Report – Hands-on setup class

Students:

Nimrod Millenium Ndulue

Salvatore Del Peschio

Salvatore Oliviero

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

a) *Modify appropriately the KDLPlanner class (files kdl_planner.h and kdl_planner.cpp) that provides a basic interface for trajectory creation. First, define a new KDLPlanner :: trapezoidal_vel function that takes the current time t and the acceleration time t c as double arguments and returns three double variables s, ṡ and s̈ that represent the curvilinear abscissa of your trajectory 1. Remember: a trapezoidal velocity profile for a curvilinear abscissa s ∈ [0, 1] is defined as follows:*

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

where t_c is the acceleration duration variable while ṡ(t) and s̈(t) can be easily retrieved calculating time derivative of (1).

We have added the function prototype in the KDLplanner class:

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

subsequently, implemented it into the kdl_planner.cpp:

```
void KDLPlanner::trapezoidal_vel(double t, double tc, double &s, double &s_dot, double &s_ddot){
  // Trapezoidal profile s : [0, 1] //

  double s_ddot_c = -1.0 / (std::pow(tc, 2) - trajDuration_*tc);
  if (t <= tc){
    s = 0.5*s_ddot_c*std::pow(t, 2);
    s_dot = s_ddot_c*t;
    s_ddot = s_ddot_c;
  }
  else if (t <= trajDuration_-tc){
    s = s_ddot_c*tc*(t - tc/2);
    s_dot = s_ddot_c*tc;
    s_ddot = 0;
  }
  else {
    s = 1 - 0.5*s_ddot_c*std::pow(trajDuration_ - t, 2);
    s_dot = s_ddot_c*(trajDuration_ - t);
    s_ddot = -s_ddot_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, ṡ and s̈ that represent the curvilinear abscissa of your trajectory. Remember, a cubic polinomial is defined as follows:*

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

Where coefficients a_3, a_2, a_1, a_0 must be calculated offline imposing boundary conditions, while ṡ(t) and s̈(t) can be easily retrieved calculating time derivative of (2).

As for the previous point, we declared the prototype of function cubic_polinomial() inside the KDLplanner class:

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

In the implementation, the curvilinear abscissa is computed solving the following equations:

$$\begin{cases} a_0 = q_i \\ a_1 = \dot{q}_\iota \\ a_3 t_f^2 + a_2 t_f^2 + a_1 t_f + a_0 = q_f \\ 3a_3 t_f^2 + 2a_2 t_f + a_1 = \dot{q}_f \end{cases}$$

Imposing $q_i = 0, \dot{q}_1 = 0, q_f = 1, \dot{q}_f = 0$ we obtain the subsequent values of the parameters:

$$\begin{cases} a_0 = 0 \\ a_1 = 0 \\ a_2 = \dfrac{3}{t_f^2} \\ a_3 = -\dfrac{2}{t_f^3} \end{cases}$$

The implementation of the above equations is shown below:

```
void KDLPlanner::cubic_polinomial(double t, double &s, double &s_dot, double &s_ddot){
  // Cubic profile s : [0, 1] // TODO: final and initial vel

  double a_3 = -2.0 / std::pow(trajDuration_, 3);
  double a_2 = 3.0 / std::pow(trajDuration_, 2);
  double a_1 = 0;
  double a_0 = 0;

  s = a_3*std::pow(t, 3) + a_2*std::pow(t, 2) + a_1*t + a_0;
  s_dot = 3*a_3*std::pow(t, 2) + 2*a_2*t + a_1;
  s_ddot = 6*a_3*t + 2*a_2;

}
```

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 _trajRadiusof your trajectory and store them in the corresponding class variables (to be created in thekdl_planner.h).*

To define a new constructor within the KDLPlanner class, we inserted the prototype in the class definition adding a default value for the acceleration duration considering the use of a cubic profile in the next sections of the homework.

```
KDLPlanner(Eigen::Vector3d _trajInit, double _trajRadius, double _trajDuration, double _accDuration = 0.5);
```

The implementation of the constructor is written in the kdl_planner.cpp as follows:

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

  trajDuration_ = _trajDuration;
  accDuration_ = _accDuration;
  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. Remember that a circular path in the $y - z$ plane can be easily defined as follows.*

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

We have chosen a modular approach to improve the readability of our code developing the function to generate the circular trajectory that is agnostic to the velocity profile, Below, are reported the prototypes and implementation of the functions.

```
trajectory_point compute_circ_traj(const double s, const double s_dot,const double s_ddot);
```

```
trajectory_point KDLPlanner::compute_circ_traj(const double s, const double s_dot,const double s_ddot){
  // Circular trajectory point //

  trajectory_point traj;

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

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

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

  return traj;

}
```

```cpp
trajectory_point KDLPlanner::compute_circ_traj_trap_prof(const double time){
  // Circular trajectory point with trapezoidal profile //

  double s, s_dot, s_ddot;
  trapezoidal_vel(time, accDuration_, s, s_dot, s_ddot);
  return compute_circ_traj(s, s_dot, s_ddot);

}

trajectory_point KDLPlanner::compute_circ_traj_cubic_prof(const double time){
  // Circular trajectory point with cubic profile //

  double s, s_dot, s_ddot;
  cubic_polinomial(time, s, s_dot, s_ddot);
  return compute_circ_traj(s, s_dot, s_ddot);

}
```

The circular trajectory is described in terms of position, velocity, and acceleration.

- The position along the x-axis of the circular trajectory is constant and equal to the initial position.
- The positions along the y-axis and z-axis are calculated with the given formulas.
- The velocity and acceleration along the y and z axes are calculated using the derivatives of circular motion.

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

We used the same approach previously described; through the following formulas we compute the trajectory point at a given time:

$$p = p_i + s\left(p_f - p_i\right) \qquad \dot{p} = \dot{s}\left(p_f - p_i\right) \qquad \ddot{p} = \ddot{s}\left(p_f - p_i\right)$$

```cpp
trajectory_point compute_rect_traj(const double s, const double s_dot,const double s_ddot);
```

```cpp
trajectory_point KDLPlanner::compute_rect_traj(const double s, const double s_dot,const double s_ddot){
  // Rectilinear trajectory point with cubic profile //

  trajectory_point traj;

  Eigen::Vector3d delta_p = trajEnd_ - trajInit_;

  traj.pos = trajInit_ + s*delta_p;
  traj.vel = s_dot*delta_p;
  traj.acc = s_ddot*delta_p;

  return traj;

}
```

```cpp
trajectory_point KDLPlanner::compute_rect_traj_trap_prof(const double time){
  // Circular rectilinear point with trapezoidal profile

  double s, s_dot, s_ddot;
  trapezoidal_vel(time, accDuration_, s, s_dot, s_ddot);
  return compute_rect_traj(s, s_dot, s_ddot);

}

trajectory_point KDLPlanner::compute_rect_traj_cubic_prof(const double time){
  // Circular rectilinear point with cubic profile //

  double s, s_dot, s_ddot;
  cubic_polinomial(time, s, s_dot, s_ddot);
  return compute_rect_traj(s, s_dot, s_ddot);

}
```

3. Test the four trajectories

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

Before any change in the main program, we implemented the getInverseKynematics function to obtain the velocity and acceleration reference in the joint space.

```cpp
void KDLRobot::getInverseKinematics(KDL::Frame &f,
                                     KDL::Twist &twist,
                                     KDL::Twist &acc,
                                     KDL::JntArray &q,
                                     KDL::JntArray &dq,
                                     KDL::JntArray &ddq){
    q = getInvKin(q,f);
    ikVelSol_->CartToJnt(q, twist, dq);
    Eigen::Matrix<double,6,7> J = toEigen(getEEJacobian());
    Eigen::Matrix<double,6,1> x_ddot = toEigen(acc);
    Eigen::Matrix<double,6,1> Jdot_qdot = getEEJacDotqDot();
    Eigen::Matrix<double,7,6> Jpinv = pseudoinverse(J);
    ddq.data = Jpinv*(x_ddot - Jdot_qdot);
}
```

For the purpose of testing the rectilinear trajectories, we need to create a planner instance as follows:

```
// Plan trajectory
double traj_duration = 1.5, acc_duration = 0.5, t = 0.0, init_time_slot = 1.0, traj_radius = 0.15;
KDLPlanner planner(init_position, end_position, traj_duration, acc_duration); // rectilinear traj
```

Furthermore, we also called the methods to obtain the trajectory's point at execution time.

```
// Extract desired pose
des_cart_vel = KDL::Twist::Zero();
des_cart_acc = KDL::Twist::Zero();
if (t <= init_time_slot) // wait a second
{
    p = planner.compute_rect_traj_trap_prof(0.0);
}
else if(t > init_time_slot && t <= traj_duration + init_time_slot)
{
    p = planner.compute_rect_traj_trap_prof(t-init_time_slot);
    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());
}
```

```
// Retrieve the first trajectory point
trajectory_point p = planner.compute_rect_traj_trap_prof(t);
```

In the case of a circular trajectory, we utilize the subsequent constructor of the class and call the compute_circ_traj_trap_prof and compute_circ_traj_cubic_prof methods.

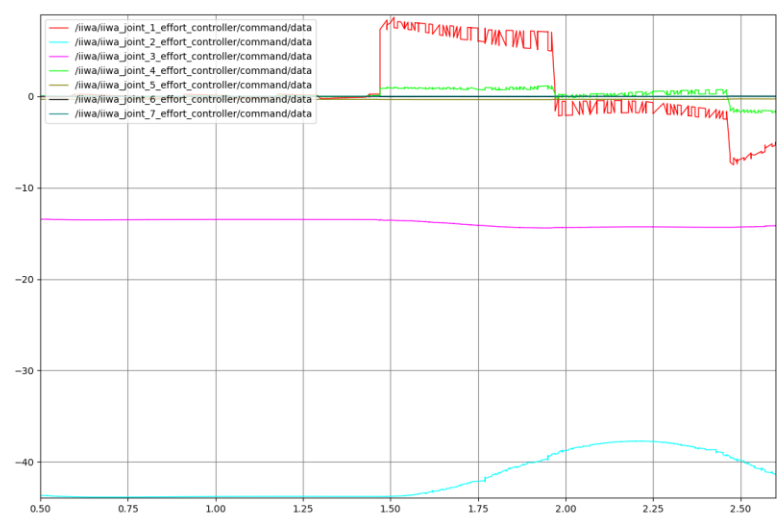
```
KDLPlanner planner(init_position, traj_radius , traj_duration, acc_duration); // circular traj
```

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*

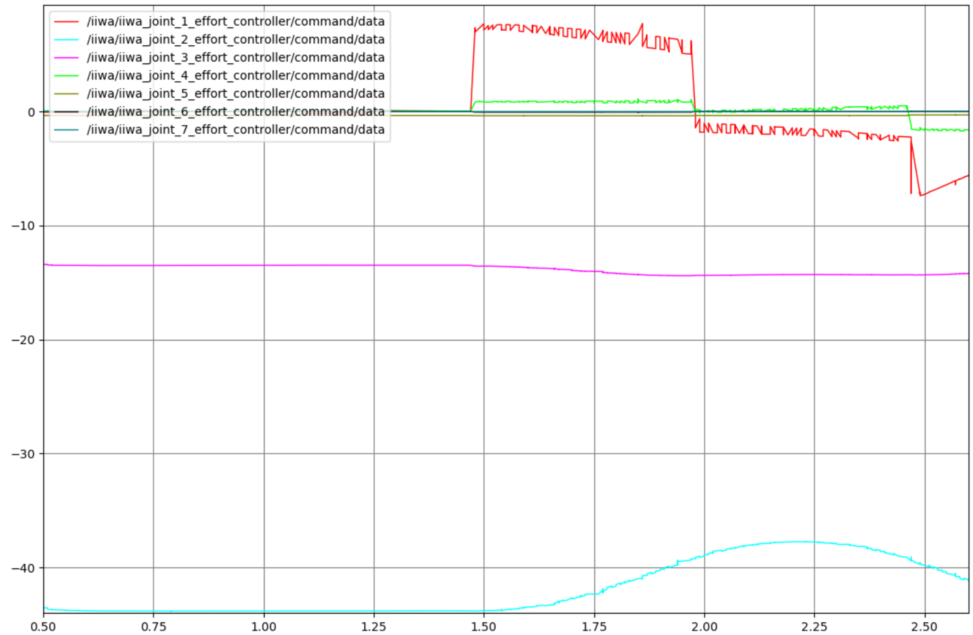
In the Linux terminal we open a rqt_plot window with the command

```
$ rqt_plot
```

After we selected the correct topics to plot for the default gains values we obtained:



Tuning with a trial-and-error approach the gains we were able to acquire have a value of $K_P = 25$ and $K_D = \sqrt{K_P}$ and ensure a smooth torques behavior. It is important to notice that a less aggressive control in the reference tracking corresponds to a bigger error.



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

Using the following bash script, we recorded the torques on the joint_effort_controller topics in a joint_torques.bag file.

#!/bin/bash

```
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

-o joint_torques.bag
```

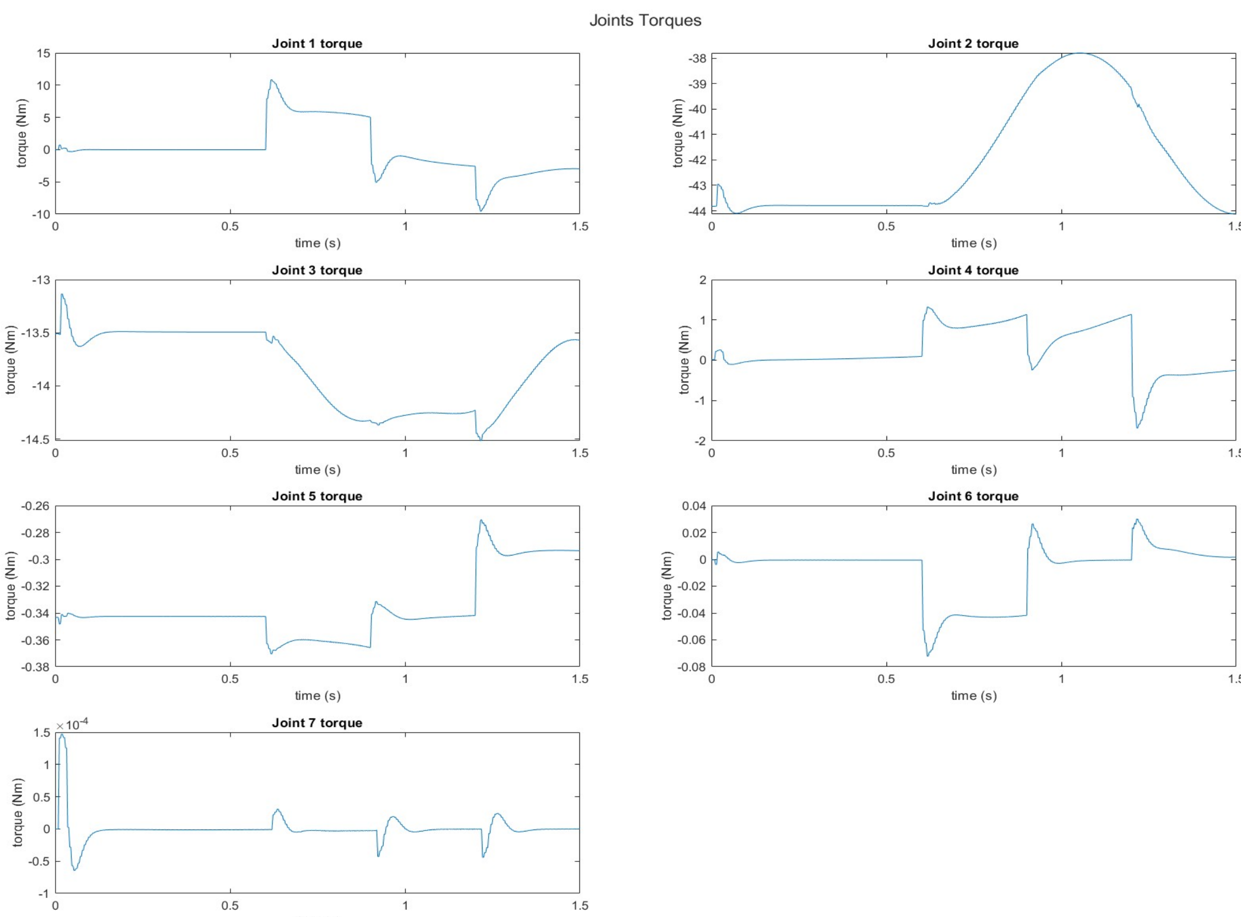
This gives us the possibility through MATLAB to plot the torques using the subsequent script.

```
bag = rosbag('joint_torques.bag');

jt_1 = select(bag,'Topic','iiwa/iiwa_joint_1_effort_controller/command');
msgStructs = readMessages(jt_1,'DataFormat','struct');
jt_1_double = cellfun(@(m) double(m.Data),msgStructs);
jt_2 = select(bag,'Topic','iiwa/iiwa_joint_2_effort_controller/command');
msgStructs = readMessages(jt_2,'DataFormat','struct');
jt_2_double = cellfun(@(m) double(m.Data),msgStructs);
jt_3 = select(bag,'Topic','iiwa/iiwa_joint_3_effort_controller/command');
msgStructs = readMessages(jt_3,'DataFormat','struct');
jt_3_double = cellfun(@(m) double(m.Data),msgStructs);
jt_4 = select(bag,'Topic','iiwa/iiwa_joint_4_effort_controller/command');
msgStructs = readMessages(jt_4,'DataFormat','struct');
jt_4_double = cellfun(@(m) double(m.Data),msgStructs);
jt_5 = select(bag,'Topic','iiwa/iiwa_joint_5_effort_controller/command');
msgStructs = readMessages(jt_5,'DataFormat','struct');
jt_5_double = cellfun(@(m) double(m.Data),msgStructs);
jt_6 = select(bag,'Topic','iiwa/iiwa_joint_6_effort_controller/command');
msgStructs = readMessages(jt_6,'DataFormat','struct');
jt_6_double = cellfun(@(m) double(m.Data),msgStructs);
jt_7 = select(bag,'Topic','iiwa/iiwa_joint_7_effort_controller/command');
msgStructs = readMessages(jt_7,'DataFormat','struct');
jt_7_double = cellfun(@(m) double(m.Data),msgStructs);

jt_double = [jt_1_double, jt_2_double, jt_3_double, jt_4_double, jt_5_double, jt_6_double, jt_7_double];
t = linspace(0, 1.5, size(jt_1_double, 1));

t_tay = tiledlayout(4,2);
t_tay.Title.String = "Joints Torques";
t_tay.Padding = "tight";
for i = 1 : size(jt_double, 2)
    nexttile;
    plot(t, jt_double(:,i))
    title('Joint ' + string(i) + ' torque')
    xlabel("time (s)")
    ylabel("torque (Nm)")
end
```



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.*

```
Eigen::VectorXd KDLController::idCntr(KDL::Frame &_desPos,
                                      KDL::Twist &_desVel,
                                      KDL::Twist &_desAcc,
                                      double _Kpp, double _Kpo,
                                      double _Kdp, double _Kdo,
                                      double &error){
    /// Inverse Dynamic Control in operational space ///

    // calculate gain matrices
    Eigen::Matrix<double,6,6> Kp = Eigen::Matrix<double,6,6>::Zero();
    Eigen::Matrix<double,6,6> Kd = Eigen::Matrix<double,6,6>::Zero();
    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 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> B = robot_->getJsim();
    // Eigen::Matrix<double,7,6> Jpinv = weightedPseudoInverse(B,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,6,1>::Zero();
    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 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>::Zero();
    Eigen::Matrix<double,6,1> dot_x_tilde = Eigen::Matrix<double,6,1>::Zero();
    x_tilde << e_p, e_o;
    error = e_p.norm();
    dot_x_tilde << dot_e_p, dot_e_o;  //-omega_e
    dot_dot_x_d << dot_dot_p_d, dot_dot_r_d;
```

We have uncommented the code previously written in the idCntr method and made some changes to allow it work correctly. For every variable we ensure a zero initialization. The available functions getEEJacobian, getEEFrame and getEEVelocity were used in place of the unimplemented ones. In addition, we computed the norm of the position error and update it as a function argument passed by reference.

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 ˙x, and acceleration ¨x, retrieve the current joint space inertia matrix M and the Jacobian (compute the analytic Jacobian) and its time derivative, compute the linear ep 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*

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

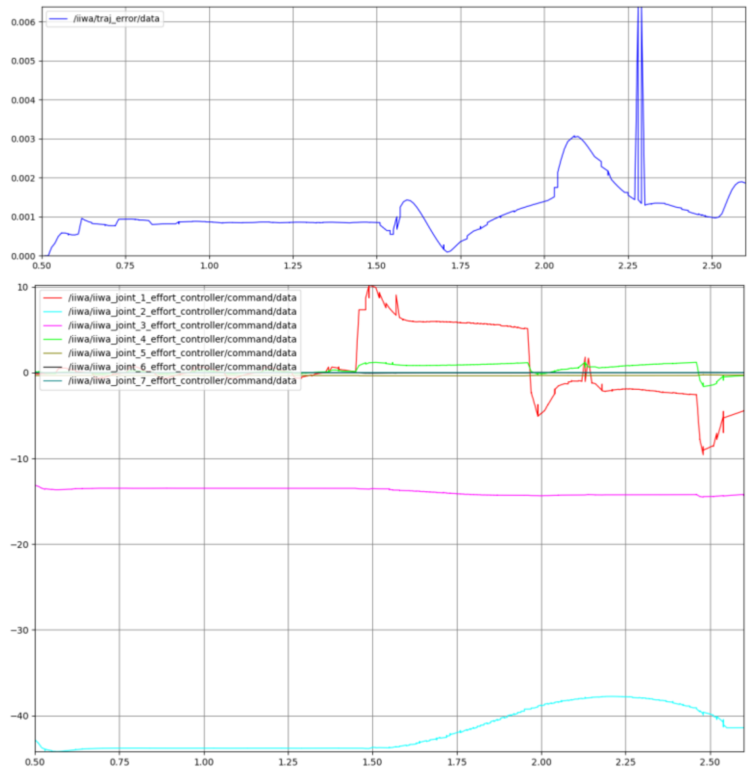
```
// inverse dynamics
Eigen::Matrix<double,6,1> y = Eigen::Matrix<double,6,1>::Zero();
Eigen::Matrix<double,6,1> J_dot_q_dot = robot_->getEEJacDotqDot();
y << (dot_dot_x_d - J_dot_q_dot + Kd*dot_x_tilde + Kp*x_tilde);
Eigen::Matrix<double,7,1> tau = Eigen::Matrix<double,7,1>::Zero();
tau = B*(Jpinv*y) + robot_->getGravity() + robot_->getCoriolis();
```

Continuing the discussion started in the previous section, once obtained the values of the errors, the current joint space inertia matrix and Jacobian, we were able to compute the correct torques for the manipulator's joints to achieve the trajectory tracking.

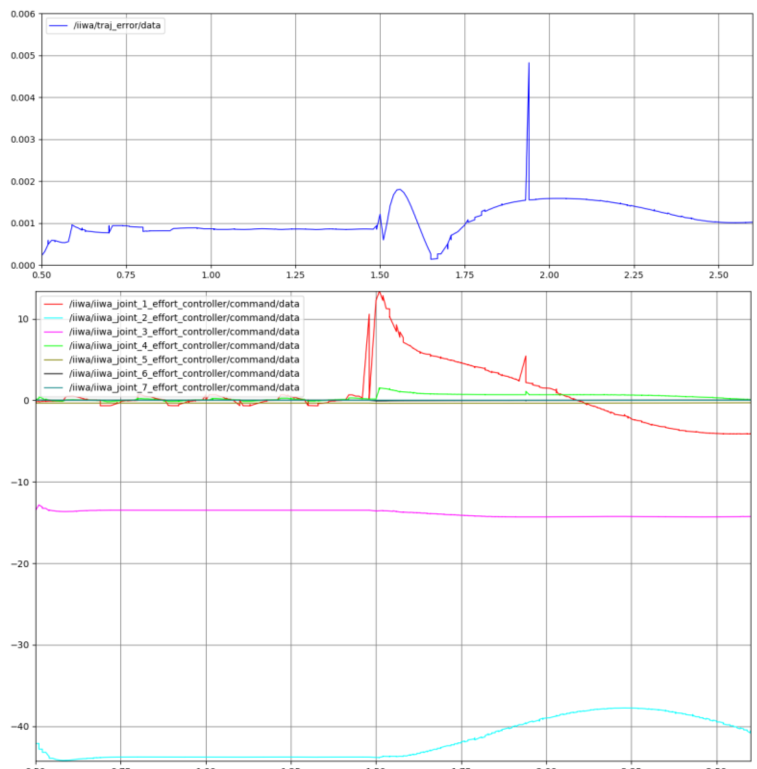
c) *Test the controller along the planned trajectories and plot the corresponding joint torque commands.*

In the same way of the third section, we plotted the joints torques and position errors for the purpose of evaluate the performance of our control. The result obtained are quite satisfactory with an error of about half centimeter and smooth enough torques for most of the cases.

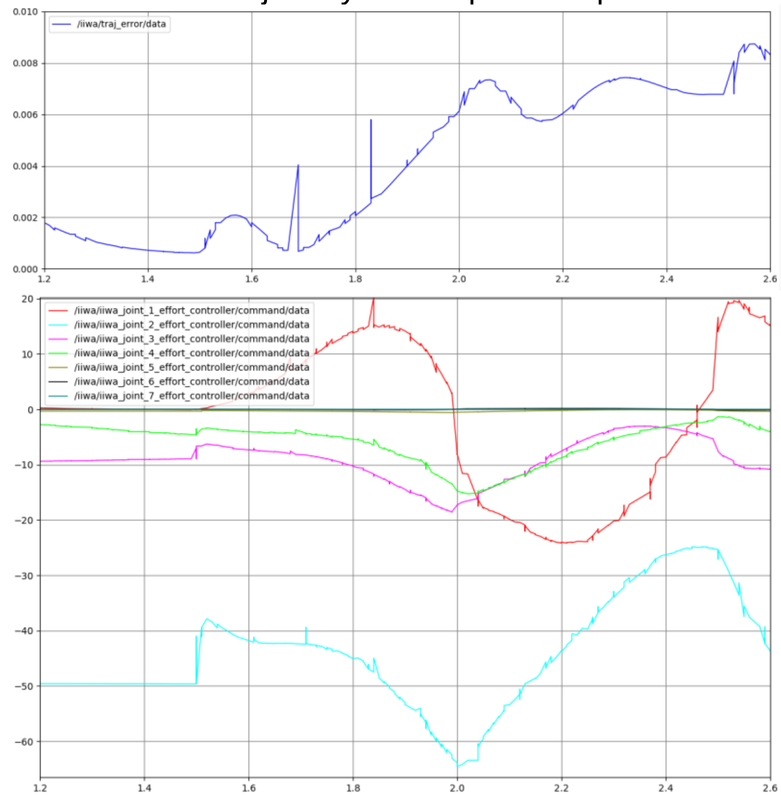
Rectilinear trajectory with trapezoidal profile



Rectilinear trajectory with cubic profile



Circular trajectory with trapezoidal profile



Circular trajectory with cubic profile