Report – Homework 2

Students:

Luca Marseglia

https://github.com/marseluca/homework2rl

Benito Vodola

https://github.com/BenitoVodola/Homework2_KDL.git

Domenico Tuccillo

https://github.com/DomenicoTuccillo/homework2.git

Debora Ippolito

https://github.com/DeboraIppolito/kdl_robot

1. Substitute the current trapezoidal velocity profile with a cubic polynomial linear trajectory

a) First, 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, ṡ and s̈ that represent the curvilinear abscissa of your trajectory.

A new method “trapezoidal_vel” for the KDLPlanner class has been defined, with the following signature:

```
void trapezoidal_vel(double time, double &s, double &dots,double &ddots);
```

That uses the private variables:

- trajDuration_ (tf)
- accDuration_ (tc)
- trajInit_ (starting point)
- trajEnd_ (final point)

Of the same class, defined as follows:

```
private:

    KDL::Path_RoundedComposite* path_;
    KDL::Path_Circle* path_circle_;
    KDL::VelocityProfile* velpref_;
    KDL::Trajectory* traject_;


    ////////////////////////////////
    double trajDuration_, accDuration_;
    Eigen::Vector3d trajInit_, trajEnd_;
    trajectory_point p;
```

Such variables are initialized with the constructor as follows:

```
KDLPlanner::KDLPlanner(double _trajDuration, double _accDuration, Eigen::Vector3d _trajInit, Eigen::Vector3d _trajEnd)
{
    trajDuration_ = _trajDuration;
    accDuration_ = _accDuration;
    trajInit_ = _trajInit;
    trajEnd_ = _trajEnd;
}
```

While trajectory_point is a struct in which have been defined the following field:

```
struct trajectory_point{
  Eigen::Vector3d pos = Eigen::Vector3d::Zero();
  Eigen::Vector3d vel = Eigen::Vector3d::Zero();
  Eigen::Vector3d acc = Eigen::Vector3d::Zero();
};
```

This function implements the following logic for the computation of s:

$$s(t) = \begin{cases} s_i + \frac{1}{2}\ddot{s}_c t^2 & 0 \le t \le t_c \\ s_i + \ddot{s}_c t_c \left(t - \frac{t_c}{2}\right) & t_c < t \le t_f - t_c \\ s_f - \frac{1}{2}\ddot{s}_c \left(t_f - t\right)^2 & t_f - t_c < t \le t_f \end{cases}$$

S_dot and s_dot_dot have been obtained deriving s.

The definition of the “trapezoidal_vel” method is as follows:

```
void KDLPlanner::trapezoidal_vel(double time, double &s, double &dots,double &ddots)
{

  double si=0;
  double sf=1;

  double ddot_traj_c = -1.0/(std::pow(accDuration_,2)-trajDuration_*accDuration_)*(sf-si);

  if(time <= accDuration_)
  {
    s = si + 0.5*ddot_traj_c*std::pow(time,2);
    dots = ddot_traj_c*time;
    ddots = ddot_traj_c;
  }
  else if(time <= trajDuration_-accDuration_)
  {
    s = si + ddot_traj_c*accDuration_*(time-accDuration_/2);
    dots = ddot_traj_c*accDuration_;
    ddots = 0;
  }
  else
  {
    s = sf - 0.5*ddot_traj_c*std::pow(trajDuration_-time,2);
    dots = ddot_traj_c*(trajDuration_-time);
    ddots = -ddot_traj_c;
  }

}
```

Where ddot_traj_c is the acceleration at the end of the linear part of velocity.

b) Create a function named KDLPlanner::cubic_polinomial that creates the cubic polynomial curvilinear abscissa for your trajectory. The function takes as argument a double t representing time and returns three double variables: s, ṡ and s̈ that represent the curvilinear abscissa of your trajectory.

A new method “cubic_polinomial” for the KDLPlanner class has been defined, with the following signature:

```
void cubic_polinomial(double time, double &s, double &dots,double &ddots);
```

This function computes the coefficients of a cubic polynomial in the form:

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

Imposing the following boundary conditions:

- $s_i = s(0) = a_0$
- $\dot{s_i} = \dot{s}(0) = a_1$
- $s_f = a_3t_f^3 + a_2\,t_f^2 + a_1\,t_f + a_0$
- $\dot{s_f} = 3a_3\,t_f^2 + 2a_2\,t_f + a_1$

Then computes s_dot and s_dot_dot deriving s.

The definition of the “cubic_polinomial” method is as follows:

```
void KDLPlanner::cubic_polinomial(double time, double &s, double &dots,double &ddots)
{

  double si=0;
  double sf=1;
  double dsi=0;
  double dsf=0;
  //finding polinomial coefficients
  double a0=si;
  double a1=dsi;
  double a2=3/std::pow(trajDuration_,2);
  double a3=-2/(std::pow(trajDuration_,3));

  s=a3*std::pow(time,3)+a2*std::pow(time,2)+a1*time+a0;
  dots=3*a3*std::pow(time,2)+2*a2*time+a1;
  ddots=6*a3*time+2*a2;
}
```

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.

Defined a new private variable for the radius:

```
double trajRadius_;
```

And a new constructor for the KDLPlanner class to also initialize this new variable:

- In “kdl_planner.h”:

```
KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double _trajRadius);
```

- In “kdl_planner.cpp”:

```
// CIRCULAR TRAJECTORY CONSTRUCTOR DEFINITION

KDLPlanner::KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double _trajRadius);
{
    trajDuration_ = _trajDuration;
    trajInit_ = _trajInit;
    traRadius_ = _trajRadius;
}
```

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.

Defined the “path_primitive_circular” method to implement the execution of the circular trajectory in the y-z plane:

- In “kdl_planner.h”:

```
trajectory_point path_primitive_circular( double &s, double &dots,double &ddots);
```

- In “kdl_planner.cpp”:

```cpp
//funzione che calcola pos, vel e acc a partire da s, s_dot, s_ddot seguendo un tratto circolare
trajectory_point KDLPlanner::path_primitive_circular( double &s, double &dots,double &ddots){
  trajectory_point traj;
  Eigen::Vector3d pi = trajInit_;
  Eigen::Vector3d pf=trajEnd_;
  Eigen::Vector3d dif=pf-pi;

  // DEFINE  THE CENTER
  Eigen::Vector3d p0;
  p0[0] = pi[0]; // centro x
  p0[1] = pi[1]+trajRadius_; // centro y
  p0[2] = pi[2]; // centro z

  // POSIZIONE
  traj.pos[0] = p0[0]; // x
  traj.pos[1] = p0[1] - trajRadius_*cos(2*M_PI*s); // y
  traj.pos[2] = p0[2] - trajRadius_*sin(2*M_PI*s); // z

  // VELOCITA
  traj.vel[0] = 0;
  traj.vel[1] = trajRadius_*(2*M_PI)*dots*sin(2*M_PI*s);
  traj.vel[2] = -trajRadius_*(2*M_PI)*dots*cos(2*M_PI*s);

  // ACCELERAZIONE
  traj.acc[0] = 0;
  traj.acc[1] = trajRadius_*(2*M_PI)*(dots*dots*2*M_PI*cos(2*M_PI*s)+ddots*sin(2*M_PI*s));
  traj.acc[2] = -trajRadius_*(2*M_PI)*(-dots*dots*2*M_PI*sin(2*M_PI*s)+ddots*cos(2*M_PI*s));

  return traj;
}
```

The trajectory was implemented following these formulas describing a circular trajectory in the yz plane:

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

c) Do the same for the linear trajectory.

Defined the “path_primitive_linear” method to implement the execution of the linear trajectory:

- In “kdl_planner.h”:

```cpp
trajectory_point path_primitive_circular( double &s, double &dots,double &ddots);
```

- In “kdl_planner.cpp”:

```cpp
trajectory_point KDLPlanner::path_primitive_linear( double &s, double &dots,double &ddots){
  trajectory_point traj;
  Eigen::Vector3d pi=trajInit_;
  Eigen::Vector3d pf=trajEnd_;
  Eigen::Vector3d dif=pf-pi;
  traj.pos=(1-s)*pi+s*pf;
  traj.vel=(-pi+pf)*dots;
  traj.acc=(-pi+pf)*ddots;
  //std::cout<<"pos: "<<traj.pos[1]<<" "<<traj.pos[2]<<" "<<traj.pos[3] <<std::endl;
  return traj;
}
```

The function was implemented following this path primitive:

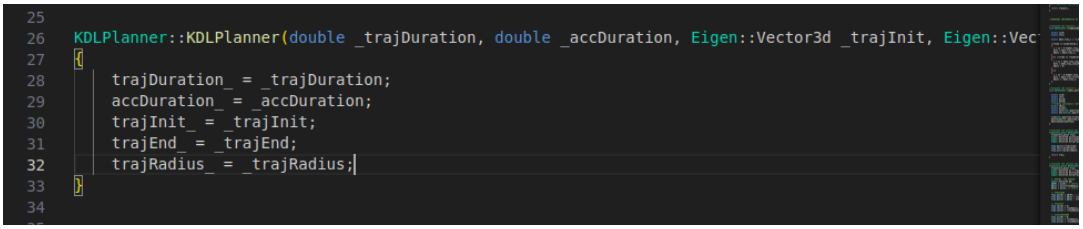
$$p(s) = (1 - s)p_i + sp_f$$

Which is well defined since

$$t = 0 => s = 0 => p(s) = p_i \; and \; t = t_f => s = 1 => p(s) = p_f.$$

NOTE:

A more general constructor was defined. This constructor initializes the variables for both the linear and the circular trajectories.



3. Test the four trajectories

a) At this point, you can create both linear and circular trajectories, each with trapezoidal velocity or 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.

We created a general function that takes as inputs the chosen path and profile and computes the trajectory.

```
// FUNZIONE GENERALE PER LA SCELTA DELLA COMBINAZIONE PATH/PROFILE
trajectory_point KDLPlanner::compute_trajectory(double time,std::string profile, std::string path)
{
  if(profile=="cubic"){
    if(path=="linear") return compute_cubic_linear(time);
    else return compute_cubic_circular(time);
  }else{
    if(path=="linear") return compute_trapezoidal_linear(time);
    else return compute_trapezoidal_circular(time);
  }

}
```

Then in the main, we extracted the position, velocity and acceleration of the computed desired trajectory:

```
//GENERAL CONSTRUCTOR
KDLPlanner planner(traj_duration, acc_duration, init_position, end_position,radius);
// Retrieve the first trajectory point
std::string profile="cubic";
std::string path="circular";
trajectory_point p = planner.compute_trajectory(t,profile,path);
```

The next step is to compute the inverse kinematics, in order to retrieve the joint positions and velocities necessary to follow the desired trajectory in the operation space.

To do that, we used the already implemented function: getInvKin which retrieves the joint position, then we also implemented a new function called getInvkinVel which derives the joint velocities. The acceleration has been kept to zero.

```cpp
//funzione che calcola l'inversione cinematica della velocita' (ricava q_dot a partire da v_e)
KDL::JntArray KDLRobot::getInvKinVel(const KDL::JntArray &qd,
                                     const KDL::Twist &eeFrameVel)
{
    KDL::JntArray jntArray_out_;
    jntArray_out_.resize(chain_.getNrOfJoints());
    int err = ikVelSol_->CartToJnt(qd, eeFrameVel, jntArray_out_);
    if (err != 0)
    {
        printf("inverse kinematics failed with error: %d \n", err);
    }
    return jntArray_out_;
}
```

In the end, the already implemented function controller_.idCntr has been used to test the trajectories with a controller in the joint space, after defining the gains Kp and Kd.

```cpp
// joint space inverse dynamics control
tau = controller_.idCntr(qd, dqd, ddqd, Kp, Kd);
```

b) Plot the torques sent to the manipulator and tune appropriately the control gains Kp and Kd until you reach a satisfactorily smooth behavior.

To plot the torques it is necessary to create some messages and publisher. Actually we also provided msgs for other usefull quantities like joint variables, velocities, errors and error norm.

```cpp
// Messages
std_msgs::Float64 tau1_msg, tau2_msg, tau3_msg, tau4_msg, tau5_msg, tau6_msg, tau7_msg;
std_msgs::Float64 qd1_msg, qd2_msg, qd3_msg, qd4_msg, qd5_msg, qd6_msg, qd7_msg;
std_msgs::Float64 dqd1_msg, dqd2_msg, dqd3_msg, dqd4_msg, dqd5_msg, dqd6_msg, dqd7_msg;
std_msgs::Float64 ddqd1_msg, ddqd2_msg, ddqd3_msg, ddqd4_msg, ddqd5_msg, ddqd6_msg, ddqd7_msg;
std_msgs::Float64 err1_msg, err2_msg, err3_msg, err4_msg, err5_msg, err6_msg, err7_msg;
std_msgs::Float64 norm_msg;
```

```
// Joints desired velocities
ros::Publisher joint1_dqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint1_desired_velocity", 1);
ros::Publisher joint2_dqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint2_desired_velocity", 1);
ros::Publisher joint3_dqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint3_desired_velocity", 1);
ros::Publisher joint4_dqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint4_desired_velocity", 1);
ros::Publisher joint5_dqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint5_desired_velocity", 1);
ros::Publisher joint6_dqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint6_desired_velocity", 1);
ros::Publisher joint7_dqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint7_desired_velocity", 1);

// Joints desired accelerations
ros::Publisher joint1_ddqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint1_desired_acceleration", 1);
ros::Publisher joint2_ddqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint2_desired_acceleration", 1);
ros::Publisher joint3_ddqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint3_desired_acceleration", 1);
ros::Publisher joint4_ddqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint4_desired_acceleration", 1);
ros::Publisher joint5_ddqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint5_desired_acceleration", 1);
ros::Publisher joint6_ddqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint6_desired_acceleration", 1);
ros::Publisher joint7_ddqd_pub = n.advertise<std_msgs::Float64>("/iiwa/joint7_desired_acceleration", 1);

// errors
ros::Publisher joint1_err = n.advertise<std_msgs::Float64>("/iiwa/joint1_err", 1);
ros::Publisher joint2_err = n.advertise<std_msgs::Float64>("/iiwa/joint2_err", 1);
ros::Publisher joint3_err = n.advertise<std_msgs::Float64>("/iiwa/joint3_err", 1);
ros::Publisher joint4_err = n.advertise<std_msgs::Float64>("/iiwa/joint4_err", 1);
ros::Publisher joint5_err = n.advertise<std_msgs::Float64>("/iiwa/joint5_err", 1);
ros::Publisher joint6_err = n.advertise<std_msgs::Float64>("/iiwa/joint6_err", 1);
ros::Publisher joint7_err = n.advertise<std_msgs::Float64>("/iiwa/joint7_err", 1);
 //norm error
ros::Publisher norm_err = n.advertise<std_msgs::Float64>("/iiwa/norm_error", 1);
```

```
Eigen::VectorXd errors =qd.data-robot.getJntValues();
// Set torques
tau1_msg.data = tau[0];
tau2_msg.data = tau[1];
tau3_msg.data = tau[2];
tau4_msg.data = tau[3];
tau5_msg.data = tau[4];
tau6_msg.data = tau[5];
tau7_msg.data = tau[6];

//creating message desired joints positions
qd1_msg.data=qd.data[0];
qd2_msg.data=qd.data[1];
qd3_msg.data=qd.data[2];
qd4_msg.data=qd.data[3];
qd5_msg.data=qd.data[4];
qd6_msg.data=qd.data[5];
qd7_msg.data=qd.data[6];

//creating message desired joints velocities
dqd1_msg.data=dqd.data[0];
dqd2_msg.data=dqd.data[1];
dqd3_msg.data=dqd.data[2];
dqd4_msg.data=dqd.data[3];
dqd5_msg.data=dqd.data[4];
dqd6_msg.data=dqd.data[5];
dqd7_msg.data=dqd.data[6];
```

```
//publishing desired joint positions
joint1_qd_pub.publish(qd1_msg);
joint2_qd_pub.publish(qd2_msg);
joint3_qd_pub.publish(qd3_msg);
joint4_qd_pub.publish(qd4_msg);
joint5_qd_pub.publish(qd5_msg);
joint6_qd_pub.publish(qd6_msg);
joint7_qd_pub.publish(qd7_msg);

// publishing desired joint velocities
joint1_dqd_pub.publish(dqd1_msg);
joint2_dqd_pub.publish(dqd2_msg);
joint3_dqd_pub.publish(dqd3_msg);
joint4_dqd_pub.publish(dqd4_msg);
joint5_dqd_pub.publish(dqd5_msg);
joint6_dqd_pub.publish(dqd6_msg);
joint7_dqd_pub.publish(dqd7_msg);

 // publishing desired joint acceleration
joint1_ddqd_pub.publish(ddqd1_msg);
joint2_ddqd_pub.publish(ddqd2_msg);
joint3_ddqd_pub.publish(ddqd3_msg);
joint4_ddqd_pub.publish(ddqd4_msg);
joint5_ddqd_pub.publish(ddqd5_msg);
joint6_ddqd_pub.publish(ddqd6_msg);
joint7_ddqd_pub.publish(ddqd7_msg);
 // publishing joint error
```

```
 // publishing joint error
joint1_err.publish(err1_msg);
joint2_err.publish(err2_msg);
joint3_err.publish(err3_msg);
joint4_err.publish(err4_msg);
joint5_err.publish(err5_msg);
joint6_err.publish(err6_msg);
joint7_err.publish(err7_msg);
//norm error
norm_err.publish(norm_msg);
```

Using the rqt_plot tool, it is possible to observe the torques computed by the controller. We tuned the parameters Kp and Kd with a trial-and-error approach for each case:

1) TRAPEZOIDAL LINEAR
2) TRAPEZOIDAL CIRCULAR

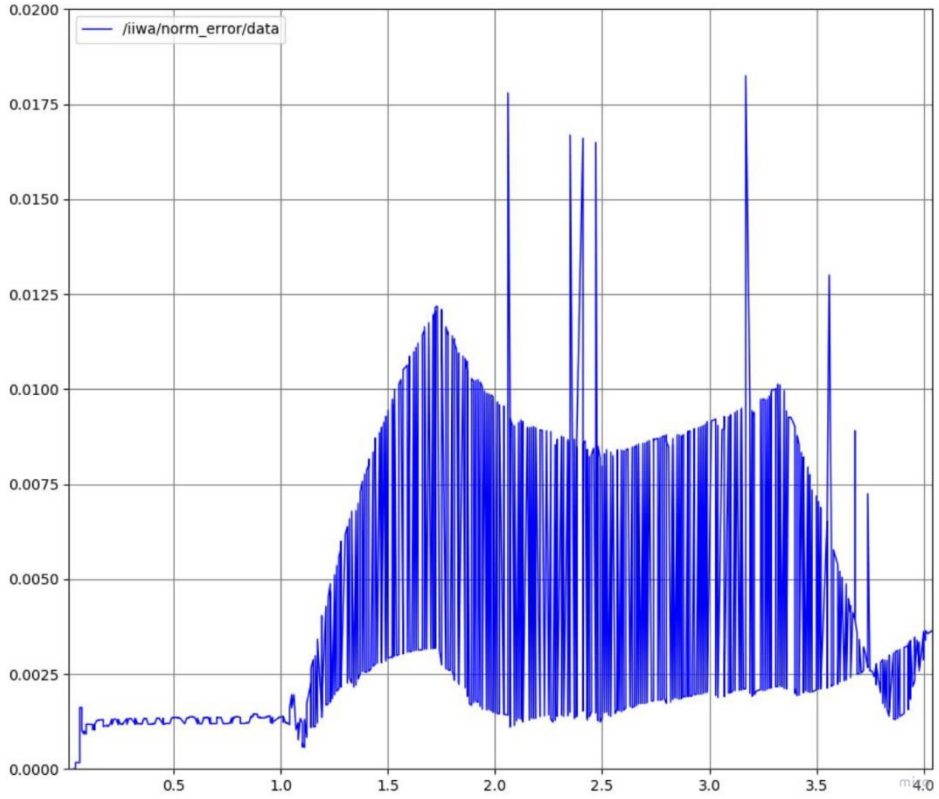
3) CUBIC LINEAR
4) CUBIC CIRCULAR

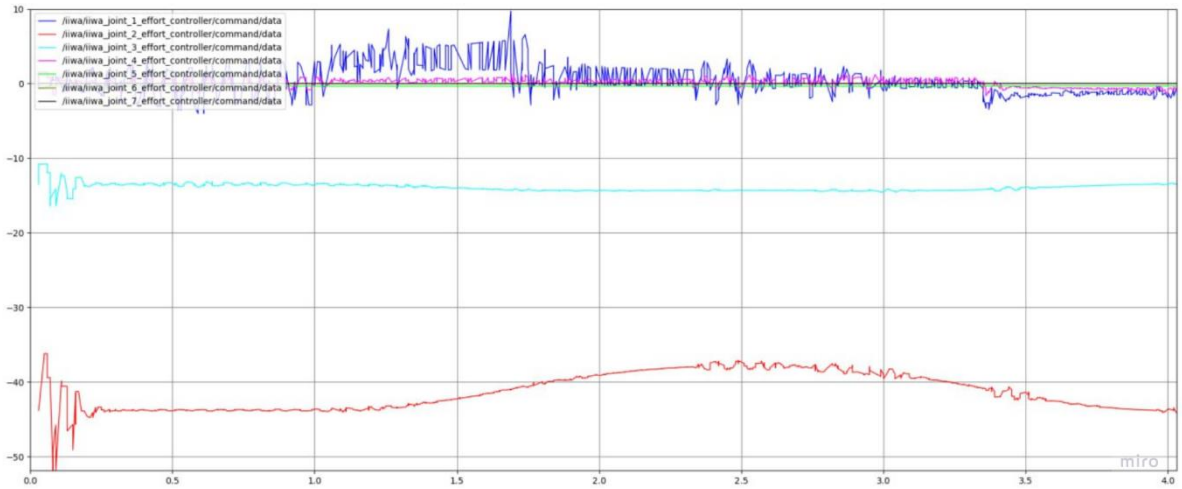
We tuned the gains to achieve the minimum error possible, but it's clear that the evolution of the torques values over time, is not very smooth. For this reason, we tuned the gains further to make the evolution of the torques values over time smoother. We observed that the price to pay is the increase of the error.

Trapezodial Linear

$$K_p = 180$$

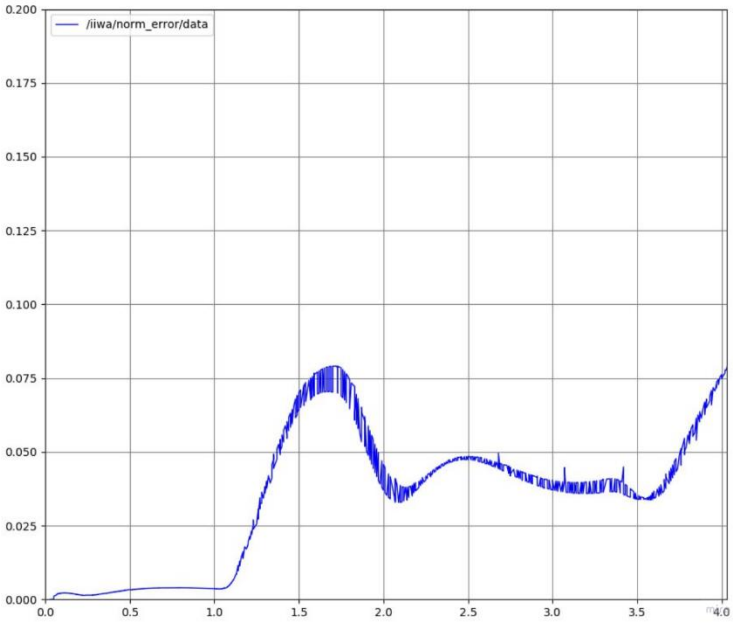
$$K_d = 80$$

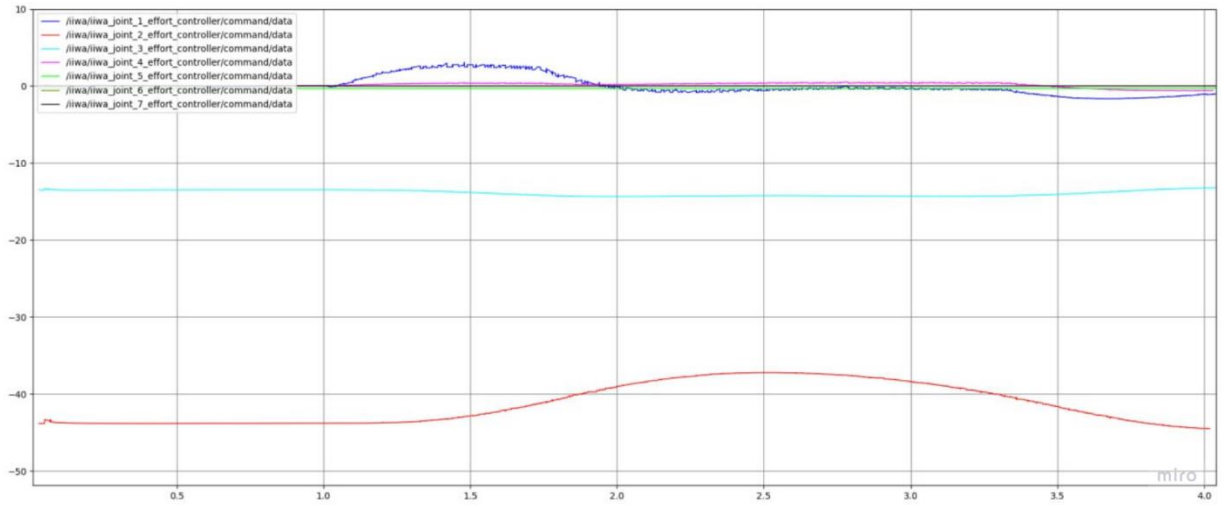




$$K_p = 20$$

$$K_d = 5$$

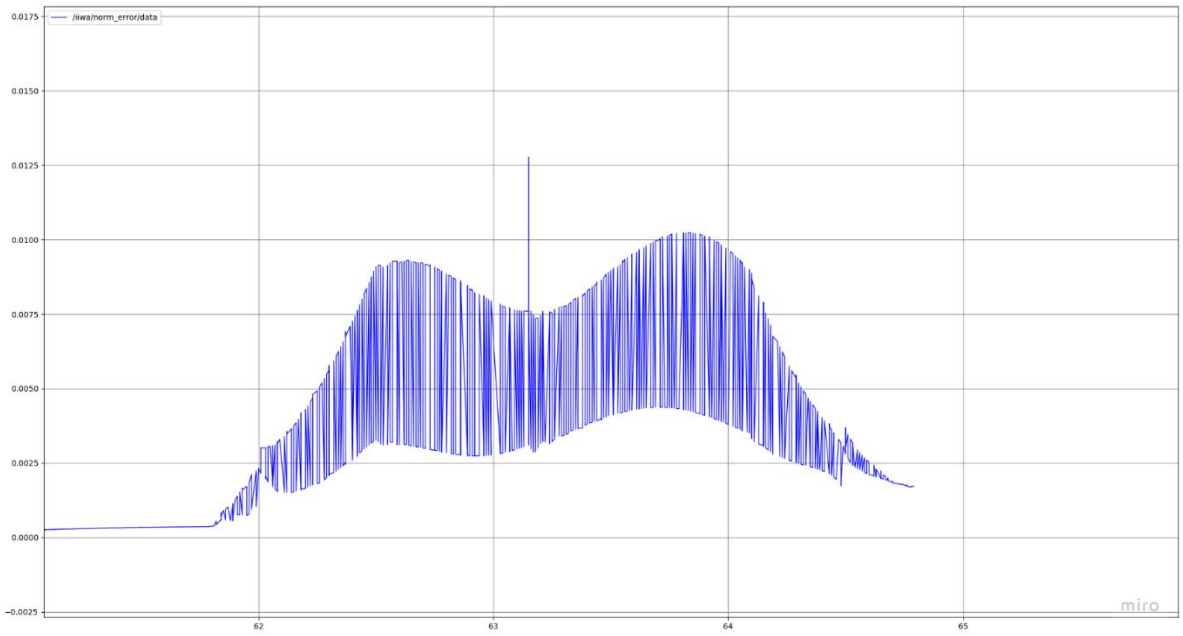

/iiwa/norm_error/data
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

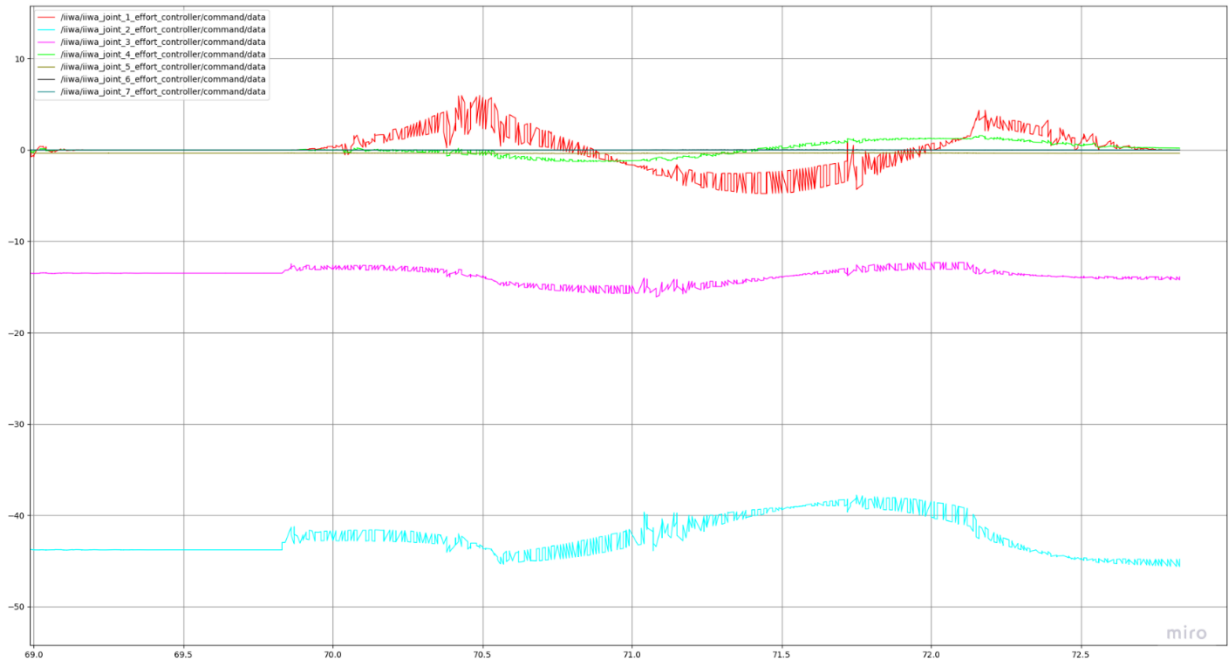

/iiwa/iiwa_joint_1_effort_controller/command/data
/iiwa/iiwa_joint_2_effort_controller/command/data
/iiwa/iiwa_joint_3_effort_controller/command/data
/iiwa/iiwa_joint_4_effort_controller/command/data
/iiwa/iiwa_joint_5_effort_controller/command/data
/iiwa/iiwa_joint_6_effort_controller/command/data
/iiwa/iiwa_joint_7_effort_controller/command/data
10
0
−10
−20
−30
−40
−50
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
miro

Trapezoidal Circular

$$K_p = 150$$

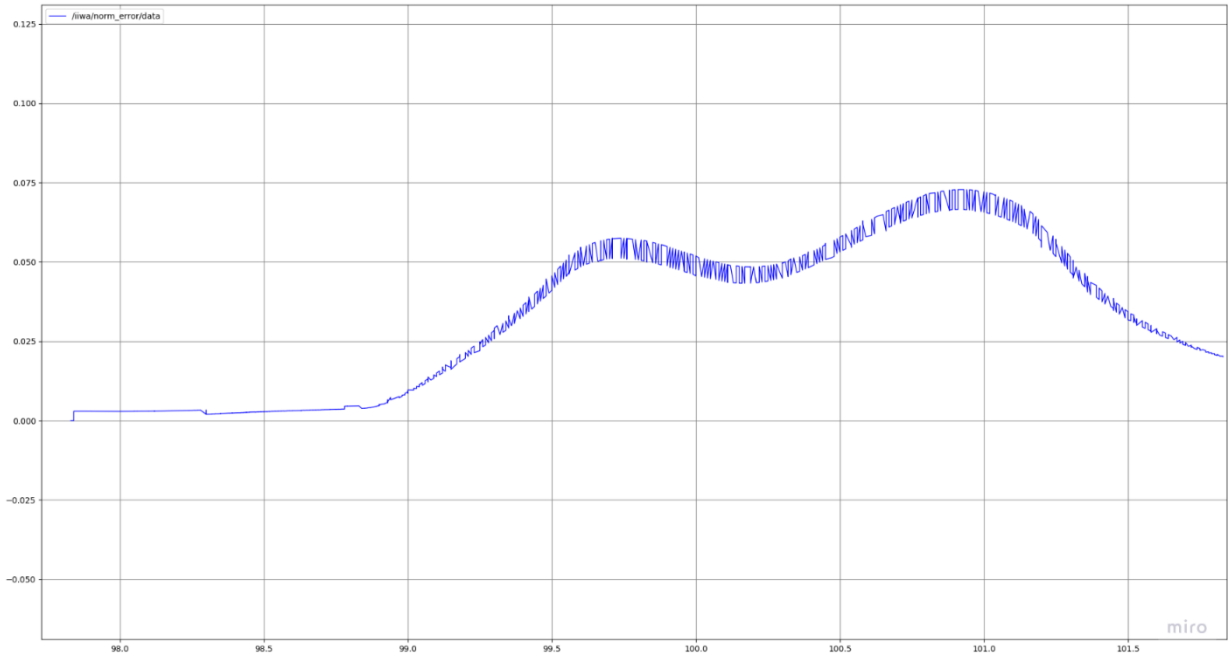
$$K_d = 80$$

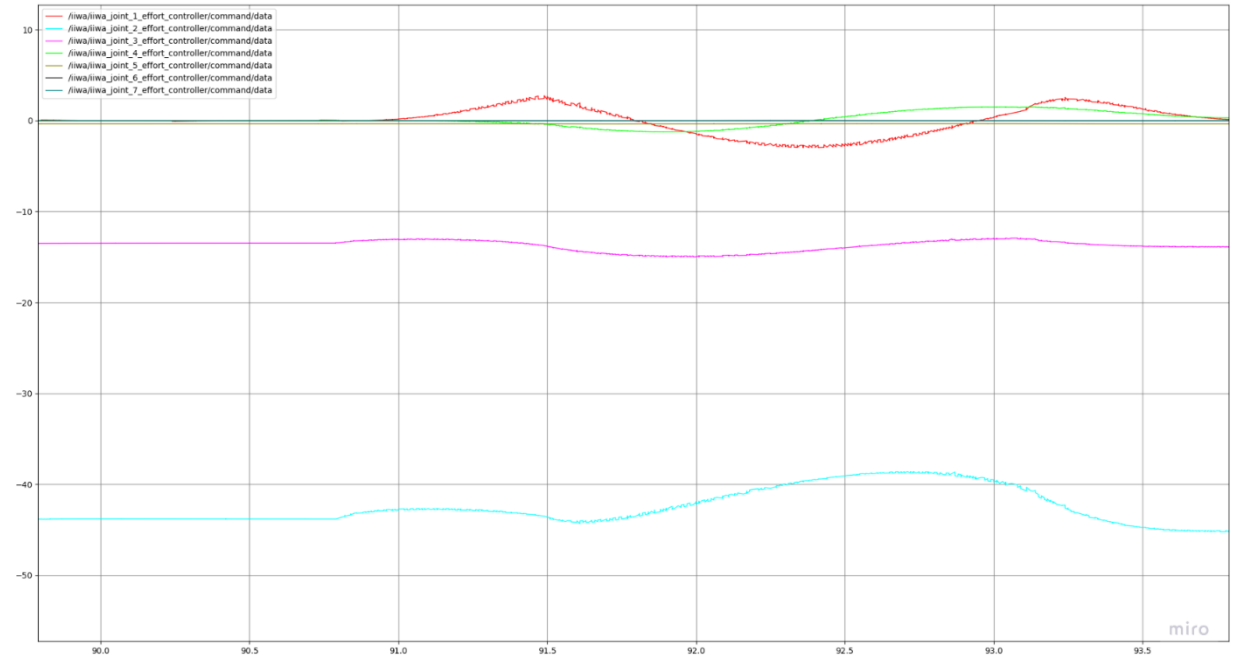




$$K_p = 10$$

$$K_d = 10$$

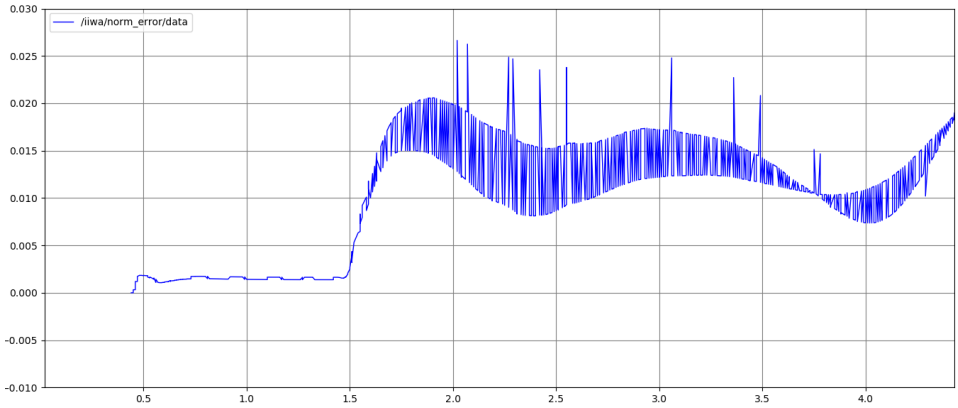

/iiwa/norm_error/data
0.125
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050
98.0
98.5
99.0
99.5
100.0
100.5
101.0
101.5
miro

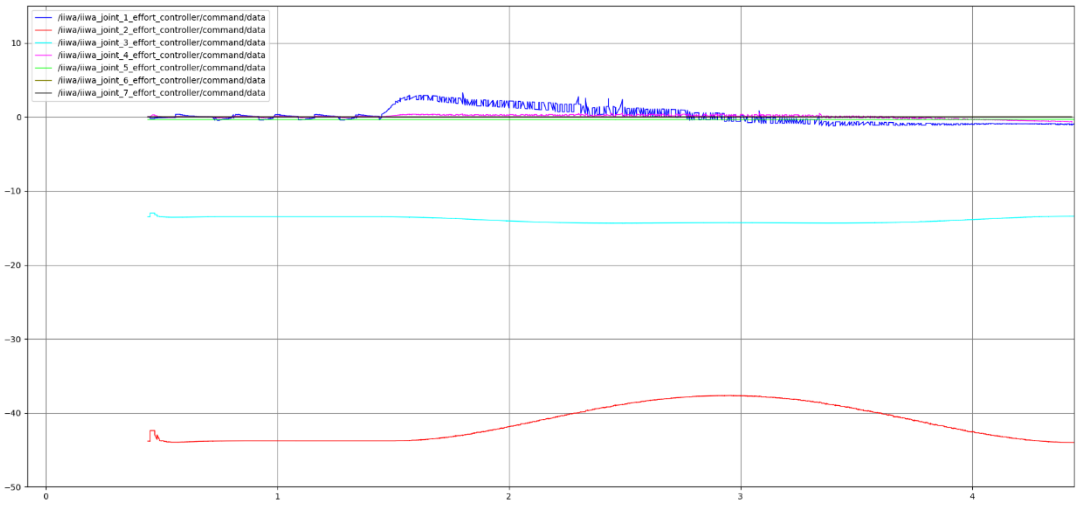

/iiwa/iiwa_joint_1_effort_controller/command/data
/iiwa/iiwa_joint_2_effort_controller/command/data
/iiwa/iiwa_joint_3_effort_controller/command/data
/iiwa/iiwa_joint_4_effort_controller/command/data
/iiwa/iiwa_joint_5_effort_controller/command/data
/iiwa/iiwa_joint_6_effort_controller/command/data
/iiwa/iiwa_joint_7_effort_controller/command/data
10
0
−10
−20
−30
−40
−50
90.0
90.5
91.0
91.5
92.0
92.5
93.0
93.5
miro

Cubic Linear

$$K_p = 70$$

$$K_d = 15$$

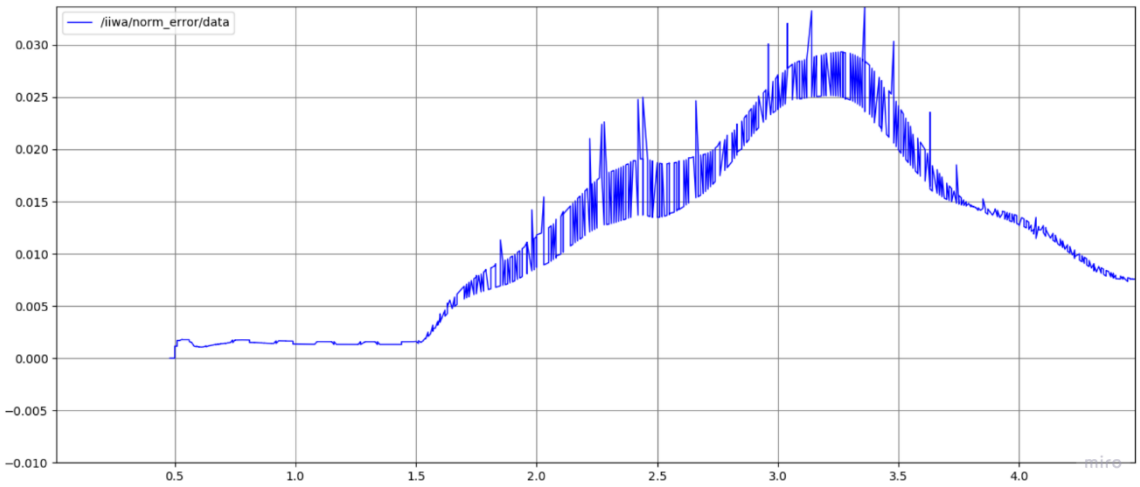


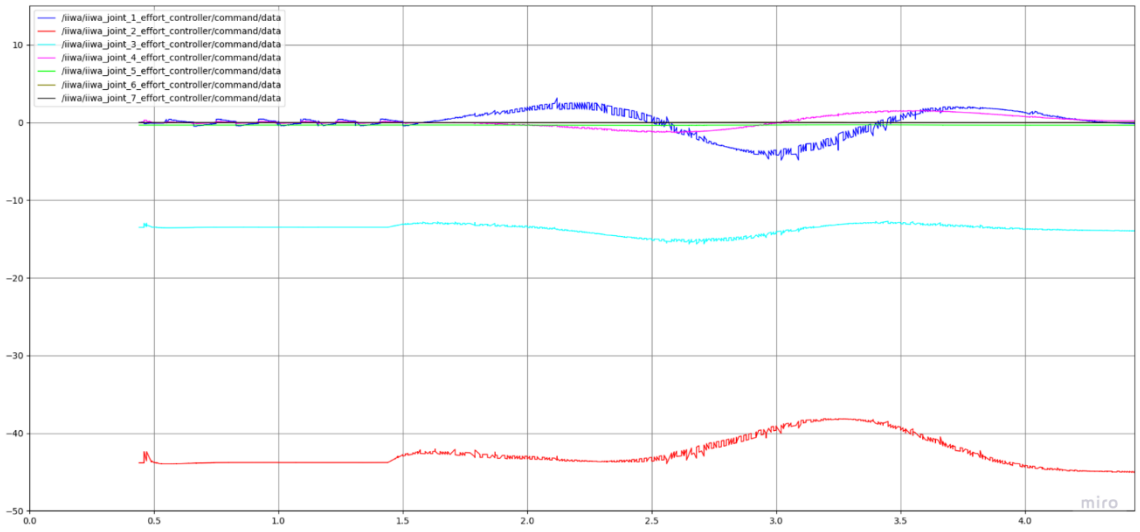


Cubic Circular

$$K_p = 80$$

$$K_d = 15$$





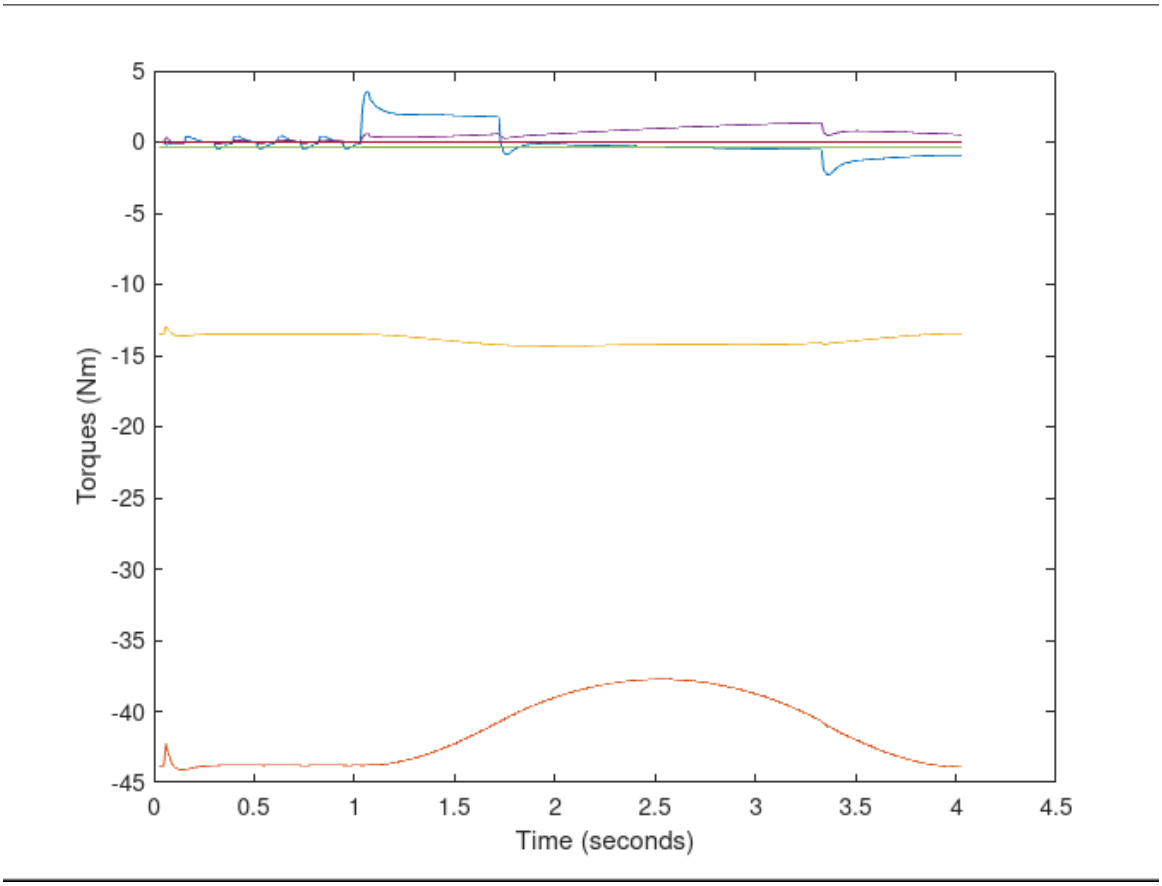
c) Optional: Save the joint torque command topics in a bag file and plot it using MATLAB.

Create a rosbag through the terminal:

```
benito@benito-HP-Laptop-15s-eq2xxx:~/my_ws$ rosbag record /iiwa/iiwa_joint_1_effort_controller/command /iiwa/iiwa_joint_2_effort_controller/command /iiwa/iiwa_joint_3_effort_controller/command /iiwa/iiwa_joint_4
_effort_controller/command /iiwa/iiwa_joint_5_effort_controller/command /iiwa/iiwa_joint_6_effort_controller/command /iiwa/iiwa_joint_7_effort_controller/command
[ INFO] [1700927072.913664299]: Subscribing to /iiwa/iiwa_joint_1_effort_controller/command
[ INFO] [1700927072.920647826]: Subscribing to /iiwa/iiwa_joint_2_effort_controller/command
[ INFO] [1700927072.924894038]: Subscribing to /iiwa/iiwa_joint_3_effort_controller/command
[ INFO] [1700927072.929580395]: Subscribing to /iiwa/iiwa_joint_4_effort_controller/command
[ INFO] [1700927072.933529947]: Subscribing to /iiwa/iiwa_joint_5_effort_controller/command
[ INFO] [1700927072.937780377]: Subscribing to /iiwa/iiwa_joint_6_effort_controller/command
[ INFO] [1700927072.944136109]: Subscribing to /iiwa/iiwa_joint_7_effort_controller/command
[ WARN] [1700927074.951984208]: /use_sim_time set to true and no clock published.  Still waiting for valid time...
[ INFO] [1700927085.937066934, 0.010000000]: Recording to '2023-11-25-16-44-45.bag'.
```

Then launch first iiwa_gazebo_effort.launch, run the controllers with rosrun and then after starting the simulation in gazebo, rosbag records the output of the simulation which has been saved in a file in the workspace.

To plot these results in matlab we had to save these data with the command rosbag. Then select the desired topic and in the end create a timeseries object to plot them. To verify the results, run the my_rosbag.m file on GitHub with the attached rosbag file.



4. Develop an inverse dynamics operational space controller

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

We first defined the four gain matrices:

```
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(); //costruisco la matrice 3x3 dei guadagni sull'errore di posizione
Kp.block(3,3,3,3) = _Kpo*Eigen::Matrix3d::Identity(); //costruisco la matrice 3x3 dei guadagni sull'errore di orientamento
Kd.block(0,0,3,3) = _Kdp*Eigen::Matrix3d::Identity();//costruisco la matrice 3x3 dei guadagni sulla derivata dell'errore di posizione
Kd.block(3,3,3,3) = _Kdo*Eigen::Matrix3d::Identity();//costruisco la matrice 3x3 dei guadagni sulla derivata dell'errore di orientamento
```

Then computed the errors of position and orientation using the already implemented functions:

```
// posizione desiderata ed effettiva
Eigen::Vector3d p_d(_desPos.p.data);
Eigen::Vector3d p_e(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);

// velocita' desiderata ed effettiva
Eigen::Vector3d dot_p_d(_desVel.vel.data);
Eigen::Vector3d dot_p_e(robot_->getEEVelocity().vel.data);
Eigen::Vector3d omega_d(_desVel.rot.data);
Eigen::Vector3d omega_e(robot_->getEEVelocity().rot.data);

// accelerazione desiderata ed effettiva
Eigen::Matrix<double,6,1> dot_dot_x_d;
Eigen::Matrix<double,3,1> dot_dot_p_d(_desAcc.vel.data);
Eigen::Matrix<double,3,1> dot_dot_r_d(_desAcc.rot.data);

//calcolo errore di posizione e velocità
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);
//calcolo errore orientamento
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);
```

b) finally compute your inverse dynamics control law following the equation:

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

```
// null space control
double cost;
Eigen::VectorXd grad = gradientJointLimits(robot_->getJntValues(),robot_->getJntLimits(),cost);


//matrice contenente y = xd_dot_dot - J_dot*q_dot + Kd*x_tilde_dot + Kp*x_tilde
Eigen::Matrix<double,6,1> y;
y << dot_dot_x_d - robot_->getEEJacDotqDot() + Kd*dot_x_tilde + Kp*x_tilde;

//restituiamo l'ingresso di controllo u = By + n
   return M * (Jpinv*y+ (I-Jpinv*J)*(/*- 10*grad */- 1*robot_->getJntVelocities()))+ robot_->getGravity() + robot_->getCoriolis();
```

Where it has been employed the function getEEJacDotqDot which at the beginning returned only the derivative of the Jacobian and has been modified to actually compute the product between it and the joint velocities

```
Eigen::VectorXd KDLRobot::getEEJacDotqDot()
{
    return s_J_dot_ee_.data*jntVel_.data;
}
```

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

1) TRAPEZOIDAL LINEAR
2) TRAPEZOIDAL CIRCULAR
3) CUBIC LINEAR
4) CUBIC CIRCULAR

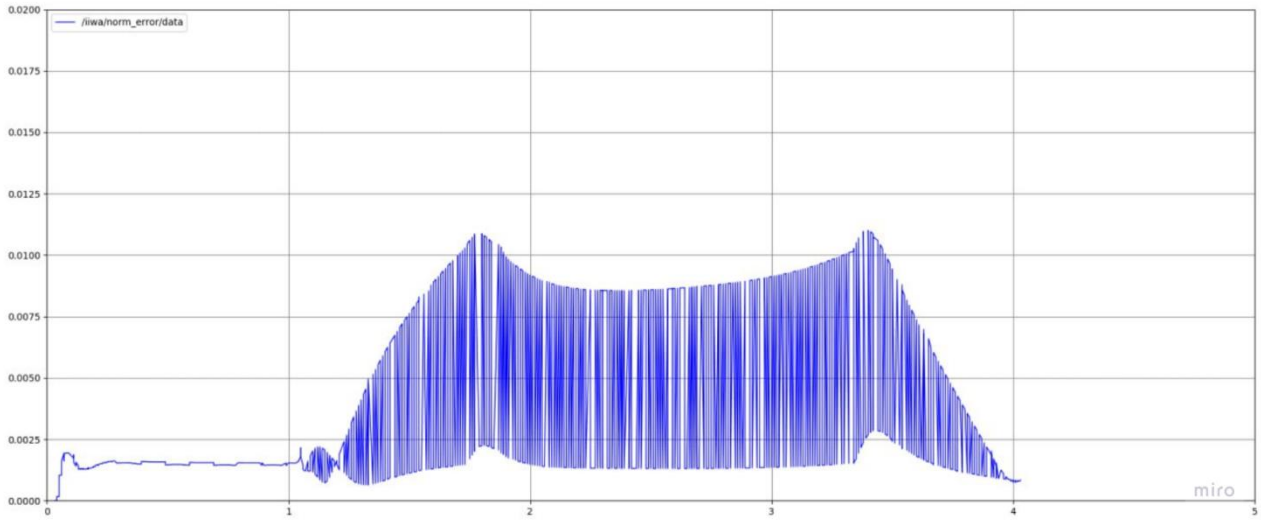
Trapezoidal Linear

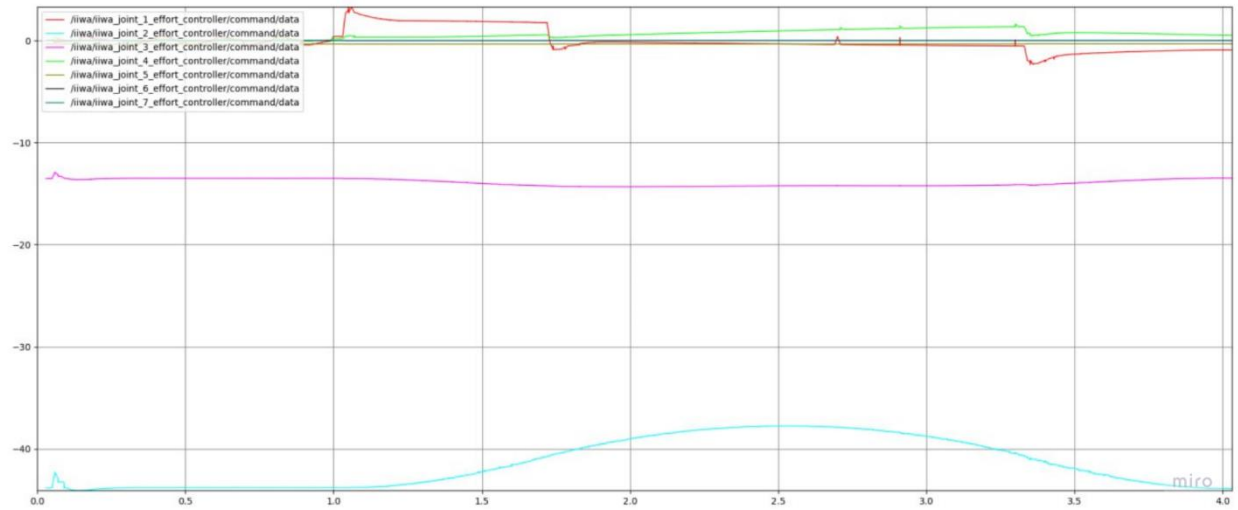
$$K_p = 70$$

$$K_o = 50$$

$$K_{dp} = 2 \cdot \sqrt{70}$$

$$K_{do} = 2 \cdot \sqrt{K_o}$$





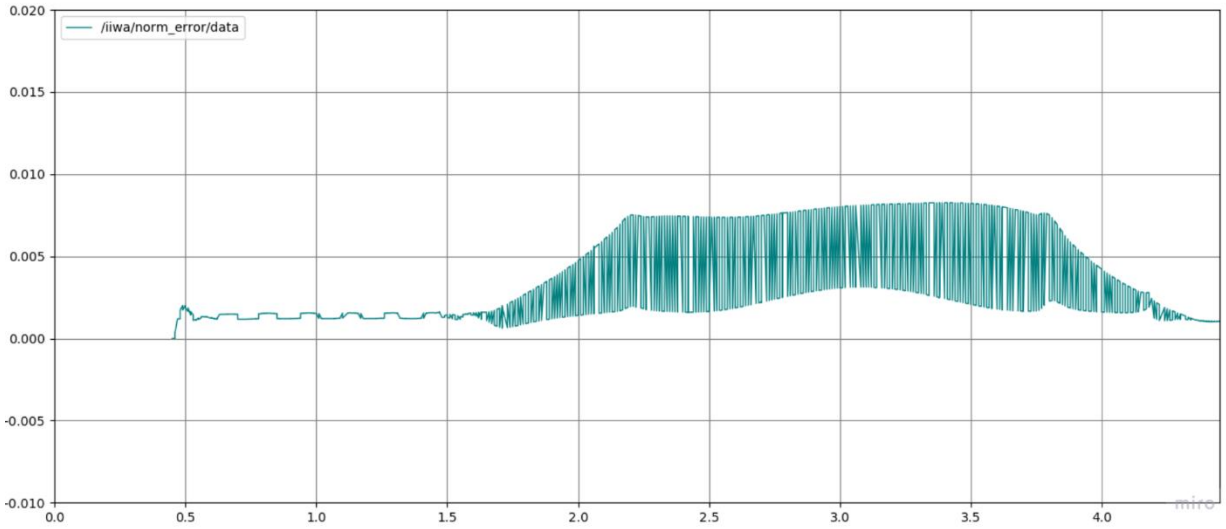
Trapezoidal Circular

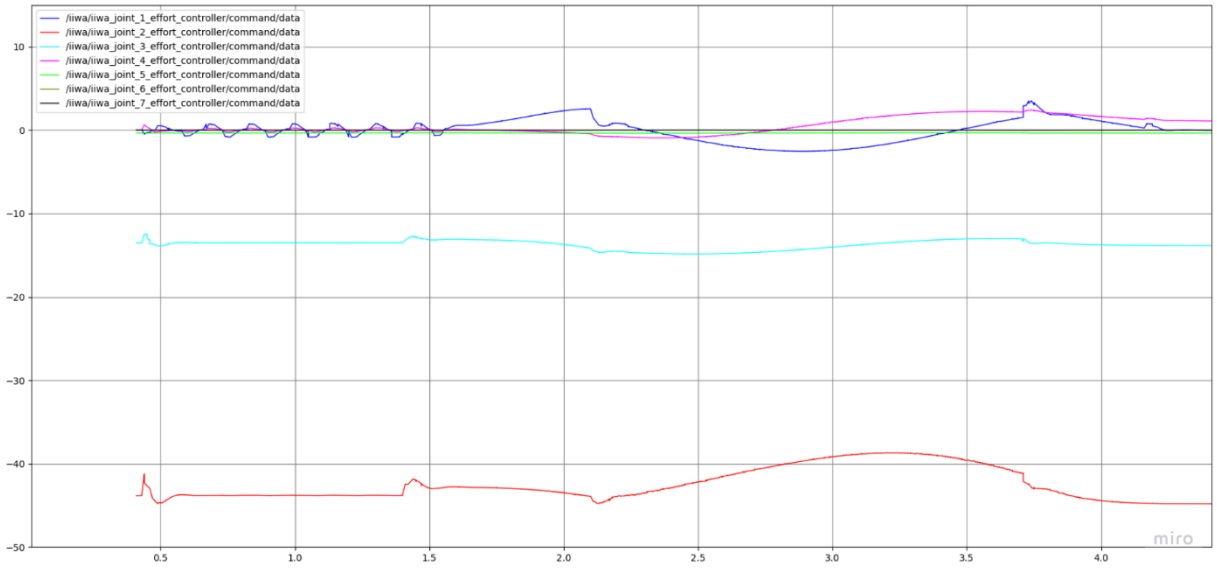
$$K_p = 100$$

$$K_o = 40$$

$$K_{dp} = 30$$

$$K_{do} = 2 \cdot \sqrt{K_o}$$





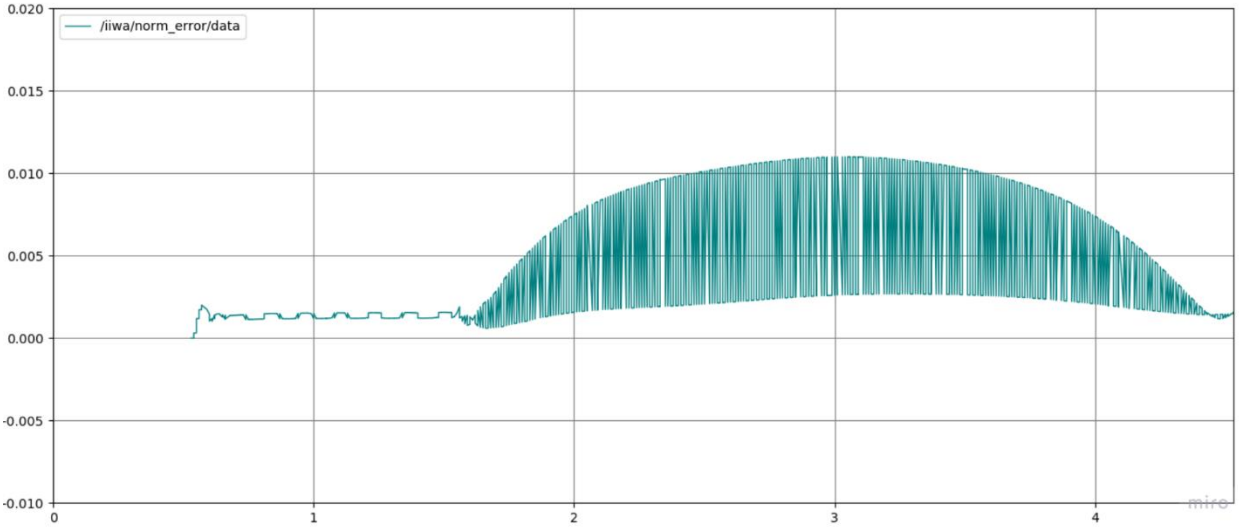
Cubic Linear

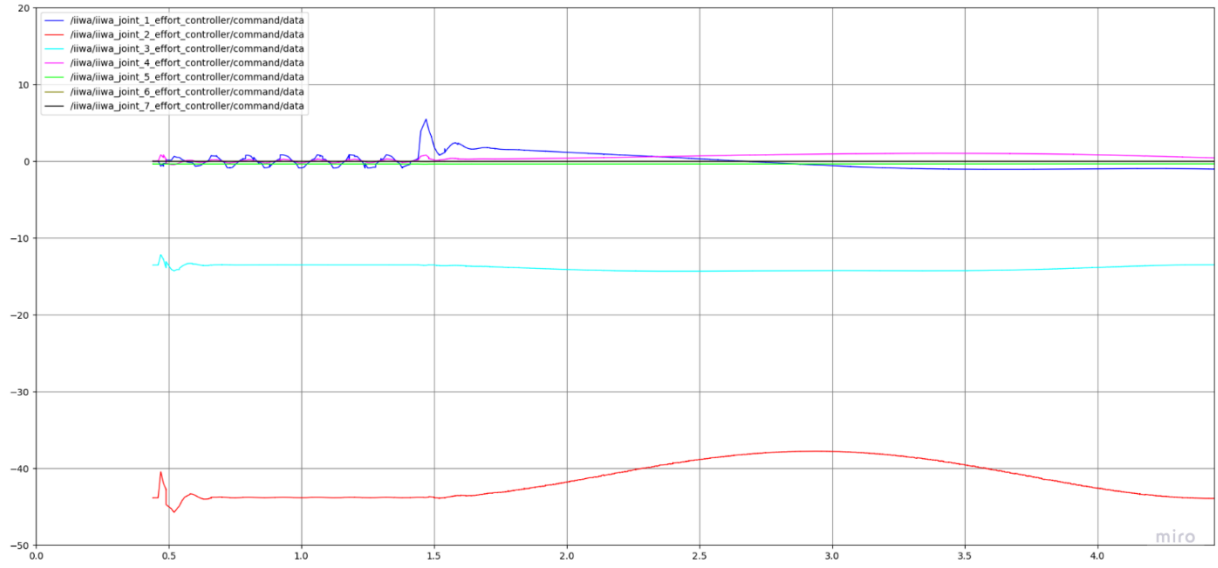
$$K_p = 80$$

$$K_o = 50$$

$$K_{dp} = 40$$

$$K_{do} = 2 \cdot \sqrt{K_o}$$





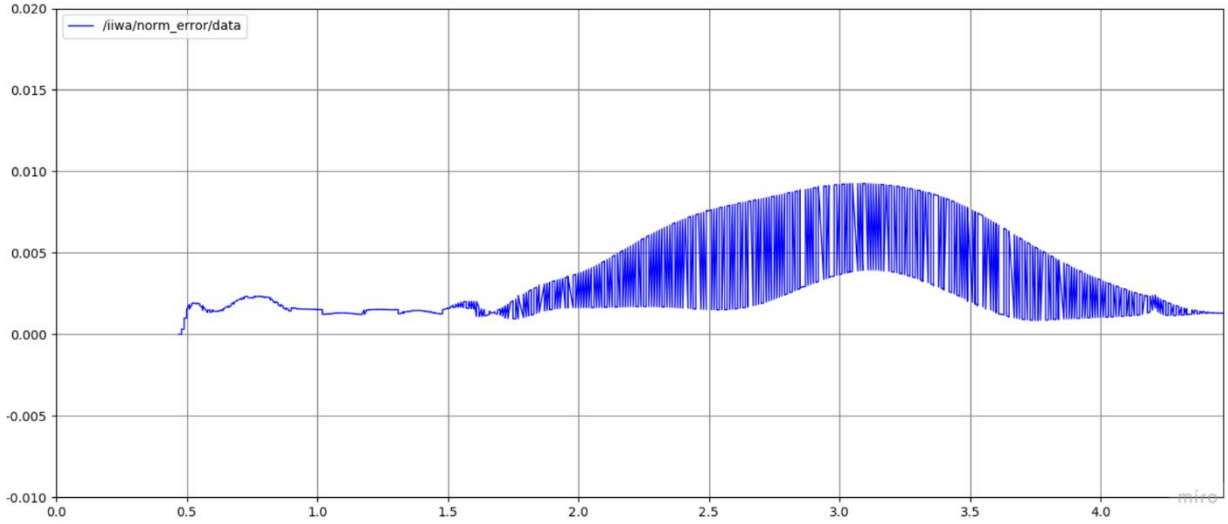
Cubic Circular

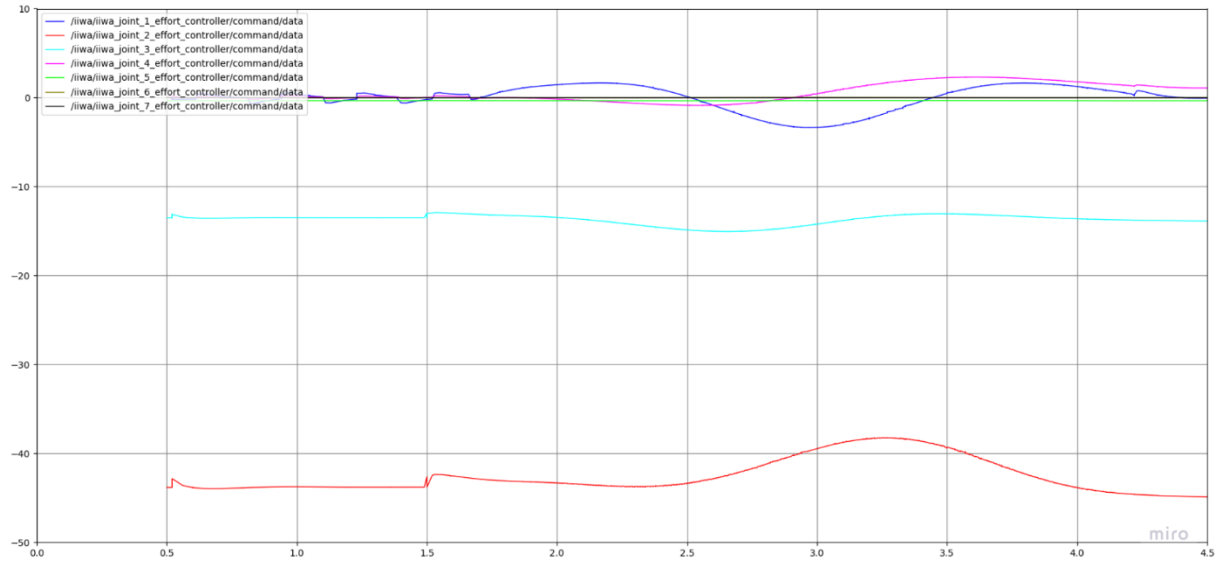
$$K_p = 100$$

$$K_o = 30$$

$$K_{dp} = 10$$

$$K_{do} = 2 \cdot \sqrt{K_o}$$





Extra 1: A different definition of the orientation error

We implemented also some functions in the utils.h file to manage the orientation error in the operational space with Euler Angles since the homework initially asked to compute the Analytical Jacobian.

At first, we created a function to compute the euler angles from the rotation matrix in input, we decided to use the ZYZ triplet:

```
inline Eigen::Matrix<double,3,1> computeEulerAngles(const Eigen::Matrix<double,3,3> &_R)
{
Eigen::Matrix<double,3,1> euler;
//scegliamo di utilizzare gli angoli di eulero ZYZ
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);
//effective angles
euler(0,0)=phi;
euler(1,0)=theta;
euler(2,0)=psi;
return euler;
}
```

At this point it is possible to easily compute the orientation error as follows:

```
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;

//desidered 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;
}
```

In order to define the Analytical Jacobian and the velocity error it is necessary to define a transformation matrix between the time derivatives of Euler angles and the angular velocity omega:

```
inline Eigen::Matrix<double,3,3> T_matrix(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;
}
```

So that:

```
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=T_matrix(phi_e);
Eigen::Matrix<double,3,3> Td=T_matrix(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;
}
```

Finally, it is possible to convert the Geometric Jacobian into the Analytical Jacobian.

```
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=Eigen::MatrixXd::Zero(6,6);
TA.block(0,0,3,3) = Eigen::Matrix3d::Identity();
TA.block(3,3,3,3) = T_matrix(euler);
 return TA.inverse()*J;
}
```

Extra 2: exploiting of redundancy by not fixing the orientation

In the previous points the orientation has been kept to the initial one, forcing the manipulator to point the end effector always upwards; since we cared only about the path of the position variables, it is possible to free the orientation and consider a reduced problem; in such ways the orientation will be just a cause of the internal reconfigurations of the manipulator. We do not choose any secondary task in the null space of the Jacobian, so the internal movements are just executed to minimize the energy.

To do that a new controller was created in the kdl_control.h:

```
Eigen::VectorXd idCntr(KDL::Frame &_desPos,
                       KDL::Twist &_desVel,
                       KDL::Twist &_desAcc,
                       double _Kpp,
                       double _Kdp);
```

The implementation was added in the kdl_control.ccp file:

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

    // calculate gain matrices
    Eigen::Matrix<double,3,3> Kp, Kd;
    Kp = _Kpp*Eigen::Matrix3d::Identity(); //costruisco la matrice 3x3 dei guadagni sull'errore di posizione
    Kd = _Kdp*Eigen::Matrix3d::Identity();//costruisco la matrice 3x3 dei guadagni sulla derivata dell'errore di posizione


    Eigen::Matrix<double,6,7> J = robot_->getEEJacobian().data;
    Eigen::Matrix<double,3,7> J_red = J.topRows(3);
    Eigen::Matrix<double,7,7> M = robot_->getJsim();
    Eigen::Matrix<double,7,3> Jpinv = pseudoinverse(J_red);

    // position
    Eigen::Vector3d p_d(_desPos.p.data);
    Eigen::Vector3d p_e(robot_->getEEFrame().p.data);

    // velocity
    Eigen::Vector3d dot_p_d(_desVel.vel.data);
    Eigen::Vector3d dot_p_e(robot_->getEEVelocity().vel.data);


    // acceleration
    Eigen::Matrix<double,3,1> dot_dot_x_d;
    Eigen::Matrix<double,3,1> dot_dot_p_d(_desAcc.vel.data);

    // compute 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);
    //ERRORE
    Eigen::Matrix<double,3,1> x_tilde;
    Eigen::Matrix<double,3,1> dot_x_tilde;
    x_tilde << e_p;
    dot_x_tilde << dot_e_p;
    dot_dot_x_d << dot_dot_p_d;
```

It's almost like the previous one but there are considered just reduced version of the errors and of the Jacobian, since we decided to free the orientation, we only need 3 DoF.

```
Eigen::Matrix<double,3,1> y;

    y << dot_dot_x_d - robot_->getEEJacDotqDot_red() + Kd*dot_x_tilde + Kp*x_tilde;
return M * (Jpinv*y)+ robot_->getGravity() + robot_->getCoriolis();


}
```

In the control law it was also necessary to define a new getEEJAcDotqDot since I need only the first three rows of the Jacobian matrix:

```
Eigen::VectorXd KDLRobot::getEEJacDotqDot_red()
{
    return s_J_dot_ee_.data.topRows(3)*jntVel_.data;
}
```