UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria dell'Automazione e Robotica

Robotics Lab Project

HOMEWORK 2

Academic Year 2024/2025

Relatore
Ch.mo prof. Mario Selvaggio
Candidato
Emmanuel Patellaro
P38000239
Repository
`https://github.com/EmmanuelPat6/Homework_2.git`

Abstract

This project focuses on developing a ROS package to dynamically control a 7-degrees-of-freedom robotic manipulator arm into the Gazebo Environment. At starting point, *ros2_kdl_package* package (`https://github.com/RoboticsLab2024/ros2_kdl_package`) has been used. First, trapezoidal velocity profile and cubic polynomial functions have been developed to allow the robot to perform both linear and circular trajectories. Then, the four trajectories have been tested with a Joint Space Inverse Dynamics Controller, in particular a PD+ Controller with Gravity Compensation. For this purpose, also some plots will be shown in the following, after an appropriate tuning of the relative gains. At the end, an Inverse Dynamics Operational Space Controller will be implemented.

Contents

Control a Manipulator to follow a Trajectory

Joint Space Trajectories

A manipulator motion can be specified in the **Joint Space** through the joint variables. In this specific case the robot is composed by 7 revolute joints. When a sequence of points is specified, there are several ways to interpolate. To do that, we need something which is simple to compute and something that minimize, for example, some curvatures and cost functions. Solving this **Constrain Optimization Problem**

$$min \int_0^{t_f} \tau^2(t)dt$$

s.t.

$$min \int_0^{t_f} \omega(t)dt = q_f - q_i$$

it has been demonstrated that the best solution, from an energetically point of view, is the **cubic polynomial** for the joint motion

$$q(t) = a_3t^3 + a_2t^2 + a_1t + a_0 \tag{1}$$

and so a **parabolic velocity profile**

$$\dot{q}(t) = 3a_3t^2 + 2a_2t + a_1 \tag{2}$$

and a **linear acceleration profile**

$$\ddot{q}(t) = 6a_3 t + 2a_2 \tag{3}$$

The four coefficients are computed offline imposing boundary conditions

$$a_0 = q_i$$

$$a_1 = \dot{q}_i$$

$$a_3 t_f^3 + a_2 t_f^2 + a_1 t_f + a_0 = q_f$$

$$3a_3 t_f^2 + 2a_2 t_f + a_1 = \dot{q}_f$$

An alternative approach involves using a **trapezoidal velocity profile**, which impose a **constant acceleration** in the start phase, a **cruise velocity**, and a **constant deceleration** in the arrival phase. This is widely used in industry because it allows for immediate verification of whether the velocities and accelerations, imposed by such motion laws, are consistent with the physical characteristics of the mechanical manipulator.

$$q(t) = \begin{cases} q_i + \frac{1}{2}\ddot{q}_c t^2 & \text{for } 0 \leq t \leq t_c \\ q_i + \ddot{q}_c t_c (t - t_c/2) & \text{for } t_c < t \leq t_f - t_c \\ q_f + \frac{1}{2}\ddot{q}_c (t_f - t)^2 & \text{for } t_f - t_c < t \leq t_f \end{cases} \tag{4}$$

In the next sections, these two will be used through the **curvilinear abscissa** $s \in [0, 1]$

Trapezoidal Velocity Profile

As previously mentioned at the beginning of this chapter, the curvilinear abscissa will be used in the following. So the system (4) becomes:

$$s(t) = \begin{cases} \frac{1}{2}\ddot{s}_c t^2 & \text{for } 0 \leq t \leq t_c \\ \ddot{q}_c t_c (t - t_c/2) & \text{for } t_c < t \leq t_f - t_c \\ 1 + \frac{1}{2}\ddot{s}_c (t_f - t)^2 & \text{for } t_f - t_c < t \leq t_f \end{cases} \tag{5}$$

Let's define a new **KDLPlanner::trapezoidal_vel** function. In *kdl_planner.h*

```
//Trapezoidal Velocity
void trapezoidal_vel(double t, double tc, double &s, double &s_dot,
    double &s_ddot);
```

Here, passage by reference to return multiple arguments is used. Then, in *kdl_planner.cpp*

```
//Trapezoidal Velocity
void KDLPlanner::trapezoidal_vel(double t, double tc, double &s,
    double &s_dot, double &s_ddot){

    //s0 = 0;
    //sf = 1;
    //Compute Acceleration
    double sc_ddot = 5.0/(std::pow(trajDuration_,2));

    //Acceleration
    if (t <= tc)
    {
        s = 0.5*sc_ddot*std::pow(t,2);
        s_dot = sc_ddot*t;
        s_ddot = sc_ddot;
    }
    //Cruise
    else if (t <= trajDuration_- tc)
    {
        s = sc_ddot*tc*(t-tc/2);
        s_dot = sc_ddot*tc;
        s_ddot = 0;
    }
    //Deceleration
    else
    {
        s = 1 - 0.5*sc_ddot*std::pow(trajDuration_-t,2);
        s_dot = sc_ddot*(trajDuration_-t);
```

```
        s_ddot = -sc_ddot;
    }
}
```

Cubic Polynomial

The same was done for the cubic polynomial defining a function named **KDLPlanner::cubic_polinomial**. In *kdl_planner.h*

```
//Cubic Polynomial
void cubic_polinomial(double t, double &s, double &s_dot, double &
    s_ddot);
```

while in *kdl_planner.cpp*

```
//Cubic Polynomial
void KDLPlanner::cubic_polinomial(double t, double &s, double &s_dot,
     double &s_ddot){
    //Remember that s(t) = a3*t^3 + a2*t^2 + a1*t + a0
    //Offline Coefficients Computation imposing Boundary Conditions

    double a0, a1, a2, a3;
    a0 = 0.0;
    a1 = 0.0;
    a2 = 3/(std::pow(trajDuration_,2));
    a3 = -2/(std::pow(trajDuration_,3));

    s = a3*std::pow(t,3) + a2*std::pow(t,2) + a1*t + a0;
    s_dot = 3*a3*std::pow(t,2) + 2*a2*t +a1;    //Parabolic Velocity
        Profile
    s_ddot = 6*a3*t +2*a2;  //Linear Acceleration Profile
}
```

As before, the equations (1), (2) and (3), become:

$$s(t) = a_3t^3 + a_2t^2 + a_1t + a_0 \tag{6}$$

$$\dot{s}(t) = 3a_3t^2 + 2a_2t + a_1 \tag{7}$$

$$\ddot{s}(t) = 6a_3t + 2a_2 \tag{8}$$

with the coefficients computed in the same way it was written before

$$a_0 = s_0$$

$$a_1 = \dot{s}_0$$

$$a_3 t_f^3 + a_2 t_f^2 + a_1 t_f + a_0 = s_f$$

$$3a_3 t_f^2 + 2a_2 t_f + a_1 = \dot{s}_f$$

obtaining

$$a_0 = 0$$

$$a_1 = 0$$

$$a_2 = \frac{3}{t_f^2}$$

$$a_3 = -\frac{2}{t_f^3}$$

After that, with time derivation, it possible to compute $\dot{s}(t)$ and $\ddot{s}(t)$

$$\dot{s} = 3a_3 t^2 + 2a_2 t + a_1$$

$$\ddot{s} = 6a_3 t + 2a_2$$

Trajectory Computation

In this chapter, the computation of both **Circular** and **Linear Trajectories** is addressed.

Circular Trajectories

First, let's define two new constructors: one for **Circular Trajectory with Trapezoidal Velocity Profile** and one for **Circular Trajectory with Cubic Polynomial**. So, in *kdl_planner.h*

```
//Constructor for Circular Trajectory with Trapezoidal Velocity
    Profile
// _trajDuration = Final Time
// _accDuration = Acceleration Time
// _trajInit = Trajectory Starting Point
// _trajRadius = Circumference Radius
KDLPlanner(double _trajDuration, double _accDuration, Eigen::Vector3d
     _trajInit, double _trajRadius);

//Constructor for Circular Trajectory with Cubic Polynomial
// _trajDuration = Final Time
// _trajInit = Trajectory Starting Point
// _trajRadius = Circumference Radius
KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double
    _trajRadius);
```

In *kdl_planner.cpp*

```
//Constructor for Circular Trajectory with Trapezoidal Velocity
    Profile
KDLPlanner::KDLPlanner(double _trajDuration, double _accDuration,
    Eigen::Vector3d _trajInit, double _trajRadius)
{
     trajDuration_ = _trajDuration;
     accDuration_ = _accDuration;
```

```
    trajInit_ = _trajInit;
    trajRadius_ = _trajRadius;
}

//Constructor for Circular Trajectory with Cubic Polynomial
KDLPlanner::KDLPlanner(double _trajDuration, Eigen::Vector3d
   _trajInit, double _trajRadius)
{
    trajDuration_ = _trajDuration;
    trajInit_ = _trajInit;
    trajRadius_ = _trajRadius;
}
```

As written in the comments of the provided code, _trajDuration is the final time of the trajectory, _accDuration is the acceleration time, _trajInit is the trajectory starting point and _trajRadius is the circumference radius. For the constructor referred to the cubic polynomial, obviously, the acceleration time is not defined.

The specifications required that the center of the trajectory be in the vertical plane containing the end-effector. Because of this, the center of the trajectory is given by adding, to the y-component of the initial point, the radius. The equations used to describe a circular path are:

$$x = x_i$$

$$y = y_i - rcos(2\pi s)$$

$$z = z_i - rsin(2\pi s)$$

Deriving:

$$\dot{x} = 0$$

$$\dot{y} = 2\pi r\dot{s}sin(2\pi s)$$

$$\dot{z} = -2\pi r\dot{s}cos(2\pi s)$$

$$\ddot{x} = 0$$

$$\ddot{y} = 2\pi r\ddot{s}sin(2\pi s) + 4\pi^2 r\dot{s}^2 cos(2\pi s)$$

$$\ddot{z} = -2\pi r\ddot{s}cos(2\pi s) + 4\pi^2 r\dot{s}^2 sin(2\pi s)$$

Directly in the function **KDLPlanner::compute_trajectory** already provided, the circular positional path as function $s(t)$ was created. To avoid redundancy, the **compute_trajectory** function will be shown only after introducing the Linear Trajectory as well.

Linear Trajectories

The same has been done for the Linear Trajectories. First, let's define two new constructors: one for **Linear Trajectory with Trapezoidal Velocity Profile** and one for **Linear Trajectory with Cubic Polynomial**. So, in *kdl_planner.h*

```
//Constructor for Linear Trajectory with Trapezoidal Velocity Profile
// _trajDuration = Final Time
// _accDuration = Acceleration Time
// _trajInit = Trajectory Starting Point
// _trajEnd = Trajectory Final Point
KDLPlanner(double _trajDuration, double _accDuration, Eigen::Vector3d
    _trajInit, Eigen::Vector3d _trajEnd);

//Constructor for Linear Trajectory with Cubic Polynomial
// _trajDuration = Final Time
// _trajInit = Trajectory Starting Point
// _trajEnd = Trajectory Final Point
KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, Eigen::
   Vector3d _trajEnd);
```

In *kdl_planner.cpp*

```
//Constructor for Linear Trajectory with Trapezoidal Velocity Profile
KDLPlanner::KDLPlanner(double _trajDuration, double _accDuration,
   Eigen::Vector3d _trajInit, Eigen::Vector3d _trajEnd)
{
    trajDuration_ = _trajDuration;
```

```
    accDuration_ = _accDuration;
    trajInit_ = _trajInit;
    trajEnd_ = _trajEnd;
}

//Constructor for Linear Trajectory with Cubic Polynomial
KDLPlanner::KDLPlanner(double _trajDuration, Eigen::Vector3d
   _trajInit, Eigen::Vector3d _trajEnd)
{
    trajDuration_ = _trajDuration;
    trajInit_ = _trajInit;
    trajEnd_ = _trajEnd;
}
```

Here, there is no more _trajRadius but there is _trajEnd, which is the trajectory final point. The equation used were

$$p(s) = p_i + s(p_f - p_i)$$

$$\dot{p}(s) = \dot{s}(p_f - p_i)$$

$$\ddot{p}(s) = \ddot{s}(p_f - p_i)$$

compute_trajectory

In this section, the implementation of the total function **compute_trajectory** will be shown. In fact, all the four trajectories implemented have been included in a single function. The choice of the trajectory is uniquely determined by the definition of the **relevant parameters** (as highlighted by the comments in the file **ros2_kdl_node.cpp**). SO, it is possible to implement four types of trajectories:

1. **Linear Trajectory with Trapezoidal Velocity Profile** (you have to define traj_duration, acc_duration, init_position, end_position)
2. **Linear Trajectory with Cubic Polynomial**(you have to define traj_duration, init_position, end_position)

3. **Circular Trajectory with Trapezoidal Velocity Profile** (you have to define traj_duration, acc_duration, init_position, radius)

4. **Circular Trajectory with Cubic Polynomial** (you have to define traj_duration, init_position, radius)

Specifically, if you want a Linear Trajectory it is necessary to impose radius = 0 and !=0 otherwise. Instead, if you want a Trapezoidal Velocity Profile, it is necessary to have the parameter acc_duration != 0. If you want a Cubic Polynomial acc_duration=0 is needed. So, to decide what type of trajectory the robot should do, the values of the parameters **radius** and **acc_duration** is fundamental. All the trajectories have been tested with a **Position Controller** defining, each time

```
////////////////////// TESTING TRAJECOTRIES ////////////////////////

//Test Linear Trajectory with Trapezoidal Velocity Profile
//planner_ = KDLPlanner(traj_duration, acc_duration, init_position,
    end_position);

//Test Linear Trajectory with Cubic Polynomial
//planner_ = KDLPlanner(traj_duration, init_position, end_position);

//Test Circular Trajectory with Trapezoidal Velocity Profile
//planner_ = KDLPlanner(traj_duration, acc_duration, init_position,
    radius);

//Test Circular Trajectory with Cubic Polynomial
//planner_ = KDLPlanner(traj_duration, init_position, radius);
```

with, for example, this definition of the parameters:

```
//double traj_duration = 1.5, acc_duration = 0.5, t = 0.0; double
    radius = 0.0;

//double traj_duration = 1.5, acc_duration = 0.0, t = 0.0; double
    radius = 0.0;

//double traj_duration = 1.5, acc_duration = 0.5, t = 0.0; double
    radius = 0.1;

//double traj_duration = 1.5, acc_duration = 0.0, t = 0.0; double
    radius = 0.1;
```

To determine which trajectory to execute, and therefore which constructor to

call, this simple control mechanism has been added in **ros2_kdl_node.cpp** after parameters definition

```
////////////////////// TRAJECTORY SELECTION ////////////////////////
if(radius == 0 && acc_duration != 0)
{
    planner_ = KDLPlanner(traj_duration, acc_duration, init_position,
        end_position);
    std::cout<< "The trajectory chosen is: Linear Trajectory with
       Trapezoidal Velocity Profile \n";
}
else if(radius == 0 && acc_duration == 0)
{
    planner_ = KDLPlanner(traj_duration, init_position, end_position)
        ;
    std::cout<< "The trajectory chosen is: Linear Trajectory with
       Cubic Polynomial \n";
}
else if(radius != 0 && acc_duration != 0)
{
    planner_ = KDLPlanner(traj_duration, acc_duration, init_position,
        radius);
    std::cout<< "The trajectory chosen is: Circular Trajectory with
       Trapezoidal Velocity Profile \n";
}
else if(radius != 0 && acc_duration == 0)
{
    planner_ = KDLPlanner(traj_duration, init_position, radius);
    std::cout<< "The trajectory chosen is: Circular Trajectory with
       Cubic Polynomial \n";
}
else
{
    std::cout<<"Not a valid trajectory chosen for you robot \n";
}
```

At the end, the final **compute_trajectory** will be

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

    if((trajRadius_ == 0 && accDuration_ != 0) || (trajRadius_ == 0
       && accDuration_ == 0))    //Linear Trajectory
    {
        if(trajRadius_ == 0 && accDuration_ != 0)
        {
            trapezoidal_vel(time, accDuration_, s, s_dot, s_ddot);
        }
        else
        {
            cubic_polinomial(time, s, s_dot, s_ddot);
        }
```

```
        traj.pos = trajInit_ + s*(trajEnd_-trajInit_);
        traj.vel = s_dot * (trajEnd_-trajInit_);
        traj.acc = s_ddot * (trajEnd_-trajInit_);
    }

    else if((trajRadius_ != 0 && accDuration_ != 0) || (trajRadius_
        != 0 && accDuration_ == 0)) //Circular Trajectory
    {
        if(trajRadius_ != 0 && accDuration_ != 0)
        {
            trapezoidal_vel(time, accDuration_, s, s_dot, s_ddot);
        }
        else
        {
            cubic_polinomial(time, s, s_dot, s_ddot);
        }
        //Circular Trajectory with the Center of the Trajectory in
           the Vertical Plane Containing the End-Effector
        //The center of the Trajectory is given by adding to the y-
           component of the initial point the radius

        //Position
        traj.pos.x() = trajInit_.x();
        traj.pos.y() = trajInit_.y() + trajRadius_ - trajRadius_*cos
           (2*M_PI*s);
        traj.pos.z() = trajInit_.z() - trajRadius_*sin(2*M_PI*s);

        //Velocity
        traj.vel.x() = 0;
        traj.vel.y() = 2*M_PI*trajRadius_*s_dot*sin(2*M_PI*s);
        traj.vel.z() = -2*M_PI*trajRadius_*s_dot*cos(2*M_PI*s);

        //Acceleration
        traj.acc.x() = 0;
        traj.acc.y() = 2*M_PI*trajRadius_*s_ddot*sin(2*M_PI*s) + 4*
           std::pow(M_PI,2)*trajRadius_*std::pow(s_dot,2)*cos(2*M_PI*
           s);
        traj.acc.z() = -2*M_PI*trajRadius_*s_ddot*cos(2*M_PI*s) + 4*
           std::pow(M_PI,2)*trajRadius_*std::pow(s_dot,2)*sin(2*M_PI*
           s);
    }
    else
    {
        std::cout<< "The trajectory is not well defined. Modify
           planner_ in ros2_kdl_node.cpp and try again.";
    }

    return traj;

}
```

To run all with a **Position Controller** and a **Position Interface**, let's send the following instructions

```
colcon build
source install/local_setup.bash
ros2 launch iiwa_bringup iiwa.launch.py
```

and, in another terminal

```
ros2 run ros2_kdl_package ros2_kdl_node
```

It is possible to do the same with a **Velocity Controller** and a **Velocity Interface** substituting the previous two instructions with

```
ros2 launch iiwa_bringup iiwa.launch.py
command_interface:="velocity"
robot_controller:="velocity_controller"
```

and

```
$ ros2 run ros2_kdl_package ros2_kdl_node
--ros-args -p cmd_interface:=velocity
```

Let's show some images

Figure 1: Position Control Start

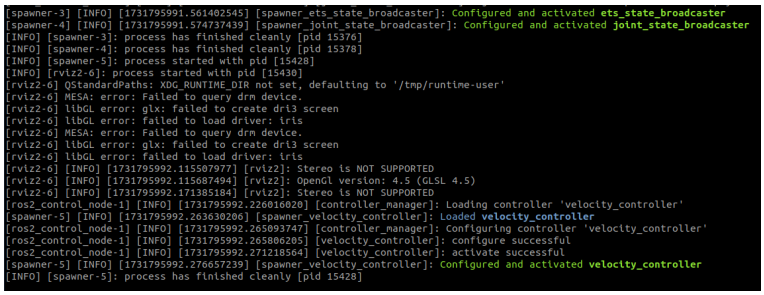

Figure 2: Velocity Control Start

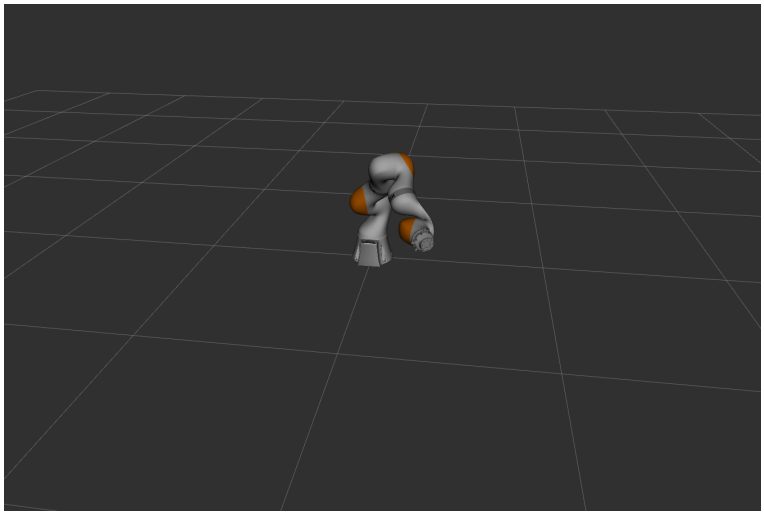

Figure 3: Robot in Rviz2

Here, the videos of the performing Linear and Circular Trajectories with Cubic Polynomial. The trajectories with the Trapezoidal Velocity Profile will not be shown because they have practically the same behaviors. The real testing will be presented in the next chapter with the **Effort Control**.

- Linear Trajectory with Position Control:
  `https://github.com/EmmanuelPat6/Homework_2/blob/main/Videos/Linear_Trajectory_Cubic_Position.webm`
- Circular Trajectory with Position Control:
  `https://github.com/EmmanuelPat6/Homework_2/blob/main/Videos/Circular_Trajectory_Cubic_Position.webm`

It is necessary to download the files to visualize them clicking to **View raw**.

Trajectory Testing

In this chapter the trajectories implemented in the previous one will be tested with the provided **Inverse Dynamics Control in the Joint Space**. To make life easier, a world in Gazebo with zero gravity has been used in order to compensate the gravity itself. For this reason, in the controller it is no more necessary to explicitly have the **Gravity Compensation Term** of the **PD+ Controller**.

Testing

At this point, it is possible to test all the trajectories:

- Linear Trajectory with Trapezoidal Velocity Profile
- Linear Trajectory with Cubic Polynomial
- Circular Trajectory with Trapezoidal Velocity Profile
- Circular Trajectory with Cubic Polynomial

In order to show it, some links will be put at the end of this chapter in order to visualize each trajectory. To do that, an **effort_interface** and an **effort_controller** have been added in such a way to be able to send torque commands to the manipulator. In **iiwa_controllers.yaml**

```
effort_controller:
```

```
    type: effort_controllers/JointGroupEffortController
...
...

effort_controller:
  ros__parameters:
    joints:
      - joint_a1
      - joint_a2
      - joint_a3
      - joint_a4
      - joint_a5
      - joint_a6
      - joint_a7
```

Furthermore, in **ros2_kdl_node.cpp**, a very important part added to compute the effort commands was

```
else if(cmd_interface_ == "effort"){


    Vector6d cartvel; cartvel << p.vel + 5*error, o_error;
    //Vector6d cartacc; cartacc << p.acc + 5*error+ 2*dot_error,
        o_dot_error;
    KDL::JntArray velocity_temp;

    velocity_temp = joint_velocities_;

    //joint_accelerations_.data = pseudoinverse(robot_->getEEJacobian
        ().data)*(cartacc - robot_->getEEJacDotqDot());

    joint_velocities_.data = pseudoinverse(robot_->getEEJacobian().
        data)*(cartvel);

    joint_positions_.data = joint_positions_.data + joint_velocities_
        .data*dt;

    joint_accelerations_.data = (joint_velocities_.data -
        velocity_temp.data)/dt;

    joint_efforts_.data = controller_.idCntr(joint_positions_,
        joint_velocities_, joint_accelerations_, 20, 5);
```

where the joint accelerations and the joint positions are computed respectively by integrating and differentiating. Because of that, the publishing period, for the publisher of the effort commands, was decreased

```
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(20),
```

```
                    std::bind(&Iiwa_pub_sub::cmd_publisher, this));

    // Send joint effort commands
    for (long int i = 0; i < joint_efforts_.data.size(); ++i) {
        desired_commands_[i] = joint_efforts_(i);
    }


}
```

For this, different parameters must be identified for the trajectories computed in this way. In order to avoid these problems, separated variables have been created.

```
// Plan trajectory
double acc_duration;
double t = 0.0;
double radius = 0.1;
double traj_duration;

if(cmd_interface_ == "position" || cmd_interface_ == "velocity"){
    traj_duration = 1.5;
    acc_duration = 0.0; //If you want a Trapezoidal Velocity Profile,
        please se this quantity to 0.7
}
else if(cmd_interface_ == "effort"){
    traj_duration = 6.0; //Longer because of faster implementation
    acc_duration = 0.0;  //If you want a Trapezoidal Velocity Profile
        with an Inverse Dynamics Control in the JOINT SPACE, please
       set this quantity to 3.5 //Instead if you it with an Inverse
       Dynamics Control in the OPERATIONAL SPACE, please set this
       quantity to 1.0 for Linear Trajectory and 2.5 for Circular
       Trajectory
}


...
...

double total_time;
int trajectory_len;
//Define Trajectory
if(cmd_interface_ == "position" || cmd_interface_ == "velocity"){
    total_time = 1.5;
    trajectory_len = 150;
}
else if(cmd_interface_ == "effort"){
    total_time = 6.0;    //Longer than the previous ones because of
       fast implementation
    trajectory_len = 600;
}
```

Remember that the **Dynamic Model** is

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F\dot{q} + g(q) = \tau \tag{9}$$

The formula used for this implementation were

$$u = B(q)(y + K_d\dot{e} + K_p e) + C(q,\dot{q})\dot{q} + g(q) \tag{10}$$

where, as I said before, $g(q)$ was not considered due to the fact that the world used is with **Zero Gravity** and

$$y = \ddot{q}$$

The function that implements this is

```
KDLController::KDLController(KDLRobot &_robot)
{
    robot_ = &_robot;
}

Eigen::VectorXd KDLController::idCntr(KDL::JntArray &_qd,
                                      KDL::JntArray &_dqd,
                                      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 = _dqd.data - dq;

    Eigen::VectorXd ddqd = _ddqd.data;
    return robot_->getJsim() * (ddqd + _Kd*de + _Kp*e)
            + robot_->getCoriolis() /*+ robot_->getGravity()*/ /*
               friction compensation?*/;
```

To run all with an **Effort Controller** and an **Effort Interface**, let's send the following instructions

```
colcon build

source install/local_setup.bash
```

```
ros2 launch iiwa_bringup iiwa.launch.py
command_interface:="effort" robot_controller:="effort_controller"
```

and, in another terminal

```
ros2 run ros2_kdl_package ros2_kdl_node --ros-args -p
cmd_interface:=effort
```

To see how the robot moves according to each trajectory, click on these links of my GitHub repository:

- Linear Trajectory with Trapezoidal Velocity Profile: `https://github.com/EmmanuelPat6/Homework_2/blob/main/Videos/Linear_Trajectory_Trapezoidal_Effort_Joint_Space.webm`
- Linear Trajectory with Cubic Polynomial: `https://github.com/EmmanuelPat6/Homework_2/blob/main/Videos/Linear_Trajectory_Cubic_Effort_Joint_Space.webm`
- Circular Trajectory with Trapezoidal Velocity Profile: `https://github.com/EmmanuelPat6/Homework_2/blob/main/Videos/Circular_Trajectory_Trapezoidal_Effort_Joint_Space.webm`
- Circular Trajectory with Cubic Polynomial: `https://github.com/EmmanuelPat6/Homework_2/blob/main/Videos/Circular_Trajectory_Cubic_Effort_Joint_Space.webm`

It is necessary to download the files to visualize them (click on **View raw**). To allow the manipulator to remain in the same position after the trajectory is completed, the following line of code has been added

```
...
//In the else after the if in which there is the command sending
if(cmd_interface_ == "effort"){
        //Send joint velocity commands
        for (long int i = 0; i < joint_efforts_.data.size(); ++i) {
```

```
            desired_commands_[i] = 0.0;
        }
    }
```

The desired commands are 0.0 because there is no gravity. If gravity had been present, these values would have needed to correspond to some values such that the gravity would have been compensated. Note that the robot tends to move very slowly. This is always due to the strange behavior caused by the lack of gravity. This behavior would likely not occur with a real robot because of phenomena due to the friction and to the inertia of the manipulator itself.

BE CAREFUL: because of some problems due to the Zero Gravity and the Effort Controller, to do all in the right way it is necessary to run Gazebo and to press very quickly on the Play Button in the bottom left. Furthermore, when a Linear Trajectory with Cubic Polynomial is desired, it is advisable to set the offset to 0.15 instead of 0.10 on the z-axis.

Torques

It is possible to plot the torques sent to the manipulator in such a way to tune appropriately the control gains **Kp** and **Kd**. In order to do this it is possible to use **rqt_plot** to visualize the torques at each run. Some screenshot have been saved in such a way to show the behavior. It is necessary to open **rqt**, in another terminal, with the very simple instruction

```
rqt
```

and to follow the passage shown in the figure below.

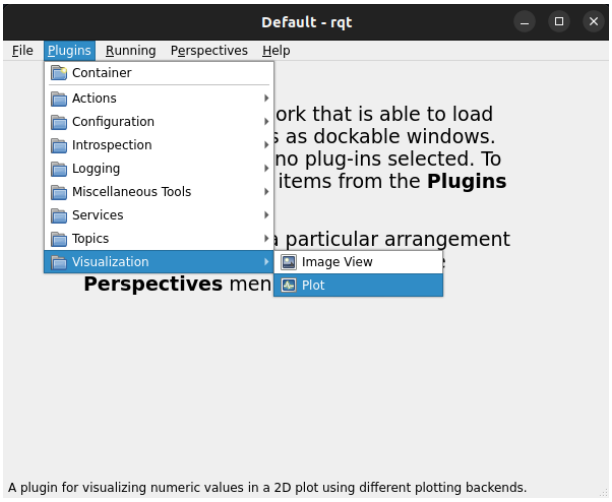

Figure 4: rqt_plot

At this point, all that remains is to insert the topics from which to extract information over time. These topics can only be specified once Gazebo has been opened with the appropriate **Effort Controller**.

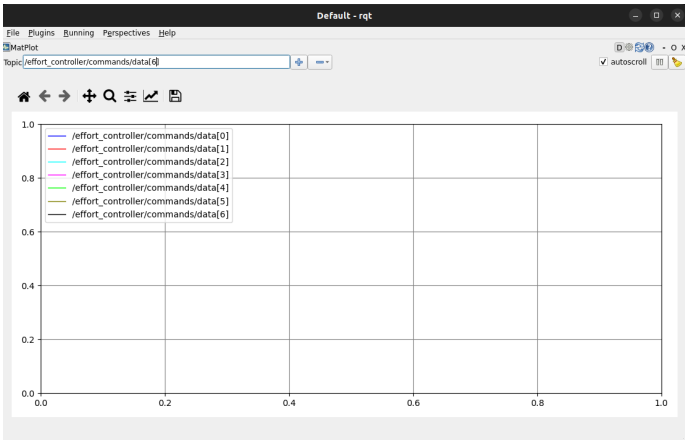

Figure 5: effort_controllers/commands/data in rqt_plot

Some screenshots will be shown below. Notice that the values shown are very zoomed in. For a more general visualization see the next section.

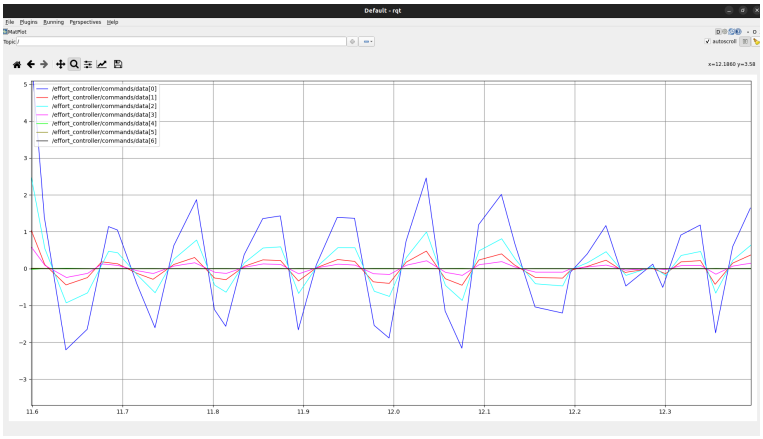

Figure 6: Torques sent to the manipulator with Linear Trajectory and Trapezoidal Velocity Profile

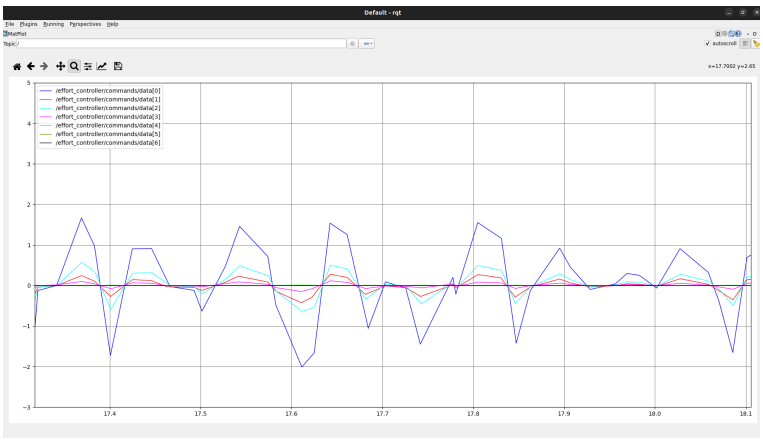

Figure 7: Torques sent to the manipulator with Linear Trajectory and Cubic Polynomial

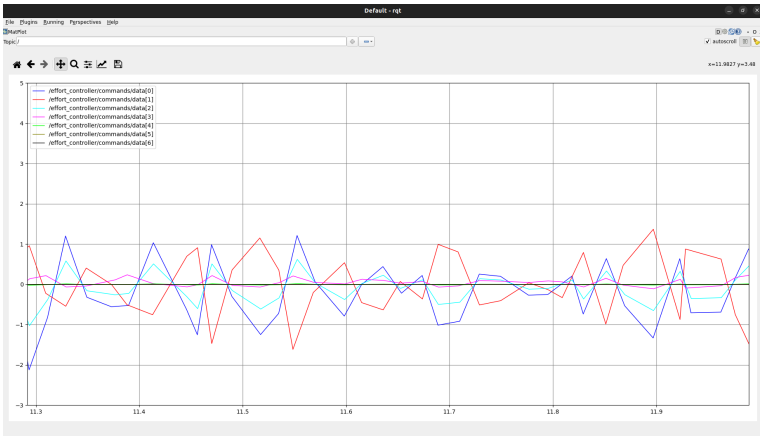

Figure 8: Torques sent to the manipulator with Circular Trajectory and Trapezoidal Velocity Profile

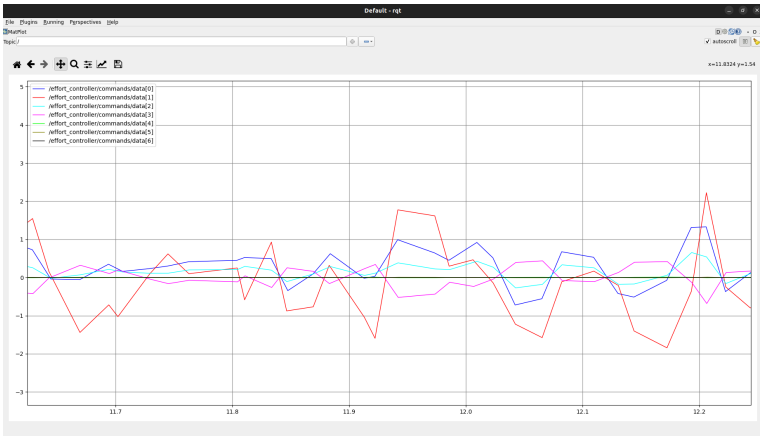

Figure 9: Torques sent to the manipulator with Circular Trajectory and Cubic Polynomial

As can be easily observed from the graphs above, the main difference between the Linear and Circular Trajectories lies in which torque command is greater for a specific joint. This is plausible since, given the completely different trajectories, the stresses and the joints that need to be actuated at a given moment are entirely different. Another point to note concerns the magnitude of the torques. Since the robot was slowed down by increasing the trajectory duration to better observe its behavior, the torques are not very high. For a robot of this size, these results are reasonable, especially considering the absence of the Gravity Compensation Term for the reasons extensively discussed earlier.

MATLAB

It is possible to save the simulation data in a **bag file**, allowing the torques to be plotted in **MATLAB**, making the plot more visible and easier to understand. Look at the active topics after the robot spawn in Gazebo

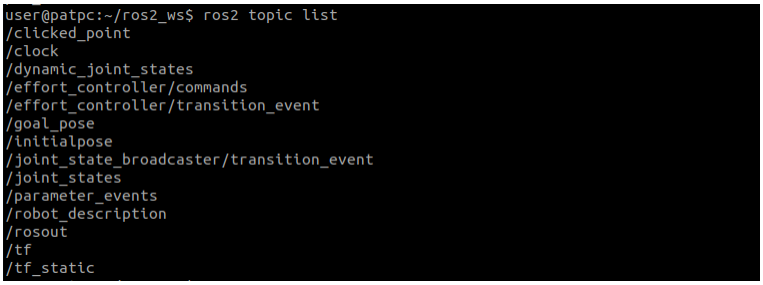

Figure 10: Topic List

Obviously, the topic I need to record is /**effort_controller/commands**. In this way, all the 7 torques sent to the manipulator will be recording in a **.db3** file. To do this, run

```
cd src
ros2 bag record /effort_controller/commands
```

A directory will be created in your folder and it is necessary to compress it in a **.zip** file in such a way to import it on MATLAB. To do this, once the zip file is loaded into the MATLAB directory, it is necessary to follow these simple steps to extract the data from the file itself and separate the various components of the seven commands (this is included in the **bag** folder in the GitHub Repository)

```
unzip("file.zip_name");
folderPath = fullfile(pwd,"folder_name");    %Folder Unzipped
bagReader = ros2bagreader(folderPath);
baginfo = ros2("bag","info",folderPath)
msgs = readMessages(bagReader);

efforts = [];
for i = 1:length(msgs)
efforts = [efforts; msgs{i}.data'];
end

t_start = 0;        % Initial Time for Plotting
num_points = #;  % Number of Values (for this, look at the number of
    the rows of the created variable 'efforts' and insert here)
t_end = num_points*0.02;        % Final Time for Plotting (20 ms
    because of the publisher)

% Time Array
time_vector = linspace(t_start, t_end, num_points);

% Timeseries Object
```

```
efforts_ts = timeseries(efforts, time_vector);

% Plotting
figure;
plot(efforts_ts.Time, efforts_ts.Data);
xlabel('Time (s)');
ylabel('Nm');
title('Efforts');
grid on;
```

Here an example on how the **MATLAB Command Window** is after these instructions

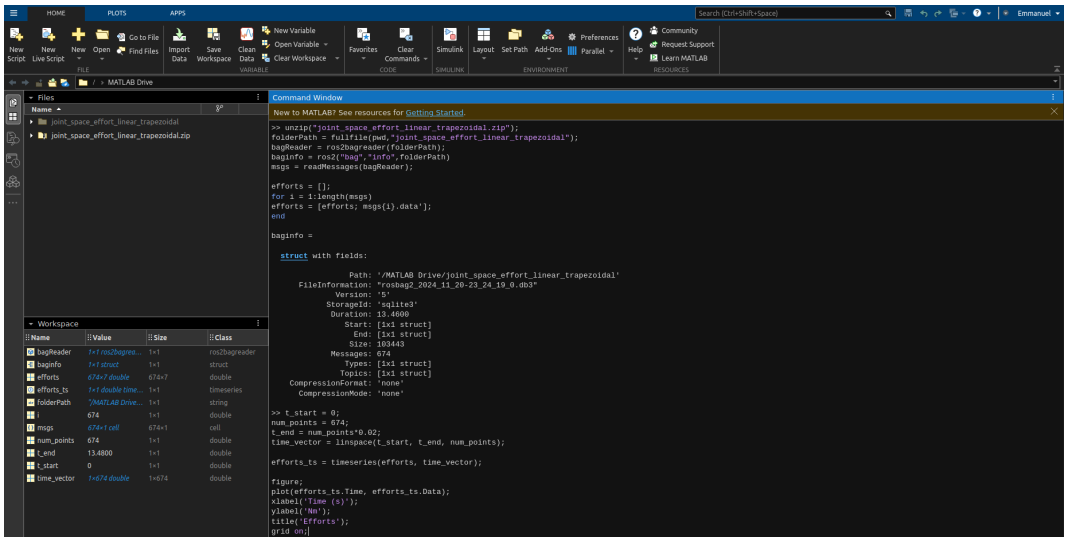

Figure 11: MATLAB Command Window

Let's plot all the bags file.

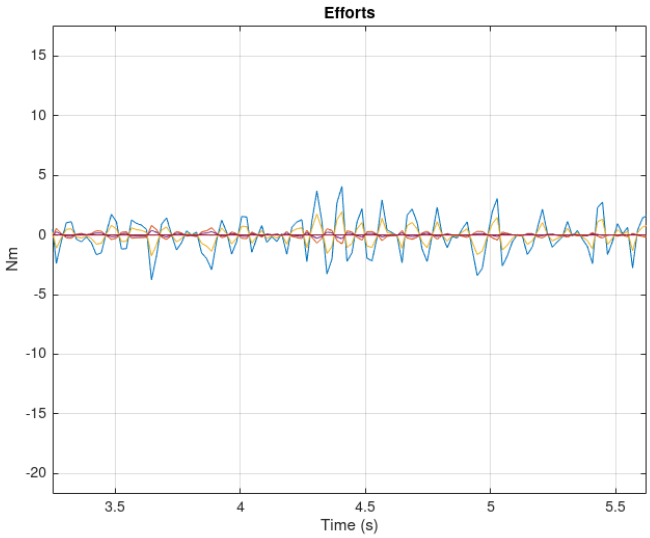

Figure 12: Torques sent to the manipulator with Linear Trajectory and Trapezoidal Velocity Profile MATLAB

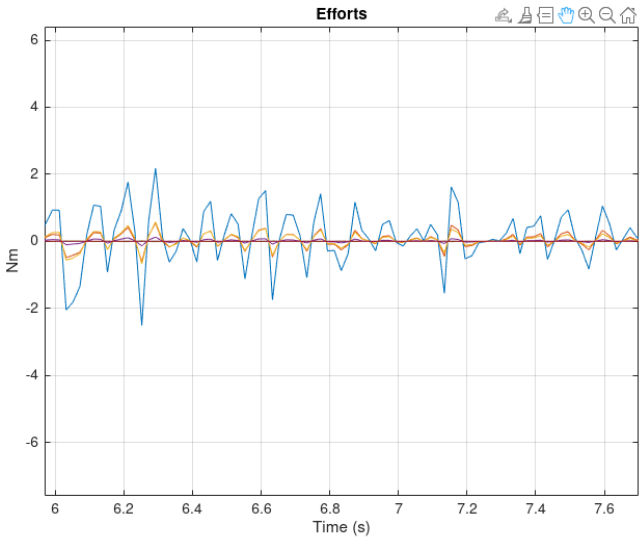

Figure 13: Torques sent to the manipulator with Linear Trajectory and Cubic Polynomial MATLAB

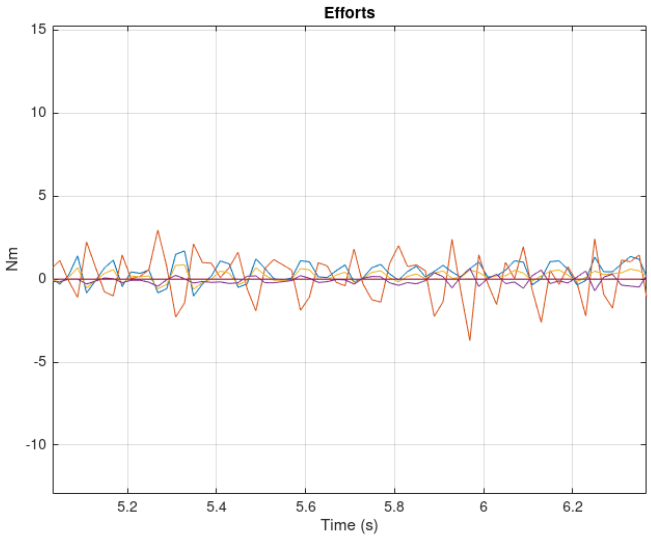

Figure 14: Torques sent to the manipulator with Circular Trajectory and Trapezoidal Velocity Profile MATLAB

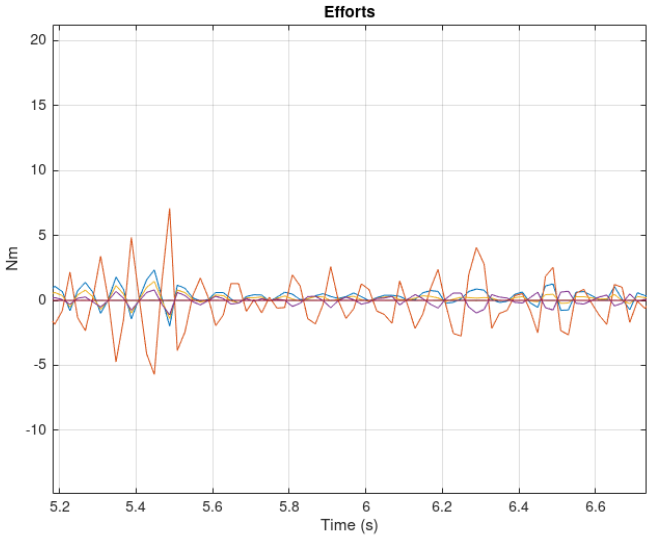

Figure 15: Torques sent to the manipulator with Circular Trajectory and Cubic Polynomial MATLAB

Inverse Dynamics Operational Space Controller

In this chapter, the trajectories, like in the previous one, are tested with another type of control: the **Inverse Dynamics Control in the Operational Space**. This control, differently from the Joint Space Inverse Dynamics Controller, computes the errors in **Cartesian Space**. In fact, usually, the motion is specified in terms of **Operational Space Variables** because this is simpler for us to identify them. Also here, the world in Gazebo with zero gravity has been used in order to compensate the gravity itself. For this reason, in the controller it is no more necessary to explicitly have the **Gravity Compensation Term** of the **PD+ Controller** as before.

Control

Into the **kdl_control.cpp** file, the **KDLController::idCntr** function was overlayed to implement the **Inverse Dynamics Operational Space Controller**. To make life easier, the implementation with **Geometric Jacobian** has been done, thanks to the **Angle-Axis** convention. Furthermore, at the end of this chapter, a trial version with the **Analytical Jacobian** has been implemented. It does not work for now. I will try to implement it in the future. Let's see the implementation with the Geometric one

```
Eigen::VectorXd KDLController::idCntr(KDL::Frame &_desPos,
                                      KDL::Twist &_desVel,
                                      KDL::Twist &_desAcc,
                                      double _Kpp, double _Kpo,
                                      double _Kdp, double _Kdo)
{

    // Compute Gain Matrices
    //_Kpp for position; _Kpo for orientation; _Kdp for velocity;
       _Kdo for ang_velocity
    //Angle-Axis Representation in such a way to use the Geometric
       Jacobian instead of the Analytical Jacobian
    Eigen::Matrix<double,6,6> Kp, Kd;
    Kp=Eigen::MatrixXd::Zero(6,6);
    Kd=Eigen::MatrixXd::Zero(6,6);
    Kp.block(0,0,3,3) = _Kpp*Eigen::Matrix3d::Identity();
    Kp.block(3,3,3,3) = _Kpo*Eigen::Matrix3d::Identity();
    Kd.block(0,0,3,3) = _Kdp*Eigen::Matrix3d::Identity();
    Kd.block(3,3,3,3) = _Kdo*Eigen::Matrix3d::Identity();

    // Read the Current State
    Eigen::Matrix<double,6,7> J = robot_->getEEJacobian().data;  //
       Jacobian
    Eigen::Matrix<double,7,7> I = Eigen::Matrix<double,7,7>::Identity
       ();    //Identity Matrix
    Eigen::Matrix<double,7,7> B = robot_->getJsim();    //Inertia
       Matrix
    //Eigen::Matrix<double,7,6> Jpinv = pseudoinverse(J);
    Eigen::Matrix<double,7,6> Jpinv = weightedPseudoInverse(B,J);
    /*
    The weighted pseudo-inverse allows calculating a least-squares
       solution that accounts
    for the robot's inertia. In other words, the resulting solution
       minimizes the robot's kinetic energy.
    */

    //Position
    Eigen::Vector3d pd(_desPos.p.data);
    Eigen::Vector3d pe(robot_->getEEFrame().p.data);
    Eigen::Matrix<double,3,3,Eigen::RowMajor> R_d(_desPos.M.data);
    Eigen::Matrix<double,3,3,Eigen::RowMajor> R_e(robot_->getEEFrame
       ().M.data);
    R_d = matrixOrthonormalization(R_d);
    R_e = matrixOrthonormalization(R_e);

    //Velocity
    Eigen::Vector3d dot_pd(_desVel.vel.data);
    Eigen::Vector3d dot_pe(robot_->getEEVelocity().vel.data);
    Eigen::Vector3d omega_d(_desVel.rot.data);
    Eigen::Vector3d omega_e(robot_->getEEVelocity().rot.data);

    //Acceleration
    Eigen::Matrix<double,6,1> dotdot_xd;
    Eigen::Matrix<double,3,1> dotdot_pd(_desAcc.vel.data);
    Eigen::Matrix<double,3,1> dotdot_rd(_desAcc.rot.data);
```

```
    //Linear Errors
    Eigen::Matrix<double,3,1> e_p = computeLinearError(pd,pe);
    Eigen::Matrix<double,3,1> dot_e_p = computeLinearError(dot_pd,
        dot_pe);

    //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);

    Eigen::Matrix<double,6,1> x_tilde;
    Eigen::Matrix<double,6,1> dotx_tilde;
    x_tilde << e_p, e_o;
    dotx_tilde << dot_e_p, dot_e_o;
    dotdot_xd << dotdot_pd, dotdot_rd;

   //Inverse Dynamics in the Operational Space

    Eigen::Matrix<double,6,1> y;

    Eigen::VectorXd Jdot_qdot=robot_->getEEJacDotqDot();

    //y << Jpinv * (dotdot_xd + Kd*dotx_tilde + Kp*x_tilde -
        Jdot_qdot);

    y << dotdot_xd - Jdot_qdot + Kd*dotx_tilde + Kp*x_tilde;

    return B * (Jpinv*y + (I-Jpinv*J)*(- 1*robot_->getJntVelocities()
        ))
             /*+ robot_->getGravity()*/ + robot_->getCoriolis();

    //return B*y + /*robot_->getGravity()*/ + robot_->getCoriolis();


}
```

The formula used were, starting from the **Dynamic Model** (9):

$$B(q)\ddot{q} + n(q,\dot{q}) = u \tag{11}$$

with $n(q,\dot{q})$ that contains all the Non-Linear Terms

$$y = \ddot{q} \tag{12}$$

Here, y has a different expression and it is to be designed so as to yeld tracking

of a trajectory specified by $x_d(t)$. So, in this case, the second-order equation:

$$\ddot{x}_e = J_A(q)\ddot{q} + \dot{J}_A(q,\dot{q})\dot{q} \tag{13}$$

suggests the choice

$$y = J_A^{\dagger}(q)(\ddot{x}_d + K_D\dot{\tilde{x}} + Kp\tilde{x} - \dot{J}_A(q,\dot{q})\dot{q}) \tag{14}$$

Also a term in the Null-Space was added. At the end, the **efforts commands** sent to the manipulator have been computed as

$$\tau = By + n \tag{15}$$

Due to the higher speed exhibited by the robot during the simulation with this type of control, it is necessary to reduce certain parameters, such as **acceleration** for the trapezoidal profile, in order to appreciate its movement. Instead, only for the **Linear Trajectory with Cubic Polynomial** it is necessary to increase a little bit more the **trajectory duration** (from 6 to 10). All the advisable parameters choices are written clearly in the code

```
// Plan trajectory
double acc_duration;
double t = 0.0;
double radius = 0.0;
double traj_duration;

if(cmd_interface_ == "position" || cmd_interface_ == "velocity"){
    traj_duration = 1.5;
    acc_duration = 0.0; //If you want a Trapezoidal Velocity Profile,
        please se this quantity to 0.7
}
else if(cmd_interface_ == "effort"){
    traj_duration = 6.0; //Longer because of faster implementation.
       For all the trajectories set this to 6. Only for the Linear
       Trajectory with Cubic Polynomial in the case of the
    //Inverse Dynamics Control in the Operational Space set this to
       10
    acc_duration = 3.5;  //If you want a Trapezoidal Velocity Profile
        with an Inverse Dynamics Control in the JOINT SPACE, please
```

```
        set this quantity to 3.5
    //Instead if you it with an Inverse Dynamics Control in the
        OPERATIONAL SPACE, please set this quantity to 1.0 for Linear
        Trajectory and 2.5 for Circular Trajectory
}
```

To start this controller it is necesasry to comment out the previous one and uncomment this

```
else if(cmd_interface_ == "effort"){

/*
    Vector6d cartvel; cartvel << p.vel + 5*error, o_error;
    //Vector6d cartacc; cartacc << p.acc + 5*error+ 2*dot_error,
        o_dot_error;
    KDL::JntArray velocity_temp;

    velocity_temp = joint_velocities_;

    //joint_accelerations_.data = pseudoinverse(robot_->getEEJacobian
        ().data)*(cartacc - robot_->getEEJacDotqDot());

    joint_velocities_.data = pseudoinverse(robot_->getEEJacobian().
        data)*(cartvel);

    joint_positions_.data = joint_positions_.data + joint_velocities_
        .data*dt;

    joint_accelerations_.data = (joint_velocities_.data -
        velocity_temp.data)/dt;

    joint_efforts_.data = controller_.idCntr(joint_positions_,
        joint_velocities_, joint_accelerations_, 40, 20);
*/
    joint_efforts_.data = controller_.idCntr(cartpos, des_vel,
        des_acc, 150, 70, 50, 40);

}
```

It is important to comment out all the instructions as shown in the previous code and not only the one that calls the controller(important!).

Results

A testing for all the 4 trajectories has been done also for this controller. Here the videos

- Linear Trajectory with Trapezoidal Velocity Profile: `https://github.`

com/EmmanuelPat6/Homework_2/blob/main/Videos/Linear_Trajectory_Trapezoidal_Effort_Operational_Space.webm

- Linear Trajectory with Cubic Polynomial: https://github.com/EmmanuelPat6/Homework_2/blob/main/Videos/Linear_Trajectory_Cubic_Effort_Operational_Space.webm
- Circular Trajectory with Trapezoidal Velocity Profile: https://github.com/EmmanuelPat6/Homework_2/blob/main/Videos/Circular_Trajectory_Trapezoidal_Effort_Operational_Space.webm
- Circular Trajectory with Cubic Polynomial: https://github.com/EmmanuelPat6/Homework_2/blob/main/Videos/Circular_Trajectory_Cubic_Effort_Operational_Space.webm

It is necessary to download the files to visualize them (click on **View raw**). Doing the same steps before, let's plot both on **rqt_plot** and **MATLAB** the torques sent to the manipulator.

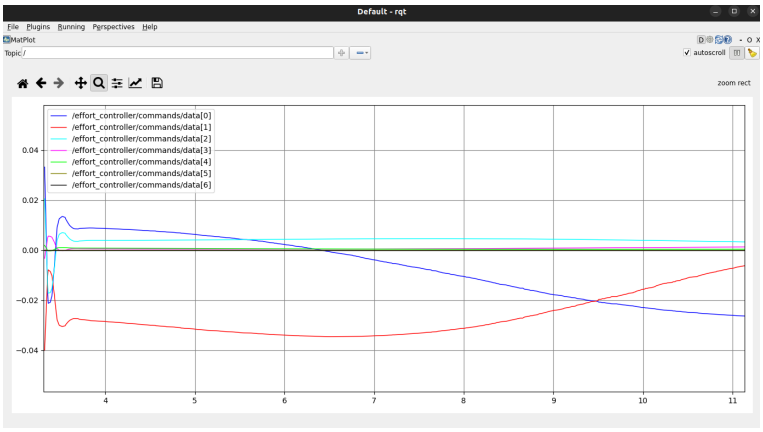

Figure 16: Torques sent to the manipulator with Linear Trajectory and Trapezoidal Velocity Profile

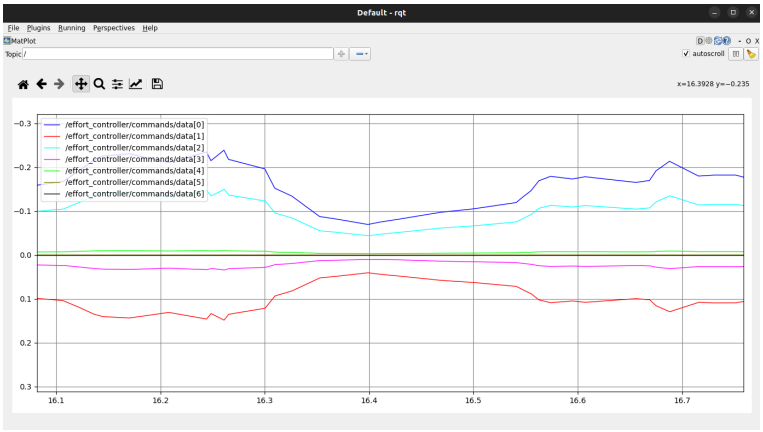

Figure 17: Torques sent to the manipulator with Linear Trajectory and Cubic Polynomial

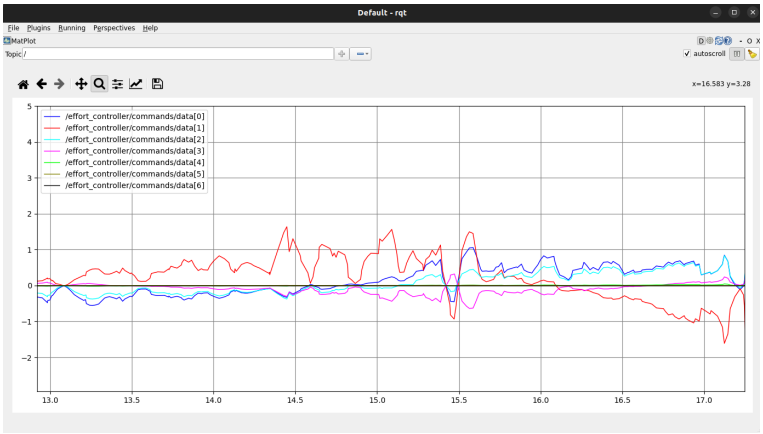

Figure 18: Torques sent to the manipulator with Circular Trajectory and Trapezoidal Velocity Profile

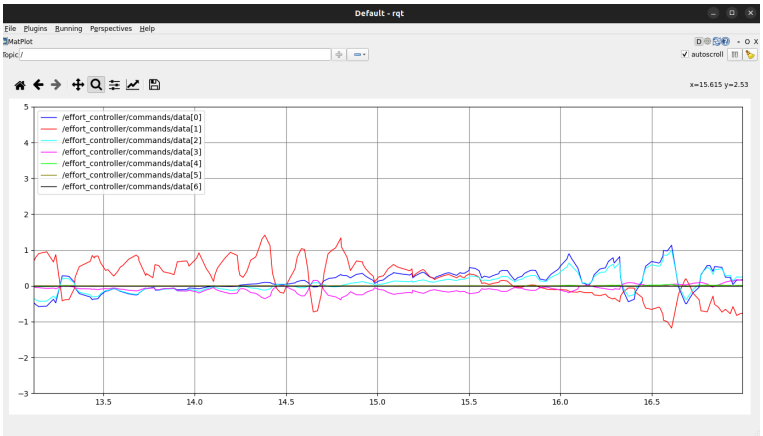

Figure 19: Torques sent to the manipulator with Circular Trajectory and Cubic Polynomial

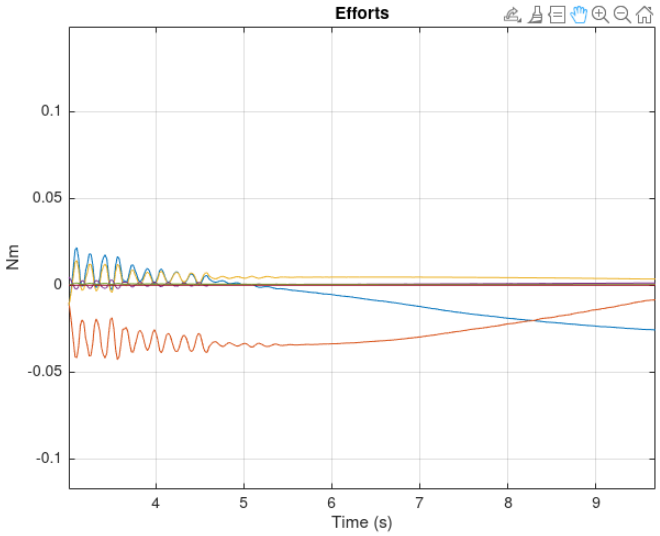

Figure 20: Torques sent to the manipulator with Linear Trajectory and Trapezoidal Velocity Profile MATLAB

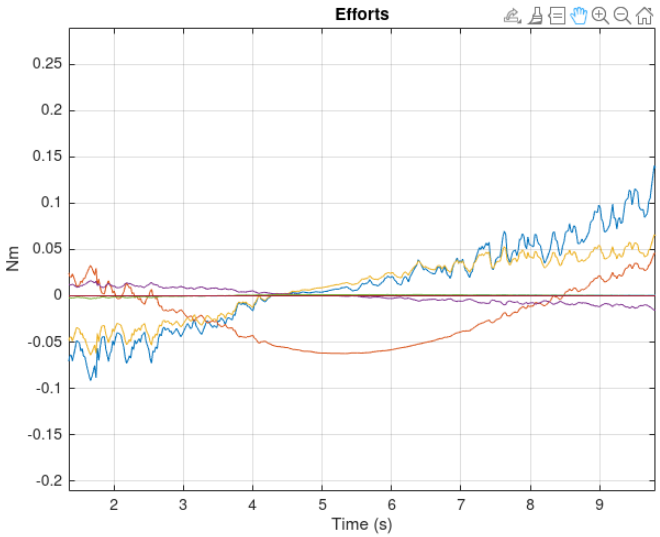

Figure 21: Torques sent to the manipulator with Linear Trajectory and Cubic Polynomial MATLAB

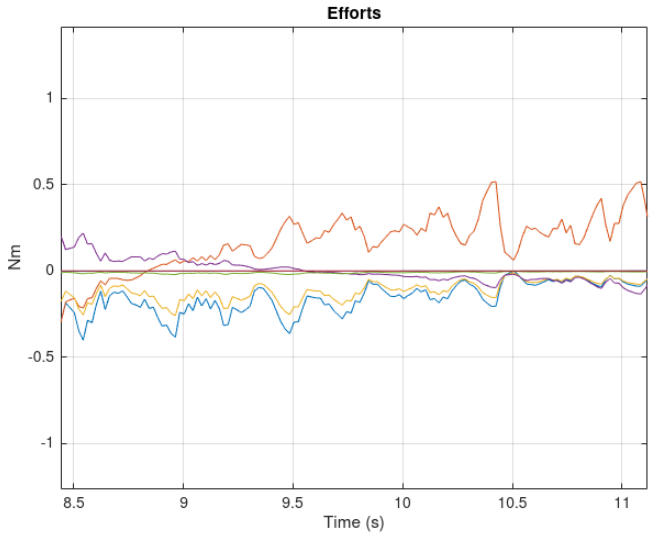

Figure 22: Torques sent to the manipulator with Circular Trajectory and Trapezoidal Velocity Profile MATLAB

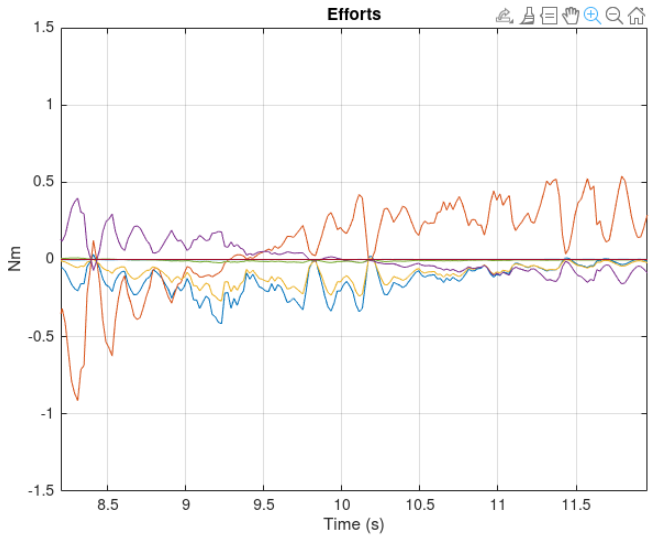

Figure 23: Torques sent to the manipulator with Circular Trajectory and Cubic Polynomial MATLAB

From these figures, it is possible to notice how with the **Inverse Dynamics Operational Space Control** the torques sent to the manipulator are more homogeneous and do not show all the peaks seen in the previous chapter.

Future Works

Analytical Jacobian

As highlighted at the beginning of the chapter, a solution with the **Analytical Jacobian** was tested. This will be revisited in the future in order to make this control method functional as well. Here, the drafts of the given solution will be shown.

```
    inline Eigen::Matrix<double,3,1> computeEulerAngles(const Eigen::
       Matrix<double,3,3> &_R)
{
    Eigen::Matrix<double,3,1> euler;

    double r13=_R(0,2);
    double r23=_R(1,2);
    double r31=_R(2,0);
    double r32=_R(2,1);
    double r33=_R(2,2);

    double phi=atan2(r23,r13);
    double theta=atan2(sqrt(r13*r13+r23*r23),r33);
    double psi=atan2(r32,-r31);

    euler(0,0)=phi;
    euler(1,0)=theta;
    euler(2,0)=psi;

    return euler;
}

inline Eigen::Matrix<double,3,1> computeOrientationErrorEuler(const
    Eigen::Matrix<double,3,3> &_R_d,
     const Eigen::Matrix<double,3,3> &_R_e)
{
    Eigen::Matrix<double,3,1> e_phi;

    //Desired Angles
    Eigen::Matrix<double,3,1> phi_d=computeEulerAngles(_R_d);
    //Effective Angles
    Eigen::Matrix<double,3,1> phi_e=computeEulerAngles(_R_e);
    e_phi=phi_d-phi_e;

    return e_phi;
}

inline Eigen::Matrix<double,3,3> computeT(const Eigen::Matrix<double
    ,3,1> &euler){
    double phi = euler(0,0);
    double theta = euler(1,0);
    Eigen::Matrix<double,3,3> T;
```

```
    T(0,0) = 0;
    T(0,1) = -sin(phi);
    T(0,2) = cos(phi)*sin(theta);
    T(1,0) = 0;
    T(1,1) = cos(phi);
    T(1,2) = sin(phi)*sin(theta);
    T(2,0)= 1;
    T(2,1)= 0;
    T(2,2)= cos(theta);

    return T;
}

inline Eigen::Matrix<double,3,3> computeTdot(const Eigen::Matrix<
    double,3,1> &euler){
    double phi = euler(0,0);
    double theta = euler(1,0);
    Eigen::Matrix<double,3,3> Tdot;

    Tdot(0,0) = 0;
    Tdot(0,1) = -cos(phi);
    Tdot(0,2) = cos(phi)*cos(theta)-sin(phi)*sin(theta);
    Tdot(1,0) = 0;
    Tdot(1,1) = -sin(phi);
    Tdot(1,2) = sin(phi)*cos(theta)+cos(phi)*sin(theta);
    Tdot(2,0)= 0;
    Tdot(2,1)= 0;
    Tdot(2,2)= -sin(theta);

    return Tdot;
}

inline  Eigen::Matrix<double,6,7> AnalitycalJacobian(const Eigen::
    Matrix<double,6,7> &J,const Eigen::Matrix<double,3,1> &euler){
    Eigen::Matrix<double,6,6> TA;
    TA.block(0,0,3,3) = Eigen::Matrix3d::Identity();
    TA.block(3,3,3,3) = computeT(euler);

    return TA.inverse()*J;
}

inline Eigen::Matrix<double,3,1> computeOrientationVelocityErrorEuler
    (const Eigen::Matrix<double,3,1> &_omega_d,
    const Eigen::Matrix<double,3,1> &_omega_e,
    const Eigen::Matrix<double,3,3> &_R_d,
    const Eigen::Matrix<double,3,3> &_R_e)
{
    //DESIDERED ANGLES
    Eigen::Matrix<double,3,1> phi_d=computeEulerAngles(_R_d);
    //EFFECTIVE ANGLES
    Eigen::Matrix<double,3,1> phi_e=computeEulerAngles(_R_e);
    Eigen::Matrix<double,3,3> Te=computeT(phi_e);
    Eigen::Matrix<double,3,3> Td=computeT(phi_d);
    Eigen::Matrix<double,3,1> dphi_d=Td.inverse()*_omega_d;
    Eigen::Matrix<double,3,1> dphi_e=Te.inverse()*_omega_e;
```

```
    return dphi_d-dphi_e;
}
```

Now the implementation to add at the one chosen in the previous section

```
//Analitycal Jacobian
Eigen::Matrix<double,3,1> euler_angles = computeEulerAngles(R_e);
Eigen::Matrix<double,6,7> JA = AnalitycalJacobian(J,euler_angles);
Eigen::Matrix<double,7,6> JApinv = weightedPseudoInverse(B,JA);

// Compute TA_inv
Eigen::Matrix<double,6,6> TA;
TA.block(0,0,3,3) = Eigen::Matrix3d::Identity();
TA.block(3,3,3,3) = computeT(euler_angles);
Eigen::Matrix<double,6,6> TA_inv = TA.inverse();

// Compute TA_dot
Eigen::Matrix<double,6,6> TA_dot;
TA_dot.block(0,0,3,3) = Eigen::Matrix3d::Identity();
TA_dot.block(3,3,3,3) = computeTdot(euler_angles);
Eigen::Matrix<double,6,7> Jdot = robot_->getEEJacDot().data;
// Compute JA_dot
Eigen::Matrix<double,6,7> JAdot = TA_inv*(Jdot - TA_dot*JA);
```