Report
Robotics Lab: Homework 2
Control a manipulator to follow a trajectory

Students:

Andrea De Pisapia P38000149

Gabriele Palmieri P38000164

Antonio Setola P38000132

Clara Solli P38000135

Git

Here is the link to….
https://github.com/antoniosetola/Homewor2.git

1. Substitute the current trapezoidal velocity profile with a cubic polynomial 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, ˙s and ¨s that represent the curvilinear abscissa of your trajectory1. Remember: a trapezoidal velocity profile for a curvilinear abscissa s ∈ [0, 1] is defined as follows:**
>
> **(1)**
>
> **where tc is the acceleration duration variable while ˙s(t) and ¨s(t) can be easily retrieved calculating time derivative of (1).**

In **kdl_planner.h** a struct vel_profile was defined, with position, velocity and acceleration:

```
struct vel_profile{
   double s=0; //pos
   double dots=0; //vel
   double ddots=0; //acc
};
```

and in KDL class (public) the prototype of the function is defined.

```
void trapezoidal_vel(double time, double accDuration, vel_profile
&vel_prof);
```

In **kdl_planner.cpp,** the function that returns the velocity profile is developed as follows:

```
void KDLPlanner::trapezoidal_vel(double time, double tc, vel_profile
&vel_prof)
{
/*  trapezoidal velocity profile with accDuration_ acceleration time
period and trajDuration_ total duration.
    time = current time
    trajDuration_   = final time
    tc    = acceleration time
    trajInit_ = trajectory initial point
    trajEnd_   = trajectory final point */

 double ddot_sc = -1.0/(std::pow(tc,2)-trajDuration_*tc);

if(time <= accDuration_)
```

```
 {
   vel_prof.s = 0.5*ddot_sc*std::pow(time,2);
   vel_prof.dots = ddot_sc*time;
   vel_prof.ddots = ddot_sc;
 }
 else if(time <= trajDuration_-tc)
 {
   vel_prof.s = ddot_sc*tc*(time-tc/2);
   vel_prof.dots = ddot_sc*tc;
   vel_prof.ddots = 0;
 }
 else
 {
   vel_prof.s = 1 - 0.5*ddot_sc*std::pow(trajDuration_-time,2);
   vel_prof.dots = ddot_sc*(trajDuration_-time);
   vel_prof.ddots = -ddot_sc;
 }


}
```

(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.
Remember, a cubic polynomial is defined as follows
(2)
where coefficients a3, a2, a1, a0 must be calculated offline imposing boundary conditions, while $\dot{s}(t)$ and $\ddot{s}(t)$ can be easily retrieved calculating time derivative of (2).

In the same way as 1a), in **kdl_planner.h** was defined the prototype of the funcion, returning the velocity profile, while in **kdl_planner.cpp** the function was developed by setting a0, a1, a2, a3 properly.

In **kdl_planner.h:**
In class (public):

```
void cubic_polinomial(double time, vel_profile &vel_prof);
```

In **kdl_planner.cpp:**

```
void KDLPlanner::cubic_polinomial(double time, vel_profile &vel_prof){
/*  trapezoidal velocity profile with accDuration_ acceleration time
period and trajDuration_ total duration.
    time = current time
    trajDuration_  = final time
    tc   = acceleration time
```

```
   trajInit_ = trajectory initial point
   trajEnd_  = trajectory final point */

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

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

The new constructor was defined in **kdl_planner.h** (and in private trajRadius_ was added).

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

```
private:

  …
   double trajDuration_, accDuration_, trajRadius_;
   Eigen::Vector3d trajInit_, trajEnd_;
};
```

In kdl_planner.cpp the constructor was developed:

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

Remember that a circular path in the y − z plane can be easily defined as follows:

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

In kdl_planner.h (public) the prototype of function `compute_trajectory_circ` was defined:

```
trajectory_point compute_trajectory_circ(double time, vel_profile
&vel_prof);
```

while in **kdl_planner.cpp** the function was developed **(vel_profile can be chosen as trapezoidal velocity profile or cubic polynomial by commenting or decommenting the highlighted lines);** the function returns the computed trajectory (traj).

```
trajectory_point KDLPlanner::compute_trajectory_circ(double time,
vel_profile &vel_prof)
{
double pi=3.14;
/* trapezoidal velocity profile with accDuration_ acceleration time
period and trajDuration_ total duration.
   time = current time
   trajDuration_  = final time
   accDuration_   = acceleration time
   trajInit_ = trajectory initial point
   trajEnd_  = trajectory final point */

trapezoidal_vel(time, accDuration_, vel_prof); // choose between
trapezoidal or cubic_polinomial velocity profile

//cubic_polinomial(time, vel_prof);

trajectory_point traj; // def. the trajectory traj

Eigen::Vector3d ddot_sc =
-1.0/(std::pow(accDuration_,2)-trajDuration_*accDuration_)*(trajEnd_-tr
ajInit_); // max acceleration

 traj.pos=
Eigen::Vector3d(trajInit_(0),trajInit_(1)-trajRadius_*cos(2*pi*vel_prof
.s),trajInit_(2)-trajRadius_*sin(2*pi*vel_prof.s)); // pos

traj.vel =
Eigen::Vector3d(0,trajRadius_*2*pi*vel_prof.dots*sin(2*pi*vel_prof.s),-
trajRadius_*2*pi*vel_prof.dots*cos(2*pi*vel_prof.s)); // vel
```

```
 traj.acc =
Eigen::Vector3d(0,trajRadius_*2*pi*(vel_prof.ddots*sin(2*pi*vel_prof.s)
+2*pi*(std::pow(vel_prof.dots,2))*cos(2*pi*vel_prof.s)),trajRadius_*2*p
i*(-vel_prof.ddots*cos(2*pi*vel_prof.s)+2*pi*(std::pow(vel_prof.dots,2)
)*sin(2*pi*vel_prof.s))); // acc
 return traj;
}
```

(c) Do the same for the linear trajectory.

In kdl_planner.h the prototype of the function was defined.

```
trajectory_point compute_trajectory_lin(double time, vel_profile
&vel_prof);
```

In **kdl_planner.cpp: (vel_profile can be chosen as trapezoidal velocity profile or cubic polinomial by commenting or decommenting the highlighted lines)**

```
trajectory_point KDLPlanner::compute_trajectory_lin(double time,
vel_profile &vel_prof)
{
/* trapezoidal velocity profile with accDuration_ acceleration time
period and trajDuration_ total duration.
   time = current time
   trajDuration_  = final time
   accDuration_   = acceleration time
   trajInit_ = trajectory initial point
   trajEnd_  = trajectory final point */

//trapezoidal_vel(time, accDuration_, vel_prof);
cubic_polinomial(time, vel_prof);

trajectory_point traj;

Eigen::Vector3d ddot_sc =
-1.0/(std::pow(accDuration_,2)-trajDuration_*accDuration_)*(trajEnd_-tr
ajInit_);

 traj.pos =
Eigen::Vector3d(trajInit_(0),trajInit_(1),trajInit_(2)+vel_prof.s/5);
 traj.vel = Eigen::Vector3d(0,0,vel_prof.dots/5);
 traj.acc = Eigen::Vector3d(0,0,vel_prof.ddots/5);

 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 polynomial curvilinear abscissa. Modify your main file kdl_robot_test.cpp and test the four trajectories with the provided joint space inverse dynamics controller.

Before testing the trajectories, a *getdesVel* function was developed: its output is the desired velocity in joint space.

In kdl_robot_test.cpp:

```
dqd = robot.getDesVel(des_cart_vel, J);
```

In kdl_robot.cpp:

```
KDL::JntArray KDLRobot::getDesVel(const KDL::Twist &des_cart_vel,
                       const KDL::Jacobian &J)
{
   KDL::JntArray jntArray_out_;
   jntArray_out_.resize(chain_.getNrOfJoints());
   Eigen::Matrix<double,6,7> J_ee = toEigen(J);
   Eigen::Matrix<double,7,6> Jpinv = pseudoinverse(J_ee);
   Eigen::Matrix<double,6,1> des_vel = toEigen(des_cart_vel);
   Eigen::VectorXd dot_qd = Jpinv * des_vel;
   jntArray_out_.data = dot_qd;

   return jntArray_out_;
}
```

Testing the trajectories:

To set the desired computation for trajectory, it will be necessary to comment/decomment the following lines inside **kdl_robot_test.cpp:**

```
   // Plan trajectory
   double traj_duration = 10, acc_duration = 1.5, t = 0.0,
init_time_slot = 0.0,traj_radius=0.1;
   KDLPlanner planner(traj_duration, init_position, traj_radius);
//decomment when using circular trajectory

//KDLPlanner planner(traj_duration, acc_duration, init_position,
end_position);      //decomment when using linear trajectory

   // Retrieve the first trajectory point
   vel_profile vel_prof;

//trajectory_point p = planner.compute_trajectory(t);
trajectory_point p = planner.compute_trajectory_circ(t, vel_prof);
//decomment when using circular trajectory
```

```
//trajectory_point p = planner.compute_trajectory_lin(t, vel_prof);
//decomment when using  linear trajectory
```

Furthermore, it will be necessary to switch between the calls:
`compute_trajectory/compute_trajectory_circ/compute_trajectory_lin`
inside the *while.*

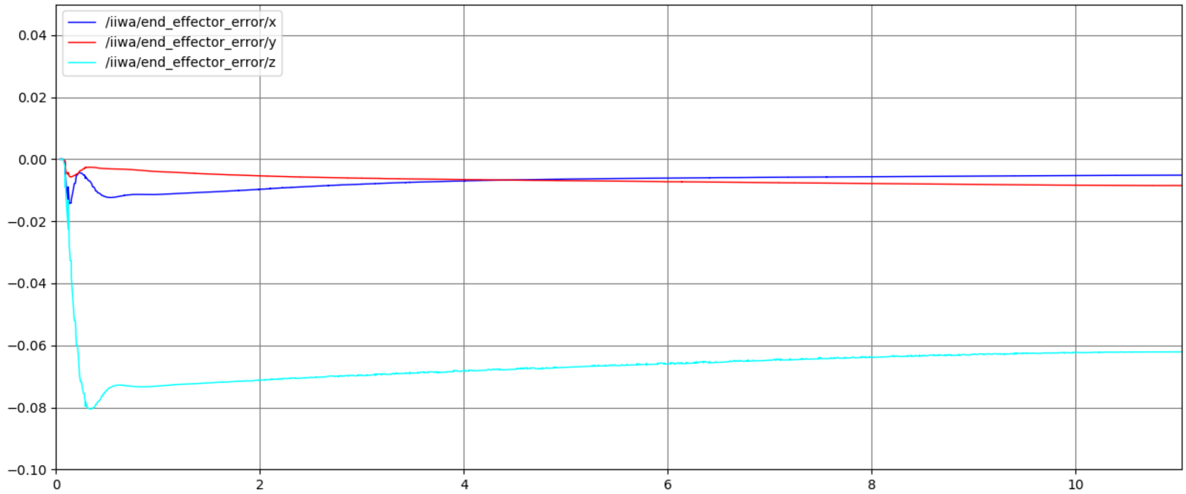
> **(b) Plot the torques sent to the manipulator and tune appropriately the control gains Kp and Kd until you reach a satisfactorily smooth behavior. You can use rqt_plot to visualize your torques at each run, save the screenshot.**

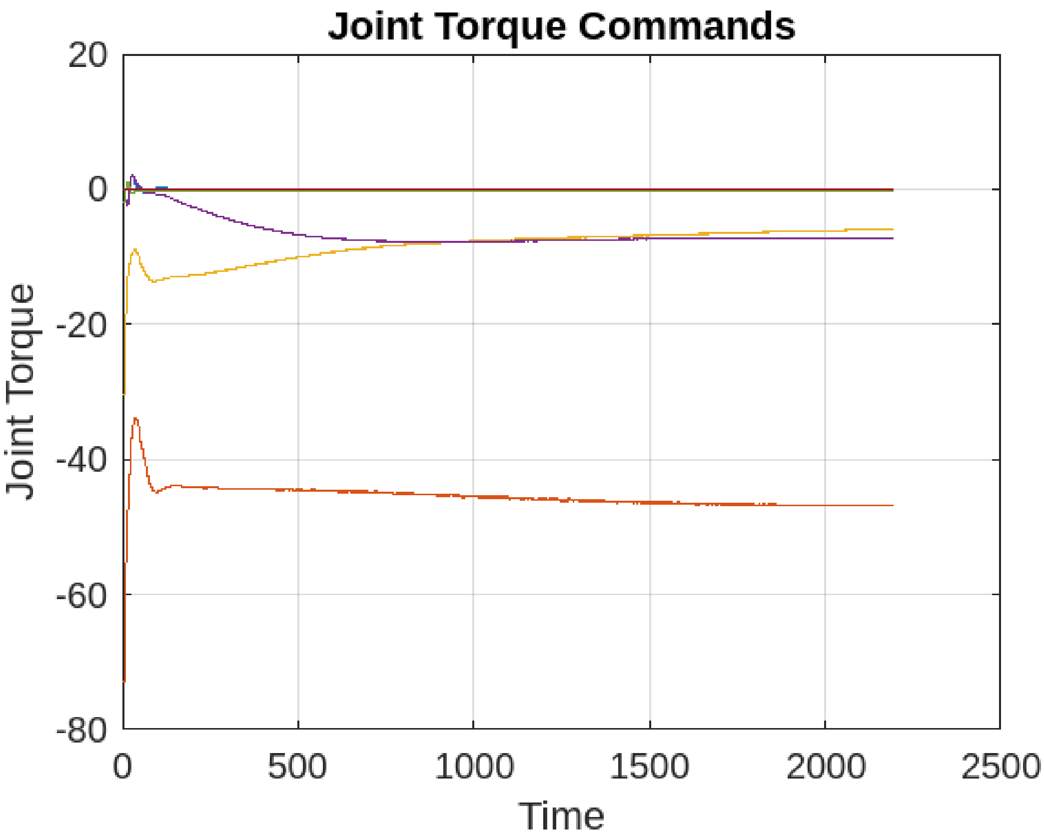
By plotting the errors, it was possible to tune the gains to make the behavior smoother.
Results are shown below.

Linear trajectory with cubic polynomial velocity profile:

Kp=70

Kd=7

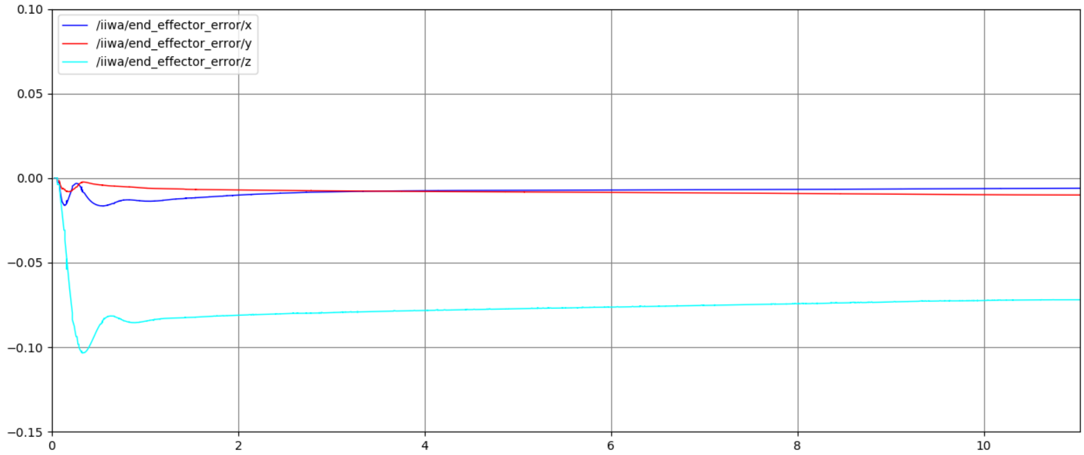


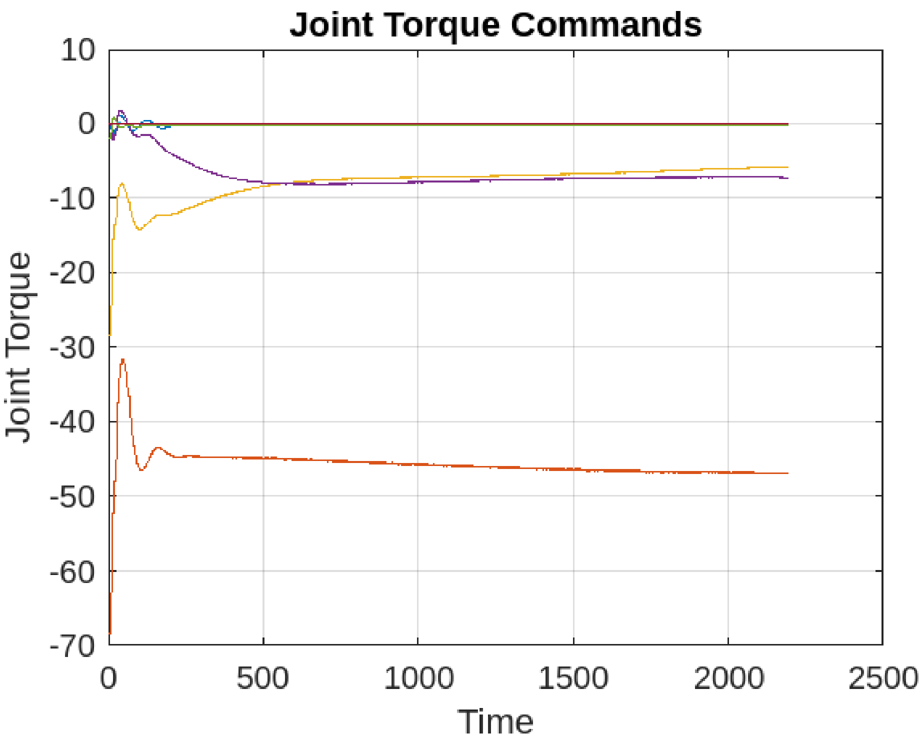


Linear trajectory, trapezoidal velocity profile:

Kp=60

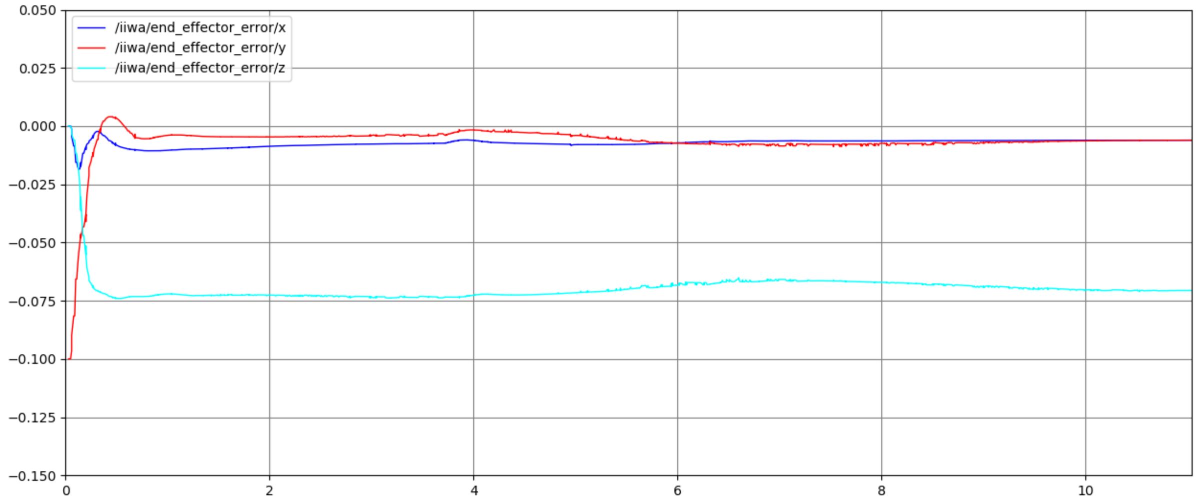
Kd=5

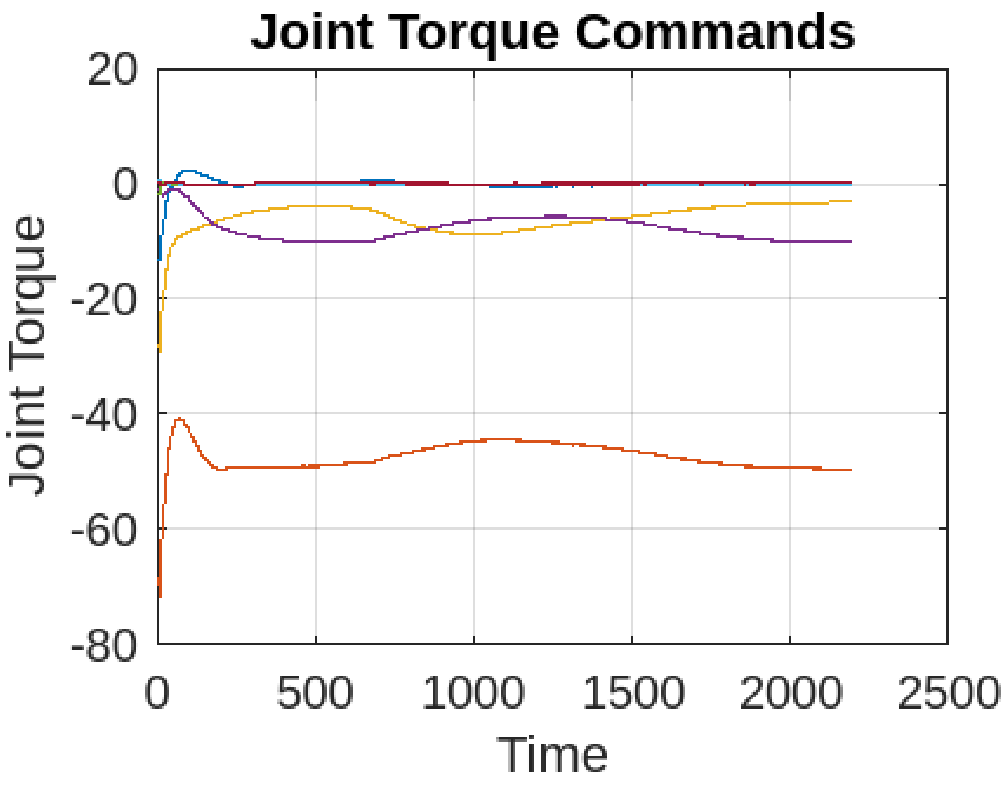




Circular trajectory, cubic polynomial velocity profile:

Kp= 70
Kd= 7

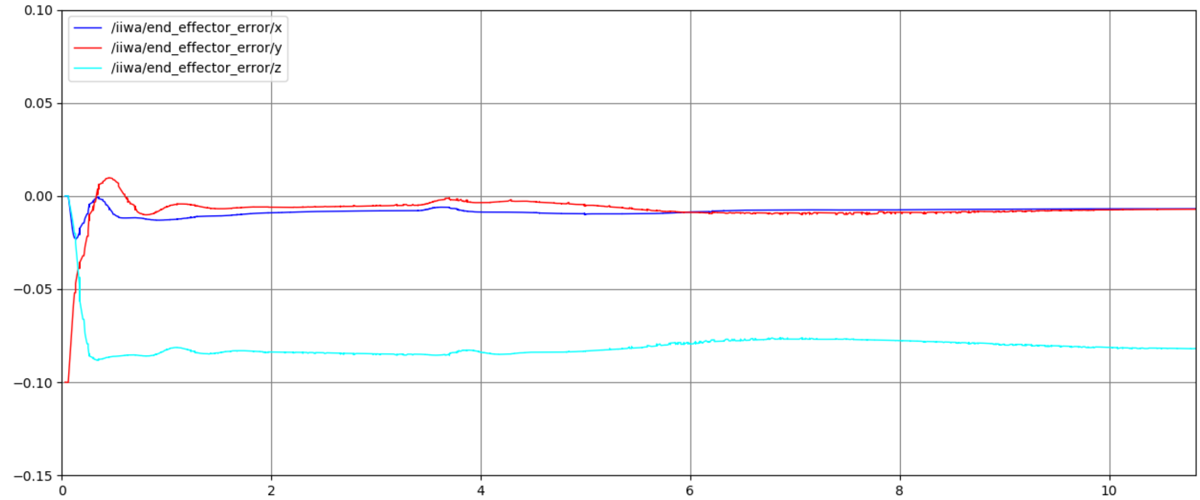


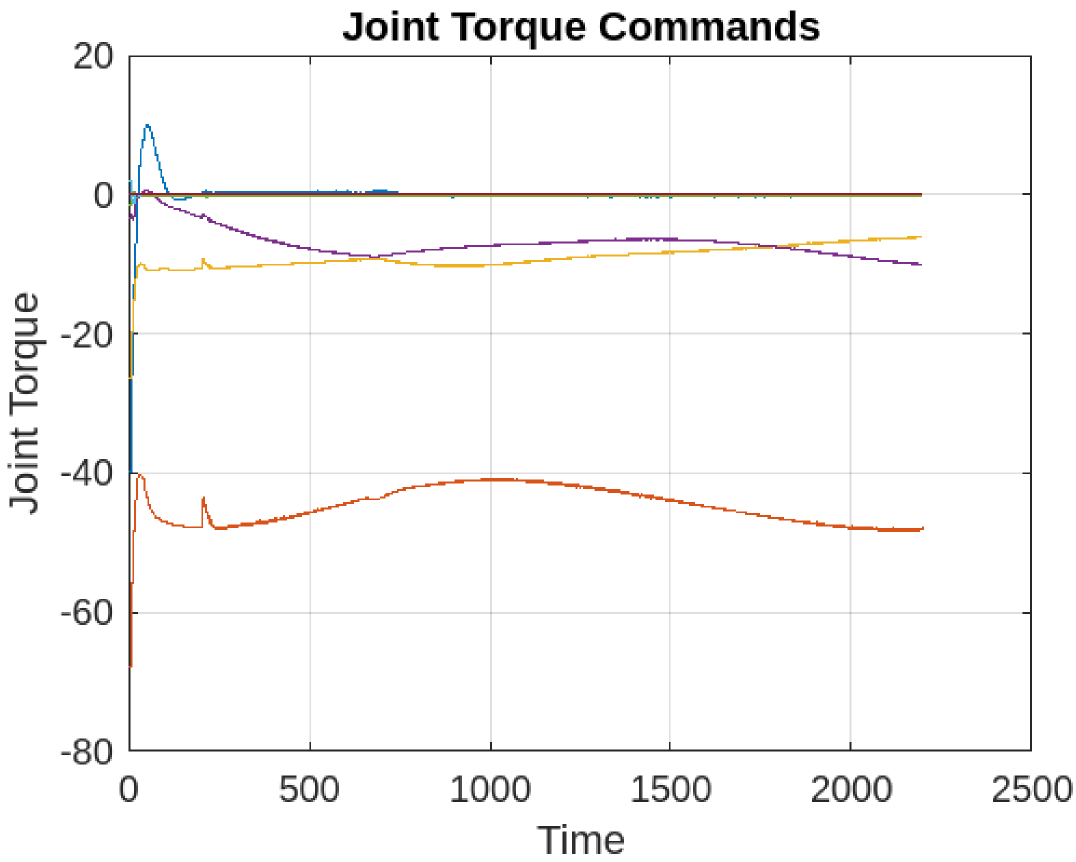


Circular trajectory, trapezoidal velocity profile

Kp=60

Kd=5

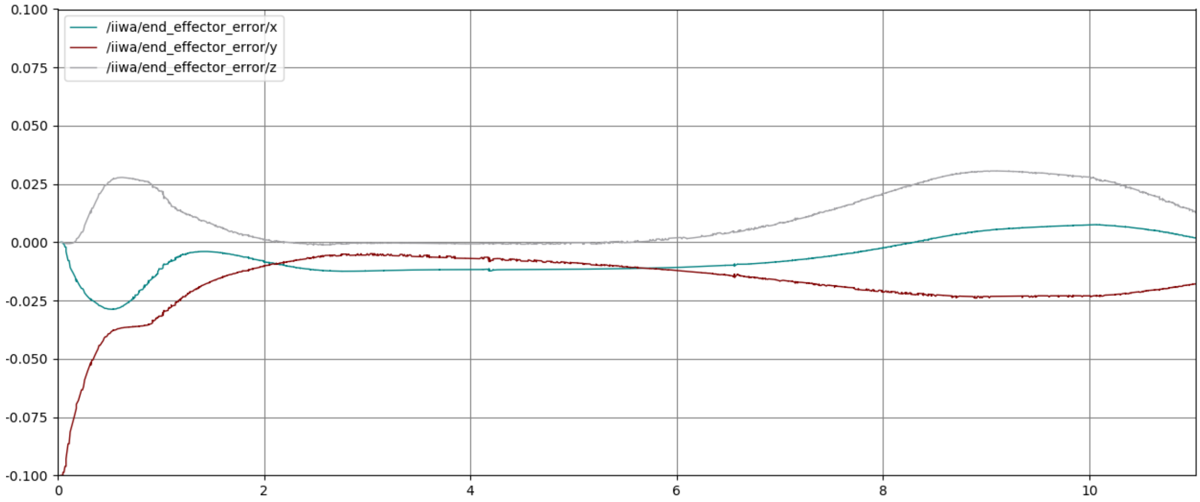




The error on axis z showing in each plot was probably due to a bad gravity compensation. Halving up the result of gravity compensation the error on axis z turns up improved.

```
return robot_->getJsim() * (ddqd + _Kd*de + _Kp*e)
            + robot_->getCoriolis() + 0.5*robot_->getGravity();
```

errors are showed below:



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

Bag files, obtained through command $rosbag record -O multi_topic_commands.bag joint_torque_commands can be found in github repo.
It's then necessary to upload the recorded bag file into load_plot_rosbag.m (in git repo).

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.

(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 $\dot{x}$, and acceleration $\ddot{x}$, 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 eo errors (some functions are provided into the include/utils.h file), finally compute your inverse dynamics control law following the equation
(4)

First of all, the Jacobian and its derivative were found through**:**
(in **kdl_control.cpp**)

```
KDL::Jacobian J_ee;
   J_ee.data.setZero();
   J_ee = robot_->getEEJacobian();
```

```
J = toEigen(J_ee);                              //Jacobian
dJ_dq=toEigen(robot_->getEEJacDotqDot()); //J dot
```

Variable returned from `getEEJacDotqDot` was modified into the right KDL::Twist (instead of KDL::Jacobian).

Position and velocity were found through getEEFrame() and getEEVelocity, already implemented in KDL library.
(in **kdl_control.cpp**)

```
  // position
   Eigen::Vector3d p_d(_desPos.p.data);
   KDL::Vector effective_p = robot_->getEEFrame().p;
   Eigen::Vector3d p_e = toEigen(effective_p);
   Eigen::Matrix<double,3,3,Eigen::RowMajor> R_d(_desPos.M.data);
   KDL::Rotation effective_r = robot_->getEEFrame().M;
   Eigen::Matrix<double,3,3,Eigen::RowMajor> R_e =
toEigen(effective_r);
   R_d = matrixOrthonormalization(R_d);
   R_e = matrixOrthonormalization(R_e);

   // velocity
   // Init of dot_p_d from _desVel.vel.data
   Eigen::Vector3d dot_p_d = Eigen::Vector3d(_desVel.vel.data);

   // Find the velocity of the end-effector and convert to
Eigen::Vector3d
   KDL::Twist eeVelocity = robot_->getEEVelocity();
   Eigen::Vector3d dot_p_e = toEigen(eeVelocity.vel);

   // Init of omega_d from _desVel.rot.data
   Eigen::Vector3d omega_d = Eigen::Vector3d(_desVel.rot.data);
   Eigen::Vector3d omega_e = toEigen(eeVelocity.rot);
```

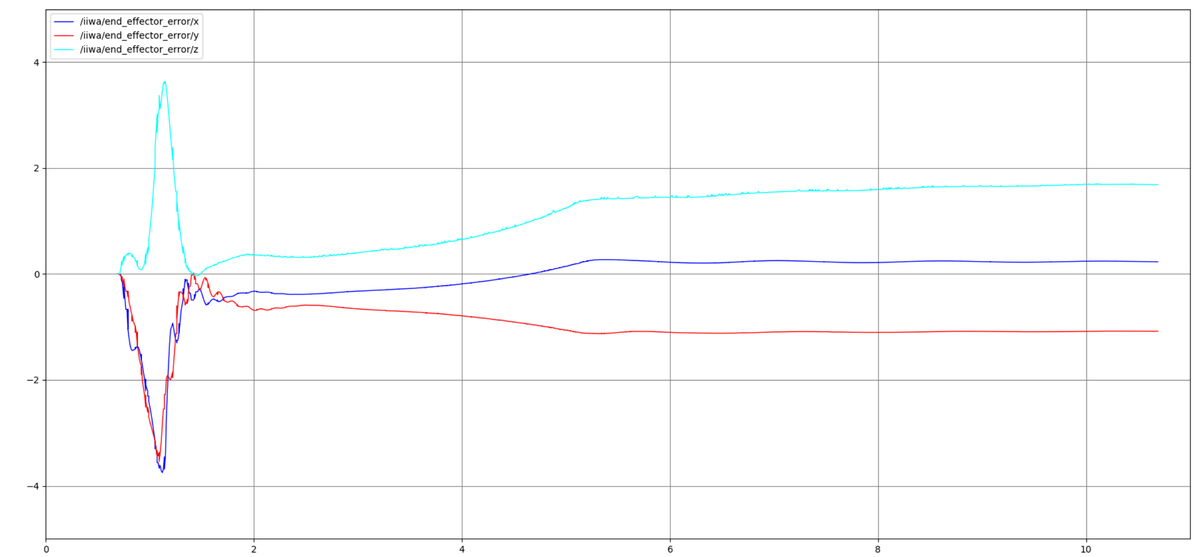
The other variables were found in the same way, by using functions already implemented into KDL library; from **utils.h** function **toEigen** was used to retrieve the desired type.

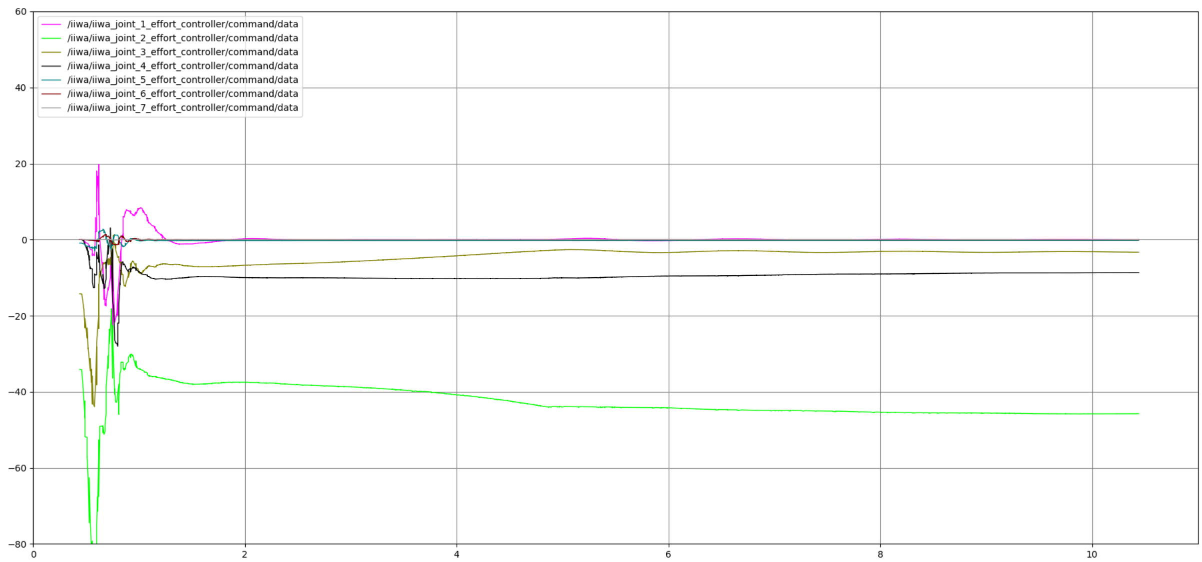
(c) Test the controller along the planned trajectories and plot the corresponding joint torque commands.

Gains:

Linear trajectory with cubic polynomial velocity profile:

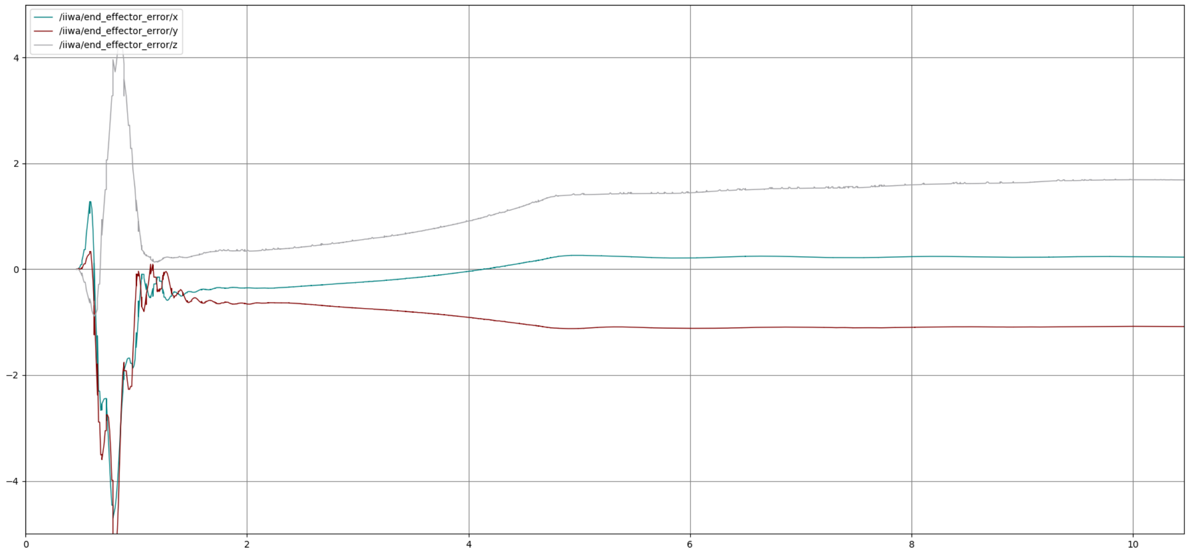
```
double Kp = 70;
            double Ko = 35;
            tau = controller_.idCntr(des_pose, des_cart_vel,
des_cart_acc, Kp, Ko,2*0.4*sqrt(Kp), 2*0.5*sqrt(Ko));
```

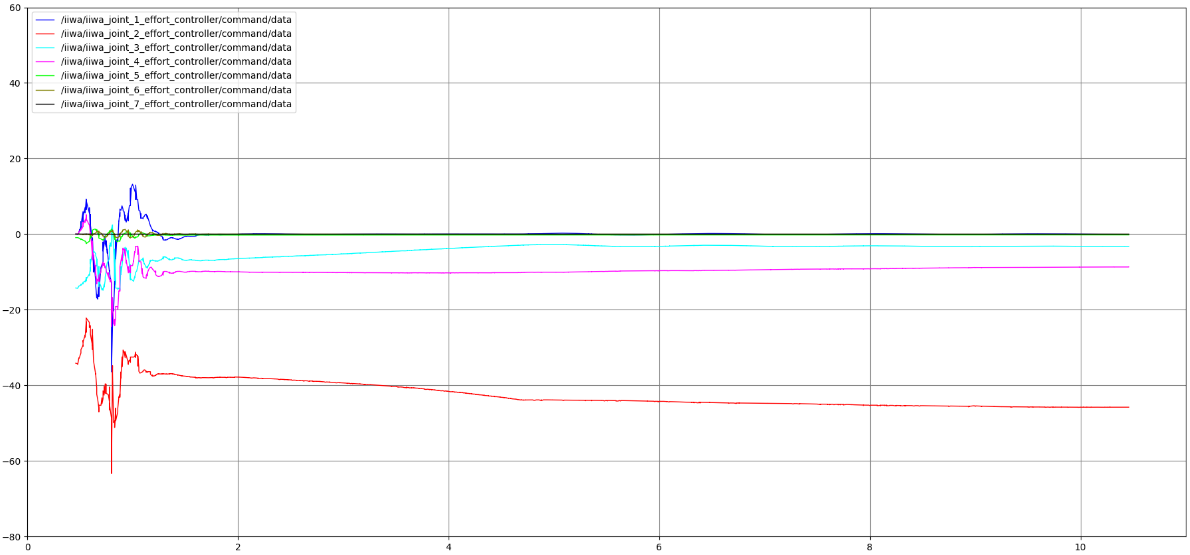




Linear trajectory with trapezoidal velocity profile:

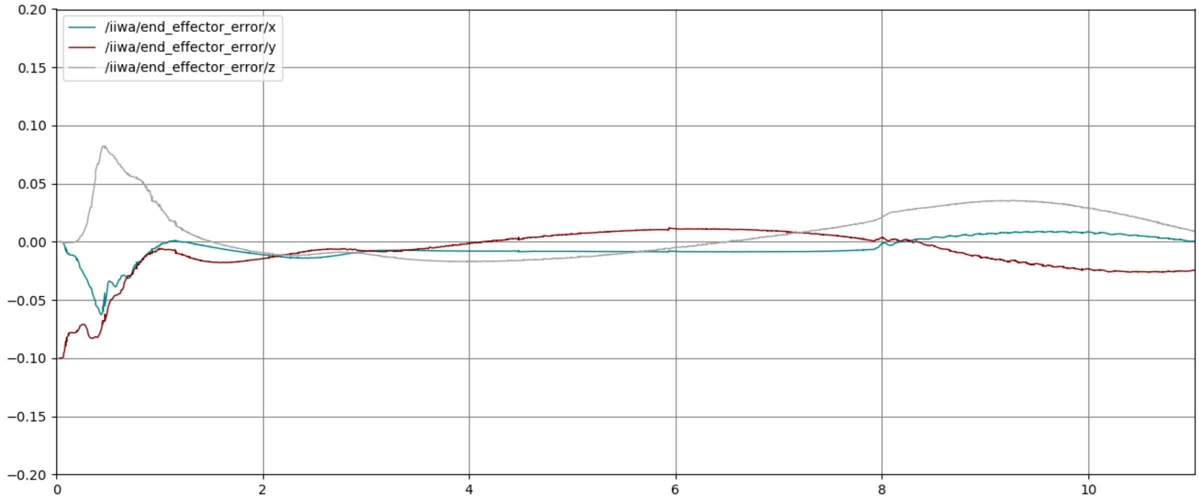
```
double Kp = 70;
            double Ko = 35;
            tau = controller_.idCntr(des_pose, des_cart_vel,
des_cart_acc, Kp, Ko,2*0.4*sqrt(Kp), 2*0.5*sqrt(Ko));
```

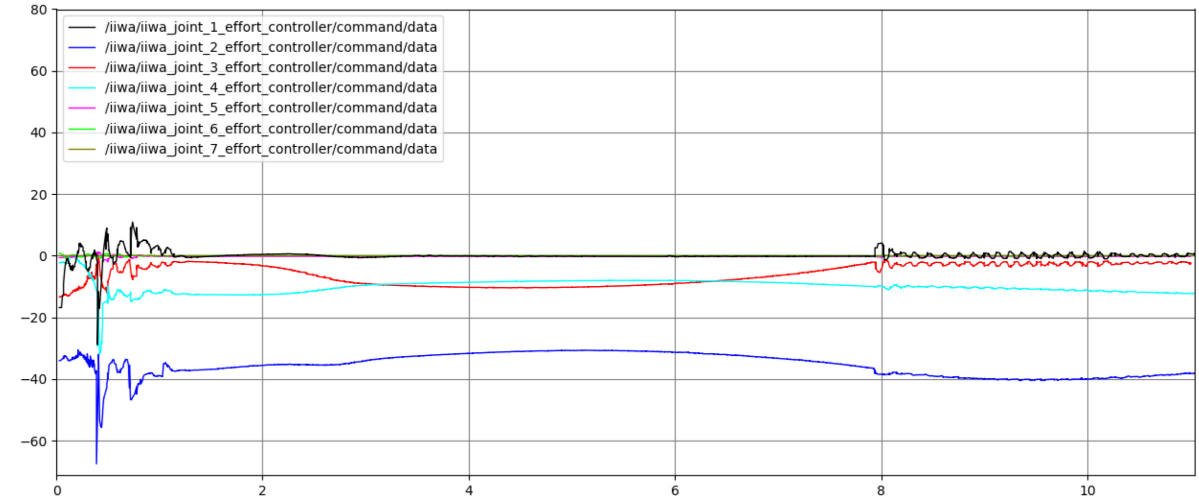




Circular trajectory, trapezoidal velocity profile:

```
double Kp = 25;
            double Ko = 10;
            tau = controller_.idCntr(des_pose, des_cart_vel,
des_cart_acc, Kp, Ko,2*0.8*sqrt(Kp), 2*0.8*sqrt(Ko));
```





Circular trajectory, cubic polynomial profile:

```
double Kp = 25;
            double Ko = 10;
            tau = controller_.idCntr(des_pose, des_cart_vel,
des_cart_acc, Kp, Ko,2*0.8*sqrt(Kp), 2*0.8*sqrt(Ko));
```