

Robotics Lab: Homework 2 Report

a.y. 2024/2025

Students

Annese Antonio	P38000296
Bosco Stefano	P38000245
Ercolanese Luciana	P38000197
Varone Emanuela	P38000284

Contents

1	Cubic polinomial linear trajectory	3
1.0.1	1.a KDLPlanner::trapezoidal_vel function	3
1.0.2	1.b KDLPlanner::cubic_polynomial function	4
2	Creation of circular and linear trajectories for the robot	5
2.0.1	2.a KDLPlanner::KDLPlanner constructors for the circular trajectory	5
2.0.2	2.b cubic_polynomial function calling	5
2.0.3	2.c Linear trajectory	6
3	Testing the four trajectories	7
3.0.1	3.a Test the four trajectories with the provided joint space inverse dynamics controller	7
3.0.2	3.b/3.c Torques plot	11
4	Inverse dynamics operational space controller	14
4.0.1	4.a/4.b Implementation of the inverse dynamics operational space controller . . .	14
4.0.2	4.c Test the controller and plot the joint torque commands	15
5	GitHub repositories links:	17

Goal: control a manipulator to follow a trajectory

This document contains a report of Homework 2 of the Robotics Lab class. 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.

1 Cubic polynomial linear trajectory

In this section, starting from a trapezoidal velocity profile, we need to implement a cubic polynomial linear trajectory.

1.0.1 1.a KDLPlanner::trapezoidal_vel function

First of all, we need to modify the KDLPlanner class (files kdl_planner.h and kdl_planner.cpp) which provides a basic interface for trajectory creation.

For that purpose, it's important to remember that a trapezoidal velocity profile for a curvilinear abscissa $s \in [0, 1]$ is defined as follows:

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

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).

Recalled that, we define a new KDLPlanner::trapezoidal_vel_function which takes the current time t and the acceleration time t_c as double arguments and returns three double variables s , s_dot and s_ddot that represent the curvilinear abscissa of trajectory, as seen in Fig.1 and Fig.2:

```
/*1a.Define a new KDLPlanner::trapezoidal_vel function that takes the current time t and the acceleration time tc as double arguments
and returns three double variables s, s_dot and s_ddot that represent the curvilinear abscissa of your trajectory.*/
void trapezoidal_vel(double t_, double tc_, double tf_, double &s, double &s_dot, double &s_ddot);
```

Figure 1: Declaration of KDLPlanner::trapezoidal_vel function in kdl_planner.h

```
/*1a.Define a new KDLPlanner::trapezoidal_vel function that takes the current time t and the acceleration time tc as double arguments
and returns three double variables s, s_dot and s_ddot that represent the curvilinear abscissa of your trajectory.*/
void KDLPlanner::trapezoidal_vel(double t_, double tc_, double tf_, double &s, double &s_dot, double &s_ddot) {
    double sc_ddot;
    sc_ddot=2/(tf_*tc_-std::pow(tc_,2));

    if(0 <= t_ && t_ <= tc_)
    {
        s = 0.5*sc_ddot*std::pow(t_,2);
        s_dot = sc_ddot*t_;
        s_ddot = sc_ddot;
    }
    else if(t_ <= tf_-tc_)
    {
        s = sc_ddot*tc_*(t_-tc_/2);
        s_dot = 0.5*sc_ddot;
        s_ddot = 0;
    }
    else if(t_ <= tf_)
    {
        s = 1 - 0.5*sc_ddot*std::pow(tf_-t_,2);
        s_dot = 0;
        s_ddot = 0;
    }
    else {
        std::cout<<"Error: t is greater than tf"<<std::endl;
    }
}
```

Figure 2: Implementation of KDLPlanner::trapezoidal_vel function in kdl_planner.cpp

1.0.2 1.b KDLPlanner::cubic_polynomial function

A cubic polynomial curvilinear abscissa is defined as follows:

$$s(t) = a_3 t^3 + a_2 t^2 + a_1 t + a_0 \quad (2)$$

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).

To create the cubic polynomial curvilinear abscissa for the trajectory, we implement a function named `KDLPlanner::cubic_polynomial` that takes as argument a double t , representing time, and returns three double s , s_dot and s_ddot , which represent the curvilinear abscissa of the trajectory, as seen in Fig.3 and Fig.4:

```
/*1b.Create a function named KDLPlanner::cubic_polynomial that creates the cubic polynomial curvilinear abscissa for you
The function takes as argument a double t representing time and returns three double s, s_dot and s_ddot that represent
void cubic_polynomial(double t_, double tf_, double &s, double &s_dot, double &s_ddot);
```

Figure 3: Declaration of `KDLPlanner::cubic_polynomial` function in `kdl_planner.h`

```
void KDLPlanner::cubic_polynomial(double t_, double tf_, double &s, double &s_dot, double &s_ddot) {
    double s0,sf,v0,vf,a0,a1,a2,a3;

    //Initialize s0,sf,v0,vf
    s0=0;
    sf=1;
    v0=0;
    vf=0;

    //Calculate coefficients a0-a3
    a0 = s0;
    a1 = v0;
    a2 = (3 * (sf - s0) / std::pow(tf_,2)) - ((2 * v0 + vf) / tf_);
    a3 = (-2 * (sf - s0) / std::pow(tf_,3)) + ((v0 + vf) / std::pow(tf_,2));

    // Calculate position s(t)
    s = a3 * std::pow(t_,3) + a2 * std::pow(t_,2) + a1 * t_ + a0;

    // Calculate velocity s_dot(t)
    s_dot = 3 * a3 * std::pow(t_,2) + 2 * a2 * t_ + a1;

    // Calculate acceleration s_ddot(t)
    s_ddot = 6 * a3 * t_ + 2 * a2;
}
```

Figure 4: Implementation of `KDLPlanner::cubic_polynomial` function in `kdl_planner.cpp`

2 Creation of circular and linear trajectories for the robot

The objective of this chapter is to obtain a `compute_trajectory` function that returns position, velocity, and acceleration for both trajectories, circular and linear.

2.0.1 2.a KDLPlanner::KDLPlanner constructors for the circular trajectory

In the `kdl_planner.h` file, we define new constructors `KDLPlanner::KDLPlanner` that takes as arguments:

- the time duration `_trajDuration`, the starting point `Eigen::Vector3d _trajInit` and the radius `_trajRadius` of the trajectory, for the circular trajectory with a cubic profile;
- the time duration `_trajDuration`, the acceleration duration `_accDuration`, the starting point `Eigen::Vector3d _trajInit` and the radius `_trajRadius` of the trajectory, for the circular trajectory with a trapezoidal profile (Fig.5).

```
//2a.New constructors KDLPlanner::KDLPlanner
//circular cubic
KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double _trajRadius);
//circular trapezoidal
KDLPlanner(double _trajDuration, double _accDuration, Eigen::Vector3d _trajInit, double _trajRadius);
```

Figure 5: Constructors for the circular trajectory

Then, we store them in the corresponding class variables (Fig6):

```
//Circular trajectory with trapezoidal profile
KDLPlanner::KDLPlanner(double _trajDuration, double _accDuration, Eigen::Vector3d _trajInit, double _trajRadius)
{
    trajDuration_ = _trajDuration;
    accDuration_ = _accDuration;
    trajInit_ = _trajInit;
    trajRadius_ = _trajRadius;
    time_law = TRAPEZOIDAL;
    path_type = CIRCULAR;
}

//Circular trajectory with cubic profile
KDLPlanner::KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double _trajRadius)
{
    trajDuration_ = _trajDuration;
    trajInit_ = _trajInit;
    trajRadius_ = _trajRadius;
    time_law = CUBIC;
    path_type = CIRCULAR;
}
```

Figure 6: Store constructors arguments in the corresponding class variables

2.0.2 2.b cubic_polinomial function calling

Recalled that a circular path in the yz plane can be defined as follows:

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

we create the positional path as function of $s(t)$ directly in the function `KDLPlanner::compute_trajectory`, by calling the `cubic_polinomial` function to retrieve s and its derivatives from t and then filling in the trajectory point fields `traj.pos`, `traj.vel`, and `traj.acc` (Fig.7):

```

trajectory_point KDLPlanner::compute_trajectory(double time)
{
    double s, s_dot, s_ddot;
    trajectory_point traj;

    // Selezione del tipo di legge temporale
    if (time_law == CUBIC) {
        cubic_polynomial(time, trajDuration_, s, s_dot, s_ddot);
    } else if (time_law == TRAPEZOIDAL) {
        trapezoidal_vel(time, accDuration_, trajDuration_, s, s_dot, s_ddot);
    }

    // Selezione del tipo di percorso
    if (path_type == CIRCULAR) {
        // Applica la posizione curvilinea al percorso circolare
        traj.pos(0)=trajInit_(0);
        traj.pos(1)=trajInit_(1)-trajRadius_*cos(2*M_PI*s);
        traj.pos(2)=trajInit_(2)-trajRadius_*sin(2*M_PI*s);

        traj.vel(0)=0;
        traj.vel(1)=trajRadius_*sin(2*M_PI*s)*2*M_PI*s_dot;
        traj.vel(2)=-trajRadius_*cos(2*M_PI*s)*2*M_PI*s_dot;

        traj.acc(0)=0;
        traj.acc(1)=4 * std::pow(M_PI,2) * trajRadius_ * cos(2 * M_PI * s) * std::pow(s_dot,2)
            + 2 * M_PI * trajRadius_ * sin(2 * M_PI * s) * s_ddot;
        traj.acc(2) = 2 * M_PI * trajRadius_ * (2 * M_PI * sin(2 * M_PI * s) * std::pow(s_dot,2) - cos(2 * M_PI * s) * s_ddot);
    }
}

```

Figure 7: Calling the cubic_polynomial function

2.0.3 2.c Linear trajectory

In the kdl_planner.h file, we define new constructors KDLPlanner::KDLPlanner that takes as arguments:

- the time duration _trajDuration, the starting point `Eigen::Vector3d _trajInit` and the end point `_trajEnd` of the trajectory, for the linear trajectory with a cubic profile;
- the time duration _trajDuration, the acceleration duration _accDuration, the starting point `Eigen::Vector3d _trajInit` and the end point `_trajEnd` of the trajectory, for the linear trajectory with a trapezoidal profile (Fig.8).

```

//linear cubic
KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, Eigen::Vector3d _trajEnd);
//linear trapezoidal
KDLPlanner(double _trajDuration, double _accDuration,
            Eigen::Vector3d _trajInit, Eigen::Vector3d _trajEnd);

```

Figure 8: Constructors for the linear trajectory

Then, we store them in the corresponding class variables (Fig.9):

```

//2a. store the arguments of the new constructors in the corresponding class variable
//Linear trajectory with trapezoidal profile
KDLPlanner::KDLPlanner(double _trajDuration, double _accDuration, Eigen::Vector3d _trajInit, Eigen::Vector3d _trajEnd)
{
    trajDuration_ = _trajDuration;
    accDuration_ = _accDuration;
    trajInit_ = _trajInit;
    trajEnd_ = _trajEnd;
    time_law = TRAPEZOIDAL;
    path_type = LINEAR;
}

//Linear trajectory with cubic profile
KDLPlanner::KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, Eigen::Vector3d _trajEnd)
{
    trajDuration_ = _trajDuration;
    trajInit_ = _trajInit;
    trajEnd_ = _trajEnd;
    time_law = CUBIC;
    path_type = LINEAR;
}

```

Figure 9: Store constructors arguments in the corresponding class variables

Next, we included the linear trajectory in the `KDLPlanner::compute_trajectory` function, as seen in (Fig.10):

```
trajectory_point KDLPlanner::compute_trajectory(double time)
{
    double s, s_dot, s_ddot;
    trajectory_point traj;

    // Selezione del tipo di legge temporale
    if (time_law == CUBIC) {
        cubic_polynomial(time, trajDuration_, s, s_dot, s_ddot);
    } else if (time_law == TRAPEZOIDAL) {
        trapezoidal_vel(time, accDuration_, trajDuration_, s, s_dot, s_ddot);
    }

    // Selezione del tipo di percorso
    if (path_type == CIRCULAR) {
        // Applica la posizione curvilinea al percorso circolare
        traj.pos(0)=trajInit_(0);
        traj.pos(1)=trajInit_(1)-trajRadius_*cos(2*M_PI*s);
        traj.pos(2)=trajInit_(2)-trajRadius_*sin(2*M_PI*s);

        traj.vel(0)=0;
        traj.vel(1)=trajRadius_*sin(2*M_PI*s)*2*M_PI*s_dot;
        traj.vel(2)=-trajRadius_*cos(2*M_PI*s)*2*M_PI*s_dot;

        traj.acc(0)=0;
        traj.acc(1)=4 * std::pow(M_PI,2) * trajRadius_ * cos(2 * M_PI * s) * std::pow(s_dot,2)
            + 2 * M_PI * trajRadius_ * sin(2 * M_PI * s) * s_ddot;
        traj.acc(2) = 2 * M_PI * trajRadius_ * (2 * M_PI * sin(2 * M_PI * s) * std::pow(s_dot,2) - cos(2 * M_PI * s) * s_ddot);
    }
    else if (path_type == LINEAR) {
        //Posizione lungo la traiettoria impostata come interpolazione lineare tra trajInit_ e trajEnd_ in funzione di s.
        traj.pos = trajInit_ + (trajEnd_ - trajInit_) * s;

        // Velocità costante lungo la traiettoria
        traj.vel = s_dot * (trajEnd_ - trajInit_).normalized(); // Normalizzo per ottenere direzione costante

        // Accelerazione nulla lungo la traiettoria lineare
        traj.acc = Eigen::Vector3d::Zero();
    }
    return traj;
}
```

Figure 10: Linear trajectory inclusion in the `KDLPlanner::compute_trajectory` function

3 Testing the four trajectories

The objective of this chapter is to:

- test both linear and circular trajectories, each with trapezoidal velocity and of cubic polynomial curvilinear abscissa;
- plot the torques sent to the manipulator.

3.0.1 3.a Test the four trajectories with the provided joint space inverse dynamics controller

At this point, we can create both linear and circular trajectories, each with trapezoidal velocity of cubic polynomial curvilinear abscissa. To test the four trajectories with the provided joint space inverse dynamics controller, we modify our main file `ros2_kdl_node.cpp`, in this way:

- First, we declared two parameters called `time_law` and `path_type`, memorized their value in the respective variables (`time_law_` and `path_type_`), printed the current value of the parameters in the log and checked whether the parameters values are valid, as seen in Fig.11:

```
// Parameters added in the constructor
declare_parameter("time_law", "trapezoidal"); // Values: "trapezoidal" or "cubic"
declare_parameter("path_type", "linear"); // Values: "linear" or "circular"

get_parameter("time_law", time_law_);
get_parameter("path_type", path_type_);

RCLCPP_INFO(get_logger(), "Current time_law is: '%s'", time_law_.c_str());

if (!(time_law_ == "trapezoidal" || time_law_ == "cubic"))
{
    RCLCPP_INFO(get_logger(), "Selected time_law is not valid!"); return;
}

RCLCPP_INFO(get_logger(), "Current path_type is: '%s'", path_type_.c_str());

if (!(path_type_ == "linear" || path_type_ == "circular"))
{
    RCLCPP_INFO(get_logger(), "Selected path_type is not valid!"); return;
}
```

Figure 11: declared time_law and path_type parameters

- Depending on the path type and time law, we initialized an instance of KDLPanner with the appropriate parameters, as seen in Fig.12:

```
// Plan trajectory
double traj_duration = 1.5, acc_duration = 0.5, t = 0.0, trajRadius = 0.2;

//planner_ = KDLPanner(traj_duration, acc_duration, init_position, end_position); // cu

//Example of planner configuration with a trajectory type
if (path_type_ == "linear") {
    if(time_law_ == "cubic"){
        planner_ = KDLPanner(traj_duration, init_position, end_position);
    }
    else{
        planner_ = KDLPanner(traj_duration, acc_duration, init_position, end_position);
    }
} else if (path_type_ == "circular") {
    if(time_law_ == "cubic"){
        planner_ = KDLPanner(traj_duration, init_position, trajRadius);
    }
    else{
        planner_ = KDLPanner(traj_duration, acc_duration, init_position, trajRadius);
    }
}
// Retrieve the first trajectory point
trajectory_point p = planner_.compute_trajectory(t);
```

Figure 12: Initialization of an instance of KDLPanner with the appropriate parameters

- Then, we have configured the node's command interface (position, velocity, or effort), checked that it is valid and printed information in the log, as seen in Fig.13:

```
// declare cmd_interface parameter (position, velocity or effort)
declare_parameter("cmd_interface", "position"); // defaults to "position"
get_parameter("cmd_interface", cmd_interface_);

RCLCPP_INFO(get_logger(), "Current cmd interface is: '%s'", cmd_interface_.c_str());

if (!(cmd_interface_ == "position" || cmd_interface_ == "velocity" || cmd_interface_ == "effort"))
{
    RCLCPP_INFO(get_logger(), "Selected cmd interface is not valid!"); return;
}
```

Figure 13: Configuration of the node's command interface

Before doing so, it was necessary to edit the iiwa_controllers.yaml file by adding an effort_controller Fig.14:

```

effort_controller:
  type: effort_controllers/JointGroupEffortController

effort_controller:
  ros__parameters:
    command_interfaces:
      - effort
    state_interfaces:
      - position
      - velocity
      - effort
    joints:
      - joint_a1
      - joint_a2
      - joint_a3
      - joint_a4
      - joint_a5
      - joint_a6
      - joint_a7

```

Figure 14: Adding effort_controller in the iiwa_controllers.yaml file

- Next, we configured a publisher for the effort interface. Specifically, we created a publisher to publish commands on the topic “/effort_controller/commands” with a queue of size 10 and introduced a timer that calls periodically (every 30 milliseconds) the cmd_publisher method (Fig.15):

```

}
} else if(cmd_interface_ == "effort"){
  // Create cmd publisher

  cmdPublisher_ = this->create_publisher<FloatArray>("/effort_controller/commands", 10);
  timer_ = this->create_wall_timer(std::chrono::milliseconds(30),
    std::bind(&Iiwa_pub_sub::cmd_publisher, this));
}

```

Figure 15: Configuration of a publisher for the effort interface

- With the provided joint space inverse dynamics controller (Fig.16):

```

Eigen::VectorXd KDLController::idCntr(KDL::JntArray &_qd,
                                       KDL::JntArray &_dq,
                                       KDL::JntArray &_ddqd,
                                       double _Kp, double _Kd)
{
  // read current state
  Eigen::VectorXd q = robot_->getJntValues();
  Eigen::VectorXd dq = robot_->getJntVelocities();

  // calculate errors
  Eigen::VectorXd e = _qd.data - q;
  Eigen::VectorXd de = _dq.data - dq;

  Eigen::VectorXd ddqd = _ddqd.data;
  return robot_->getJsim() * (ddqd + _Kd*de + _Kp*e)
    + robot_->getCoriolis();
}

```

Figure 16: First inverse dynamics controller

we calculated torque values after updating joint velocities, positions and accelerations (Fig.17):

```

}
else if(cmd_interface_ == "effort"){
    //Store the current joint velocity as a reference to calculate the next acceleration:
    joint_velocity_old.data=joint_velocities_.data;

    //Combine a desired velocity p.vel with an error term for correction:
    NOTE: The three zeros represent rotation components not considered here!*/
    Vector6d cartvel; cartvel << p.vel + error, 0,0,0;

    //Update joint velocities, using the pseudoinverse of the end-effector Jacobian to map the desired Cartesian velocity (cartvel)
    dq_des.data = pseudoinverse(robot_->getEEJacobian().data)*cartvel;

    //Calculate the new joint positions by integrating the velocities (joint_velocities_) with the time step dt:
    q_des.data = joint_positions_.data + joint_velocities_.data*dt;

    //Calculate joint acceleration by discrete numerical derivative:
    joint_acceleration_d_.data=(joint_velocities_.data-joint_velocity_old.data)/dt;

    //Use the first method (idCntr) to calculate the required joint torques:
    torque_values = controller_.idCntr(q_des,dq_des,joint_acceleration_d_, _Kp, _Kd);
}

```

Figure 17: torque_values

Finally, we tested the four trajectories, using the following commands:

- cubic-circular:

```
ros2 launch iiwa_bringup iiwa.launch.py time_law:="cubic" path_type:="circular" use_sim:="true"
command_interface:="effort" robot_controller:="effort_controller"
```

```
ros2 run ros2_kdl_package ros2_kdl_node --ros-args -p time_law:="cubic" -p path_type:="circular"
-p cmd_interface:="effort"
```

The attached video ("video_cubic_circular.mp4") shows the cubic circular trajectory executed by the robot.

- trapezoidal-circular:

```
ros2 launch iiwa_bringup iiwa.launch.py time_law:="trapezoidal" path_type:="circular"
use_sim:="true" command_interface:="effort" robot_controller:="effort_controller"
```

```
ros2 run ros2_kdl_package ros2_kdl_node --ros-args -p time_law:="trapezoidal" -p path_type:="circular"
-p cmd_interface:="effort"
```

The attached video ("video_trapezoidal_circular.mp4") shows the trapezoidal circular trajectory executed by the robot.

- cubic-linear:

```
ros2 launch iiwa_bringup iiwa.launch.py time_law:="cubic" path_type:="linear" use_sim:="true"
command_interface:="effort" robot_controller:="effort_controller"
```

```
ros2 run ros2_kdl_package ros2_kdl_node --ros-args -p time_law:="cubic" -p path_type:="linear"
-p cmd_interface:="effort"
```

The attached video ("video_cubic_linear.mp4") shows the cubic linear trajectory executed by the robot.

- trapezoidal-linear:

```
ros2 launch iiwa_bringup iiwa.launch.py time_law:="trapezoidal" path_type:="linear"
use_sim:="true" command_interface:="effort" robot_controller:="effort_controller"
```

```
ros2 run ros2_kdl_package ros2_kdl_node --ros-args -p time_law:="trapezoidal" -p path_type:="linear"
-p cmd_interface:="effort"
```

The attached video ("video_trapezoidal_linear.mp4") shows the trapezoidal linear trajectory executed by the robot.

NOTES:

To ensure that the robot, once the trajectory is completed, remains stationary in the position reached, we proceeded as seen in Fig.18:

```

RCLCPP_INFO_ONCE(this->get_logger(), "Trajectory executed successfully ...");
// Send joint effort commands
if(cmd_interface_ == "effort" || "effort_cartesian"){

    KDLController controller_(*robot_);
    q_des.data=joint_positions_.data;
    // Azzerare i valori di qd (velocità dei giunti)
    dq_des.data = Eigen::VectorXd::Zero(7,1);
    // Azzerare i valori di qdd (accelerazioni dei giunti)
    joint_acceleration_d_.data = Eigen::VectorXd::Zero(7,1);

    torque_values = controller_.idCntr(q_des,dq_des,joint_acceleration_d_, _Kp, _Kd);

    // // Update KDLrobot structure
    robot_->update(toStdVector(joint_positions_.data),toStdVector(joint_velocities_.data));

    for (long int i = 0; i < torque_values.size(); ++i) {

        desired_commands_[i] = torque_values(i);
    }
}

```

Figure 18: Code implemented to ensure that the robot remains stationary in the position reached

In the implemented code, the velocity and acceleration values are initialized to zero, in order to ensure that the joints are stationary. With the "torque_values = controller_.idCntr(joint_positions_, qd, qdd, _Kp, _Kd)" instruction, the values of the torques with zero velocities and accelerations are re-calculated. Then the internal structure of the robot is updated with the current positions and velocities of the joints. For each joint, the calculated torque value (torque_values(i)) is transferred into desired_commands_, a buffer used to send commands to the robot.

3.0.2 3.b/3.c Torques plot

After tuning the control gains K_p and K_d and choosing the values:

- $K_p = 170$;
- $K_d = 30$;

we plotted the torques sent to the manipulator.

The **first method** pursued to extrapolate the plot was based on the use of `rqt_plot`. The attached video ("video_rqt_plot.mp4") shows how, after launching the `iiwa.launch.py` and the `ros2_kdl_node.cpp` files, the `rqt_plot` plugin works: it enables to visualize the torques sent to the manipulator at each run. The **second method** pursued to extrapolate the plot consisted of saving the joint torque command topics in a bag file and plotting it using MATLAB (Fig. 19):

```

Plot effort

Variable Define

1  path = 'Circular';
2  %path = 'Linear';
3
4  %time_low = 'Trapezoidal';
5  time_low = 'Cubic';
6
7  %esercizio = '3';
8  esercizio = '4';
9
10
11  trajectoryName = [path, ' ', time_low, 'Trajectory'];
12  fileName = [path, '-', time_low, '-', esercizio, '.png'];
13  recName = [path, '-', time_low, '-', esercizio, '.db3'];
14
15
16  imgFolderPath = "C:\Users\anton\Desktop\Robotic\Immagini\";
17  recFolderPath = "C:\Users\anton\Desktop\Robotic\my_recording\";

File bag loading

18  % Using ros2bagreader to read .db3 file
19  bag = ros2bagreader(fullfile(recFolderPath, recName));
20
21  % Disp topics in the bag file
22  disp(bag.AvailableTopics)
23
24  % Select the topic
25  msgs = readMessages(select(bag, 'Topic', '/effort_controller/commands'));

Messages copying

26  % Number of messages
27  n = numel(msgs);
28
29  % Pre-allocate matrix with dimension (# of message x # of joint)
30  torqueValues = zeros(n, 7);
31
32  % Copying torque value from msgs data traspost
33  for i = 1:n
34      torqueValues(i, :) = msgs{i}.data';
35  end

Data plotting

36  % defining time
37  loop_rate = 150/1.5;
38  n_of_point = 1:n;
39  time = n_of_point/loop_rate;
40
41  figure;
42  hold on;
43
44  for j = 1:7
45      plot(time, torqueValues(:, j), 'DisplayName', ['Torque ' num2str(j)], 'LineWidth', 3);
46  end
47
48  % Labels
49  xlabel('Time');
50  ylabel('Torque');
51  title([path, ' ', time_low, ' ', esercizio]);
52  legend show;
53  grid on;
54  hold off;
55  saveas(gcf, fullfile(imgFolderPath, fileName))

```

Figure 19: Script Matlab

The figures (Fig. 20, Fig. 21, Fig. 22, Fig. 23) provided below show the plots obtained for each of the four trajectories:

- Cubic-Circular:

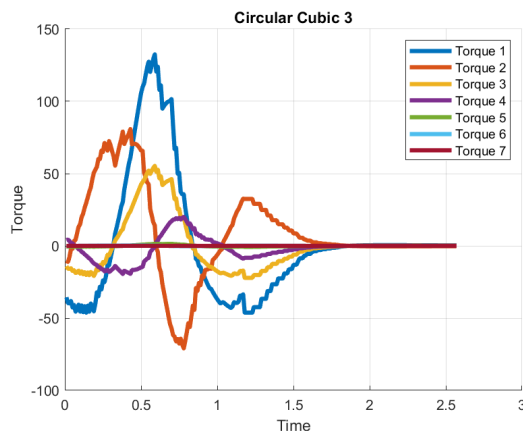


Figure 20: Torques sent to the manipulator during the cubic circular trajectory

- Trapezoidal-Circular:

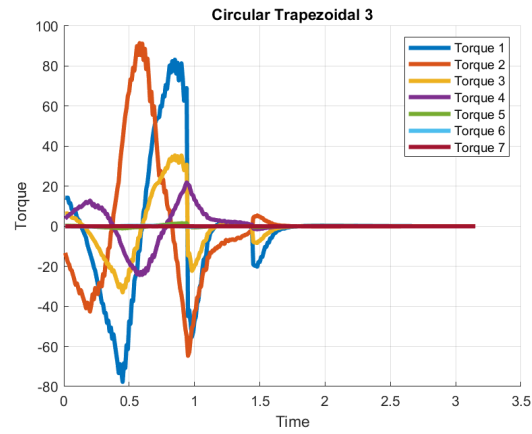


Figure 21: Torques sent to the manipulator during the trapezoidal circular trajectory

- Cubic-Linear:

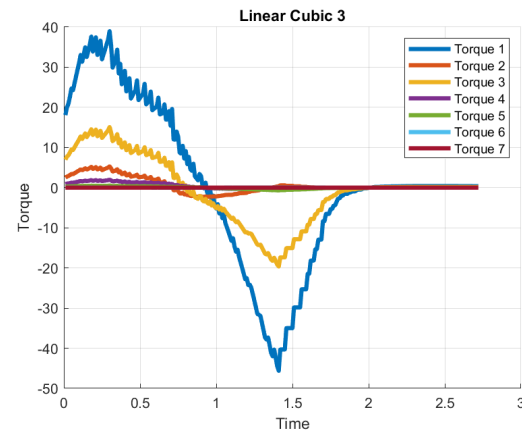


Figure 22: Torques sent to the manipulator during the cubic linear trajectory

- Trapezoidal-Linear:

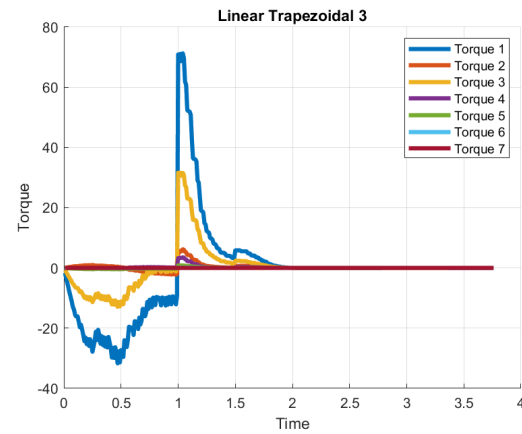


Figure 23: Torques sent to the manipulator during the trapezoidal linear trajectory

4 Inverse dynamics operational space controller

The objective of this chapter is to implement an inverse dynamics operational space controller, test the controller along the planned trajectories and plot the corresponding joint torque commands.

4.0.1 4.a/4.b Implementation of the inverse dynamics operational space controller

Differently from joint space inverse dynamics controller implemented in the first method (idCntr), the operational space controller computes the errors in Cartesian space.

Into the kdl_control.cpp file, to implement the inverse dynamics operational space controller, we use the second KDLController::idCntr function, which takes as arguments the desired KDL::Frame pose, the KDL::Twist velocity, the KDL::Twist acceleration and four gains as arguments: Kpp, position error proportional gain, Kdp, position error derivative gain, and so on for the orientation (Fig.24):

```
Eigen::VectorXd idCntr(KDL::Frame & desPos,
                       KDL::Twist & desVel,
                       KDL::Twist & desAcc,
                       double _Kpp,
                       double _Kpo,
                       double _Kdp,
                       double _Kdo);
```

Figure 24: KDLController::idCntr function arguments

The logic behind the implementation of the controller is sketched within the function:

- calculate the gain matrices, Fig.25:

```
//Calculation of gain matrices
//Both matrices are initialized to zero:
Eigen::Matrix<double, 6, 6> Kp = Eigen::MatrixXd::Zero(6, 6);
Eigen::Matrix<double, 6, 6> Kd = Eigen::MatrixXd::Zero(6, 6);
//Update the 3x3 block in the upper left part of Kp: represents linear proportional gains
Kp.block(0, 0, 3, 3) = _Kpp * Eigen::Matrix3d::Identity();
//Update the 3x3 block in the lower right part of Kp: represents angular proportional gains
Kp.block(3, 3, 3, 3) = _Kpo * Eigen::Matrix3d::Identity();
//Update the 3x3 block in the upper left part of Kd: represents linear derivative gains.
Kd.block(0, 0, 3, 3) = _Kdp * Eigen::Matrix3d::Identity();
//Update 3x3 block in the lower right part of Kd: represents angular derivative gains.
Kd.block(3, 3, 3, 3) = _Kdo * Eigen::Matrix3d::Identity();
```

Figure 25: Gain matrices

- read the current Cartesian state of the manipulator in terms of endeffector parametrized pose x, velocity, \dot{x} , and acceleration, \ddot{x} Fig.26:

```
//Position:
//Current position and orientation of the EE
KDL::Frame cart_pose = robot->getEEFrame();
//Desired position of the EE
Eigen::Vector3d p_d(desPos.p.data);
//Current position of the EE
Eigen::Vector3d p_e(cart_pose.p.data);

//Desired rotation matrix:
Eigen::Matrix<double, 3, 3, Eigen::RowMajor> R_d(desPos.M.data);
//Current rotation matrix:
Eigen::Matrix<double, 3, 3, Eigen::RowMajor> R_e(cart_pose.M.data);
R_d = matrixOrthonormalization(R_d);
R_e = matrixOrthonormalization(R_e);

//Velocity:
//Linear and angular velocity of EE:
KDL::Twist cart_twist = robot->getEEVelocity();
//Desired linear velocity:
Eigen::Vector3d dot_p_d(desVel.vel.data);
//Current linear velocity:
Eigen::Vector3d dot_p_e(cart_twist.vel.data);
//Desired angular velocity:
Eigen::Vector3d omega_d(desVel.rot.data);
//Current angular velocity:
Eigen::Vector3d omega_e(cart_twist.rot.data);

//Acceleration:
Eigen::Matrix<double, 6, 1> dot_dot_x_d;
//Desired accelerations (linear and angular):
Eigen::Matrix<double, 3, 1> dot_dot_p_d(desAcc.vel.data);
Eigen::Matrix<double, 3, 1> dot_dot_r_d(desAcc.rot.data);
```

Figure 26: Cartesian state of the manipulator

- retrieve the current joint space inertia matrix M , the Jacobian and its time derivative Fig.27:

```
//Jacobian
KDL::Jacobian JEE = robot->getEEJacobian();
//Identity matrix
Eigen::Matrix<double, 7, 7> I = Eigen::Matrix<double, 7, 7>::Identity();
//Inertia matrix
Eigen::Matrix<double, 7, 7> M = robot->getJsim();
//Pseudoinverse of the Jacobian: used to map forces from the operational space to the joints.
Eigen::Matrix<double, 7, 6> Jpinv = pseudoinverse(robot->getEEJacobian().data);
```

Figure 27

- compute the linear ep and the angular eo errors Fig.28:

```
//Linear errors:
Eigen::Matrix<double, 3, 1> e_p = computeLinearError(p_d, p_e);
Eigen::Matrix<double, 3, 1> dot_e_p = computeLinearError(dot_p_d, dot_p_e);

//Orientation errors
Eigen::Matrix<double, 3, 1> e_o = computeOrientationError(R_d, R_e);
Eigen::Matrix<double, 3, 1> dot_e_o = computeOrientationVelocityError(omega_d, omega_e, R_d, R_e);

//Construction of states
//Combine linear (ep) and angular (eo) errors into a single vector: this is the global error in the operational space
Eigen::Matrix<double, 6, 1> x_tilde;
x_tilde << e_p; e_o;
//Combine the linear (dot_e_p) and angular (dot_e_o) velocity errors into a single vector: representing the time derivative of the error
Eigen::Matrix<double, 6, 1> dot_x_tilde;
dot_x_tilde << dot_e_p; dot_e_o;
```

Figure 28: Errors computation

- finally, compute (Fig.29) the inverse dynamics control law following the equation:

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

```
//Represent the desired acceleration for the end-effector in linear (dot_dot_p_d) and angular motion (dot_dot_r_d)
dot_dot_x_d << dot_dot_p_d; dot_dot_r_d;

//Inverse Dynamics
Eigen::Matrix<double, 6, 1> y = dot_dot_x_d - robot->getEEJacDot().data * robot->getJntVelocities()
+ Kd * dot_x_tilde + Kp * x_tilde;

//Torques
return M * (Jpinv * y) + robot->getCoriolis();
```

Figure 29: Inverse dynamics control computation

4.0.2 4.c Test the controller and plot the joint torque commands

In the `ros2.kdl.node.cpp` file we introduced a new command interface, called "effort_cartesian" (Fig.30). Selecting this type of interface will use the `idCntr` function, that implements the inverse dynamics operational space controller, to calculate torque values.

```
if (!cmd_interface_ == "position" || "velocity" || "effort" || "effort_cartesian")
{
    RCLCPP_INFO(get_logger(), "Selected cmd interface is not valid!"); return;
}

else if(cmd_interface_ == "effort" || "effort_cartesian")
{
    // Create cmd publisher
    cmdPublisher_ = this->create_publisher<FloatArray>("/effort_controller/commands", 10);
    timer_ = this->create_wall_timer(std::chrono::milliseconds(10),
        std::bind(&Giwa_pub_sub::cmd_publisher, this));

    for (long int i = 0; i < nj; ++i) {
        desired_commands_[i] = 0;
    }
}

else if(cmd_interface_ == "effort_cartesian")
{
    Vector6d cartacc; cartacc << p.acc + error/dt, 0,0,0;
    desVel = KDL::Twist(KDL::Vector(p.vel[0], p.vel[1], p.vel[2]), KDL::Vector::Zero());
    desAcc = KDL::Twist(KDL::Vector(p.acc[0], p.acc[1], p.acc[2]), KDL::Vector::Zero());
    desPos.M = desFrame.M;
    desPos.p = desFrame.p;

    //Use the second method (idCntr) to calculate the required joint torques:
    torque_values=controller_.idCntr(desPos,desVel,desAcc,_Kpp,_Kpo,_Kdp,_Kdo);
}
```

Figure 30: Inverse dynamics control computation

After tuning the control gains and choosing the values:

- $K_{pp} = 90$;
- $K_{dp} = 2 \cdot \sqrt{K_{pp}}$;
- $K_{po} = 100$;
- $K_{do} = 2 \cdot \sqrt{K_{po}}$;

we tested the controller and plotted the torques sent to the manipulator.

The attached video:

- ("video_cubic_circular_4c.mp4") shows the cubic circular trajectory executed by the robot;
- ("video_trapezoidal_circular_4c.mp4") shows the trapezoidal circular trajectory executed by the robot;
- ("video_cubic_linear_4c.mp4") shows the cubic linear trajectory executed by the robot;
- ("video_trapezoidal_linear_4c.mp4") shows the trapezoidal linear trajectory executed by the robot.

The method pursued to extrapolate the plot consisted of saving the joint torque command topics in a bag file and plotting it using MATLAB. The figures (Fig. 31, Fig. 32, Fig. 33, Fig. 34) provided below show the plots obtained for each of the four trajectories:

- Cubic-Circular:

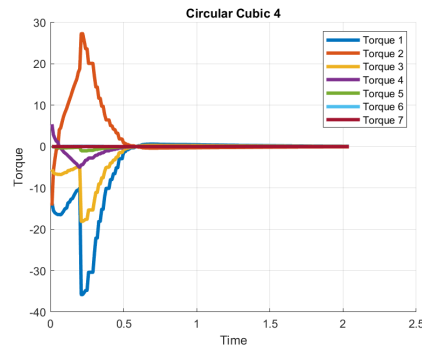


Figure 31: Torques sent to the manipulator during the cubic circular trajectory

- Trapezoidal-Circular:

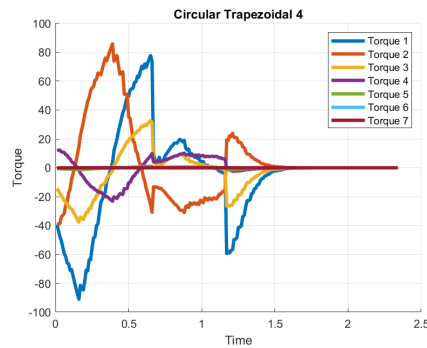


Figure 32: Torques sent to the manipulator during the trapezoidal circular trajectory

- Cubic-Linear:

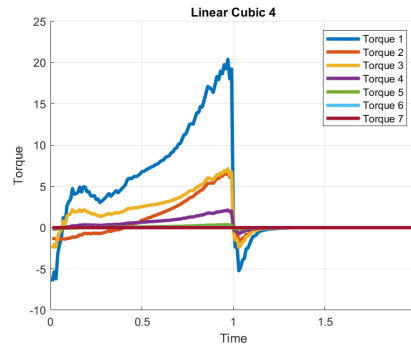


Figure 33: Torques sent to the manipulator during the cubic linear trajectory

- Trapezoidal-Linear:

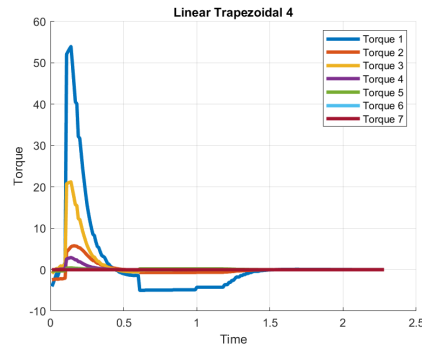


Figure 34: Torques sent to the manipulator during the trapezoidal linear trajectory

5 GitHub repositories links:

Students

Annese Antonio	https://github.com/antann1/HomeworkRL_2.git
Bosco Stefano	https://github.com/SteBosco/HMW2_RL_2024.git
Ercolanese Luciana	https://github.com/LErcolanese/RL_Homework2.git
Varone Emanuela	https://github.com/Emanuela-var/Homework2_RL_.git