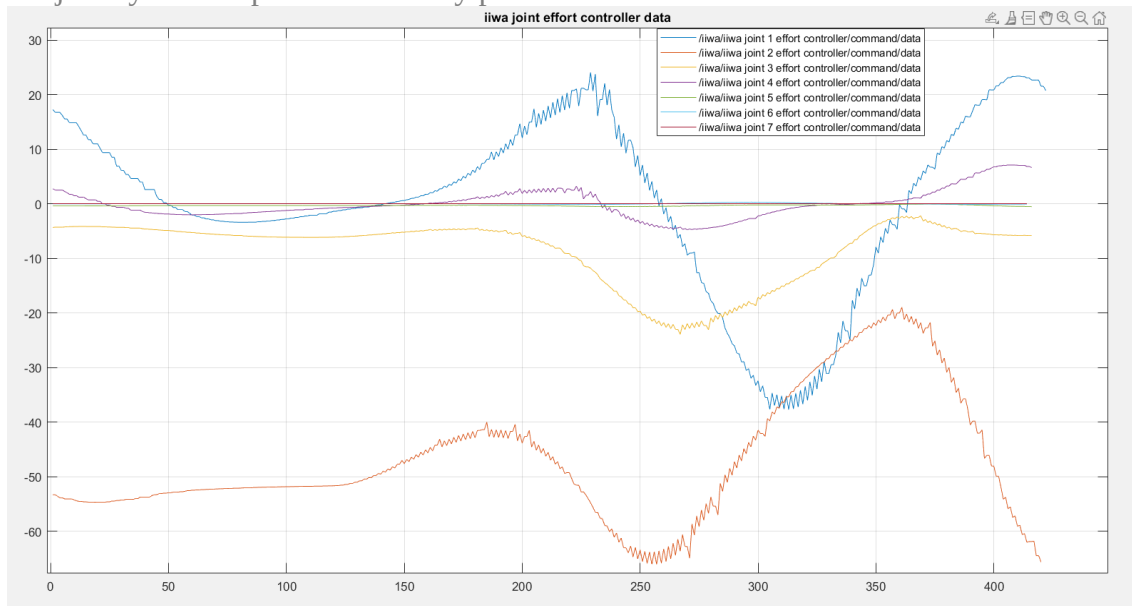
Linear trajectory with trapezoidal velocity profile:



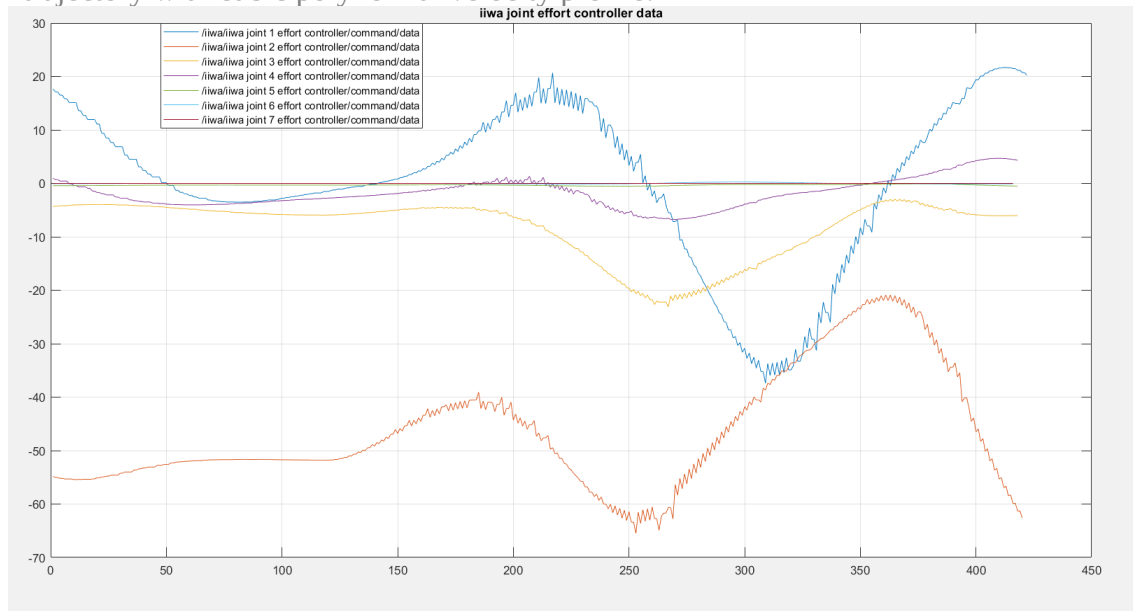
Linear trajectory with cubic polynomial velocity profile:



Circular trajectory with trapezoidal velocity profile:



Circular trajectory with cubic polynomial velocity profile:



4. Develop an inverse dynamics operational space controller

- (a) Into the `kdl_control.cpp` file, fill the empty overlaid `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.

In this step the file “`kdl_control.cpp`” has been modified in particular the function “`KDLController::idCntr`” in order to implement the inverse dynamics operational space controller.

```
26  /*FUNCTION TO IMPLEMENT INVERSE DYNAMICS OPERATIONAL SPACE CONTROLLER*/
27  Eigen::VectorXd KDLController::idCntr(KDL::Frame &_desPos,
28                                         KDL::Twist &_desVel,
29                                         KDL::Twist &_desAcc,
30                                         double _Kpp, double _Kpo,
31                                         double _Kdp, double _Kdo)
32  {
33  }
```

- (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/utls.h` file), finally compute your inverse dynamics control law following the equation

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

In the “`kdl_control.cpp`” we compute desired and effective position and orientation, desired and effective linear and angular velocity and desired acceleration. So we compute errors with “`computeLinearError`”, “`computeOrientationError`”, “`computeOrientationVelocityError`” functions which defined in “`utls.h`” file.

In order to calculate the inverse dynamics control following the equation 4 we have applied both the geometric Jacobian and analytic Jacobian.

The geometric Jacobian is computed by “*getEEJacobian()*” function, while the analytic Jacobian is computed as shown below:

```

125 //DEFINE j_dot
126 Eigen::Matrix<double,6,7> Jdot = robot_->getEEJacDotqDot().data;
127
128 //Compute Analytical Jacobian Ja=Ta^(-1)*J
129 Eigen::Matrix<double,3,1> euler = computeEulerAngles(R_e);
130 Eigen::Matrix<double,6,7> JA = AnalyticalJacobian(J,euler);
131 Eigen::Matrix<double,7,6> JApinv = pseudoinverse(JA);
132
133 // Compute TA_inv
134 //Eigen::Matrix<double,6,> TA_inv = JA * J.inverse();
135 Eigen::Matrix<double,6,6> TA;
136 TA.block(0,0,3,3) = Eigen::Matrix3d::Identity();
137 TA.block(3,3,3,3) = T_matrix(euler);
138 Eigen::Matrix<double,6,6> TA_inv = TA.inverse();
139
140 // Compute TA_dot
141 Eigen::Matrix<double,6,6> TA_dot;
142 TA_dot.block(0,0,3,3) = Eigen::Matrix3d::Identity();
143 TA_dot.block(3,3,3,3) = Tdot_matrix(euler);
144
145 // Compute JA_dot
146 Eigen::Matrix<double,6,7> JAdot = TA_inv*(Jdot - TA_dot*JA);
147
148 /*APPLYING FORMULAE 4*/
149 Eigen::Matrix<double,6,1> y;
150 //y << dot_dot_x_d - Jdot*robot_->getJntVelocities() + Kd*dot_x_tilde + Kp*x_tilde;
151 y << dot_dot_x_d - JAdot*robot_->getJntVelocities() + Kd*dot_x_tilde + Kp*x_tilde;
152
153 //RETURN By + n
154 //return M * (Jpinv*y )+ robot_->getGravity() + robot_->getCoriolis();
155 return M * (JApinv*y )+ robot_->getGravity() + robot_->getCoriolis();

```

As you can see, the commented lines 150 and 154 are referred to geometric Jacobian while the uncommented lines 151 and 155 are referred to analytic Jacobian.

In order to implement these code lines we have created “*AnalyticalJacobian*”, “*computeEulerAngles*”, “*T_matrix*”, “*Tdot_matrix*”, functions in “*utils.h*” file.

Used commands

```
$ cd catkin_ws
```

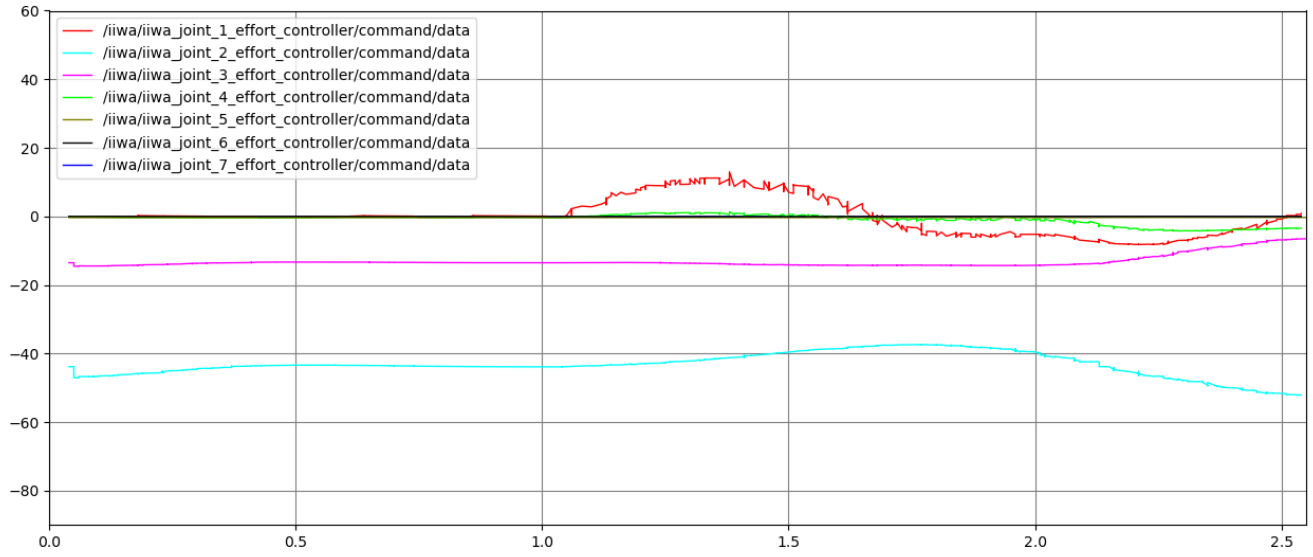
```
$ roscore
```

```
$ roslaunch iiwa_gazebo iiwa_gazebo_effort.launch
```

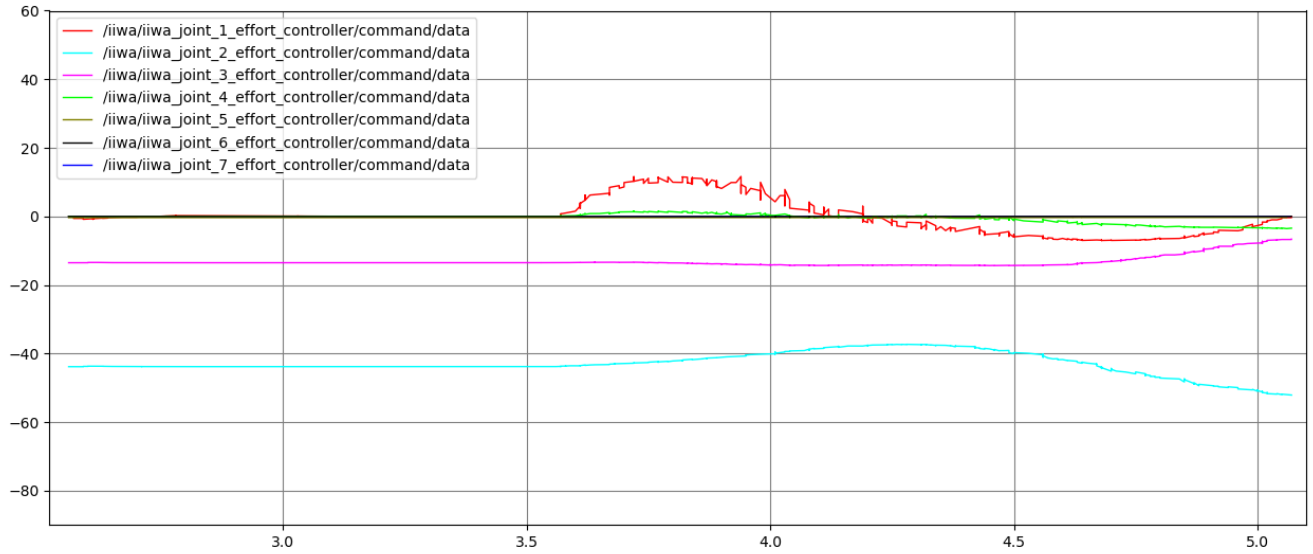
```
$ rosrun kdl_ros_control kdl_robot_test ./src/iiwa_stack/iiwa_description/urdf/iiwa14.urdf
```

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

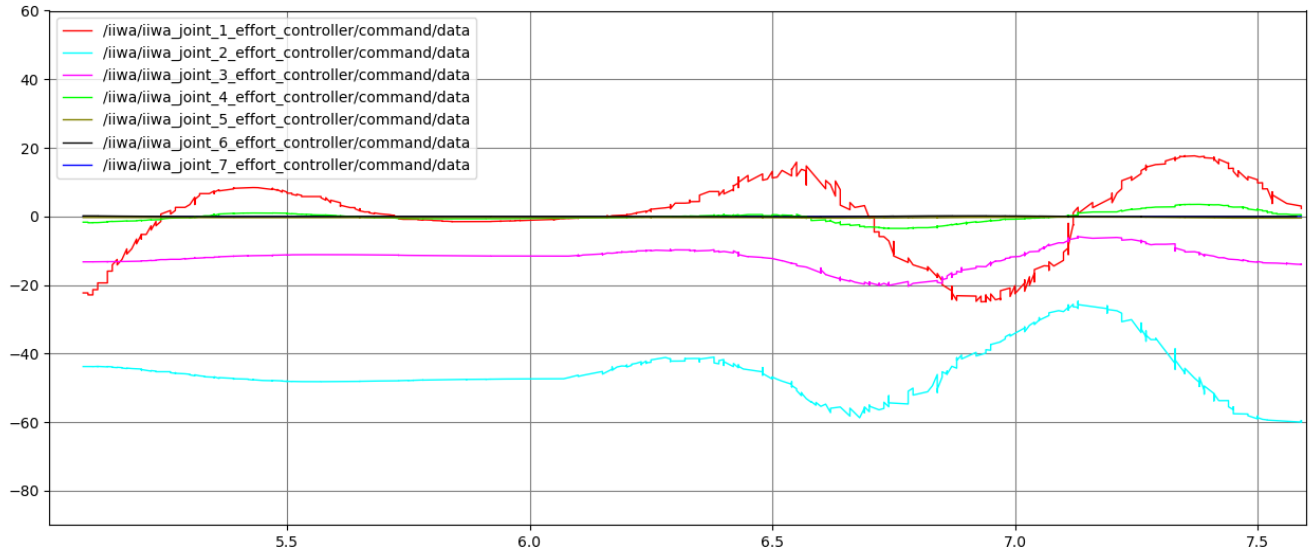
Choice 1 → linear trajectory with trapezioidal velocity profile with $x=0.6$ $y=0.5$ $z=0.4$



Choice 2 → linear trajectory with cubic polynomial velocity profile with $x=0.6$ $y=0.5$ $z=0.4$



Choice 3 → circular trajectory with trapezioidal velocity profile with $r=0.1$



Choice 4 → circular trajectory with cubic polynomial velocity profile with $r=0.1$

