UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

Automation and Robotics Engineering

ROBOTICS LAB

HOMEWORK 2

Control a manipulator to follow a trajectory

Instructor:
Mario Selvaggio

Student:
Francesco Grasso
P38000046

Accademic Year 2023/2024

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

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

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

The following function returns the curvilinear abscissa of the trajectory. (s, $\dot{s}$ and $\ddot{s}$):

```
kdl_planner.cpp

////////////////////////// (1.a) Definition of trapezoidal_vel function ////////////////////////////////////////////

void KDLPlanner::trapezoidal_vel(double time, double &s, double &dots,double &ddots)
{

  double si=0;
  double sf=1;

  double ddot_traj_c = -1.0/(std::pow(accDuration_,2)-trajDuration_*accDuration_)*(sf-si);

  if(time <= accDuration_)
  {
    s = si + 0.5*ddot_traj_c*std::pow(time,2);
    dots = ddot_traj_c*time;
    ddots = ddot_traj_c;
  }
  else if(time <= trajDuration_-accDuration_)
  {
    s = si + ddot_traj_c*accDuration_*(time-accDuration_/2);
    dots = ddot_traj_c*accDuration_;
    ddots = 0;
  }
  else
  {
    s = sf - 0.5*ddot_traj_c*std::pow(trajDuration_-time,2);
    dots = ddot_traj_c*(trajDuration_-time);
    ddots = -ddot_traj_c;
  }

}
```

Figura 1: trapezoidal_vel function

1.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 polinomial is defined as follows $s(t) = a_3t^3 + a_2t^2 + a_1t + a_0$

The following function returns the cubic polynomial curvilinear abscissa of the trajectory. (s, $\dot{s}$ and $\ddot{s}$):

```cpp
//////////////////////////  (1.b) Definition of cubic_polinomial function   ///////////////////////////////


void KDLPlanner::cubic_polinomial(double time, double &s, double &dots,double &ddots)
{

  double si=0;
  double sf=1;
  double dsi=0;
  double dsf=0;

  double a0=si;
  double a1=dsi;
  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;
  dots=3*a3*std::pow(time,2)+2*a2*time+a1;
  ddots=6*a3*time+2*a2;
}
```

Figura 2: cubic_polinomial function

2. **Create circular trajectories for your robot**

2.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).

```
///////////////////////// (2.a) Definition of Circ Constructor ///////////////////////////////////////////////////

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

```

Figura 3: Constructor for for circular trajectories

The following lines of code have been added to the file:**kdl_planner.h** in the public and private methods respectively:

1) //**Circular Constructor**

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

2) **double trajRadius_;**

2.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

$x = x_i$

$y = y_i - rcos(2\pi s)$

$z = z_i - rsin(2\pi s)$

```cpp
//////////////////////////  (2.b) Definition of Circ_Cubic trajectory  /////////////////////////////////////////////////////


trajectory_point KDLPlanner::compute_cubic_circular( double time, double trajRadius_ ){

  trajectory_point traj;

  double s;
  double ds;
  double dds;
  cubic_polinomial(time, s, ds, dds);

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

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

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

  return traj;

}
```

Figura 4: Circular Trajectory with cubic_polinomial function

```cpp
trajectory_point KDLPlanner::compute_trapezoidal_circular(double time, double trajRadius_)
{

  trajectory_point traj;

  double s;
  double ds;
  double dds;
  trapezoidal_vel(time,s,ds,dds);


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

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

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

  return traj;
}
```

Figura 5: Circular Trajectory with trapezoidal function

2.c) Do the same for the linear trajectory

```
/////////////////////////  (2.c) Definition of Lin_Cubic trajectory  ////////////////////////////////////////////////////

trajectory_point KDLPlanner::compute_cubic_linear( double time){
  double s;
  double ds;
  double dds;
  cubic_polinomial(time,s,ds,dds);

  trajectory_point traj;
  Eigen::Vector3d pi=trajInit_;
  Eigen::Vector3d pf=trajEnd_;
  Eigen::Vector3d dif=pf-pi;

  traj.pos=(1-s)*pi+s*pf;
  traj.vel=(-pi+pf)*ds;
  traj.acc=(-pi+pf)*dds;

    return traj;
}
```

Figura 6: Linear Trajectory with cubic_polinomial function

```
/////////////////////////  (2.c) Definition of Lin_Trap trajectory  ////////////////////////////////////////////////////

trajectory_point KDLPlanner::compute_trapezoidal_linear( double time){
  double s;
  double ds;
  double dds;
  trapezoidal_vel(time,s,ds,dds);

  trajectory_point traj;
  Eigen::Vector3d pi=trajInit_;
  Eigen::Vector3d pf=trajEnd_;
  Eigen::Vector3d dif=pf-pi;

  traj.pos=(1-s)*pi+s*pf;
  traj.vel=(-pi+pf)*ds;
  traj.acc=(-pi+pf)*dds;

    return traj;
}
```

Figura 7: Linear Trajectory with trapezoidal function

```
trajectory_point KDLPlanner::compute_trajectory(double time,double trajRadius_,std::string profile, std::string traje)
{
  if(profile=="cubic" && traje == "linear"){
     return compute_cubic_linear(time);
   }

   if(profile=="cubic" && traje == "circular"){
     return compute_cubic_circular(time,trajRadius_);
   }

   if(profile=="trapezoidal" && traje == "linear"){
     return compute_trapezoidal_linear(time);
   }

  if(profile=="trapezoidal" && traje == "circular"){
     return compute_trapezoidal_circular(time,trajRadius_);
   }

}
```

Figura 8: compute_trajectory function

3. **Test the four trajectories**

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

By specifying the **traje** and **profile** values,it is possible to select one of the four previously defined trajectories.

In this example, a linear trajectory with a cubic profile has been defined.

```
    // Plan trajectory
    double traj_duration = 1.5, acc_duration = 0.5, t = 0.0, init_time_slot = 1.0, radius=0.1;

    std::string profile="cubic";
    std::string traje="linear";

    //constructor
    KDLPlanner planner(traj_duration, acc_duration, init_position,end_position);
    //KDLPlanner planner(traj_duration, acc_duration, radius);

    trajectory_point p = planner.compute_trajectory(t,radius,profile,traje);
```

Figura 9

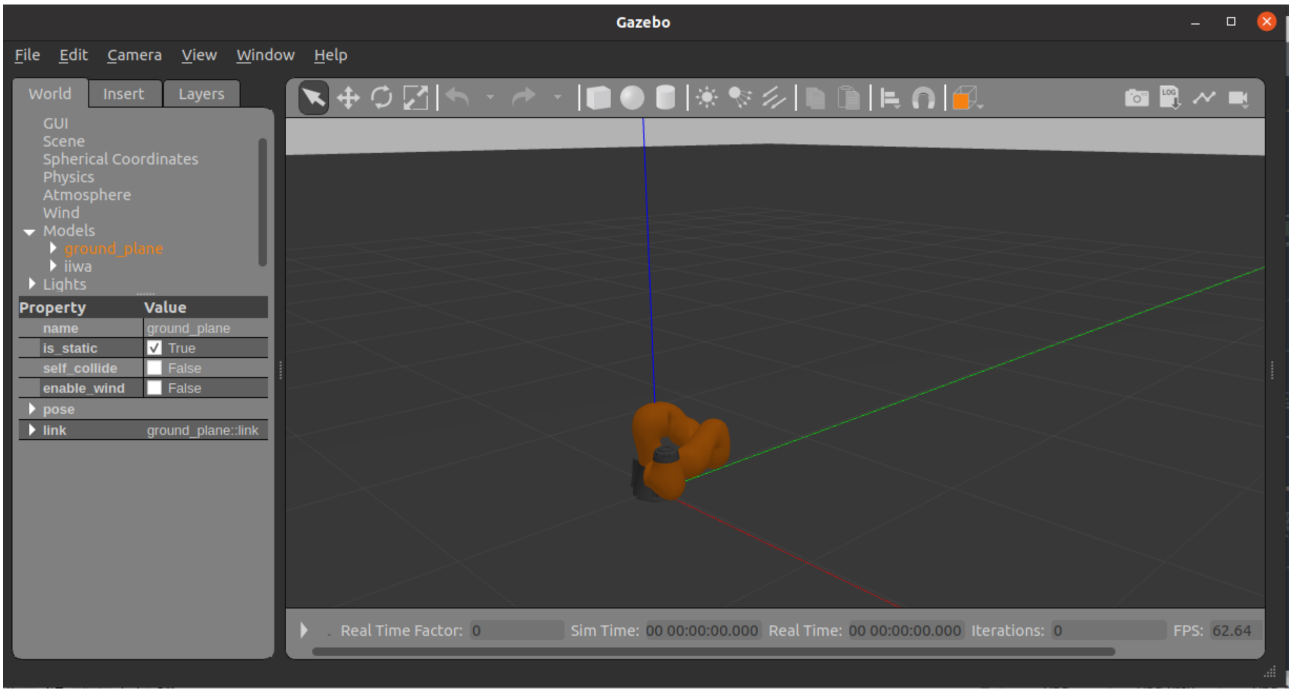

Figura 10: Iiwa Robot in initial configuration

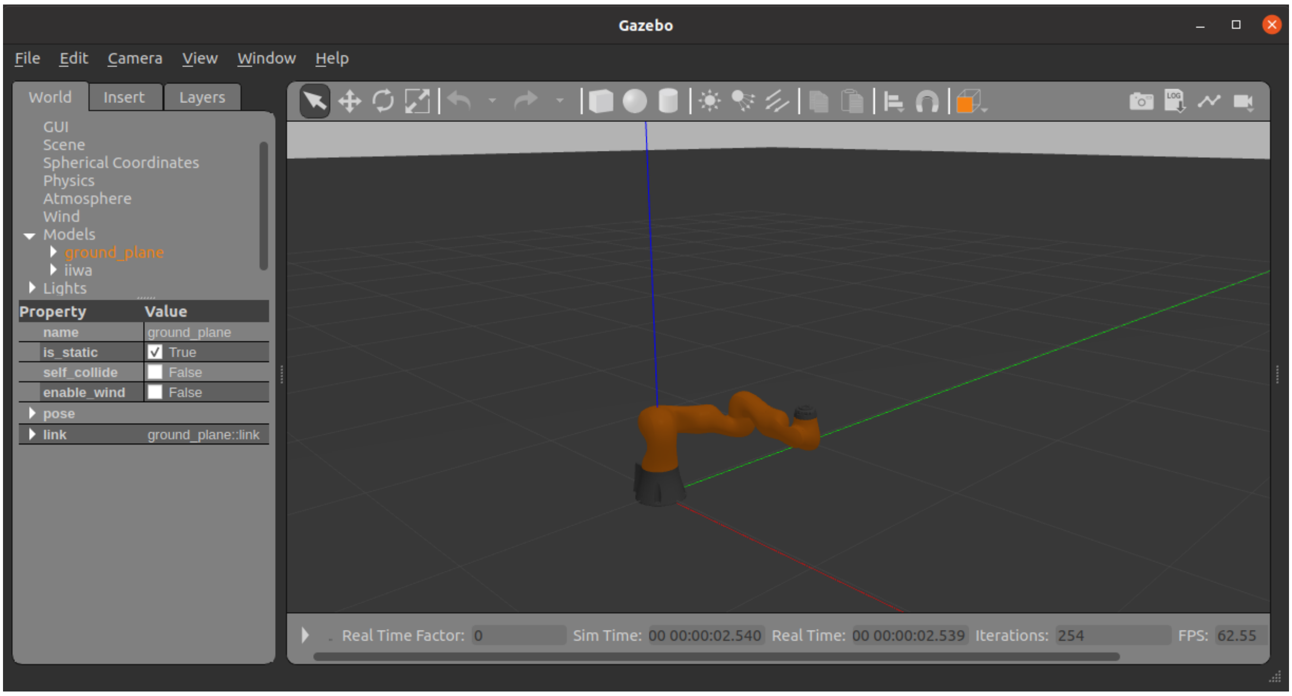

Figura 11: Iiwa Robot in final configuration

3.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.**

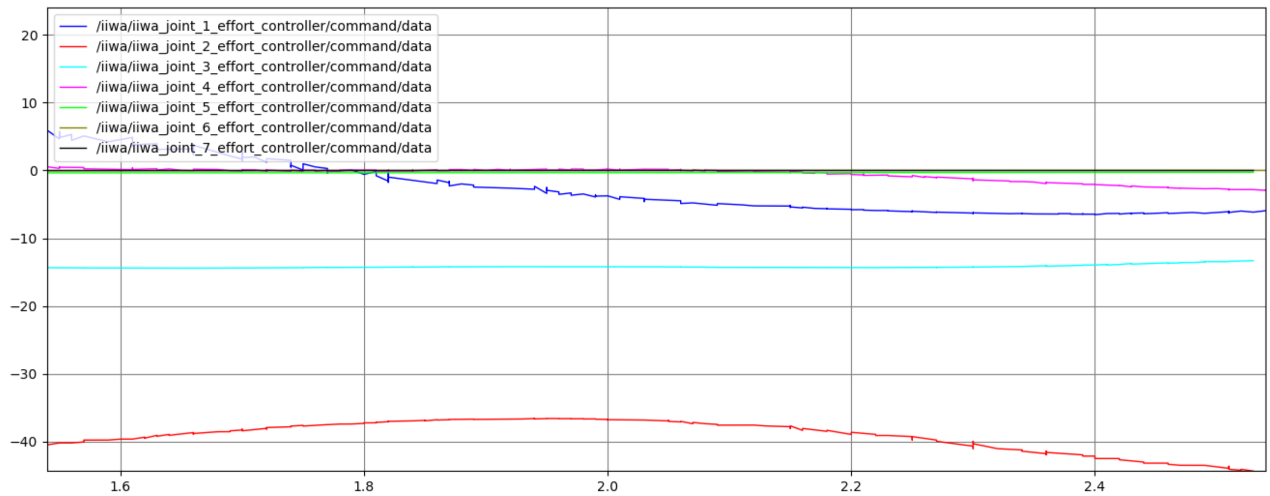

Figura 12: Linear Trajectory with cubic profile , Kp = 19, Kd = sqrt(Kp)

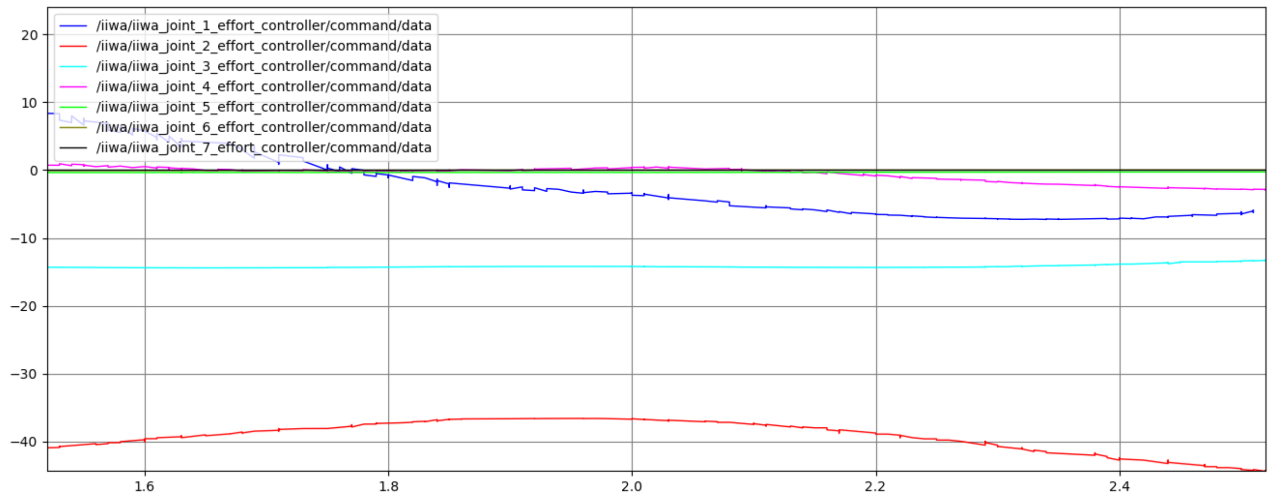

Figura 13: Linear Trajectory with trapezoidal profile , Kp = 19, Kd = sqrt(Kp)

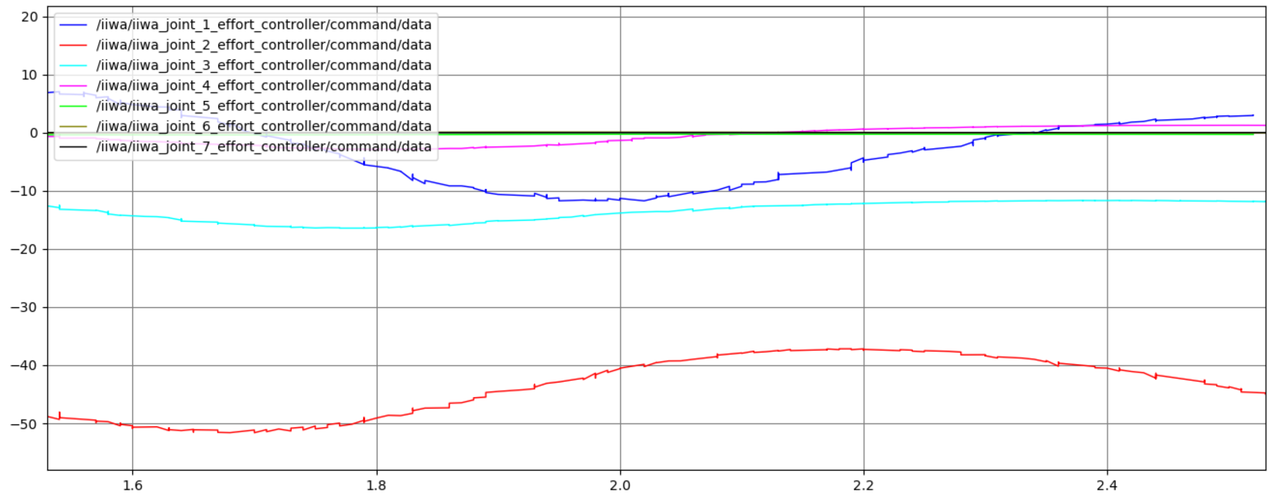

Figura 14: Circular Trajectory with cubic profile , Kp = 13, Kd = sqrt(Kp)

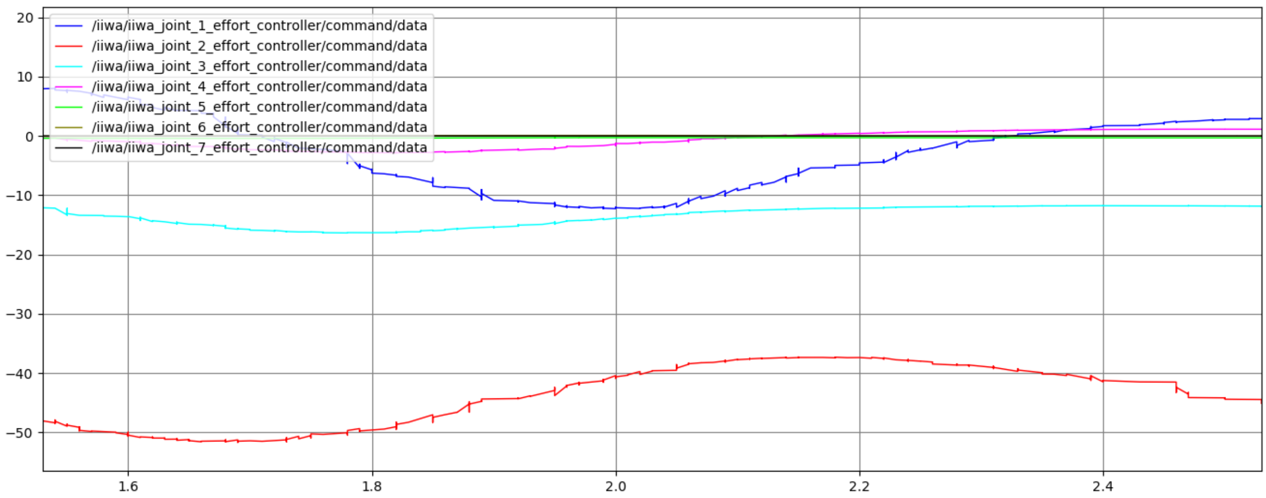

Figura 15: Circular Trajectory with trapezoidal profile , Kp = 13, Kd = sqrt(Kp)

4. **Develop an inverse dynamics operational space controller**

4.a-b) 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.

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

$$\tau = By + n \tag{2}$$

$$y = J_A^{\dagger}(\ddot{x}_d + K_D\dot{\tilde{x}} + K_P\tilde{x} - \dot{J}_A\dot{q} \tag{3}$$

```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, Kd;
    Kp=Eigen::MatrixXd::Zero(6,6);
    Kd=Eigen::MatrixXd::Zero(6,6);
    Kp.block(0,0,3,3) = _Kpp*Eigen::Matrix3d::Identity();
    Kp.block(3,3,3,3) = _Kpo*Eigen::Matrix3d::Identity();
    Kd.block(0,0,3,3) = _Kdp*Eigen::Matrix3d::Identity();
    Kd.block(3,3,3,3) = _Kdo*Eigen::Matrix3d::Identity();

    // read 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 = pseudoinverse(J);
```

Figura 16: function's arguments, gain matrices and functions to read the current state of the robot

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

    // velocity
    Eigen::Vector3d dot_p_d(_desVel.vel.data);
    Eigen::Vector3d dot_p_e(robot_->getEEVelocity().vel.data);
    Eigen::Vector3d omega_d(_desVel.rot.data);
    Eigen::Vector3d omega_e(robot_->getEEVelocity().rot.data);

    // acceleration
    Eigen::Matrix<double,6,1> dot_dot_x_d;
    Eigen::Matrix<double,3,1> dot_dot_p_d(_desAcc.vel.data);
    Eigen::Matrix<double,3,1> dot_dot_r_d(_desAcc.rot.data);
```

Figura 17: functions to read current and desired position, velocity and acceleration

```
    // compute linear errors
    Eigen::Matrix<double,3,1> e_p = computeLinearError(p_d,p_e);
    Eigen::Matrix<double,3,1> dot_e_p = computeLinearError(dot_p_d,dot_p_e);

    // compute orientation errors
    Eigen::Matrix<double,3,1> e_o = computeOrientationError(R_d,R_e);
    Eigen::Matrix<double,3,1> dot_e_o = computeOrientationVelocityError(omega_d,omega_e,R_d,R_e);
    Eigen::Matrix<double,6,1> x_tilde;
    Eigen::Matrix<double,6,1> dot_x_tilde;
    x_tilde << e_p, e_o;
    dot_x_tilde << dot_e_p, -omega_e;//dot_e_o;
    dot_dot_x_d << dot_dot_p_d, dot_dot_r_d;

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

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

     return M * (Jpinv*y)+ robot_->getGravity() + robot_->getCoriolis();
```

Figura 18: calculation of linear and orientation errors with inverse dynamics control law

The line of code relating to the inverse dynamics control law in **kdl_robot_test.cpp** has been uncommented.

The **getEEJacDotqDot()** function in **kdl_robot.cpp** was modified by multiplying the $\dot{J}$ matrix by $\dot{q}$.

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

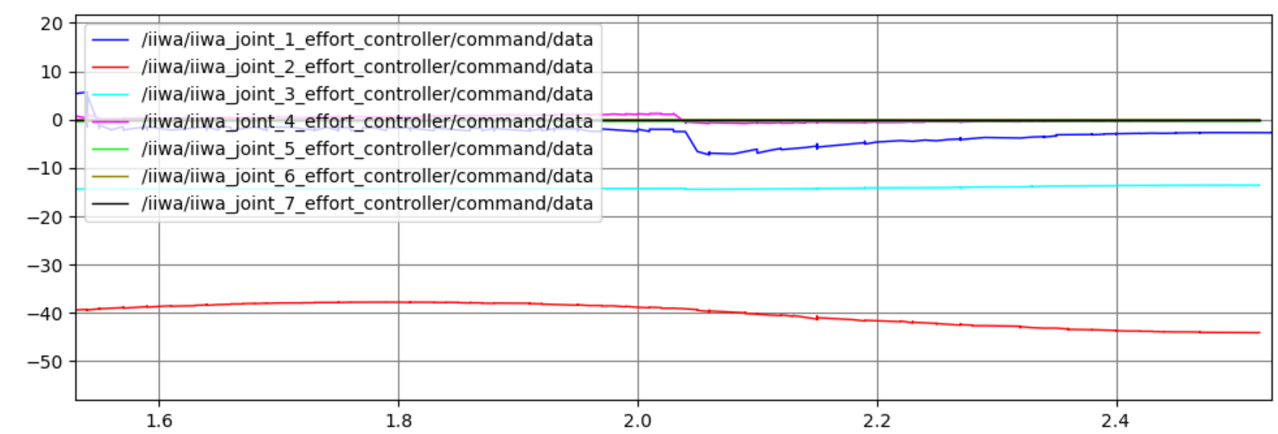

Figura 19: Linear Trajectory with trapezoidal profile , Kp = 20, Ko= 10, Kdp = sqrt(Kp) and Kdo = sqrt(Ko)

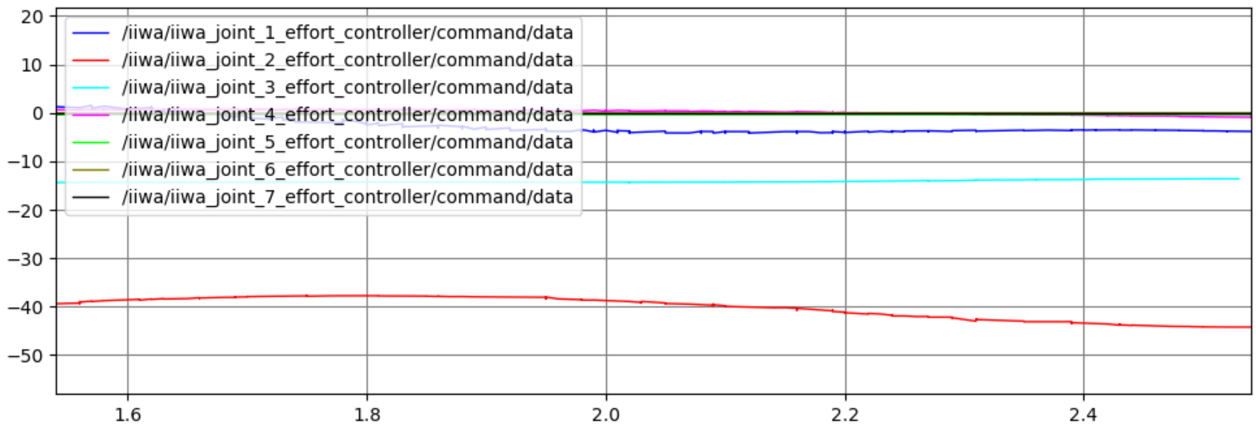

Figura 20: Linear Trajectory with cubic profile , Kp = 25, Ko= 10, Kdp = sqrt(Kp) and Kdo = sqrt(Ko)

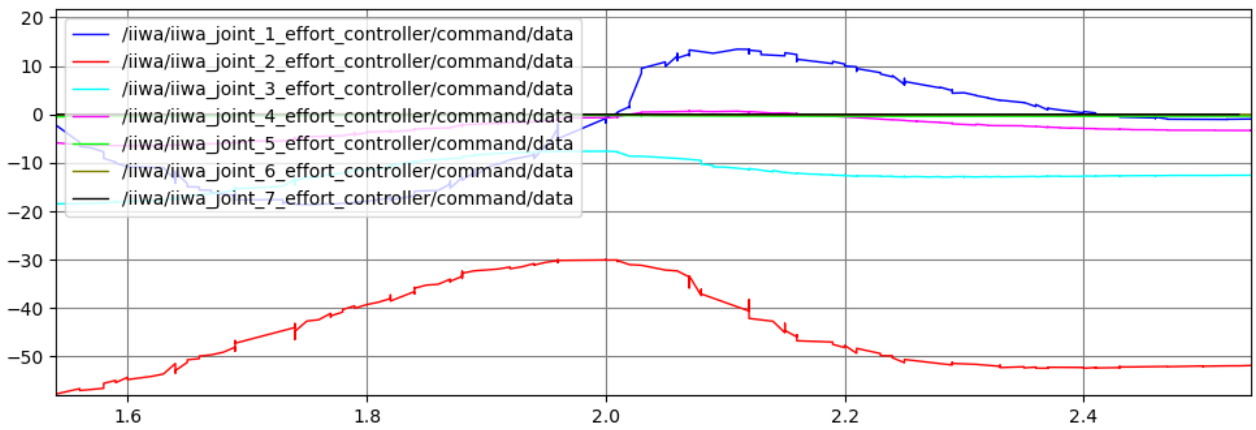

Figura 21: Circular Trajectory with trapezoidal profile , Kp = 30, Ko= 25, Kdp = sqrt(Kp) and Kdo = sqrt(Ko)

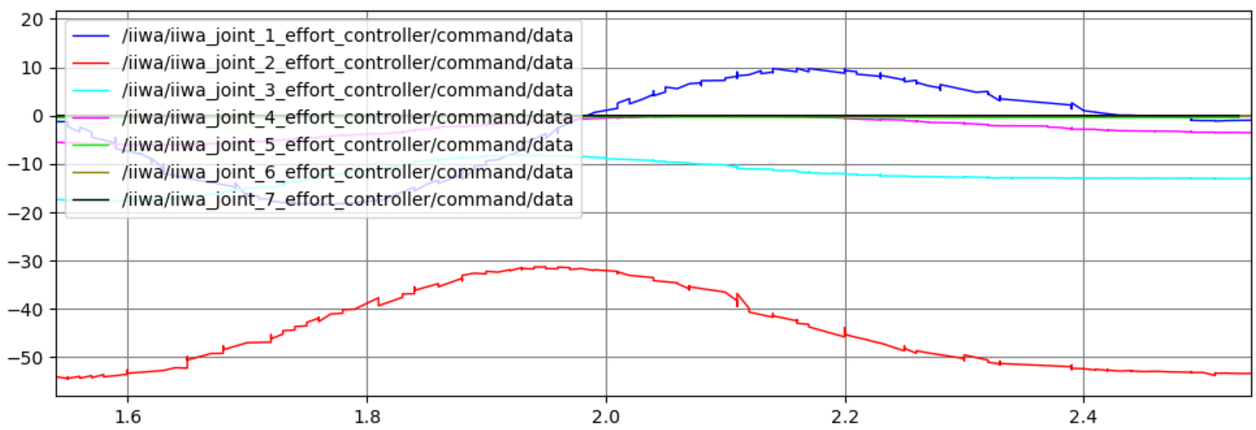

Figura 22: Circular Trajectory with cubic profile , Kp = 40, Ko= 40, Kdp = 1.5*sqrt(Kp) and Kdo = 1.5*sqrt(Ko)