Report - Homework 2

Giulia Gelsomina Romano P38000223

Group:
Emanuele Cuzzocrea
Silvia Leo
Vito Daniele Perfetta
Giulia Gelsomina Romano

The code of this project and the related video can be found in my public github repository: `https://github.com/giuliagromano/rl_hw2.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 t_c as double arguments and returns three double variables s, $\dot{s}$ and $\ddot{s}$ 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

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

where t_c 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.h this function prototype has been inserted

```
void trapezoidal_vel(double t, double &s, double &sdot, double &sdotdot)
    ;
```

It was preferred to take only the current time t as input, since the acceleration time t_c is already a private attribute of the KDLPLanner class. This function has been implemented as follows in kdl_planner.cpp

```
void KDLPlanner::trapezoidal_vel(double t, double &s, double &sdot,
    double &sdotdot) {
  double scdd = -1.0/(std::pow(accDuration_,2)-(trajDuration_*
      accDuration_));

  if(t <= accDuration_) {
    s = 0.5*scdd*std::pow(t,2);
    sdot = scdd*t;
    sdotdot = scdd;
  }
  else if(t <= trajDuration_- accDuration_) {
    s = scdd*accDuration_*(t-accDuration_/2);
    sdot = scdd*accDuration_;
    sdotdot = 0.0;
  }
  else {
    s = 1 - 0.5*scdd*std::pow(trajDuration_-t,2);
    sdot = scdd*(trajDuration_-t);
    sdotdot = -scdd;
  }

  return;
}
```

(b) Create a function named KDLPlanner::cubic_polynomial 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

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

where coefficients a_3, a_2, a_1, a_0 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 order to compute the coefficients a_3, a_2, a_1, a_0, the following boundary conditions have been imposed

$$\begin{cases} s_i = 0 \\ s_f = 1 \\ \dot{s}_i = 0 \\ \dot{s}_f = 0 \end{cases}$$

Solving the system, the four values are

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

The following prototype and implementation are respectively shown in kdl_planner.h and kdl_planner.cpp

```
void cubic_polynomial(double t, double &s, double &sdot, double &sdotdot
    );
```

```
void KDLPlanner::cubic_polynomial(double t, double &s,double &sdot,
    double &sdotdot){
  double a0,a1,a2,a3;

  a0 = 0.0;
  a1 = 0.0;
  a2 = 3/std::pow(trajDuration_,2);
  a3 = -2/std::pow(trajDuration_,3);

  s = a3*std::pow(t,3)+a2*std::pow(t,2)+a1*t+a0;
  sdot = 3*a3*std::pow(t,2)+2*a2*t+a1;
  sdotdot = 6*a3*t+2*a2;

  return;
}
```

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).

It is preferable to introduce two new constructors related to the circular trajectory, instead of only one. This is because the parameters required by the trapezoidal velocity profile and cubic polynomial are slightly different. In this way, the user is not forced to assign unnecessary parameters. The following prototype and implementation are respectively shown in kdl_planner.h and kdl_planner.cpp

```
KDLPlanner(double _trajDuration, double _accDuration, Eigen::Vector3d
    _trajInit, double _trajRadius);       //constructor for circular
    trajectory with trapezoidal velocity
KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double
    _trajRadius);     //constructor for circular trajectory with cubic
    polynomial
```

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

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 DLPlanner::compute_trajectory: first, call the cubic_polynomial 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

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

In order to select the desired trajectory, an additional integer parameter has been used. The compute_trajectory function takes this parameter as input, and uses it as a decision variable inside a switch structure

```
trajectory_point compute_trajectory(double time, int a);
```

The requested circular trajectory with cubic polynomial profile is implemented in "case 3". In view of testing the circular trajectory with trapezoidal velocity profile too, the "case 2" is implemented as shown below

```
trajectory_point KDLPlanner::compute_trajectory(double time, int a) {
    trajectory_point traj;

    double s,sd,sdd;

    // 0-> Linear Trajectory TV
    // 1-> Linear Trajectory CP
```

```
    // 2 -> Circular Trajectory TV
    // 3-> Circular Trajectory CP

    switch(a){
        ...
        case 2:
        {
          trapezoidal_vel(time,s,sd,sdd);

          Eigen::Vector3d rot_component(0.0,-trajRadius_*std::cos(2*KDL
              ::PI*s),-trajRadius_*std::sin(2*KDL::PI*s));
          Eigen::Vector3d centr_acc(0.0,4*std::pow(KDL::PI,2)*
              trajRadius_*std::pow(sd,2)*std::cos(2*KDL::PI*s),4*std::pow
              (KDL::PI,2)*trajRadius_*std::pow(sd,2)*std::sin(2*KDL::PI*s
              ));
          Eigen::Vector3d tang_acc(0,2*KDL::PI*trajRadius_*sdd*std::sin
              (2*KDL::PI*s),-2*KDL::PI*trajRadius_*sdd*std::cos(2*KDL::PI
              *s));

          traj.pos = trajInit_ + rot_component;
          traj.vel = Eigen::Vector3d(0.0,2*KDL::PI*trajRadius_*sd*std::
              sin(2*KDL::PI*s),-2*KDL::PI*trajRadius_*sd*std::cos(2*KDL::
              PI*s));
          traj.acc = centr_acc + tang_acc;
          //std::cout << "Circular Trajectory TV" << std::endl;
          break;
        }
        case 3:
        {
          cubic_polynomial(time,s,sd,sdd);

          Eigen::Vector3d rot_component(0.0,-trajRadius_*std::cos(2*KDL
              ::PI*s),-trajRadius_*std::sin(2*KDL::PI*s));
          Eigen::Vector3d centr_acc(0.0,4*std::pow(KDL::PI,2)*
              trajRadius_*std::pow(sd,2)*std::cos(2*KDL::PI*s),4*std::pow
              (KDL::PI,2)*trajRadius_*std::pow(sd,2)*std::sin(2*KDL::PI*s
              ));
          Eigen::Vector3d tang_acc(0,2*KDL::PI*trajRadius_*sdd*std::sin
              (2*KDL::PI*s),-2*KDL::PI*trajRadius_*sdd*std::cos(2*KDL::PI
              *s));

          traj.pos = trajInit_ + rot_component;
          traj.vel = Eigen::Vector3d(0.0,2*KDL::PI*trajRadius_*sd*std::
              sin(2*KDL::PI*s),-2*KDL::PI*trajRadius_*sd*std::cos(2*KDL::
              PI*s));
          traj.acc = centr_acc + tang_acc;
          //std::cout << "Circular Trajectory CP" << std::endl;
          break;
        }
        default:
            std::cout << "Error planner selection" << std::endl;
    }
  return traj;
}
```

(c) Do the same for the linear trajectory.

According to the previous observation, two different constructors have been introduced here too

```
KDLPlanner(double _trajDuration, double _accDuration, Eigen::Vector3d
    _trajInit, Eigen::Vector3d _trajEnd);     //constructor for linear
    trajectory with trapezoidal velocity
KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, Eigen::
    Vector3d _trajEnd);      //constructor for linear trajectory with
    cubic polynomial
```

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

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

Given the curvilinear abscissa, the linear trajectory has been computed as follows

$$\begin{cases} p(s) = p_i + s(p_f - p_i) \\ \dot{p}(s) = \dot{s}(p_f - p_i) \\ \ddot{p}(s) = \ddot{s}(p_f - p_i) \end{cases}$$

where $s \in [0, 1]$. The following two switch "case" are referred to the linear trajectory with trapezoidal velocity profile and cubic polynomial profile respectively

```
        ...
        case 0:
        {
          trapezoidal_vel(time,s,sd,sdd);

          Eigen::Vector3d diff = trajEnd_ - trajInit_;

          traj.pos = trajInit_ + s*(diff);   //s in {0,1}
          traj.vel = sd*(diff);
          traj.acc = sdd*(diff);
          //std::cout << "Linear Trajectory TV" << std::endl;
          break;
        }
        case 1:
        {
          cubic_polynomial(time,s,sd,sdd);

          Eigen::Vector3d diff= trajEnd_ - trajInit_;

          traj.pos = trajInit_ + s*(diff);   //s in {0,1}
          traj.vel = sd*(diff);
          traj.acc = sdd*(diff);
          //std::cout << "Linear Trajectory CP" << std::endl;
          break;
        }
        ...
```

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.

The selection of the desired type of trajectory is made by changing the "a" variable value (which will be sent to the compute_trajectory function), and uncommenting the corresponding constructor

```
...
    // Plan trajectory
    double traj_duration = 5, acc_duration = 1, t = 0.0, init_time_slot
       = 1.0, radius = 0.1;

    int a = 0;      //0-> Linear Trajectory TV, 1-> Linear Trajectory CP
       , 2 -> Circular Trajectory TV, 3-> Circular Trajectory CP

    KDLPlanner planner(traj_duration, acc_duration, init_position,
       end_position);    //0
    //KDLPlanner planner(traj_duration, init_position, end_position);
                        //1
    //KDLPlanner planner(traj_duration, acc_duration, init_position,
       radius);           //2
    //KDLPlanner planner(traj_duration, init_position,radius);
                               //3

    trajectory_point p = planner.compute_trajectory(t,a);
...
```

```
...
    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 << 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_trajectory(0.0, a);
            }
            else if(t > init_time_slot && t <= traj_duration +
               init_time_slot) {
                p = planner.compute_trajectory(t-init_time_slot, a);
                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;
            }
```

```
            des_pose.p = KDL::Vector(p.pos[0],p.pos[1],p.pos[2]);

            // EE's trajectory position
            KDL::Frame pose = robot.getEEFrame();
            Eigen::Vector3d position_now(pose.p.data);
            Eigen::Vector3d epos = p.pos - position_now;
...
```

As the joint space inverse dynamics control needs as reference values the desired position, velocity and acceleration in the joint space, the inverse kinematics problem has been tackled with the three following functions

```
...
            /////// INVERSE KINEMATICS ///////
            //retrieve current values
            qd.data << jnt_pos[0], jnt_pos[1], jnt_pos[2], jnt_pos[3],
                jnt_pos[4], jnt_pos[5], jnt_pos[6];
            dqd.data << jnt_vel[0], jnt_vel[1], jnt_vel[2], jnt_vel[3],
                jnt_vel[4], jnt_vel[5], jnt_vel[6];

            //update desired values
            ddqd = robot.getInvAccKin(qd,dqd,des_cart_acc);
            dqd = robot.getInvVelKin(qd, des_cart_vel);
            qd = robot.getInvKin(qd, des_pose);

            /////// JOINT SPACE INVERSE DYNAMICS CONTROL  ///////
            // Gains
            double Kp = 130, Kd = 10;

            tau = controller_.idCntr(qd, dqd, ddqd, Kp, Kd);
...
```

The inverse kinematics functions prototype and implementation have been coded in the kdl_robot.h and kdl_robot.cpp respectively

```
    KDL::JntArray getInvKin(const KDL::JntArray &q, const KDL::Frame &
        eeFrame);
    KDL::JntArray getInvVelKin(const KDL::JntArray &q, const KDL::Twist
        &Cartesian_vel);
    KDL::JntArray getInvAccKin(const KDL::JntArray &q, const KDL::
        JntArray &dq, const KDL::Twist &Cartesian_acc);
```

```
KDL::JntArray KDLRobot::getInvKin(const KDL::JntArray &q, const KDL::
    Frame &eeFrame) {

    KDL::JntArray jntArray_out_;
    jntArray_out_.resize(chain_.getNrOfJoints());
    int err = ikSol_->CartToJnt(q, eeFrame, jntArray_out_);
    if (err != 0) {
        printf("inverse kinematics failed with error: %d \n", err);
    }
    return jntArray_out_;
}


KDL::JntArray KDLRobot::getInvVelKin(const KDL::JntArray &q,
                                     const KDL::Twist &Cartesian_vel) {

    KDL::JntArray jntArray_out_;
    jntArray_out_.resize(chain_.getNrOfJoints());
```

```
    int err = ikVelSol_->CartToJnt(q, Cartesian_vel, jntArray_out_);
    if (err != 0) {
        printf("velocity inverse kinematics failed with error: %d \n",
            err);
    }
    return jntArray_out_;
}

KDL::JntArray KDLRobot::getInvAccKin(const KDL::JntArray &q, const KDL::
    JntArray &dq, const KDL::Twist &Cartesian_acc) {

    KDL::JntArray jntArray_out_;
    jntArray_out_.resize(chain_.getNrOfJoints());

    KDL::Twist J_ddqd_T = Cartesian_acc - s_J_dot_q_dot_ee_;
    Eigen::Matrix<double,6,1> J_ddqd = Eigen::MatrixXd::Zero(6, 1);
    J_ddqd.head(3) << J_ddqd_T.vel.x(), J_ddqd_T.vel.y(), J_ddqd_T.vel.z
        ();
    J_ddqd.tail(3) << J_ddqd_T.rot.x(), J_ddqd_T.rot.y(), J_ddqd_T.rot.z
        ();

    Eigen::Matrix<double,6,7> J(getEEJacobian().data);
    Eigen::Matrix<double,7,6> Jpinv = pseudoinverse(J);

    Eigen::Matrix<double,7,1> ddqd = Jpinv*(J_ddqd);
    for (int i = 0; i < ddqd.rows(); i++) {
        jntArray_out_(i) = ddqd(i, 0);
    }

    return jntArray_out_;
}
```

It can be noticed that for the position and velocity inverse kinematics functions the KDL solver ChainIkSolverPos_NR_JL and ChainIkSolverVel_wdls have been used. For what concern the acceleration inverse kinematics, a feedforward solution has been implemented (since the KDL::ChainIkSolverAcc is an abstract class).

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

Following a trial and error strategy, the K_P and K_D gains have been chosen equal to

$$K_P = 130, \qquad K_D = 10$$

Used commands
Run each command in a different terminal

```
$ roslaunch iiwa_gazebo iiwa_gazebo_effort.launch
```

```
$ rosrun kdl_ros_control kdl_robot_test ./src/iiwa_stack/
    iiwa_description/urdf/iiwa7.urdf
```

```
$ rqt_plot
```

The results obtained in the four cases are shown below

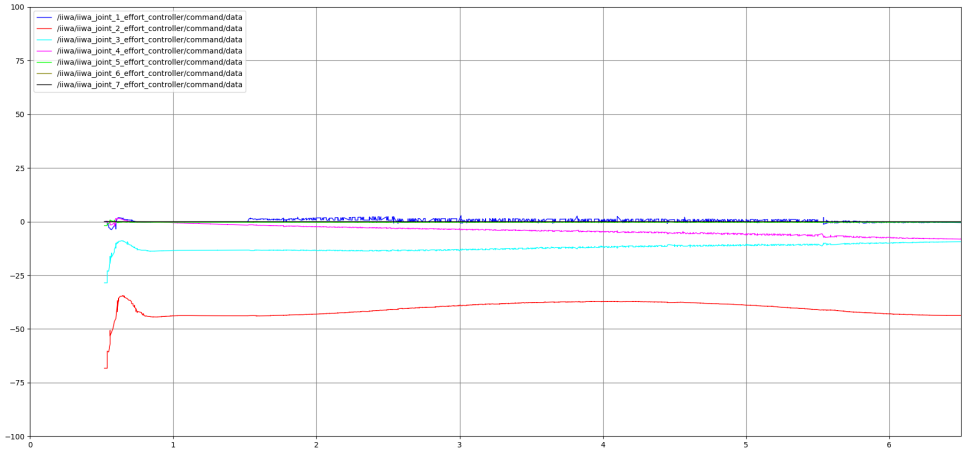

Figure 1: Linear trajectory with trapezoidal profile. Inverse dynamics joint space control

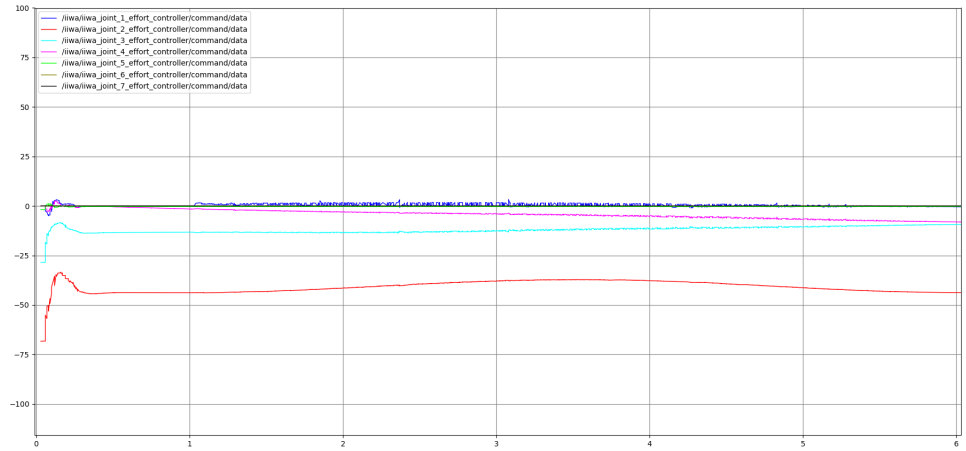

Figure 2: Linear trajectory with cubic profile. Inverse dynamics joint space control

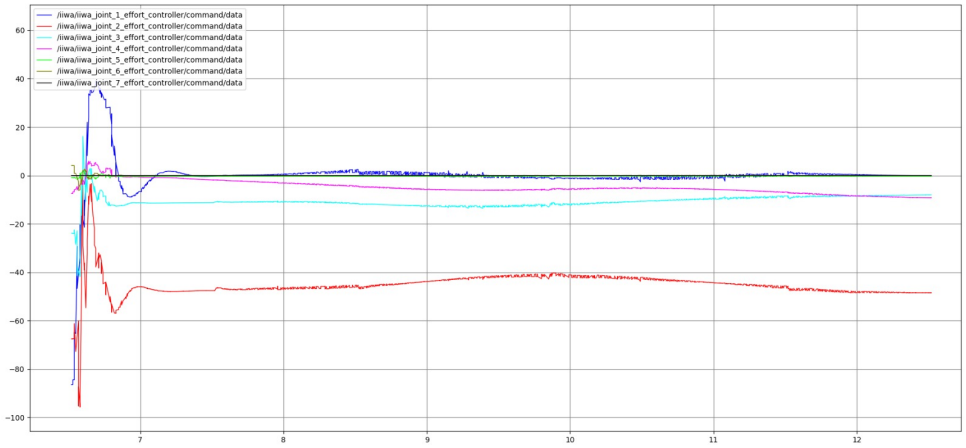

Figure 3: Circular trajectory with trapezoidal profile. Inverse dynamics joint space control

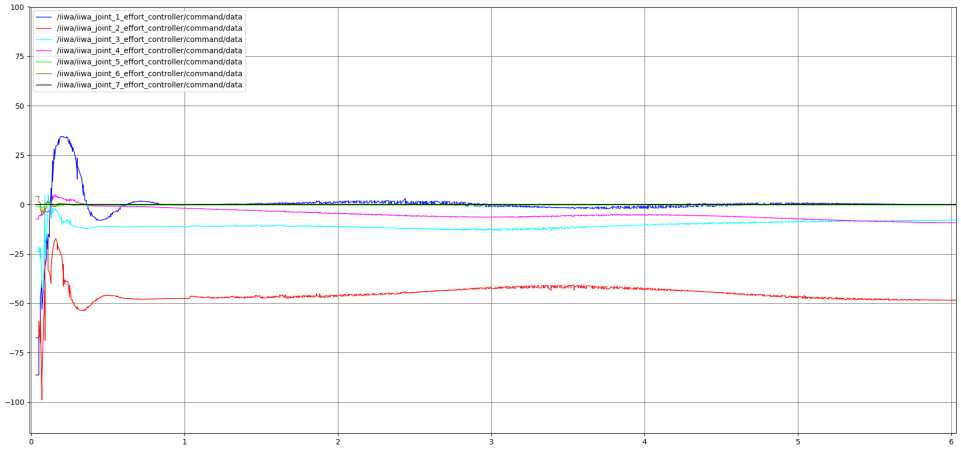

Figure 4: Circular trajectory with cubic profile. Inverse dynamics joint space control

The K_P and K_D gains have been selected in such a way to obtain as much as smooth torque behaviour as possible while ensuring the position error norm under a threshold of 3 cm in all four cases. More specifically, the greater error happened to be along the z axis, while along the x and y axes the error was of the order of mm.

(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`.

Used commands

Run each command in a different terminal to record the topics of interest

```
$ roslaunch iiwa_gazebo iiwa_gazebo_effort.launch
```

```
$ 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
```

```
$ rosrun kdl_ros_control kdl_robot_test ./src/iiwa_stack/
   iiwa_description/urdf/iiwa7.urdf
```

To visualize the information about the recorded topics, it is possible to follow the step below

```
Command Window
  >> rosbag info 'hw2_3c0.bag'
  Path:     C:\uni\P38 magistrale unina\robotics lab\handson_classes\hw2\matlab\hw2_3c0.bag
  Version:  2.0
  Duration: 6.0s
  Start:    gen 01 1970 01:00:00.55 (0.55)
  End:      gen 01 1970 01:00:06.55 (6.55)
  Size:     550.9 KB
  Messages: 8414
  Types:    std_msgs/Float64 [fdb28210bfa9d7c91146260178d9a584]
  Topics:   /iiwa/iiwa_joint_1_effort_controller/command  1202 msgs  : std_msgs/Float64
            /iiwa/iiwa_joint_2_effort_controller/command  1202 msgs  : std_msgs/Float64
            /iiwa/iiwa_joint_3_effort_controller/command  1202 msgs  : std_msgs/Float64
            /iiwa/iiwa_joint_4_effort_controller/command  1202 msgs  : std_msgs/Float64
            /iiwa/iiwa_joint_5_effort_controller/command  1202 msgs  : std_msgs/Float64
            /iiwa/iiwa_joint_6_effort_controller/command  1202 msgs  : std_msgs/Float64
            /iiwa/iiwa_joint_7_effort_controller/command  1202 msgs  : std_msgs/Float64

fx >>
```

Figure 5: Rosbag info

To import the topics data in Matlab environment the following procedure can be taken

```
%loading file
%bag=rosbag('hw2_3c0.bag');
%bag=rosbag('hw2_3c1.bag');
%bag=rosbag('hw2_3c2v2.bag');
bag=rosbag('hw2_3c3v2.bag');
%loading info
bSel1 = select(bag,'Topic','/iiwa/iiwa_joint_1_effort_controller/command');
msgStructs = readMessages(bSel1,'DataFormat','struct');
eff1 = cellfun(@(m) double(m.Data),msgStructs);
bSel2 = select(bag,'Topic','/iiwa/iiwa_joint_2_effort_controller/command');
msgStructs = readMessages(bSel2,'DataFormat','struct');
eff2 = cellfun(@(m) double(m.Data),msgStructs);
bSel3 = select(bag,'Topic','/iiwa/iiwa_joint_3_effort_controller/command');
msgStructs = readMessages(bSel3,'DataFormat','struct');
eff3 = cellfun(@(m) double(m.Data),msgStructs);
bSel4 = select(bag,'Topic','/iiwa/iiwa_joint_4_effort_controller/command');
msgStructs = readMessages(bSel4,'DataFormat','struct');
eff4 = cellfun(@(m) double(m.Data),msgStructs);
bSel5 = select(bag,'Topic','/iiwa/iiwa_joint_5_effort_controller/command');
msgStructs = readMessages(bSel5,'DataFormat','struct');
eff5 = cellfun(@(m) double(m.Data),msgStructs);
bSel6 = select(bag,'Topic','/iiwa/iiwa_joint_6_effort_controller/command');
msgStructs = readMessages(bSel6,'DataFormat','struct');
eff6 = cellfun(@(m) double(m.Data),msgStructs);
bSel7 = select(bag,'Topic','/iiwa/iiwa_joint_7_effort_controller/command');
msgStructs = readMessages(bSel7,'DataFormat','struct');
eff7 = cellfun(@(m) double(m.Data),msgStructs);
t=0:0.005:10;
```

Figure 6: Rosbag load

The results obtained in the four cases are shown below

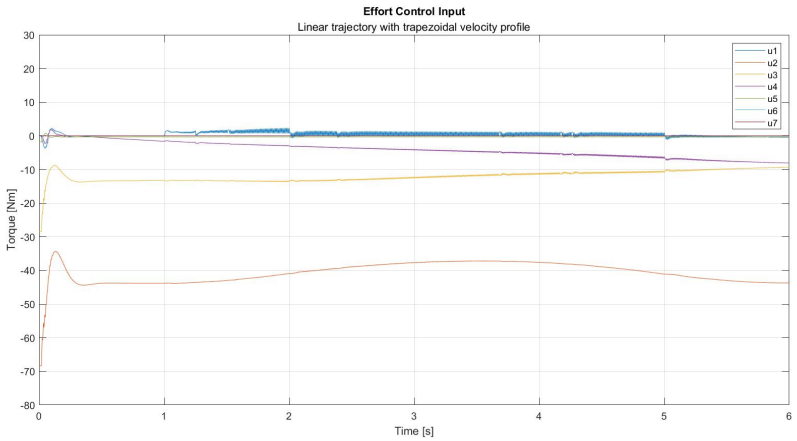

Figure 7: Linear trajectory with trapezoidal profile. Inverse dynamics joint space control (Matlab)

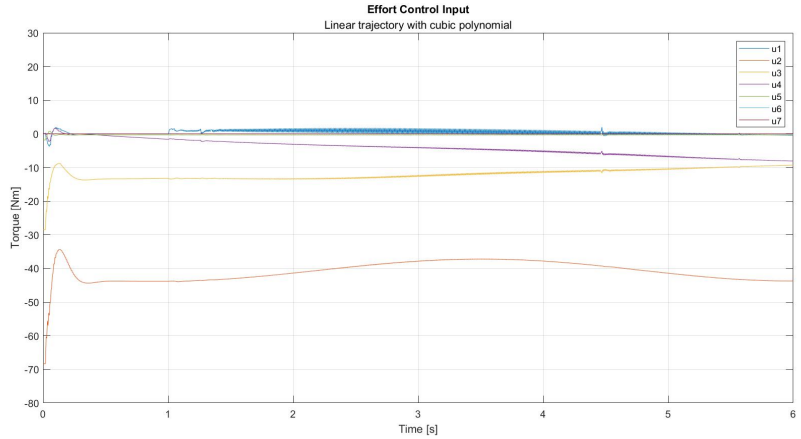

Figure 8: Linear trajectory with cubic profile. Inverse dynamics joint space control (Matlab)

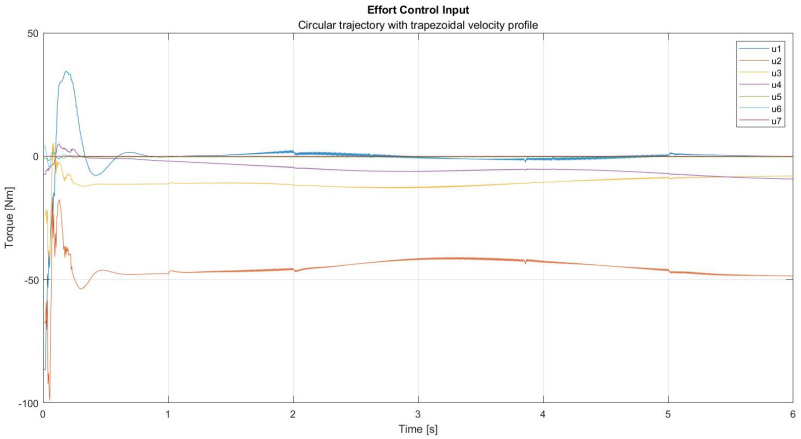

Figure 9: Circular trajectory with trapezoidal profile. Inverse dynamics joint space control (Matlab)

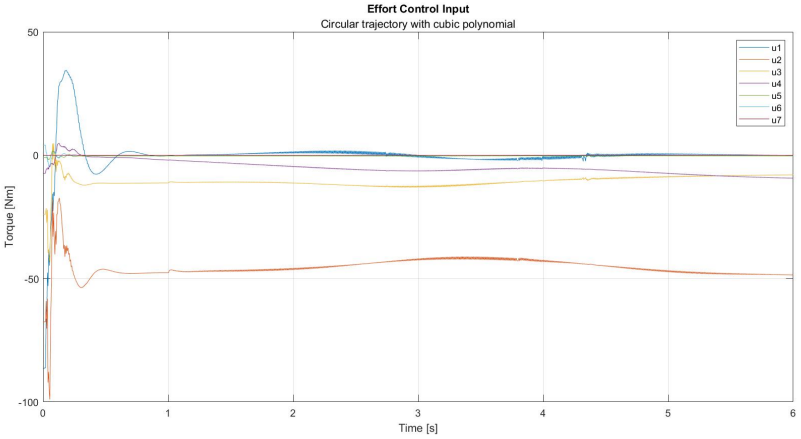

Figure 10: Circular trajectory with cubic profile. Inverse dynamics joint space control (Matlab)

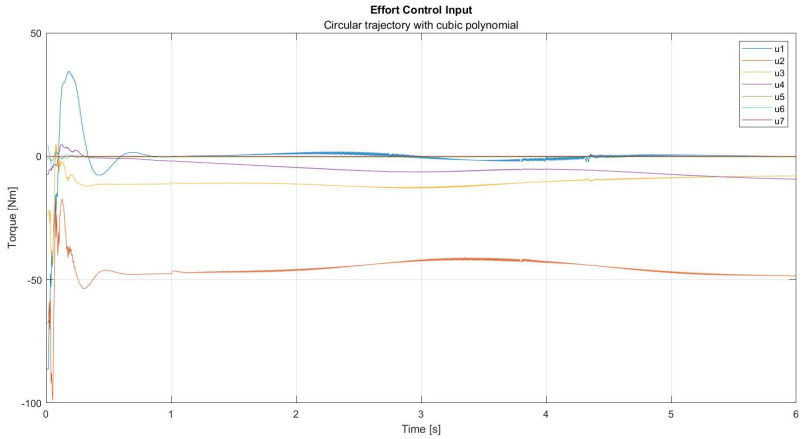

Figure 11: Circular trajectory with cubic profile. Inverse dynamics joint space control (Matlab)

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.

First of all, the prototype already inside kdl_control.h was used

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

(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 endeffector 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 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

$$\tau = By + n, \qquad y = J_A^{\dagger}(\ddot{x}_d + K_D\dot{\tilde{x}} + K_P\tilde{x} - \dot{J}_A\dot{q}) \tag{4}$$

The inverse dynamics operational space controller has been developed in kdl_control.cpp

```
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 = Eigen::MatrixXd::Zero(6, 6);
    Eigen::Matrix<double,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();
    Kp(2,2) = _Kpp + 170;       //gain offset along z-axis

    Kp(3,3)=_Kpo+30;

    //DEBUG
    //std::cout << "Kp: " << std::endl << Kp << std::endl;

    // 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::MatrixXd::Zero(6, 1);

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::MatrixXd::Zero(6, 1);
Eigen::Matrix<double,6,1> dot_x_tilde = Eigen::MatrixXd::Zero(6, 1);
x_tilde << e_p, e_o;
dot_x_tilde << dot_e_p, dot_e_o;
dot_dot_x_d << dot_dot_p_d, dot_dot_r_d;

// null space control
double cost=0.0;
Eigen::VectorXd grad = gradientJointLimits(robot_->getJntValues(),
   robot_->getJntLimits(),cost);

// inverse dynamics
Eigen::Matrix<double,6,1> y = Eigen::MatrixXd::Zero(6, 1);
Eigen::Matrix<double,6,1> Jdqd(robot_->getEEJacDotqDot());

y << dot_dot_x_d - Jdqd + Kd*dot_x_tilde + Kp*x_tilde;

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

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

It is worth remarking that K_P component along the z axis has been increased to reduce the corresponding position error. Furthermore, in order to better compensate the orientation error, also one of the associated K_P component has been enhanced.
It is preferred to exploit the null space in order to maximize the distance of each joint with respect to the corresponding joint limits. Equivalently, to minimize the distance from the center of their ranges.

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

The kdl_robt_test.cpp has been modified as follows in order to test the new controller

```
...
            ///////  OPERATIONAL SPACE INVERSE DYNAMICS CONTROL  ///////

            // Gains
            double Kp = 15, Ko = 20;

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

The results are shown below

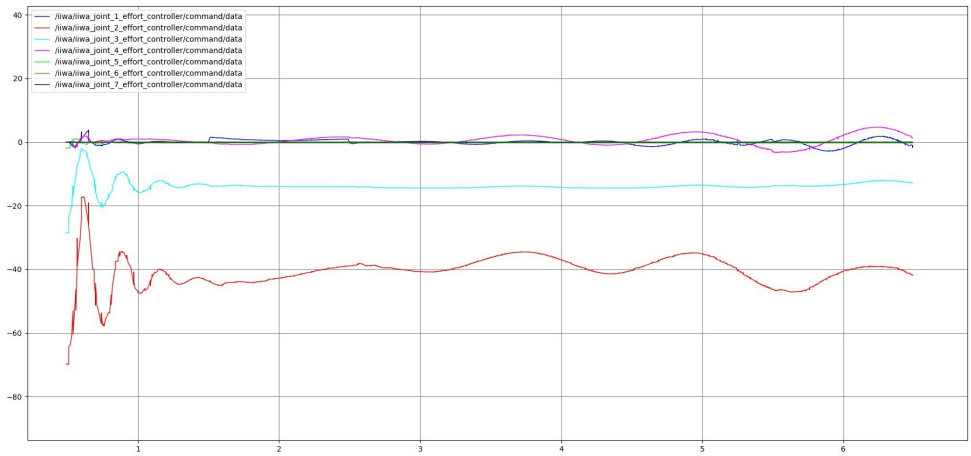

Figure 12: Linear trajectory with trapezoidal profile. Inverse dynamics operational space control

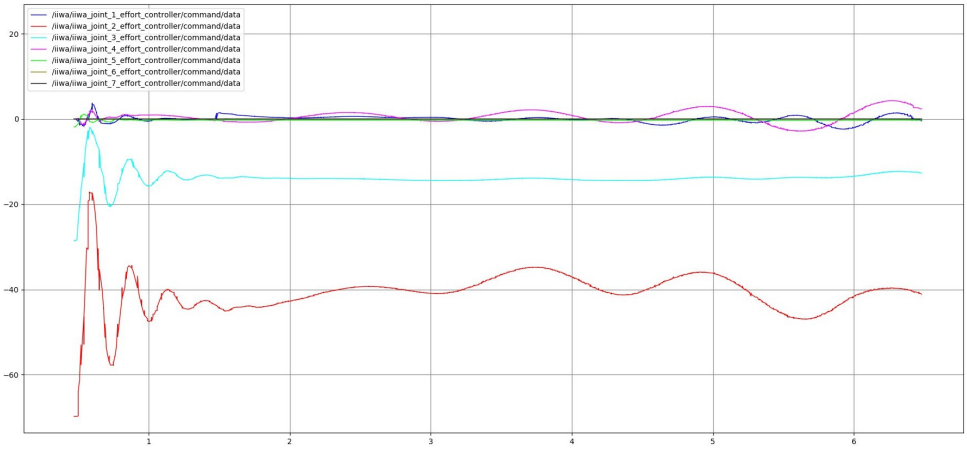

Figure 13: Linear trajectory with cubic profile. Inverse dynamics operational space control

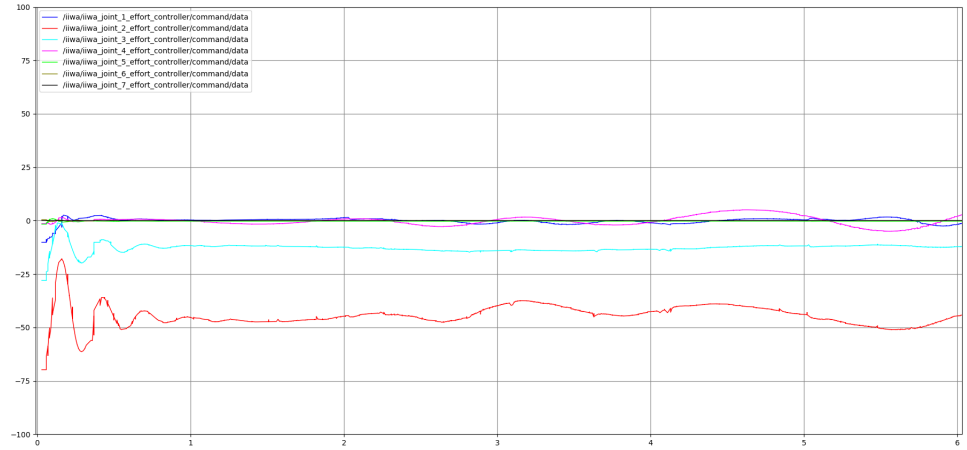

Figure 14: Circular trajectory with trapezoidal profile. Inverse dynamics operational space control

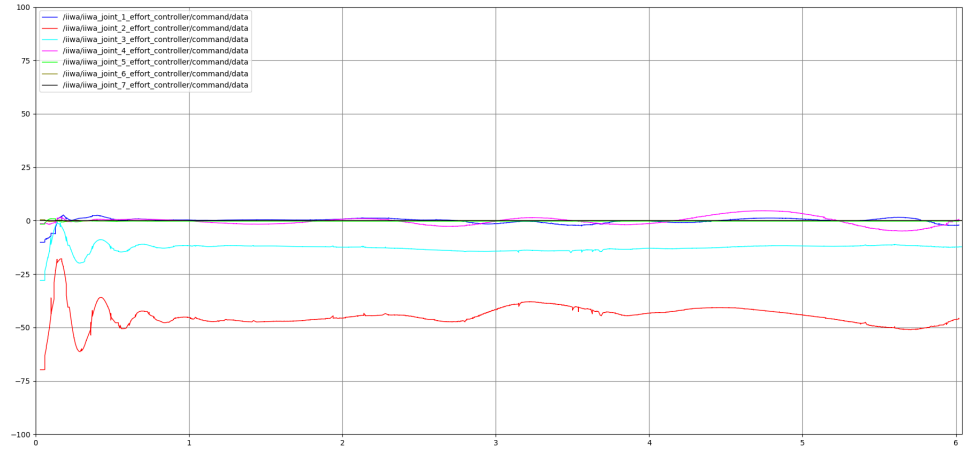

Figure 15: Circular trajectory with cubic profile. Inverse dynamics operational space control

Alternative solution: inverse dynamics operational space controller using Euler angles

Instead of computing the orientation error using the angle and axis representation, it is possible to consider a different control algorithm using Euler angles and the analytical Jacobian. The corresponding prototype and implementation are shown below

```
...
    Eigen::VectorXd idCntr_euler(KDL::Frame &_desPos,
                                 KDL::Twist &_desVel,
                                 KDL::Twist &_desAcc,
                                 double _Kpp,
                                 double _Kpo,
                                 double _Kdp,
                                 double _Kdo);
...
```

```
Eigen::VectorXd KDLController::idCntr_euler(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 = Eigen::MatrixXd::Zero(6,6);
    Eigen::Matrix<double,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();
    Kp(2,2) = Kp(2,2) + 150;
    Kp(1,1) = Kp(1,1) + 40;

    // read current state
    Eigen::Matrix<double,7,1> q = robot_->getJntValues();
    Eigen::Matrix<double,7,1> dq = robot_->getJntVelocities();
    Eigen::Matrix<double,7,7> B = robot_->getJsim();
    Eigen::Matrix<double,7,7> I = Eigen::Matrix<double,7,7>::Identity();
    Eigen::Matrix<double,6,7> J = robot_->getEEJacobian().data;



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

    // orientation
    Eigen::Vector3d euler_d(0,0,0);
    Eigen::Vector3d euler_e(0,0,0);
    Eigen::Vector3d e_euler(0,0,0);
    double alpha_e = 0, beta_e = 0, gamma_e = 0;
    double alpha_d = 0, beta_d = 0, gamma_d = 0;

    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);

    // conversion from Eigen::Matrix to KDL::Rotation (needed for euler
       angle extraction)
    KDL::Rotation R_d2 = KDL::Rotation::Identity();
    KDL::Rotation R_e2 = KDL::Rotation::Identity();
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j)
            R_d2(i,j) = R_d(i,j);
    }
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j)
            R_e2(i,j) = R_e(i,j);
    }

    R_d2.GetEulerZYZ(alpha_d,beta_d,gamma_d);
    R_e2.GetEulerZYZ(alpha_e,beta_e,gamma_e);
    euler_d << alpha_d,beta_d,gamma_d;
    euler_e << alpha_e,beta_e,gamma_e;
    e_euler = euler_d - euler_e;

    // x_tilde
    Eigen::Matrix<double,6,1> x_tilde;
    x_tilde.setZero();
    x_tilde << e_p, e_euler;



////// dot_x_tilde computation //////
    // conversion from KDL::Twist to Eigen::Matrix
    Eigen::Matrix<double,6,1> ve_d = Eigen::Matrix<double,6,1>::Zero();
    ve_d.head(3) << _desVel.vel.x(), _desVel.vel.y(), _desVel.vel.z();
    ve_d.tail(3) << _desVel.rot.x(), _desVel.rot.y(), _desVel.rot.z();
    Eigen::Matrix<double,6,1> ve_e = Eigen::Matrix<double,6,1>::Zero();
    ve_e.head(3) << robot_->getEEVelocity().vel.x(), robot_->
       getEEVelocity().vel.y(), robot_->getEEVelocity().vel.z();
    ve_e.tail(3) << robot_->getEEVelocity().rot.x(), robot_->
       getEEVelocity().rot.y(), robot_->getEEVelocity().rot.z();

    // TA_d
    Eigen::MatrixXd s_TA_d = Eigen::MatrixXd::Zero(6,6);
    s_TA_d.block<3,3>(0,0) = Eigen::Matrix<double, 3, 3>::Identity();
    Eigen::Matrix3d s_T_d = Eigen::Matrix3d::Zero();
    eul_kin_ZYZ(beta_d, alpha_d, s_T_d);
    s_TA_d.block<3,3>(3,3) = s_T_d;

    // TA_e
    Eigen::MatrixXd s_TA_e = Eigen::MatrixXd::Zero(6,6);
    s_TA_e.block<3,3>(0,0) = Eigen::Matrix<double, 3, 3>::Identity();
    Eigen::Matrix3d s_T_e = Eigen::Matrix3d::Zero();
    eul_kin_ZYZ(beta_e, alpha_e, s_T_e);
    s_TA_e.block<3,3>(3,3) = s_T_e;

    // dot_x_d, dot_x_e
    Eigen::Matrix<double,6,1> dot_x_d = s_TA_d.inverse() * ve_d;
    Eigen::Matrix<double,6,1> dot_x_e = s_TA_e.inverse() * ve_e;
    Eigen::Matrix<double,6,1> dot_x_tilde;
    dot_x_tilde.setZero();
```

```
    dot_x_tilde << (dot_x_d - dot_x_e);



////// s_JA_ee_dot computation//////
    // JA_d
    Eigen::MatrixXd s_JA_ee_d(6,7);
    s_JA_ee_d.setZero();
    s_JA_ee_d = s_TA_d.inverse() * J;

    // JA_e
    Eigen::MatrixXd s_JA_ee_e(6,7);
    s_JA_ee_e.setZero();
    s_JA_ee_e = s_TA_e.inverse() * J;

    // s_TA_dot
    Eigen::MatrixXd s_TA_dot_ = Eigen::MatrixXd::Zero(6,6);
    Eigen::Matrix3d s_T_dot = Eigen::Matrix3d::Zero();
    eul_kin_ZYZ_dot(beta_e, alpha_e, dot_x_e(4), dot_x_e(3), s_T_dot);
    s_TA_dot_.block<3,3>(3,3) = s_T_dot;

    // s_JA_ee_dot
    Eigen::MatrixXd s_JA_ee_dot(6,7);
    Eigen::Matrix<double,6,7> J_dot = robot_->getEEJacDot().data;
    s_JA_ee_dot.setZero();
    s_JA_ee_dot = s_TA_e.inverse()*(J_dot - s_TA_dot_*s_JA_ee_e);



////// dot_dot_x_d computation //////
    // conversion from KDL::Twist to Eigen::Matrix
    Eigen::Matrix<double,6,1> dot_ve_d;
    dot_ve_d.setZero();
    dot_ve_d.head(3) << _desAcc.vel.x(), _desAcc.vel.y(), _desAcc.vel.z
       ();
    dot_ve_d.tail(3) << _desAcc.rot.x(), _desAcc.rot.y(), _desAcc.rot.z
       ();
    Eigen::Matrix<double,6,1> dot_dot_x_d;
    dot_dot_x_d.setZero();
    dot_dot_x_d = s_TA_d.inverse()*(dot_ve_d - s_TA_dot_*dot_x_d);



////// tau computation //////
    Eigen::Matrix<double,6,1> y;
    y.setZero();
    y <<  dot_dot_x_d + Kd*dot_x_tilde + Kp*x_tilde - s_JA_ee_dot*dq;

    // JApinv
    Eigen::Matrix<double,7,6> JApinv = weightedPseudoInverse(I,s_JA_ee_e
       );

    // null space control
    double cost;
    Eigen::VectorXd grad = Eigen::VectorXd::Zero(7);
    grad = gradientJointLimits(robot_->getJntValues(),robot_->
       getJntLimits(),cost);
    Eigen::VectorXd tau = Eigen::VectorXd::Zero(7);
```

```
    tau = B*(JApinv*y + (I-JApinv*s_JA_ee_e)*(-grad- 1*robot_->
        getJntVelocities())) + robot_->getGravity() + robot_->getCoriolis
        ();

    return tau;
}
```

In order to compute the $T(\phi)$ matrix and its derivative, that gives the relationship between the analytical Jacobian and the geometric Jacobian, it has been necessary to introduce two additional functions. These have been inserted inside the utils.h file

```
inline void eul_kin_ZYZ(const double beta, const double alpha, Eigen::
    Matrix3d &T) {
          T <<
            0, -sin(alpha), cos(alpha) * sin(beta),
            0, cos(alpha), sin(alpha) * sin(beta),
            1, 0, cos(beta);
}

inline void eul_kin_ZYZ_dot(const double beta, const double alpha, const
     double beta_dot, const double alpha_dot, Eigen::Matrix3d &T_dot) {
          T_dot <<
              0, -cos(alpha) * alpha_dot, -sin(alpha) * sin(beta) *
                 alpha_dot + cos(alpha) * cos(beta) * beta_dot,
              0, -sin(alpha) * alpha_dot, cos(alpha) * sin(beta) *
                 alpha_dot + cos(beta) * sin(alpha) * beta_dot,
              0, 0, -sin(beta) * beta_dot;
}
```

Inside kdl_robot_test.cpp the following parameters have been changed in order to optimize the performance of this control

```
...
        double traj_duration = 8, acc_duration = 2, t = 0.0,
           init_time_slot = 2.0, radius = 0.1;    // Euler angles
           parameters
...
            // Gains
            double Kp = 15, Ko = 20;

            // Alternative solution : operational space inverse dynamics
                control using euler angles
            tau = controller_.idCntr_euler(des_pose, des_cart_vel,
               des_cart_acc, Kp, Ko, 2*sqrt(Kp), 2*sqrt(Ko));
...
```

The result in case of linear trajectory with trapezoidal velocity profile is shown below

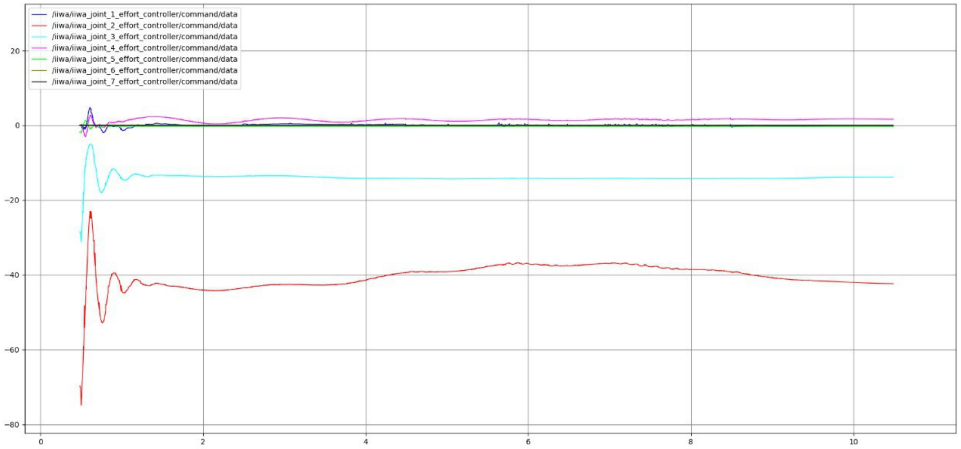

Figure 16: Linear trajectory with trapezoidal profile. Inverse dynamics operational space control with Euler angles