Robotics Lab: Homework 2

Control a manipulator to follow a trajectory

Gaetano Torella
Stefano Riccardi
Marco Maffeo
Antonio D'Angelo

GitHub: https://github.com/marcomaffeo/homework2.git

Control a manipulator to follow a trajectory

The goal of this homework is to develop a ROS package to dynamically control a 7-degrees-of-freedom robotic manipulator arm into the Gazebo environment. The **kdl_ros_control** package (at the following link: https://github.com/mrslvg/kdl_robot must be used as starting point. The student is requested to address the following points and provide a detailed report of the employed methods. In addition, a personal GitHub repo with all the developed code must be shared with the instructor. The report is due in one week from the homework release.

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 $s\ \dot{s}\ \ddot{s}$ that represent the curvilinear abscissa of your trajectory.
Remember: a trapezoidal velocity profile for a curvilinear abscissa $s \in [0, 1]$ is defined as follows

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

where tc is the acceleration duration variable while $\dot{s}\ (t)$ and $\ddot{s}\ (t)$ can be easily retrieved calculating time derivative of (1).

Inside the **kdl_planner.cpp** we define the new **trapezoidal_vel** function as follow

```
void KDLPlanner::trapezoidal_vel(double time, double timeC, double &s, double &dot_s, double
&ddot_s)
{
  double s_i = 0;
  double s_f = 1;
  double ddot_sc = -1.0/(std::pow(timeC,2) - trajDuration *timeC) * (s_f-s_i);

  if(time <= timeC)
  {
      s = 0.5*ddot_sc*std::pow(time,2);
      dot_s = ddot_sc*time;
      ddot_s = ddot_sc;
  }
  else if(time <= trajDuration -timeC)
  {
      s = ddot_sc*timeC*(time-timeC/2);
      dot_s = ddot_sc*timeC;
      ddot_s= 0;
  }
  else
  {
      s = s_f - 0.5*ddot_sc*std::pow(trajDuration -time,2);
      dot_s = ddot_sc*(trajDuration -time);
      ddot_s = - ddot_sc;
  }
}
```

Starting from the given **compute_trajectory** function we declared a void type function that changes the value of $\boldsymbol{s}\ \dot{\boldsymbol{s}}\ \ddot{\boldsymbol{s}}$. Inside that we declared as double **s_i**, **s_f** and **ddot_sc** and compute the values of curvilinear abscissa.

In the **kdl_planner.h** we add the declaration of the **trapezoidal_vel** function prototype inside the public section of the class **KDLPlanner** as follow

```
void trapezoidal vel(double time, double timeC, double &s, double &dot s, double &ddot s);
```

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 doubles $\boldsymbol{s}\ \dot{\boldsymbol{s}}\ \ddot{\boldsymbol{s}}$ that represent the curvilinear abscissa of your trajectory. Remember, a cubic polynomial is defined as follows

$$s(t) = a_3 t^3 + a_2 t^2 + a_1 t + a_0 \tag{2}$$

where coefficients $a_0\ a_1\ a_2\ a_3$ must be calculated offline imposing boundary conditions, while $\dot{\boldsymbol{s}}\ (\boldsymbol{t})$ and $\ddot{\boldsymbol{s}}\ (\boldsymbol{t})$ can be easily retrieved calculating time derivative of (2).

First of all, we calculate the coefficients as follow where q represent our s variable.

$$\begin{aligned} a_0 &= q_i \\ a_1 &= \dot{q}_i \\ a_3 t_f^3 + a_2 t_f^2 + a_1 t_f + a_0 &= q_f \\ 3a_3 t_f^2 + 2a_2 t_f + a_1 &= \dot{q}_f, \end{aligned}$$

Inside the **kdl_planner.cpp** we define the new **cubic_polinomial** function

```
void KDLPlanner::cubic polinomial(double time, double &s, double &dot s, double &ddot s)
{
  double a0 = 0;       // s0 = 0
  double a1 = 0;       // dot so = 0
  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;
  dot s = 3*a3*std::pow(time,2) + 2*a2*time + a1;
  ddot s = 6*a3*time + 2*a2;
}
```

As we did for the **trapezoidal_vel** function, we go in the **kdl_planner.h** and add the declaration of the **cubic_polinomial** function inside the public section of the class **KDLPlanner** as follow

```
void cubic polinomial(double time, double &s, double &dot s, double &ddot s);
```

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**).

In the **kdl_planner.h** we add the new construct for circular trajectory as follow and we add the **trajRadius_** like class variable.

```
class KDLPlanner{
public:
       KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double _trajRadius);
. . .
private:
    double trajDuration_, accDuration_, trajRadius_;
```

In the **kdl_planner.cpp** we declared the function as follow

```
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

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

First of all we create a **trajectory_point KDLPlanner::compute_circolar_trajectory** that takes as arguments the current time, the acceleration duration **_timeC** and the values of **s**, **dot_s** and **ddot_s** and returns a **trajectory_point**.
Inside the function we reclaimed the **cubic_polinomial** (or the **trapezoidal_vel**) function to compute the values of the curvilinear abscissa, we define a **trajectory_point traj** and we compute the velocity and acceleration equation for each component from the circular path expression.

```
trajectory_point KDLPlanner::compute_circolar_trajectory(double time, double timeC, double &s,double
&dot_s,double &ddot_s)
{
  //trapezoidal_vel(time, timeC, s, dot_s, ddot_s);
  cubic_polinomial(time, s, dot_s, ddot_s);
```

```
  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_ * sin(2*M_PI*s) * dot_s ;
  traj.vel(2) = - 2*M_PI * trajRadius_ * cos(2*M_PI*s) * dot_s;

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

  return traj;
}
```

Starting from that we can simply select the trapezoidal or the cubical profile by comment or uncomment the chosen one.

c) Do the same for the linear trajectory.

We define a **trajectory_point compute_linear_trajectory** function that takes as arguments the current time, the acceleration duration **_timeC** and the values of **s**, **dot_s** and **ddot_s** and returns a **trajectory_point**. Also, for this function we can choose the profile that we want, and we compute the trajectory values for position, velocity and acceleration using the convex combination as follow

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

The code we have implemented is as follow

```
trajectory point KDLPlanner::compute linear trajectory(double time, double timeC, double &s,double
&dot_s,double &ddot_s)
{
  trajectory_point traj;

  //trapezoidal vel(time, timeC, s, dot s, ddot s);
  cubic_polinomial(time, s, dot_s, ddot_s);
  traj.pos = (1-s) * trajInit_ + s * trajEnd_;
  traj.vel = (-trajInit  + trajEnd )*dot s;
  traj.acc = Eigen::Vector3d::Zero();

  return traj;
}
```

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.

For try the circular and the linear trajectories we must modify the **kdl_robot_test.cpp**.
First of all, if we choose the linear trajectory we have to use the given planner

```
    double traj_duration = 1.5, acc_duration = 0.5, t = 0.0, init_time_slot = 1.0;
    KDLPlanner planner(traj_duration, acc_duration, init_position, end_position);
```

Otherwise, we must use the planner for the circular trajectory

```
    double radius = 0.1;
    KDLPlanner planner(traj_duration, init_position, radius);
```

After that we declared 3 double variables **s**, **dot_s**, **ddot_s** to pass on to **compute_trajectory** functions

```
    double s = 0;
    double dot_s = 0;
    double ddot_s = 0;
```

We initialize a **trajectory_point p** that use our planner as follow

```
    trajectory_point p = planner.compute_circolar_trajectory(t,acc_duration,s,dot_s,ddot_s);
    //trajectory_point p = planner.compute_linear_trajectory(t,acc_duration,s,dot_s,ddot_s);
```

To change the values of the trajectory, inside the update block of the main function we add the two functions

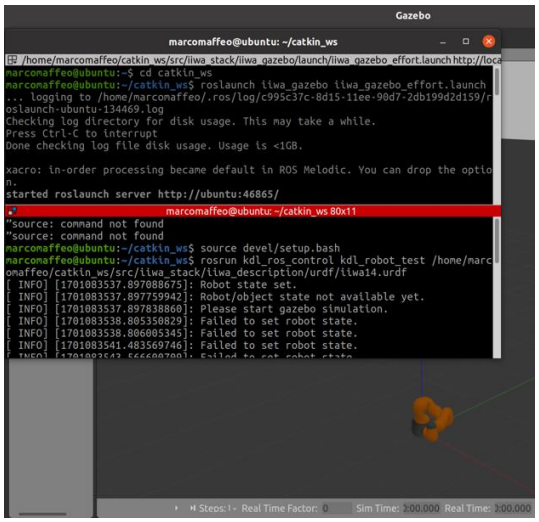
```
while ((ros::Time::now()-begin).toSec() < 2*traj_duration + init_time_slot)
   {
     if (robot state available)
      {
       // Update robot
       robot.update(jnt pos, jnt vel);
       // Update time
       t = (ros::Time::now()-begin).toSec();
       std::cout << "time: " << t << " s: " << s << " dot s: "<< dot s << " ddot s: " << ddot s <<
std::endl;

       // 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 circolar trajectory(0.0,acc duration,s,dot s,ddot s);
         //p = planner.compute_linear_trajectory(0.0,acc_duration,s,dot_s,ddot_s);
```

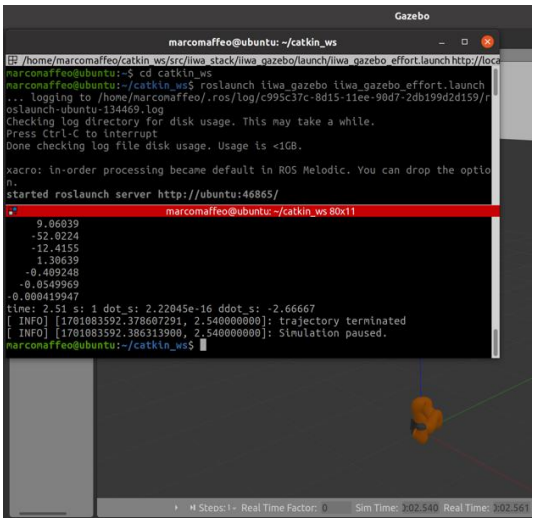
```
        }
else if(t > init_time_slot && t <= traj_duration + init_time_slot)
    {
      p = planner.compute_circolar_trajectory(t-init_time_slot,acc_duration,s,dot_s,ddot_s);
      //p = planner.compute_linear_trajectory(t-init_time_slot,acc_duration,s,dot_s,ddot_s);

      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());
    }
else
    {
      ROS_INFO_STREAM_ONCE("trajectory terminated");
      break;
    }
```

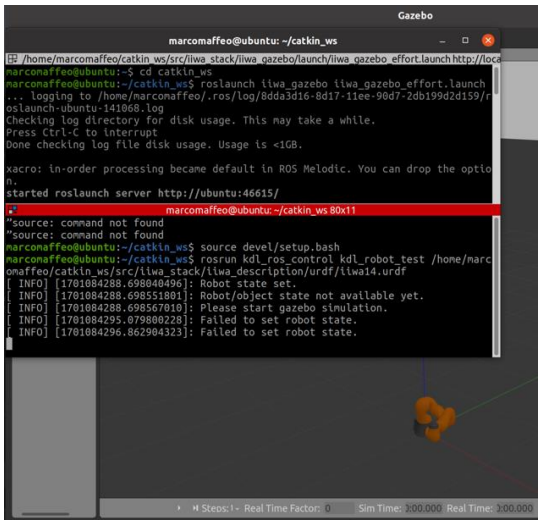
With gazebo we can see the motion of the robot as follow where are visualize the initial and final states of the circular and linear trajectory. The type of profile is chosen arbitrarily, in this case we use the trapezoidal one.



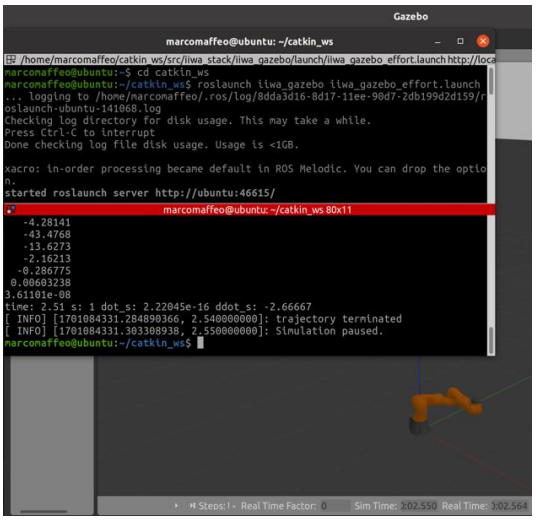
Initial state of circular trajectory



Final state of circular trajectory

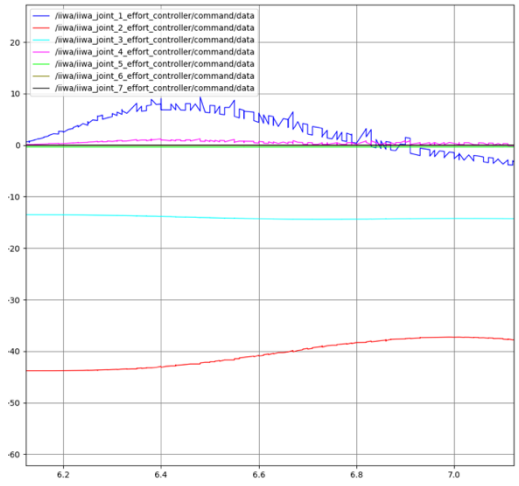


Initial state of linear trajectory

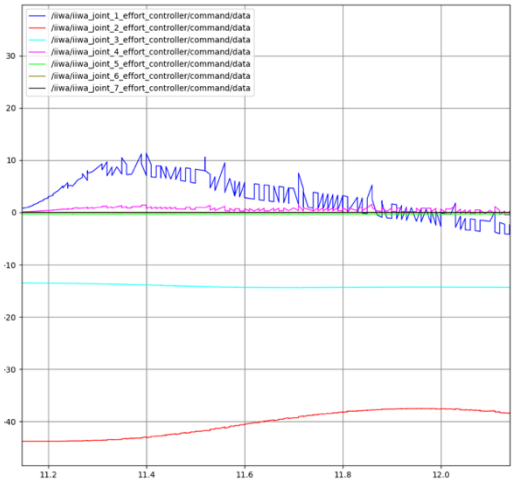


Final state of linear trajectory

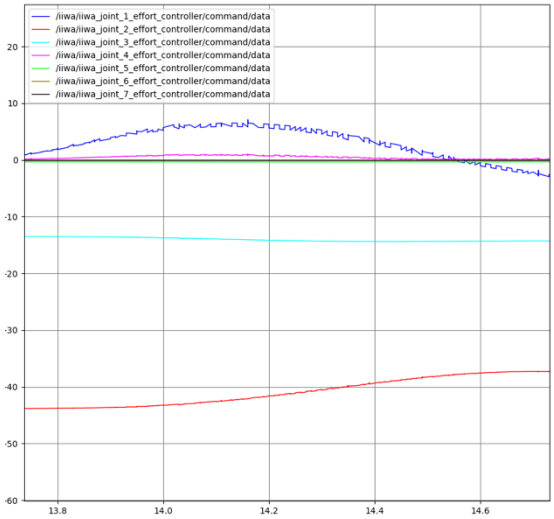
b) Plot the torques sent to the manipulator and tune appropriately the control gains **Kp** and **Kd** until you reach a satisfactorily smooth behavior. You can use **rqt_plot** to visualize your torques at each run, save the screenshot.



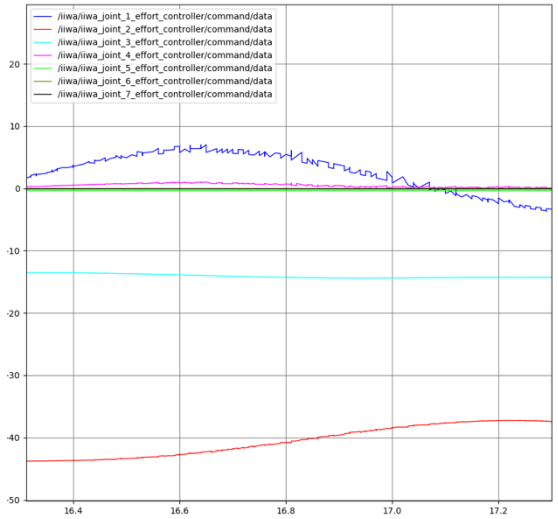
Pid 1: Kp = 50, Kd = 7



Pid 2: Kp = 100, Kd = 10



Pid 3: Kp = 20, Kd = 4



Pid 4: Kp = 50, Kd = 0.5

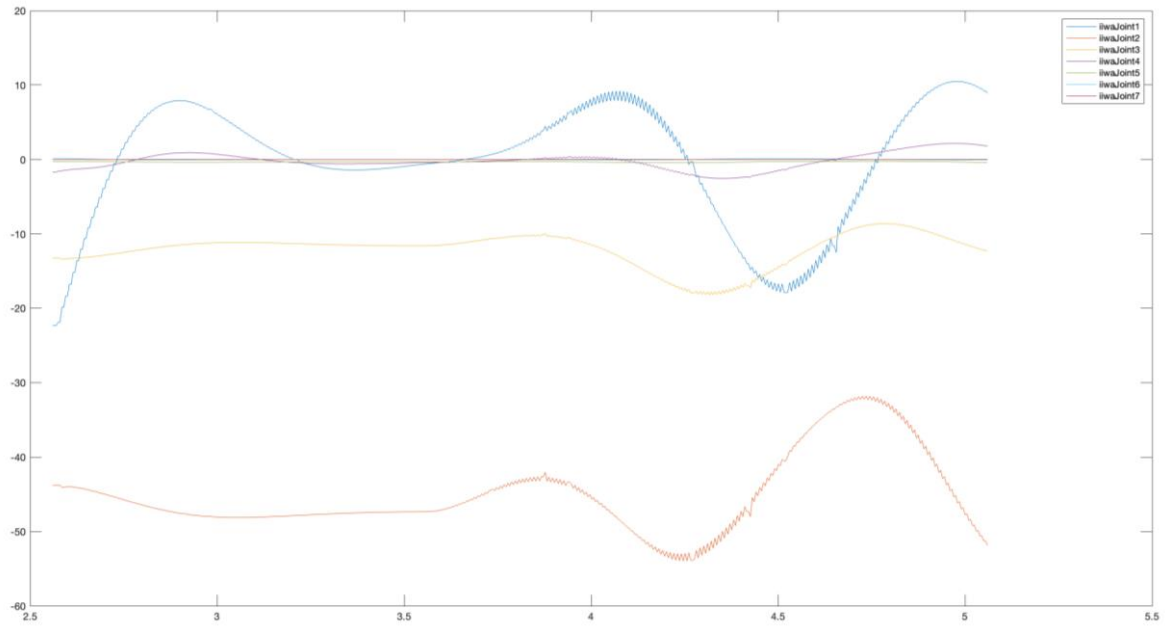
We tested the behavior of the manipulator arm with various regulation parameters and found that the regulator with the smoothest behavior is the third one with values of Kp = 20 and Kd = 4.

c) 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.

We use the **rosbag** record command to save the data from the **iiwa joint effort controller** and use this MATLAB script to plot the joint torque

```
bag = rosbag('subset.bag');
jointData = struct('joint1', [], 'joint2', [], 'joint3', [], 'joint4', [], 'joint5', [], 'joint6', [], 'joint7', []);
jointData.joint1 = select(bag,'Topic','/iiwa/iiwa_joint_1_effort_controller/command');
jointData.joint2 = select(bag,'Topic','/iiwa/iiwa_joint_2_effort_controller/command');
jointData.joint3 = select(bag,'Topic','/iiwa/iiwa_joint_3_effort_controller/command');
jointData.joint4 = select(bag,'Topic','/iiwa/iiwa_joint_4_effort_controller/command');
jointData.joint5 = select(bag,'Topic','/iiwa/iiwa_joint_5_effort_controller/command');
jointData.joint6 = select(bag,'Topic','/iiwa/iiwa_joint_6_effort_controller/command');
jointData.joint7 = select(bag,'Topic','/iiwa/iiwa_joint_7_effort_controller/command');

for R = 1:7
    jointTopic = sprintf('joint%d', R);
    data = jointData.(jointTopic);
    timeVector = linspace(data.StartTime, data.EndTime, data.NumMessages);
    msgStructs = readMessages(data,'DataFormat','struct');
    Points = cellfun(@(m) double(m.Data),msgStructs);
    plot(timeVector,Points)
    hold on
end
legend('iiwaJoint1','iiwaJoint2','iiwaJoint3','iiwaJoint4','iiwaJoint5','iiwaJoint6','iiwaJoint7');
```



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.

Frist of all we uncomment the code of the **KDLController::idCntr** function and we analyze the previous version of the controller. After that we fix some problem with the end effector frame, jacobian and velocity functions:

```
Eigen::VectorXd KDLController::idCntr(KDL::Frame & desPos, KDL::Twist & desVel, KDL::Twist & desAcc,
                                      double _Kpp, double _Kpo,double _Kdp, double _Kdo){
   // calculate gain matrices
   Eigen::Matrix<double,6,6> Kp, Kd;
   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> 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);
```

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 $\boldsymbol{x}$, velocity $\dot{\boldsymbol{x}}$, and acceleration $\ddot{\boldsymbol{x}}$, retrieve the current joint space inertia matrix **M** and the **Jacobian** (compute the analytic Jacobian) and its time derivative, compute the linear $\boldsymbol{e_p}$ and the angular $\boldsymbol{e_o}$ errors (some functions are provided into the include/utils.h file), finally compute your inverse dynamics control law following the equation

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

First of all we implement a function called **getEEJacDotqDot1** in **kdl_robot.cpp** (and in its .h file) file because of a bug that happen when we use the original **getEEJacDotqDot** function; different from the original one, which return the **J_dot** in **VectorXd**, our function returns the **J_dot** as **KDL::Jacobian**

```
KDL::Jacobian KDLRobot::getEEJacDotqDot1(){
    return s_J_dot_ee_;
}
```

We implement the inverse dynamics in the operational space as follow:

```
//// inverse dynamics

   Eigen::Matrix<double,6,1> y;
   KDL::Jacobian dotJacEE;
   dotJacEE = robot_->getEEJacDotqDot1();

       y << dot dot x d - dotJacEE.data * robot ->getJntVelocities() + Kd*dot x tilde + Kp*x tilde;

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