Robotics Lab: Homework 2

Control a manipulator to follow a trajectory

Giuseppe Saggese
P38000213

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

a) ***Modify appropriately the KDLPlanner class (files kdl_planner.h and kdl_planner.cpp) that provides a basic interface for trajectory creation. First, define a new KDLPlanner::trapezoidal_vel function that takes the current time t and the acceleration time tc as double arguments and returns three double variables s, $\dot{s}$ and $\ddot{s}$ that represent the curvilinear abscissa of your trajectory.***

In order to do that we implemented our trapezoidal_vel function.

```
void KDLPlanner::trapezoidal_vel (double time, double &s, double &s_d, double &s_dd)
{
  double s_f=1;
  double s_i=0;

  double ddot_traj_c=-(s_f-s_i)/(std::pow(accDuration_,2)-trajDuration_*accDuration_);

  if(time <= accDuration_)
  {
    s = s_i+ 0.5*ddot_traj_c*std::pow(time,2);
    s_d = ddot_traj_c*time;
    s_dd = ddot_traj_c;
  }
  else if(time <= trajDuration_-accDuration_)
  {
    s = s_i + ddot_traj_c*accDuration_*(time-accDuration_/2);
    s_d = ddot_traj_c*accDuration_;
    s_dd = 0;
  }
  else
  {
    s = s_f - 0.5*ddot_traj_c*std::pow(trajDuration_-time,2);
    s_d = ddot_traj_c*(trajDuration_-time);
    s_dd = -ddot_traj_c;
  }
}
```

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

Then we also want to implement a function that creates a cubic polynomial curvilinear abscissa.
We did it like that:

```
void KDLPlanner::cubic_polinomial (double time, double &s, double &s_d, double &s_dd)
{
  double s_f=1;
  double s_i=0;

  double a0 = s_i;
  double a1 = 0;
  double a2 = 3*(s_f-s_i)/std::pow(trajDuration_,2);
  double a3 = -2*(s_f-s_i)/std::pow(trajDuration_,3);

  s = a3*std::pow(time,3) + a2* std::pow(time,2) + a1*time + a0;
  s_d = 3*a3* std::pow(time,2) + 2*a2*time + a1;
  s_dd = 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 (to be created in the kdl_planner.h).***

```
//Constructor for Circular Trajectory
KDLPlanner::KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double _trajRadius, double _accDuration)
{
  accDuration_ = _accDuration;
  trajDuration_ = _trajDuration;
  trajInit_ = _trajInit;
  trajRadius_=_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: first, call the cubic_polinomial function to retrieve s and its derivatives from t; then fill in the trajectory_point fields traj.pos, traj.vel, and traj.acc.***
As you can see we set both the velocity and acceleration on the x axis equal to zero

```
trajectory_point KDLPlanner::compute_CircCubicTrajectory(double time, double trajRadius_) //Circ Cubic
{
  trajectory_point traj;

  double s,s_d,s_dd;
  cubic_polinomial(time, s, s_d, s_dd);

  traj.pos(0) = trajInit_(0);
  traj.pos(1) = trajInit_(1)- trajRadius_*(std::cos(2*M_PI*s));
  traj.pos(2) = trajInit_(2) - trajRadius_*(std::sin(2*M_PI*s));

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

  traj.acc(0) = 0;
  traj.acc(1) = 2*M_PI*trajRadius_*(std::cos(2*M_PI*s)*2*M_PI*std::pow(s_d,2)+std::sin(2*M_PI*s)*s_dd);
  traj.acc(2) = 2*M_PI*trajRadius_*(std::sin(2*M_PI*s)*2*M_PI*std::pow(s_d,2)-std::cos(2*M_PI*s)*s_dd);

  return traj;
}
```

c) ***Do the same for the linear trajectory.***

```
//Constructor for Prof Linear Trajectory & Lin
KDLPlanner::KDLPlanner(double _trajDuration, double _accDuration, Eigen::Vector3d _trajInit, Eigen::Vector3d _trajEnd)
{
  trajDuration_ = _trajDuration;
  accDuration_ = _accDuration;
  trajInit_ = _trajInit;
  trajEnd_ = _trajEnd;
}
```

```
trajectory_point KDLPlanner::compute_CubicLinearTrajectory(double time) //Lin Cubic
{
  trajectory_point traj;

  double s, s_d, s_dd;
  cubic_polinomial(time, s, s_d, s_dd);

  traj.pos = trajInit_ + s * (trajEnd_ - trajInit_);
  traj.vel = s_d * (trajEnd_ - trajInit_);
  traj.acc = s_dd * (trajEnd_ - trajInit_);

  return traj;
}
```

3. Test the four trajectories

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

So at point 1.a we have done the trapezoidal_vel function which is used to start a simulation with a trapezoidal profile for the velocity. Then we also implemented a function to compute the cubic polinomial curvilinear abscissa.

These two algorithms have already been explained in the previous sections, now we have to test the 4 trajectories:

- Linear Trajectory with trapezoidal velocity
- Linear Trajectory with cubic polinomial curvilinear abscissa
- Circular Trajectory with cubic polinomial curvilinear abscissa
- Circular Trajectory with trapezoidal velocity

To implement and test all the 4 trajectories we have implemented an index based selection like shown in figure:

Choosing a number from 0 to 3 you can choose between all the different trajectories.

```
int trajIndex = 3;  /*0)LinearTrap - enabled
                      ProfLinear - commented
                     1)CubicLinear - enabled
                      CubicLinear - commented
                     2)Circular Cubic
                     3)Circular Trap*/
```

Then you only have to uncomment the right constructor.
As you can see in the figure, we implemented different functions for each different trajectory.

```
//Constructors
//KDLPlanner planner(traj_duration, init_position, traj_radius, acc_duration); //Circular Trap & Cubic
KDLPlanner planner(traj_duration, acc_duration, init_position, end_position); // Lin Trap & Cubic

// Gains
double Kp;
double Kd;

// Retrieve the first trajectory point
trajectory_point p;
if(trajIndex == 0)
{
    Kp = 20;
    Kd = sqrt(Kp);
    p = planner.compute_LinTrapTrajectory(t); //Lin Trap
    //p = planner.compute_trajectory(t);
}
else if(trajIndex == 1)
{
    Kp = 20;
    Kd = sqrt(Kp)-1;
    p = planner.compute_CubicLinearTrajectory(t); //CubicLinear
    //p = planner.compute_CubicLinearTrajectory2(t); //CubicLinear
}
else if(trajIndex == 2)
{
    Kp = 20;
    Kd = sqrt(Kp);
    p = planner.compute_CircCubicTrajectory(t, traj_radius); //Circular Cubic
}
else if(trajIndex == 3)
{
    Kp = 20;
    Kd = sqrt(Kp)+0.3;
    p = planner.compute_CircTrapTrajectory(t, traj_radius); //Circular Trap
}
```

Then of course the same happens in the while loop during the execution of the control. Inside the while loop will be called the right function thanks to that method.

```
if (t <= init_time_slot) // wait a second
{
    if(trajIndex == 0)
    {
        p = planner.compute_LinTrapTrajectory(0.0);  //Lin Traj
        //p = planner.compute_trajectory(0.0); //Prof Traj
    }
    else if(trajIndex == 1)
    {
        p = planner.compute_CubicLinearTrajectory(0.0); //Cubic Lin Traj1
        //p = planner.compute_CubicLinearTrajectory2(0.0); //Cubic Lin Traj2
    }
    else if(trajIndex == 2)
    {
        p = planner.compute_CircCubicTrajectory(0.0, traj_radius); //Circular Traj
    }
    else if(trajIndex == 3)
    {
        p = planner.compute_CircTrapTrajectory(0.0, traj_radius);
    }
}
```

```
else if(t > init_time_slot && t <= traj_duration + init_time_slot)
{
    if(trajIndex == 0)
    {
        p = planner.compute_LinTrapTrajectory(t-init_time_slot);  //Lin Traj
        //p = planner.compute_trajectory(t-init_time_slot); //Prof Traj
    }
    else if(trajIndex == 1)
    {
        p = planner.compute_CubicLinearTrajectory(t-init_time_slot); //Cubic Lin Traj
        //p = planner.compute_CubicLinearTrajectory2(t-init_time_slot); //Cubic Lin Traj2
    }
    else if(trajIndex == 2)
    {
        p = planner.compute_CircCubicTrajectory(t-init_time_slot, traj_radius); //Circular Traj
    }
    else if(trajIndex == 3)
    {
        p = planner.compute_CircTrapTrajectory(t-init_time_slot, traj_radius);
    }

    des_cart_vel = KDL::Twist(KDL::Vector(p.vel[0], p.vel[1], p.vel[2]),KDL::Vector::Zero());
    des_cart_acc = KDL::Twist(KDL::Vector(p.acc[0], p.acc[1], p.acc[2]),KDL::Vector::Zero());
}
```

Then we also noticed that in the main function there were no functions to compute the values for the velocities for the inverse kinematic, so we implemented the function getInverseKinematics in which we pass by reference the joints velocity variable, and then the function assigns the right values to it.

```
// inverse kinematics
qd.data << jnt_pos[0], jnt_pos[1], jnt_pos[2], jnt_pos[3], jnt_pos[4], jnt_pos[5], jnt_pos[6];
qd = robot.getInvKin(qd, des_pose);
robot.getInverseKinematics(des_pose, des_cart_vel, des_cart_acc,qd,dqd,ddqd);

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

In the figure we show the implementation of the getInverseKinematics function:

```
void KDLRobot::getInverseKinematics(KDL::Frame &f,KDL::Twist &twist,KDL::Twist &acc,
                                    KDL::JntArray &q,KDL::JntArray &dq,KDL::JntArray &ddq)
{
    Eigen::Matrix<double,3,1> dot_p(twist.vel.data);
    Eigen::Matrix<double,3,1> dot_r(twist.rot.data);
    Eigen::Matrix<double,6,1> dp;
    dp << dot_p, dot_r;
    Eigen::Matrix<double,7,6> Jpinv = weightedPseudoInverse(jsim_.data, s_J_ee_.data);
    dq.resize(chain_.getNrOfJoints());
    dq.data = Jpinv*dp;
}
```

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

Then we tuned the gains Kp and Kd using the rqt_plot Ros tool.

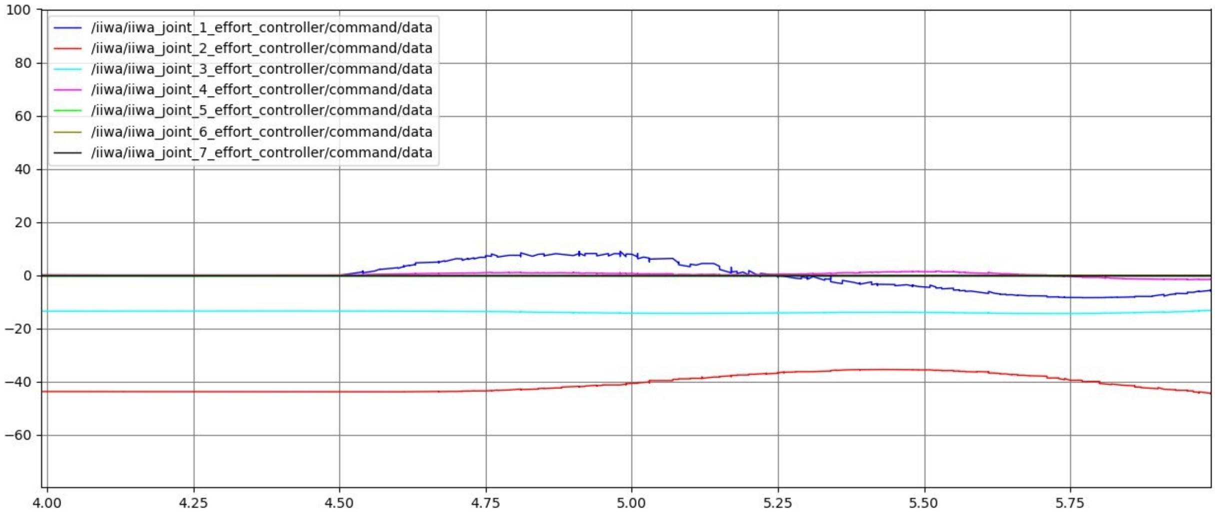
With this tool we choose to visualize the torques topics and then we tune the gains.

We show our best values plotted in the figure:

- Trajectory : Linear with trapezoidal velocity profile

$K_p = 20$;

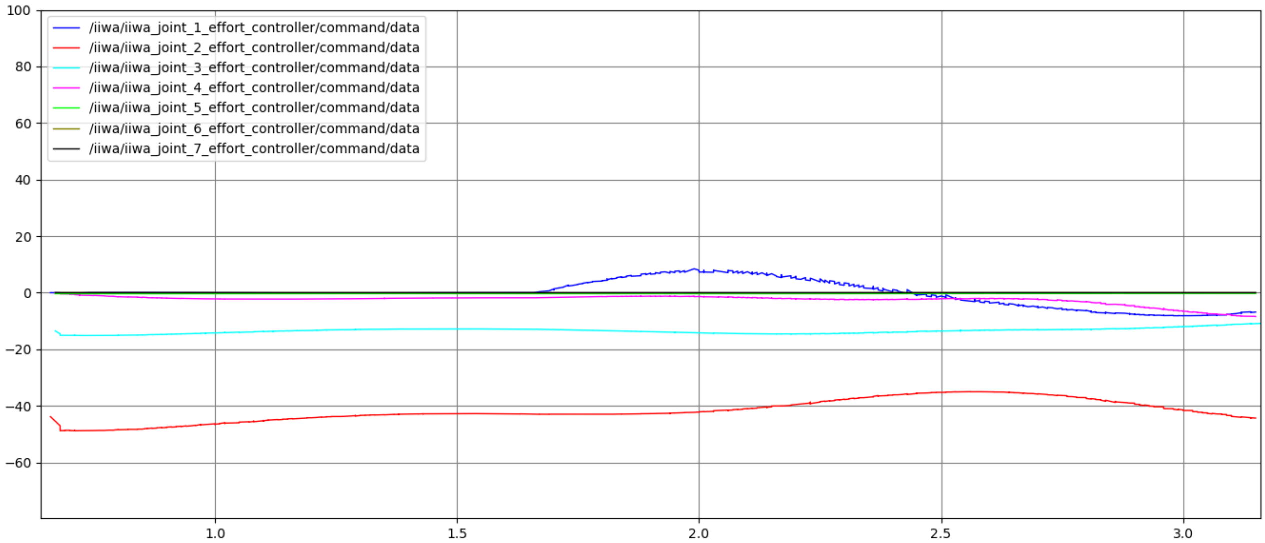
$K_d = \sqrt{(K_p)}$.



- Trajectory : Linear with cubic polinomial curvilinear abscissa

$K_p = 20$;

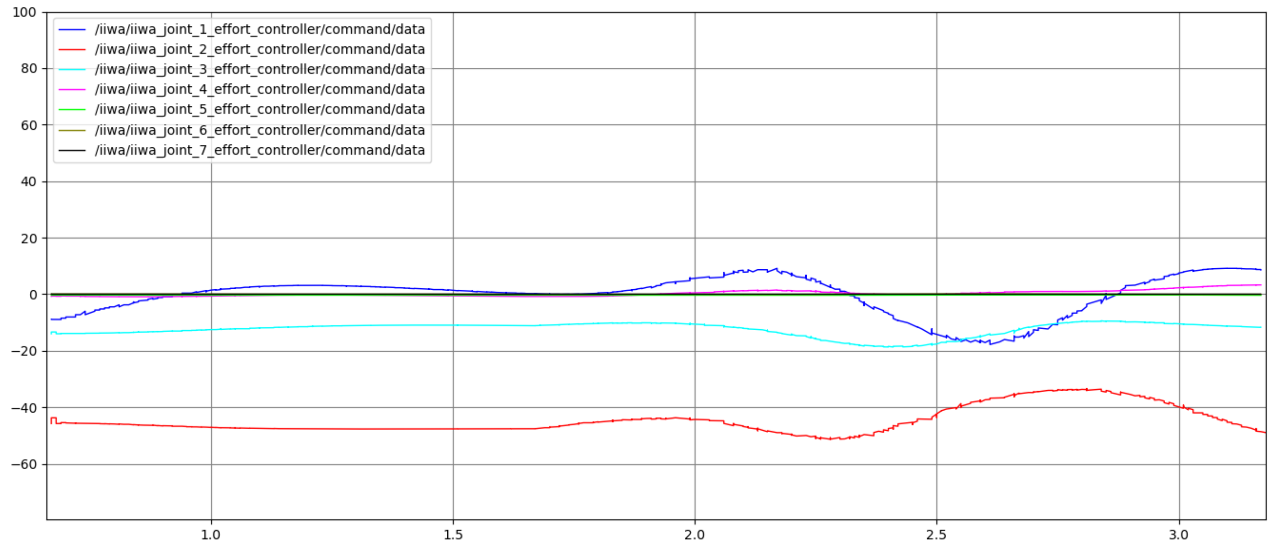
$K_d = \sqrt{(K_p)} - 1$.



- Trajectory : Circular with cubic polinomial curvilinear abscissa

$K_p = 20$;

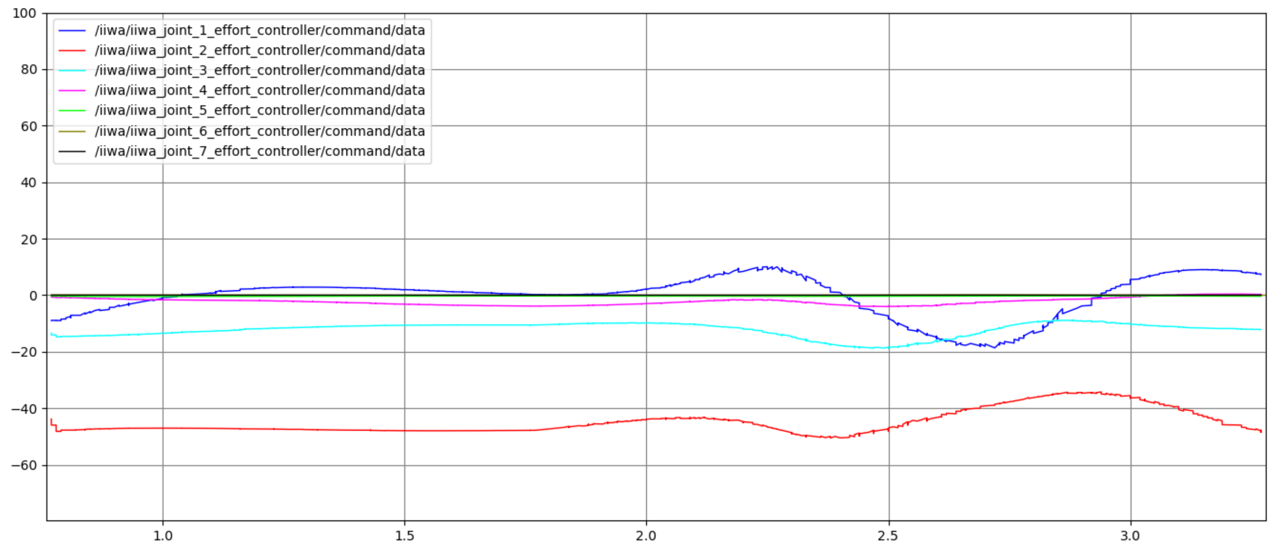
$K_d = \sqrt{(K_p)}$.



- Trajectory : Circular with trapezoidal velocity profile

$K_p = 20$;

$K_d = \sqrt{(K_p)}+0{,}3$.



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

Other then using rqt_plot it is also possible to show the plot in MATLAB.
In order to do that we are going to use the rosbag function.

First of all it was necessary to create a variable .bag (EJ1.bag, … , EJ7.bag).

In that variable there will be stored the stream of the topic that we choose using the following command in the bash:

rosbag record -O /home/davide/Documents/EJ1.bag /iiwa/iiwa_joint_1_effort_controller/command

That command has to be applied for each effort topic.

Once the info relative to the torques have been saved in that file we have to show them in MATLAB.

- In order to do that first of all we have to install the Ros ToolBox in MATLAB.

- Then launch the rosinit command.

- Then we create a variable 'bag1' in MATLAB in which we store the bag file 'EJ1.bag' previously created with the command rosbag(path)

- Then we choose the topic to read from the bag file and we store in a new variable.

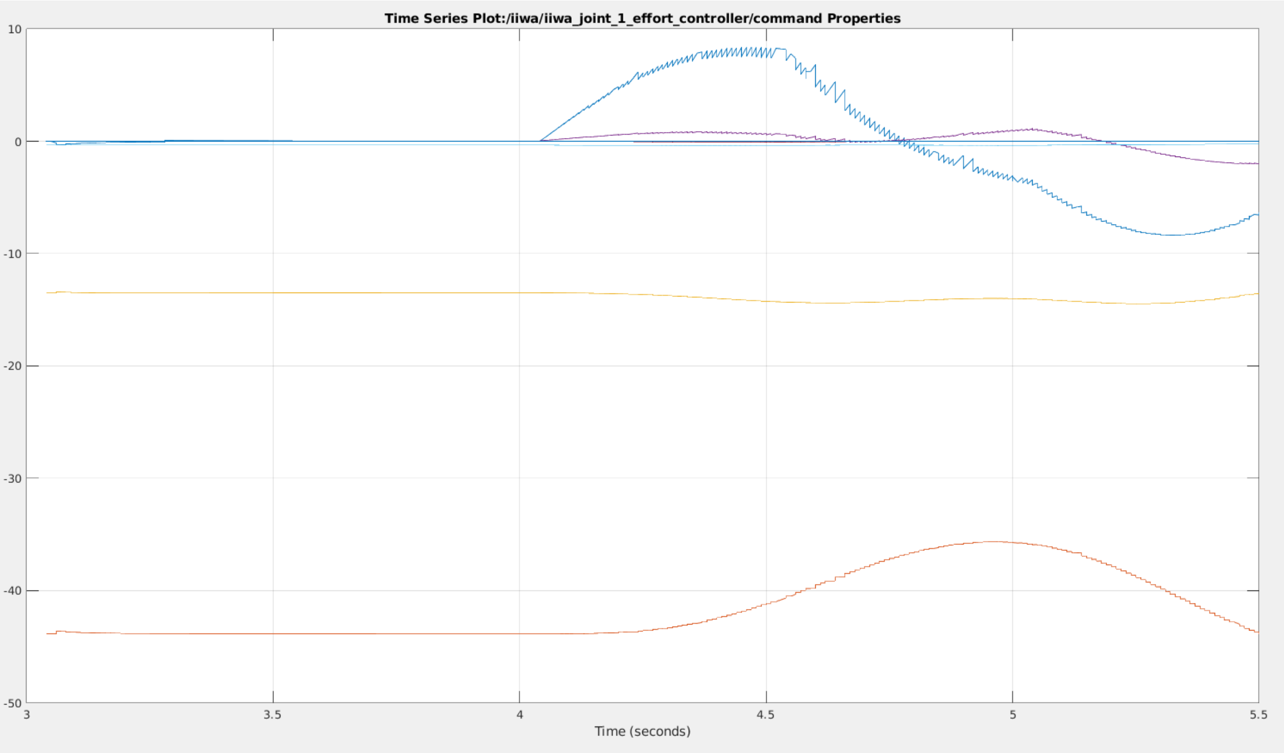
- Then with the command timeseries we obtain a return of a timeseries object from the variable in which we stored the topic.

- Then we can plot that last variable.

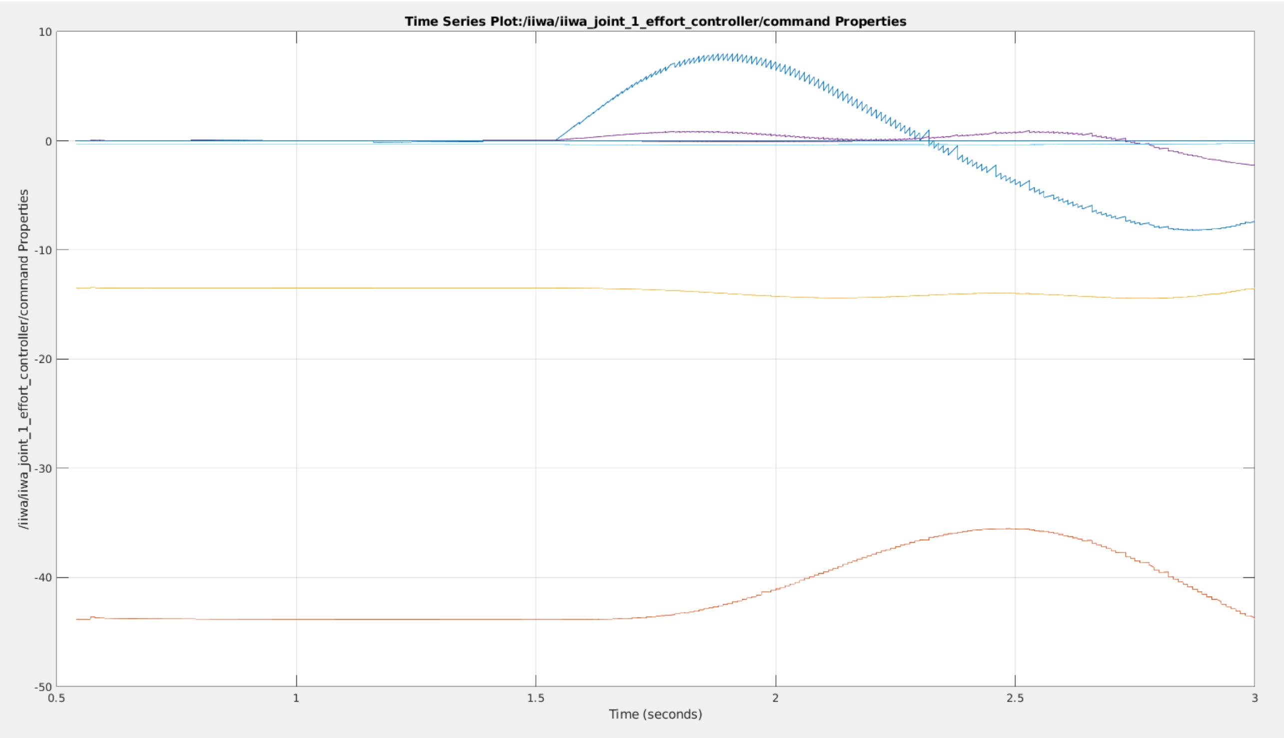
We apply that for every Effort Joint and trajectory.

Now we plot all the efforts for each trajectory with a MATLAB code shown in the last page.

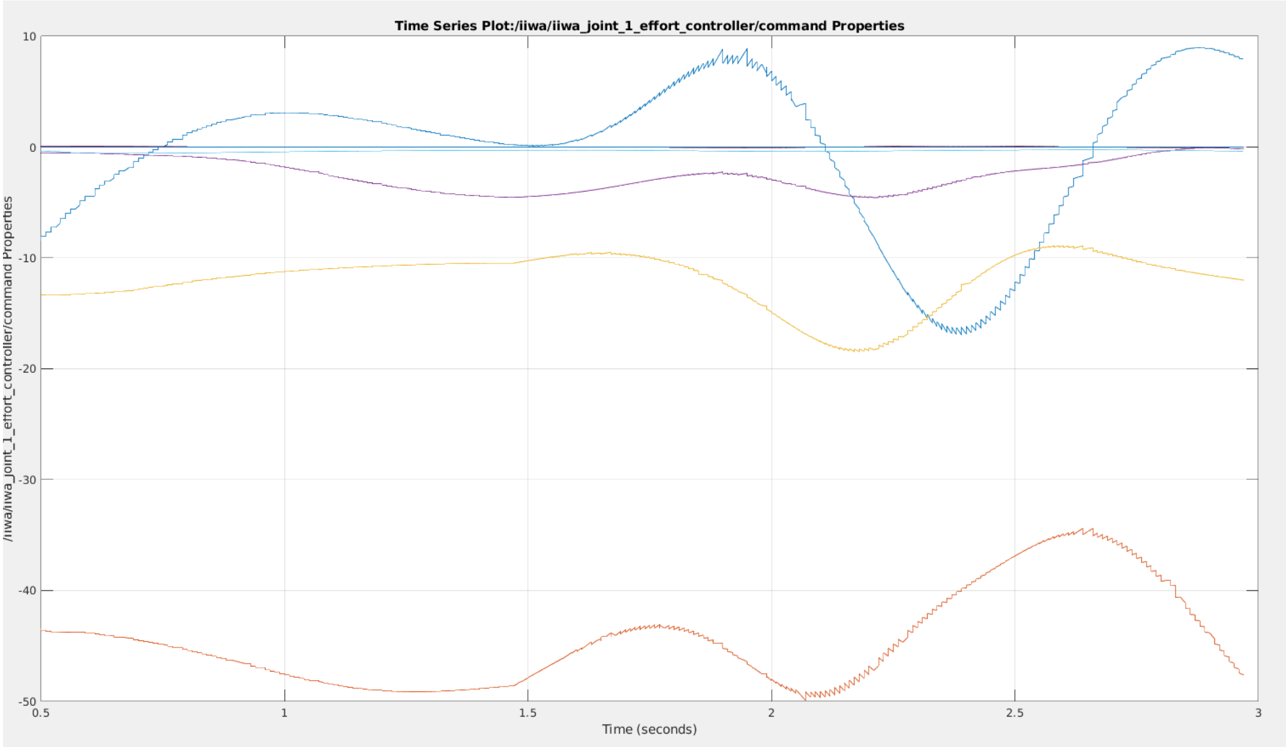
- Trajectory : Linear with trapezoidal velocity profile



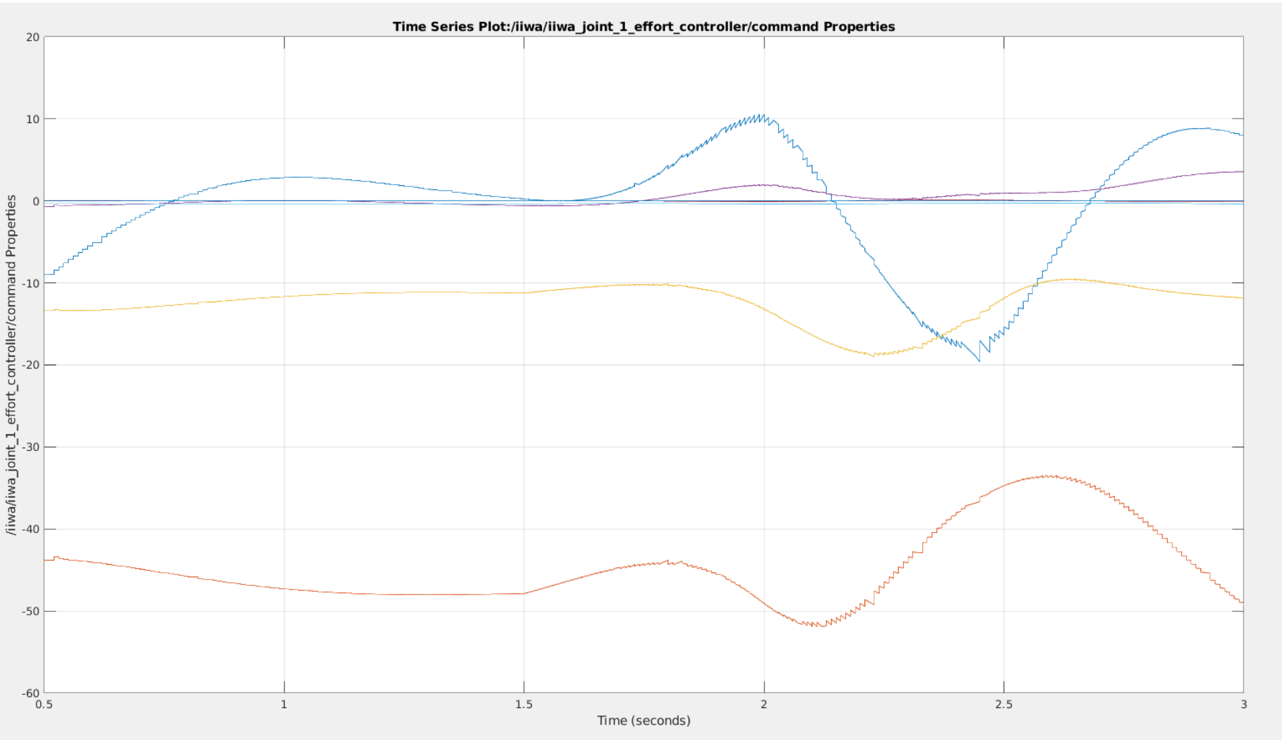
- Trajectory : Linear with cubic polinomial curvilinear abscissa



- Trajectory : Circular with cubic polinomial curvilinear abscissa



- Trajectory : Circular with trapezoidal velocity profile



4. *Develop an inverse dynamics operational space controller*

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

In order to use the operational space controller we uncommented the KDLController::idCntr but we noted that some functions in the code were not defined and implemented in the kdl_robot.h and kdl_robot.cpp files. So we substituted the getJacobian(), getCartesianPose(), getCartesianVelocity() and getJacDotqDot() functions respectively with getEEJacobian(), getEEFrame(), getEEVelocitiy() and getEEJacDotqDot(). Moreover we initialized the K_p and K_d matrices to zero.
In the kdl_robot_test.cpp file we commented the line in which calls back the joint space inverse dynamics controller and we uncommented the line of the operational space controller with its gains K_p and K_o. We used differently proportional and derivative gain for each trajectories.

```cpp
// inverse kinematics
qd.data << jnt_pos[0], jnt_pos[1], jnt_pos[2], jnt_pos[3], jnt_pos[4], jnt_pos[5], jnt_pos[6];
qd = robot.getInvKin(qd, des_pose);
//robot.getInverseKinematics(des_pose, des_cart_vel, des_cart_acc,qd,dqd,ddqd);

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

// Gains Operational Space Controller
double Kp;
double Ko;

if(trajIndex == 0)
{
    Kp=70;
    Ko=66;

// Cartesian space inverse dynamics control
tau = controller_.idCntr(des_pose, des_cart_vel, des_cart_acc, Kp, Ko, 1.7*sqrt(Kp), 2*sqrt(Ko));

}
else if(trajIndex == 1)
{
    Kp=60;
    Ko=65;

// Cartesian space inverse dynamics control
tau = controller_.idCntr(des_pose, des_cart_vel, des_cart_acc, Kp, Ko, 1.7*sqrt(Kp), 2*sqrt(Ko));
}
else if(trajIndex == 2)
{
    Kp=40;
    Ko=38;

// Cartesian space inverse dynamics control
tau = controller_.idCntr(des_pose, des_cart_vel, des_cart_acc, Kp, Ko, 1.3*sqrt(Kp), 1.5*sqrt(Ko));
}
else if(trajIndex == 3)
{
    Kp=30;
    Ko=30;

// Cartesian space inverse dynamics control
tau = controller_.idCntr(des_pose, des_cart_vel, des_cart_acc, Kp, Ko, 1.5*sqrt(Kp), 1.7*sqrt(Ko));

}
```

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

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

Calculate gain matrices:

```
//     // calculate 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();
       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 Cartesian state of your manipulator in terms of end-effector parametrized pose x, velocity ẋ, and acceleration ẍ, retrieve the current joint space inertia matrix M and the Jacobian and its time derivative:

```
    // read current state
    Eigen::Matrix<double,6,7> J = robot_->getEEJacobian().data;
    Eigen::Matrix<double,7,7> I = Eigen::Matrix<double,7,7>::Identity();
    Eigen::Matrix<double,7,7> M = robot_->getJsim();
    Eigen::Matrix<double,7,6> Jpinv = weightedPseudoInverse(M,J);
    //Eigen::Matrix<double,7,6> Jpinv = pseudoinverse(J);

    // position
    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);

    // velocity
    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);

    // acceleration
    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);
```

Compute the linear e_p and the angular e_o errors:

```
// 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);
```

```
// compute 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> dot_x_tilde;
x_tilde << e_p, e_o;
dot_x_tilde << dot_e_p, -omega_e;//dot_e_o;
dot_dot_x_d << dot_dot_p_d, dot_dot_r_d;
```

Compute the inverse dynamics control law:

```
// inverse dynamics
Eigen::Matrix<double,6,1> y;
y << dot_dot_x_d - robot_->getEEJacDotqDot() + Kd*dot_x_tilde + Kp*x_tilde;

return M * (Jpinv*y + (I-Jpinv*J)*(/*- 10*grad */- 1*robot_->getJntVelocities()))
        + robot_->getGravity() + robot_->getCoriolis();
}
```

At this point we noticed that the getEEJacDotqDot() function returned only the $\dot{J}$ matrix thus we modified the function in the kdl_robot.cpp multiplying $\dot{J}$ by $\dot{q}$

```
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.***

We tested the operational space controller for each trajectories, we tried several proportional and derivative gains in order to reach a smooth behavior of the joint torque commands. We found the following gains:

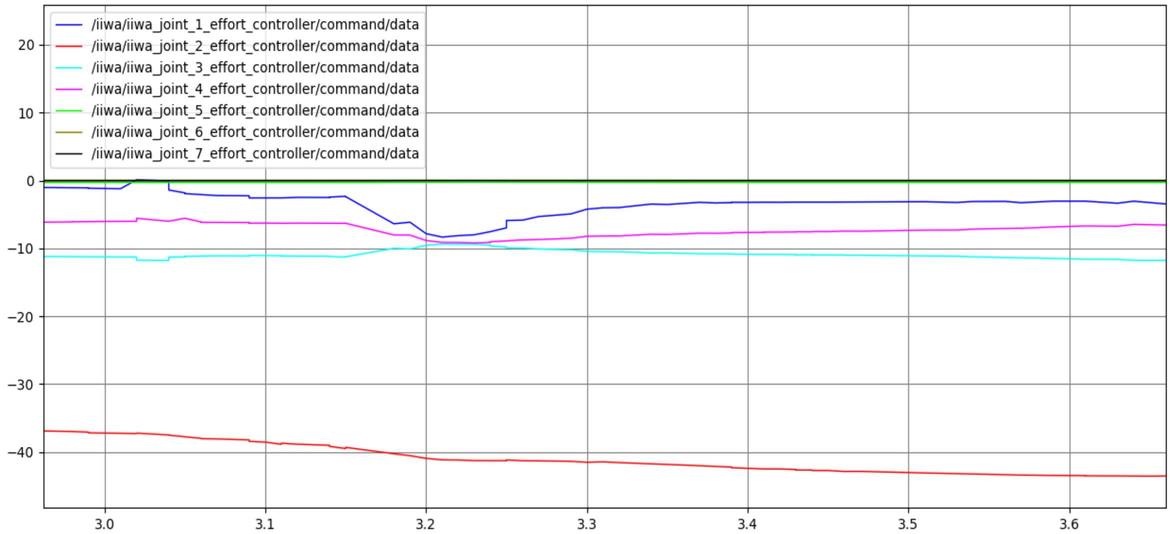
- Trajectory: Linear with trapezoidal velocity profile

$K_{pp} = 70$;

$K_{op} = 66$;

$K_{pd} = 1{,}7 * \sqrt{(K_{pp})}$;

$K_{od} = 2 * \sqrt{(K_{op})}$.



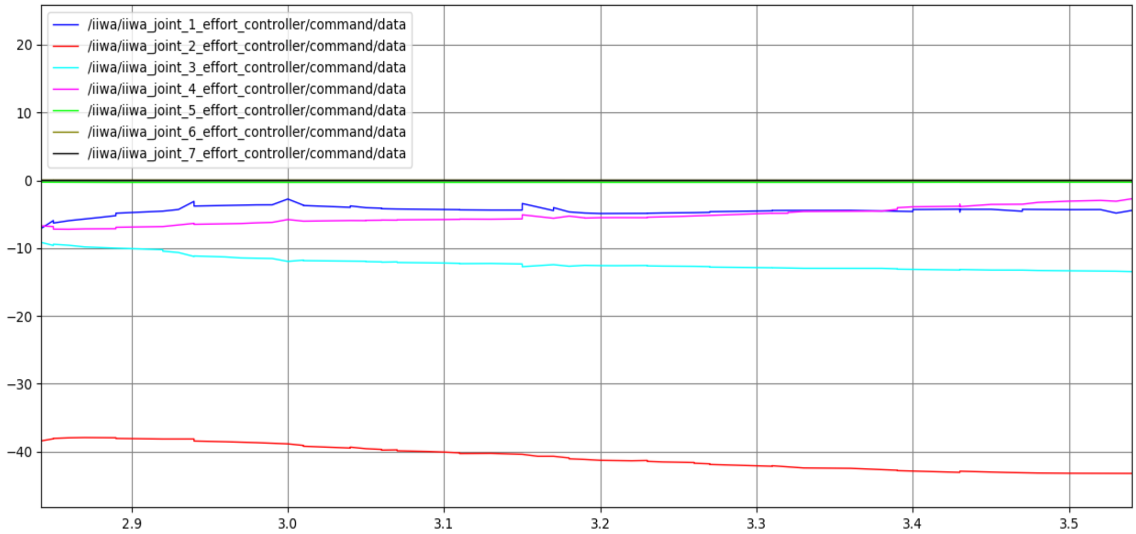
- Trajectory: Linear with cubic polinomial curvilinear abscissa

$K_{pp} = 60$;

$K_{op} = 65$;

$K_{pd} = 1{,}7 * \sqrt{(K_{pp})}$;

$K_{od} = 2 * \sqrt{(K_{op})}$.



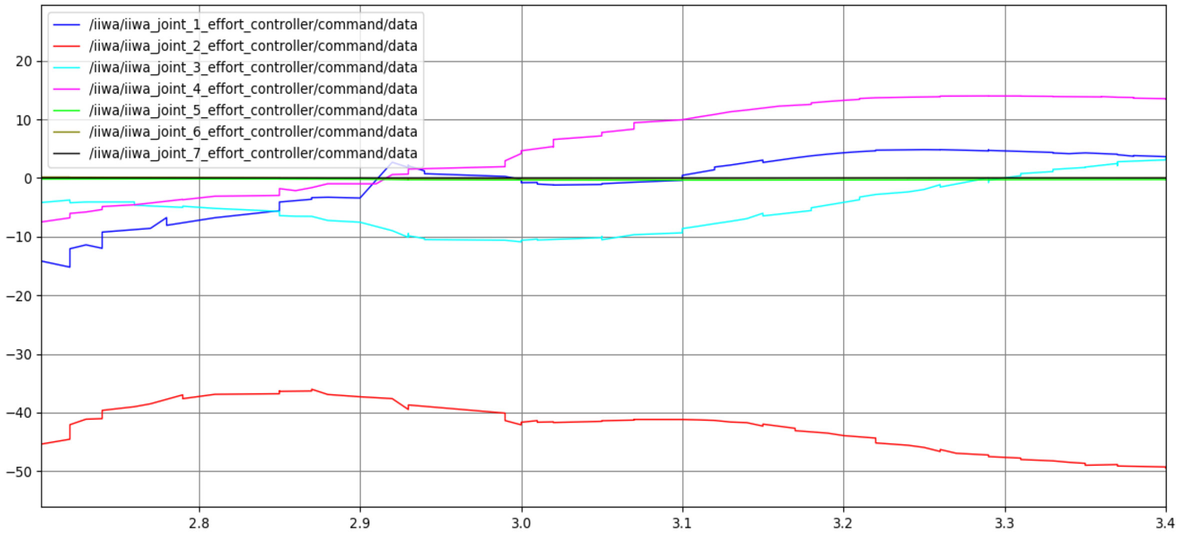
- Trajectory: Circular with cubic polinomial curvilinear abscissa

$K_{pp} = 40$;

$K_{op} = 38$;

$K_{pd} = 1{,}3 * \sqrt{(K_{pp})}$;

$K_{od} = 1{,}5 * \sqrt{(K_{op})}$.



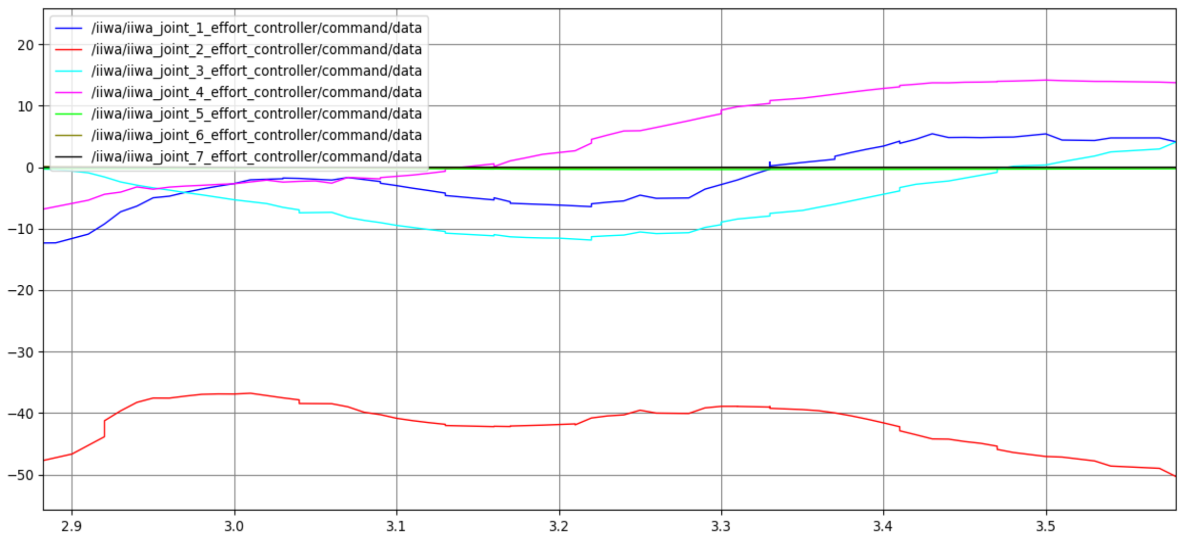
- Trajectory: Circular with trapezoidal velocity profile

$K_{pp} = 30$;

$K_{op} = 30$;

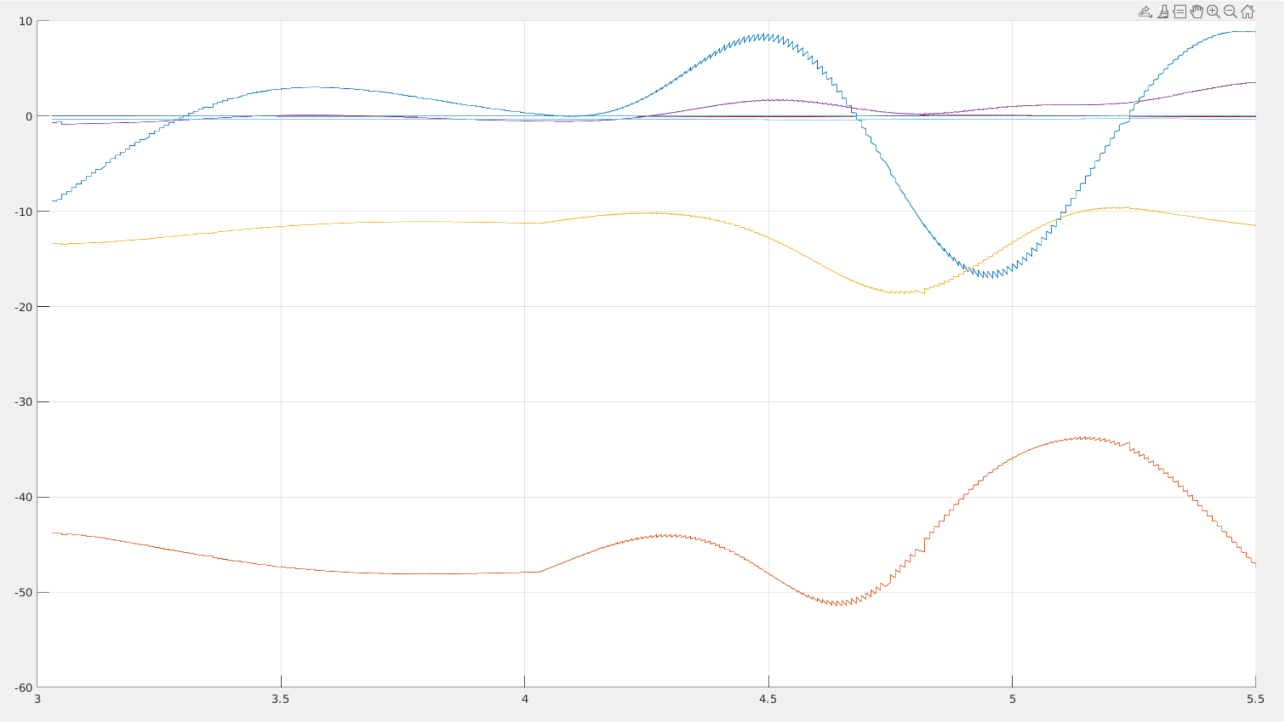
$K_{pd} = 1{,}5 * \sqrt{(K_{pp})}$;

$K_{od} = 1{,}7 * \sqrt{(K_{op})}$.

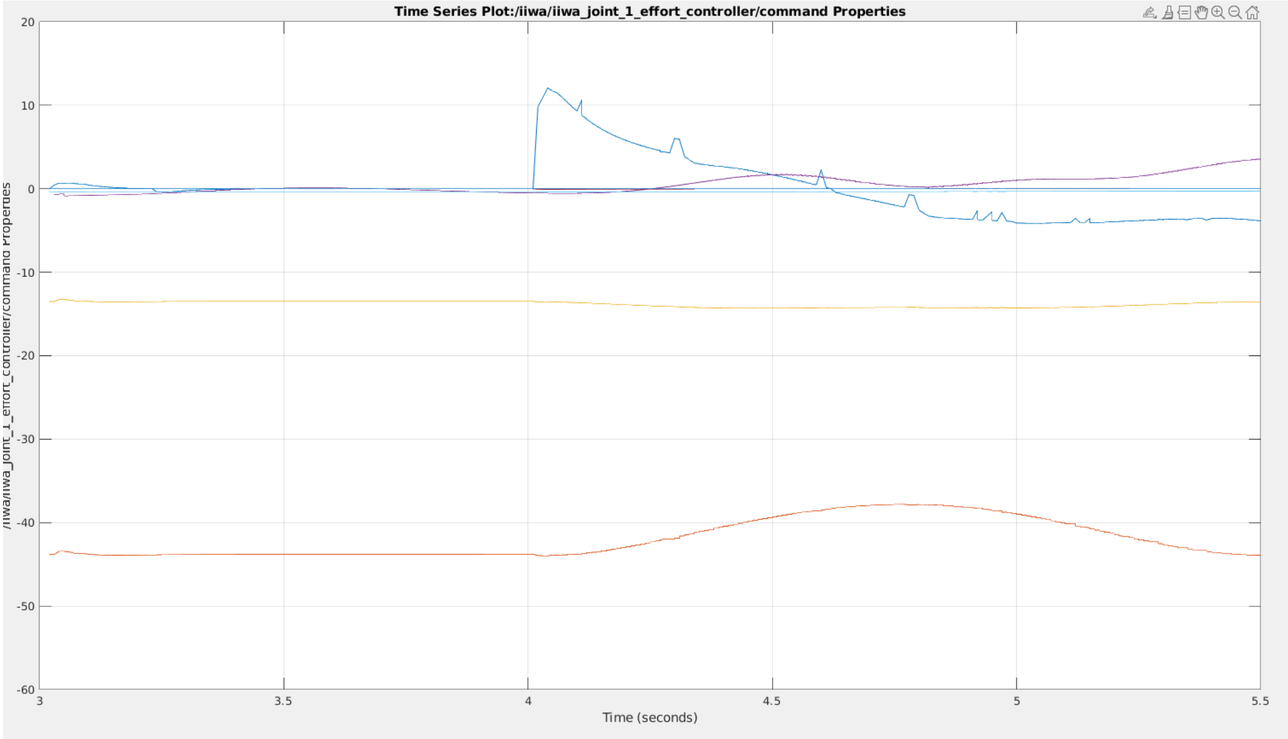


We also plotted the same joint torque commands with MATLAB

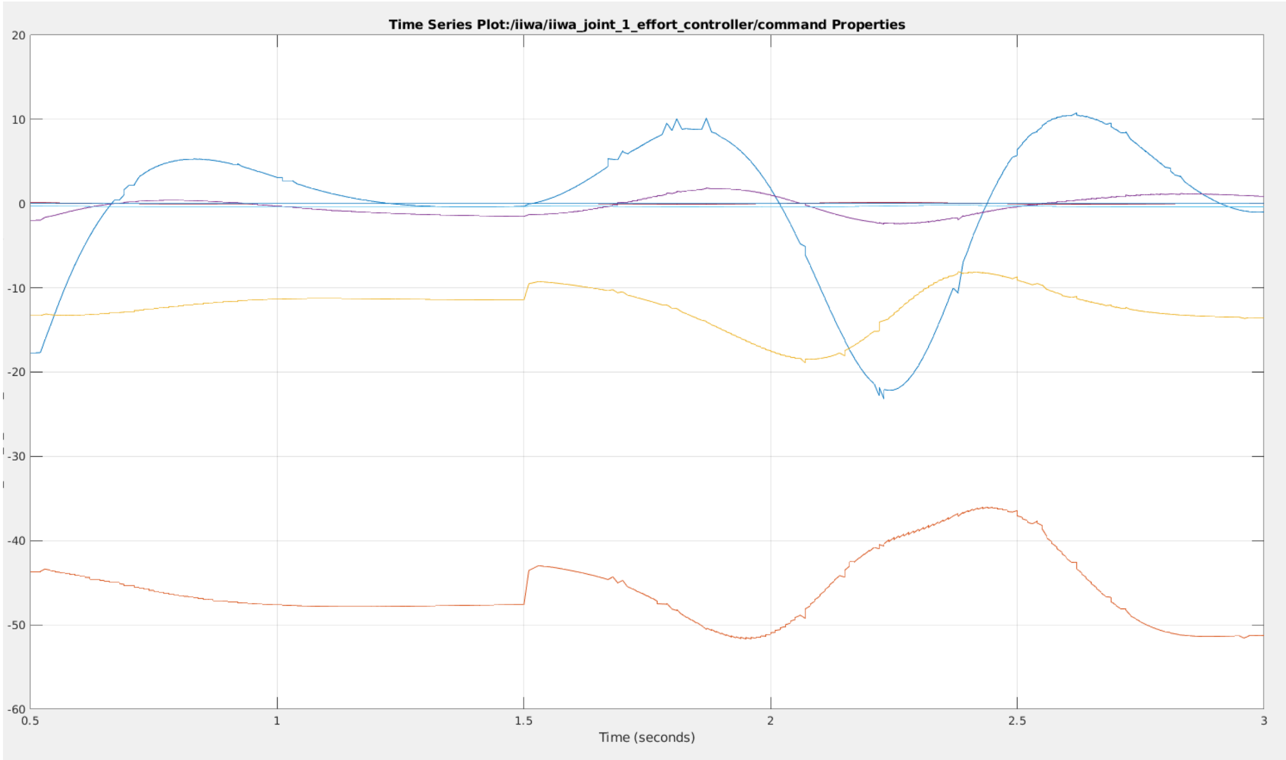
- Trajectory : Linear with trapezoidal velocity profile



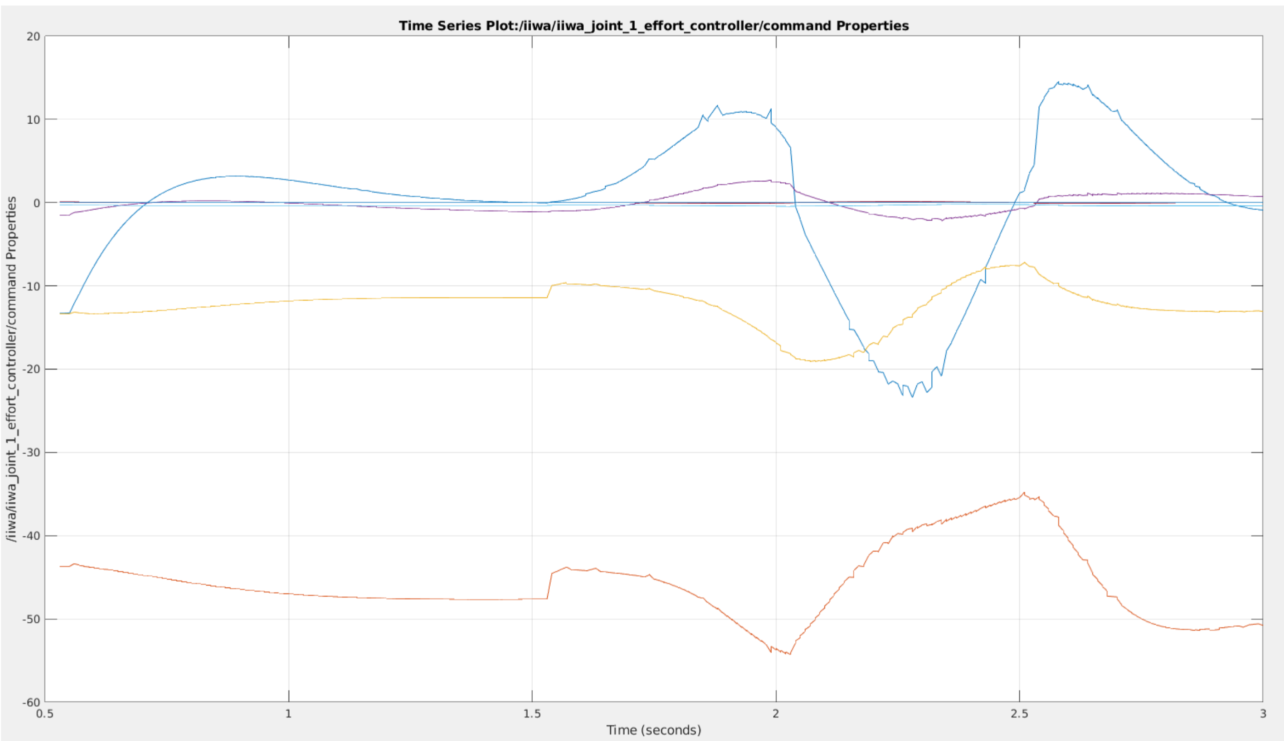
- Trajectory: Linear with cubic polinomial curvilinear abscissa



- Trajectory: Circular with cubic polinomial curvilinear abscissa



- Trajectory: Circular with trapezoidal velocity profile



This is the code used in MATLAB:

```
clear all
close all

bag1 = rosbag('/home/davide/Documents/EJ1.bag');
bag2 = rosbag('/home/davide/Documents/EJ2.bag');
bag3 = rosbag('/home/davide/Documents/EJ3.bag');
bag4 = rosbag('/home/davide/Documents/EJ4.bag');
bag5 = rosbag('/home/davide/Documents/EJ5.bag');
bag6 = rosbag('/home/davide/Documents/EJ6.bag');
bag7 = rosbag('/home/davide/Documents/EJ7.bag');
T1 = select(bag1, 'Topic', '/iiwa/iiwa_joint_1_effort_controller/command');
T2 = select(bag2, 'Topic', '/iiwa/iiwa_joint_2_effort_controller/command');
T3 = select(bag3, 'Topic', '/iiwa/iiwa_joint_3_effort_controller/command');
T4 = select(bag4, 'Topic', '/iiwa/iiwa_joint_4_effort_controller/command');
T5 = select(bag5, 'Topic', '/iiwa/iiwa_joint_5_effort_controller/command');
T6 = select(bag6, 'Topic', '/iiwa/iiwa_joint_6_effort_controller/command');
T7 = select(bag7, 'Topic', '/iiwa/iiwa_joint_7_effort_controller/command');
EJ1 = timeseries(T1);
EJ2 = timeseries(T2);
EJ3 = timeseries(T3);
EJ4 = timeseries(T4);
EJ5 = timeseries(T5);
EJ6 = timeseries(T6);
EJ7 = timeseries(T7);
plot(EJ1)
hold on
plot(EJ2)
plot(EJ3)
plot(EJ4)
plot(EJ5)
plot(EJ5)
plot(EJ6)
plot(EJ7)

grid()
```

You can find all the files at the following GitHub url: https://github.com/peppesagg/Homework2.git

Group members:
Davide Busco
Pietro Falco
Davide Rubinacci
Giuseppe Saggese