UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria dell'Automazione e Robotica

Report

HOMEWORK 2

Academic Year 2024/25

Professor
Mario Selvaggio

Candidates
Pasquale Farese matr. **P38000285**
Nello Di Chiaro matr. **P38000286**
Gianmarco Corrado matr. **P38000287**
Andrea Colapinto matr. **P38000309**

Abstract

In this report will show how we succeeded in completing each task of the Homework 2.

The first task asks us to create trapezoidal and cubic polynomial velocity profiles.
The second assignment requires us to generate a linear and a circular trajectory, using the previously defined velocity profiles.
The third task requests the testing of both linear and circular trajectories using the joint space inverse dynamics controller, in order to verify that the motion adhered to the expected parameters.
The last one asks to develop an inverse dynamic operational space controller to allow the robot to accurately follow the planned trajectories.

Repositories

- Pasquale Farese: `https://github.com/PasFar/Homework-2.git`
- Nello Di Chiaro: `https://github.com/Nellodic34/Homework-2.git`
- Gianmarco Corrado: `https://github.com/giancorr/Homework-2.git`
- Andrea Colapinto: `https://github.com/colandrea02/Homework-2.git`

Contents

Chapter 1

Implementation of Velocity Profiles

Regarding the first task of the homework, first of all we cloned the repository https://github.com/RoboticsLab2024/ros2_kdl_package, since we used this package as starting point, and then we modified the kdl_planner.h and the kdl_planner.cpp in order to implement the trapezoidal velocity function, which is computed as follows:

Acceleration Phase $(0 \leq t < t_c)$:

In this phase, the velocity increases linearly while the acceleration remains constant at its maximum value $\ddot{s_c}$:

$$s(t) = \frac{1}{2}\ddot{s_c}t^2, \quad \dot{s}(t) = \ddot{s_c}t, \quad \ddot{s}(t) = \ddot{s_c}$$

Constant Velocity Phase ($t_c \leq t < t_f - t_c$):

Here, the system moves at a constant velocity, and the acceleration drops to zero:

$$s(t) = \ddot{s}_c t_c \left(t - \frac{t_c}{2} \right), \quad \dot{s}(t) = \ddot{s}_c t_c, \quad \ddot{s}(t) = 0$$

Deceleration Phase ($t_f - t_c \leq t \leq t_f$):

During deceleration, velocity decreases linearly while acceleration is negative and constant:

$$s(t) = 1 - \frac{1}{2}\ddot{s}_c (t_f - t)^2, \quad \dot{s}(t) = \ddot{s}_c (t_f - t), \quad \ddot{s}(t) = -\ddot{s}_c$$

It is possible to see how this has been implemented in the following picture.

```
void KDLPlanner::trapezoidal_vel(double time,
                                 double &s,
                                 double &s_dot,
                                 double &s_ddot)
{
  double acc_time = 0.5;
  double fin_time = trajDuration_;
  double s_ddot_c = 1/(acc_time*(fin_time-acc_time));

  if(time <= acc_time){
    s = 0.5*s_ddot_c*std::pow(time,2);
    s_dot = s_ddot_c * time;
    s_ddot = s_ddot_c;

  }
  else if(time <= fin_time - acc_time){
    s = s_ddot_c*acc_time*(time-acc_time/2);
    s_dot = s_ddot_c * acc_time;
    s_ddot = 0;

  }
  else
  {
    s = 1 - 0.5*s_ddot_c*std::pow(fin_time-time,2);
    s_dot = s_ddot_c * (fin_time-time);
    s_ddot = -s_ddot_c;

  }
}
```

Figure 1.1: Code snippet of *kdl_planner.cpp* - Trapezoidal velocity profile.

After doing that, we also implemented the cubic polynomial curvilinear abscissa in a specific function in kdl_planner.cpp, in which we set

$$a_0 = 0, \quad a_1 = 0, \quad a_2 = \frac{3}{t_f^2}, \quad a_3 = \frac{-2}{t_f^3}$$

in order to get the following boundary conditions:

$$s(0) = s_0 = 0, \quad s(t_f) = s_{tf} = 1, \quad \dot{s}(0) = 0, \quad \dot{s}(t_f) = 0$$

The code is presented below.

```
void KDLPlanner::cubic_polinomial(double time,
                                  double &s,
                                  double &s_dot,
                                  double &s_ddot)
{
  double fin_time = trajDuration_;
  double a_0, a_1, a_2, a_3;
  a_0 = 0.0;
  a_1 = 0.0;
  a_2 = 3/std::pow(fin_time, 2);
  a_3 = -2/std::pow(fin_time, 3);

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

Figure 1.2: Code snippet of *kdl_planner.cpp* - Cubic polynomial velocity profile.

Chapter 2

Trajectory planning

To complete the second task of the homework, we had to implement a linear and a circular trajectory.

Trajectory planning

The `circular_trajectory` function generates a circular trajectory given the starting point `trajInit_`, the radius `radius_` and uses the parameters s, $\dot{s}$, and $\ddot{s}$ of the chosen velocity profile to compute position, velocity, and acceleration. As the first point of the trajectory we have chosen the spawning point of the manipulator. Since a circular path in y-z plane is described as shown below,

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

where x_i, y_i and z_i are the coordinates of the center, we computed the

position coordinates as follows.

$$\texttt{traj.pos[0]} = \texttt{trajInit[0]}$$

$$\texttt{traj.pos[1]} = (\texttt{trajInit[1]} + \texttt{radius}) - \texttt{radius} \cdot \cos(2\pi \texttt{s})$$

$$\texttt{traj.pos[2]} = \texttt{trajInit[2]} - \texttt{radius} \cdot \cos(2\pi \texttt{s})$$

Then, we computed velocities and accelerations by differentiating the above equations. Moreover, we added a constructor for the planner that takes as arguments the trajectory duration, the radius of the circumference and the starting point of the trajectory. In the following picture it is possible to see how we implemented it.

```
void KDLPlanner::circular_trajectory(trajectory_point &traj, double s, double s_dot, double s_ddot)
{
  traj.pos[0] = trajInit_[0];
  traj.pos[1] = (trajInit_[1]+radius_)-radius_*cos(2*M_PI*s);
  traj.pos[2] = (trajInit_[2])-radius_*sin(2*M_PI*s);

  traj.vel[0] = 0.0;
  traj.vel[1] = 2*M_PI*radius_*sin(2*M_PI*s)*s_dot;
  traj.vel[2] = -2*M_PI*radius_*cos(2*M_PI*s)*s_dot;

  traj.acc[0] = 0.0;
  traj.acc[1] = std::pow(2*M_PI,2)*radius_*cos(2*M_PI*s)*std::pow(s_dot,2) + 2*M_PI*radius_*sin(2*M_PI*s)*s_ddot;
  traj.acc[2] = std::pow(2*M_PI,2)*radius_*sin(2*M_PI*s)*std::pow(s_dot,2) - 2*M_PI*radius_*cos(2*M_PI*s)*s_ddot;
}
```

Figure 2.1: Code snippet of *kdl_planner.cpp* - Circular trajectory.

A similar procedure is followed to generate a linear trajectory where positions, velocities, and accelerations are interpolated linearly between `trajInit_` and `trajEnd_`:

$$\texttt{traj.pos[i]} = \texttt{trajInit[i]} + s \cdot (\texttt{trajEnd[i]} - \texttt{trajInit[i]})$$

The code implementation is showed next.

```
void KDLPlanner::linear_trajectory(trajectory_point &traj, double s, double s_dot, double s_ddot)
{
  for(int i=0; i<3; i++){
    traj.pos[i] = trajInit_[i] + s*(trajEnd_[i]-trajInit_[i]);
  }
  for(int i=0; i<3; i++){
    traj.vel[i] = + s_dot*(trajEnd_[i]-trajInit_[i]);
  }
  for(int i=0; i<3; i++){
    traj.acc[i] = + s_ddot*(trajEnd_[i]-trajInit_[i]);
  }
}
```

Figure 2.2: Code snippet of *kdl_planner.cpp* - Linear trajectory.

Trajectories computation

```
trajectory_point KDLPlanner::compute_trajectory(double time)
{
  double s = 0.0, s_dot = 0.0, s_ddot = 0.0;

  if(vel_prof_ == "cubic"){
    cubic_polinomial(time, s,s_dot, s_ddot);
  }else if (vel_prof_ == "trapezoidal"){
    trapezoidal_vel(time, s, s_dot, s_ddot);
  }

  trajectory_point traj;

  if(trajectory_ == "circular"){
    circular_trajectory(traj, s, s_dot, s_ddot);
  }
  else if(trajectory_ == "linear"){
    linear_trajectory(traj, s, s_dot, s_ddot);
  }

  return traj;

}
```

Figure 2.3: Code snippet of *kdl_planner.cpp* - Compute trajectory.

This function computes the trajectory based on input parameters, such as the trajectory duration, the starting point, the ending point etc., combining the trajectory type (linear or circular) with the curvilinear abscissa profile (trapezoidal velocity or cubic position). The

function first determines the curvilinear abscissa profile for the trajectory using, as anticipated, either a cubic polynomial or a trapezoidal velocity profile. If the trajectory type is linear, the function calls `linear_trajectory` to compute the position, velocity, and acceleration along a straight line connecting the starting and the ending poses. If the trajectory type is circular, the function calls `circular_trajectory` to define a circular motion in the y−z plane while keeping the x-coordinate constant. After computing the trajectory, the function returns a `trajectory_point` object containing the computed **positions, velocities** and **accelerations**.

Chapter 3

Trajectory Testing

The third point of the homework requires using the provided joint space inverse dynamics controller to test the two previously presented trajectories (i.e., linear and circular), with one of the velocity profiles (i.e., trapezoidal or polynomial). In order to do that, we first created a new *empty.world* in which we set the gravity to 0. This allows us to have the robot spawn in the defined initial position without falling down due the effects of the gravity. Moreover, the simulation starts in running mode, enabling the controller and the broadcasters to correctly spawn, without worrying about the timeout. Below is shown how we managed to do it.

```
declared_arguments.append(DeclareLaunchArgument('gz_args', default_value=iiwa_simulation_world,
                          description='Arguments for gz_sim'),)

gazebo = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            [PathJoinSubstitution([FindPackageShare('ros_gz_sim'),
                                   'launch',
                                   'gz_sim.launch.py'])]),
        launch_arguments={'gz_args': [LaunchConfiguration('gz_args'), ' -r'],}.items(),
        condition=IfCondition(use_sim),
)
```

Figure 3.1: Code snippet of *iiwa.launch.py* - Defining the gazebo argument for world spawning.

Next, we added the *effort_ controller* to the manipulator controllers. After doing that, the robot is spawned in *Gazebo* with the specified controller and with *effort* as hardware interfaces.

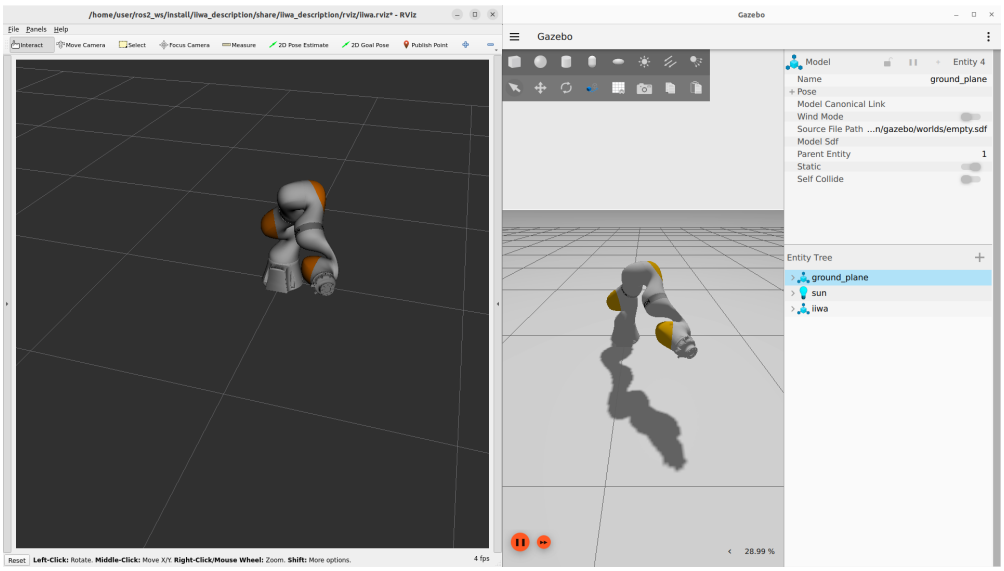

Figure 3.2: Manipulator in Gazebo and Rviz

Then we proceeded to the crucial point of the third task. We exploited the provided joint space inverse dynamics controller in order to give torques commands to the manipulator to solve the tracking

problem. The dynamic model of the robot is as follows:

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F\dot{q} + g(q) = \tau$$

In our problem, we omitted the friction value and the gravity one. This is because we don't have friction forces, and the manipulator is in a zero-gravity world.

Then we proceeded to compute the joint positions, velocities and accelerations. This can be done by exploiting the closed loop inverse kinematic algorithm, which enables us to compute the velocities of the joint using the following equation:

$$\dot{q} = J^{\dagger}(\dot{x}_d + Ke)$$

in which we set $K = 10$. Moreover, we used the angle-axis representation that allows us to use the geometric Jacobian.

Then, we can compute the joint accelerations by differentiating $\dot{q}$, and also the joint positions by integrating $\dot{q}$. After doing this, we can use the controller that computes the torques as following:

$$\tau = B(q)(\ddot{q}_d + K_D\dot{\tilde{q}} + K_P\tilde{q}) + C(q,\dot{q})\dot{q}$$

where $\ddot{q}$ is the desired joint acceleration, $\dot{\tilde{q}}$ is the joint velocities error and $\tilde{q}$ is the joint positions error.

The code is as follows.

```
Eigen::VectorXd KDLController::idCntr(KDL::JntArray &_qd,
                                      KDL::JntArray &_dqd,
                                      KDL::JntArray &_ddqd,
                                      double _Kp, double _Kd)
{
    // read current state
    Eigen::VectorXd q = robot_->getJntValues();
    Eigen::VectorXd dq = robot_->getJntVelocities();

    // calculate errors
    Eigen::VectorXd e = _qd.data - q;
    Eigen::VectorXd de = _dqd.data - dq;

    Eigen::VectorXd ddqd = _ddqd.data;

    return robot_->getJsim() * (ddqd + _Kd*de + _Kp*e)
            + robot_->getCoriolis()
            //+ robot_->getGravity() // due to Zero gravity world
            ;
}
```

Figure 3.3: Code snippet of *kdl_control.cpp* - Joint Space Inverse Dynamics Controller

3.1 Results

In this section it is possible to see the results of the torques computed with the joint space inverse dynamics controller with the two trajectories and with the two different velocity profiles.

- Linear trajectory - trapezoidal velocity profile.
- Circular trajectory - trapezoidal velocity profile.
- Linear trajectory - polynomial velocity profile.
- Circular trajectory - polynomial velocity profile.

```
Eigen::VectorXd KDLController::idCntr(KDL::JntArray &_qd,
                                      KDL::JntArray &_dqd,
                                      KDL::JntArray &_ddqd,
                                      double _Kp, double _Kd)
{
    // read current state
    Eigen::VectorXd q = robot_->getJntValues();
    Eigen::VectorXd dq = robot_->getJntVelocities();

    // calculate errors
    Eigen::VectorXd e = _qd.data - q;
    Eigen::VectorXd de = _dqd.data - dq;

    Eigen::VectorXd ddqd = _ddqd.data;

    return robot_->getJsim() * (ddqd + _Kd*de + _Kp*e)
           + robot_->getCoriolis()
           //+ robot_->getGravity() // due to Zero gravity world
           ;
}
```

Figure 3.4: Code snippet of *kdl_control.cpp* - Joint Space Inverse Dynamics Controller

3.1.1 Linear trajectory - trapezoidal velocity profile

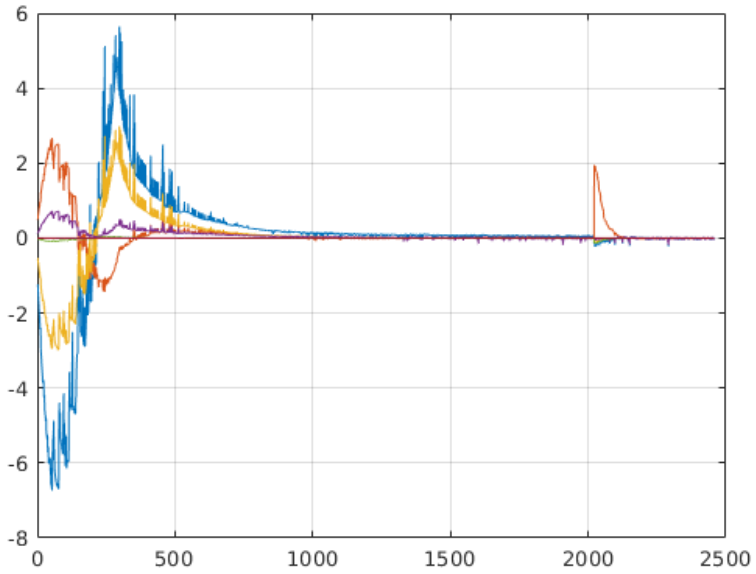

Figure 3.5: Torque commands - Linear trajectory, trapezoidal velocity profile

3.1.2 Circular trajectory - trapezoidal velocity profile

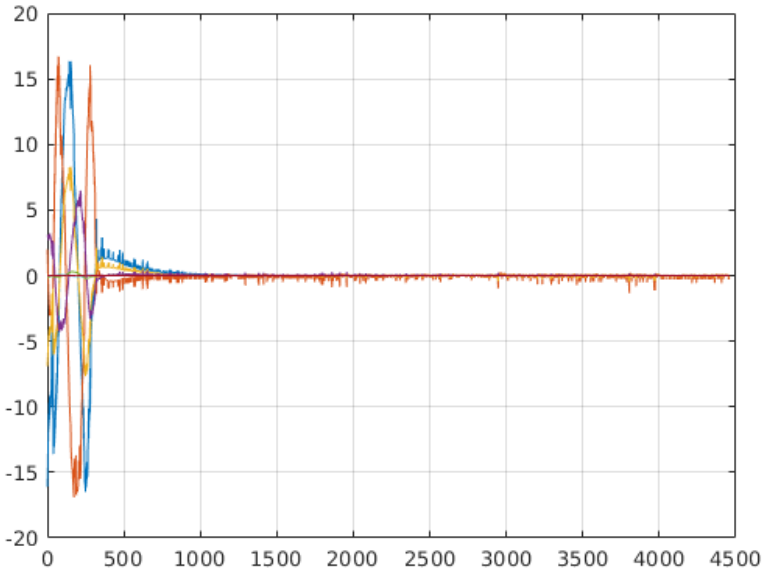

Figure 3.6: Torque commands - Circular trajectory, trapezoidal velocity profile

3.1.3 Linear trajectory - polynomial velocity profile

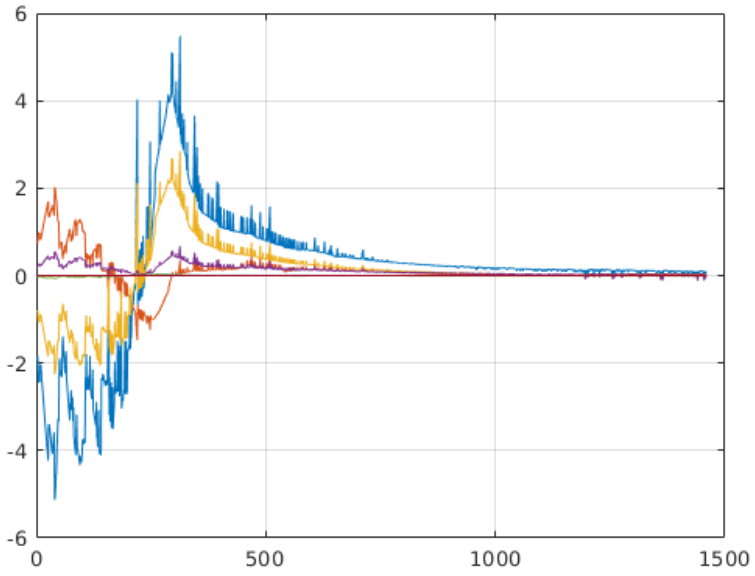

Figure 3.7: Torque commands - Linear trajectory, polynomial velocity profile

3.1.4 Circular trajectory - polynomial velocity profile

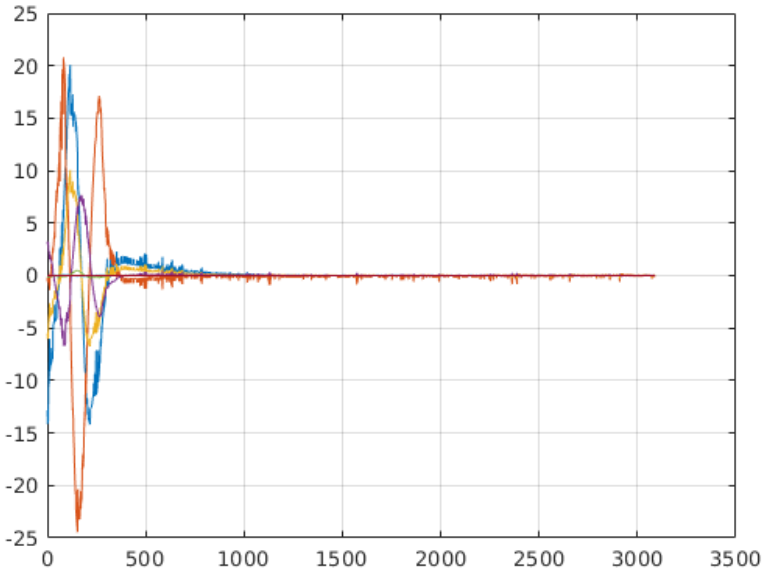

Figure 3.8: Torque commands - Circular trajectory, polynomial velocity profile

Chapter 4

Operational space inverse dynamics controller

The fourth point requires to implement an inverse dynamics operational space controller. In this case the error is defined in the cartesian space. In order to accomplish this task, we compute the frame of the desired pose, the desired velocities and the desired accelerations. Then we call the control function that computes the linear error and the orientation error (using angle-axis representation) and then computes the torques as follows:

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

where $\ddot{x}_d$ is the desired cartesian acceleration, $\dot{\tilde{x}}$ is the cartesian velocity error and $\tilde{x}$ is the cartesian positions error.

The code is as follows.

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

{

   Eigen::MatrixXd Jac = robot_->getEEJacobian().data;

   //read current cart state
   KDL::Frame cart_pos =robot_->getEEFrame();
   KDL::Twist cart_twist= robot_->getEEVelocity();


   //Define vel and pos error vectors
   Vector6d x_tilde;
   Vector6d dx_tilde;

   Eigen::VectorXd desAcc_; desAcc_=toEigen(_desAcc);

   //Transform Kp and Kd into matrixes
   Eigen::MatrixXd Kp(6,6);
   Eigen::MatrixXd Kd(6,6);
   Eigen::MatrixXd I= Eigen::MatrixXd::Identity(3,3);

   Kp.topLeftCorner(3,3)=I*_Kpp;
   Kp.bottomRightCorner(3,3)=I*_Kpo;
   Kd.topLeftCorner(3,3)=I*_Kdp;
   Kd.bottomRightCorner(3,3)=I*_Kdo;

   computeErrors(_desPos, cart_pos,_desVel, cart_twist, x_tilde, dx_tilde);

   Eigen::VectorXd y; y= pseudoinverse(Jac)*(desAcc_ + Kd*dx_tilde + Kp*x_tilde - robot_->getEEJacDot());

   return robot_->getJsim()*y + robot_->getCoriolis()
   //+ robot_->getGravity() Zero gravity world
   ;
}
```

Figure 4.1: Code snippet of *kdl_ control.cpp* - Operational Space Inverse Dynamics Controller

4.1 Results

In this section it is possible to see the results of the torques computed with the operational space inverse dynamics controller with the two trajectories and with the two different velocity profiles.

- Linear trajectory - trapezoidal velocity profile.
- Circular trajectory - trapezoidal velocity profile.
- Linear trajectory - polynomial velocity profile.

- Circular trajectory - polynomial velocity profile.

4.1.1 Linear trajectory - trapezoidal velocity profile

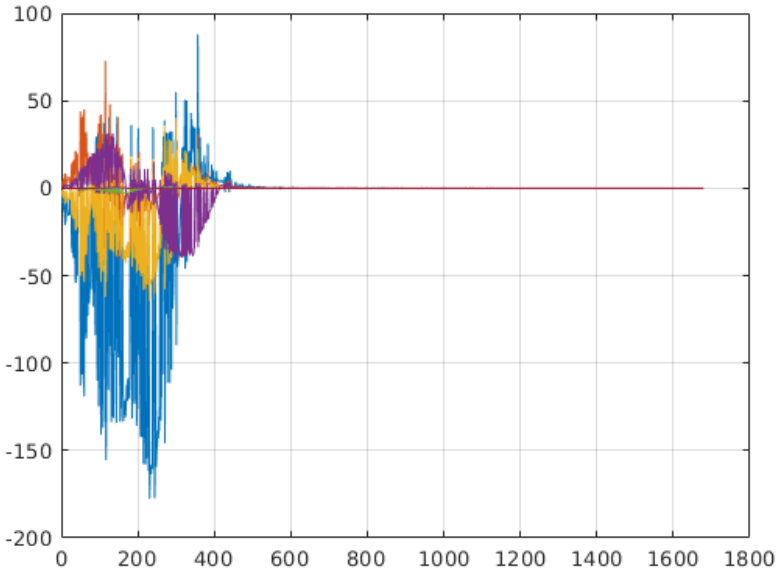

Figure 4.2: Torque commands - Linear trajectory, trapezoidal velocity profile

4.1.2 Circular trajectory - trapezoidal velocity profile

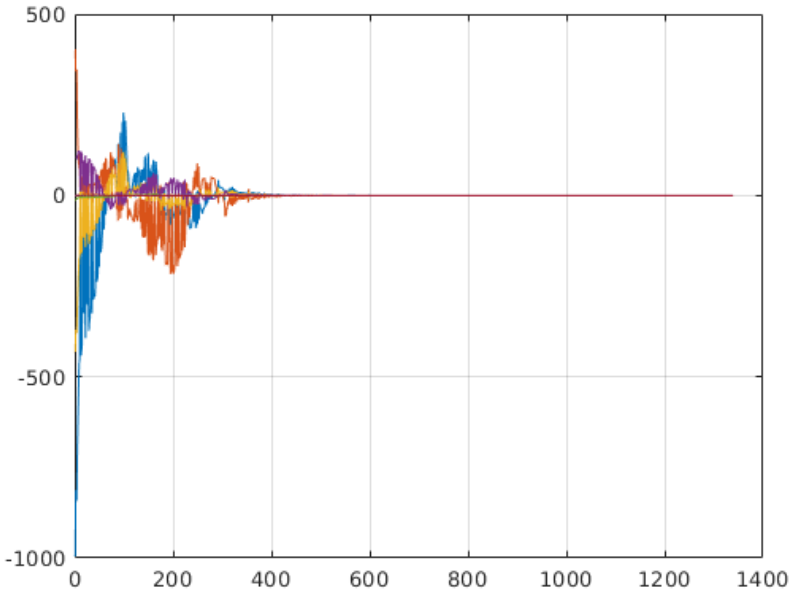

Figure 4.3: Torque commands - Circular trajectory, trapezoidal velocity profile

4.1.3 Linear trajectory - polynomial velocity profile

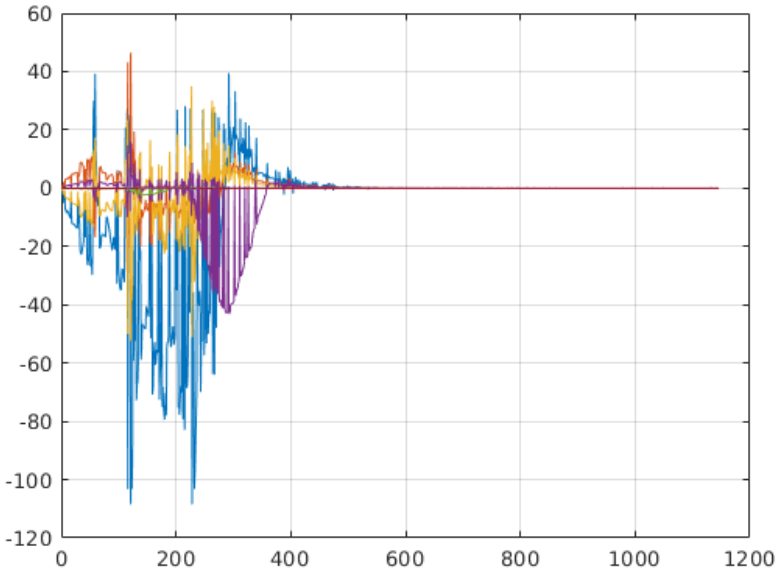

Figure 4.4: Torque commands - Linear trajectory, polynomial velocity profile

4.1.4 Circular trajectory - polynomial velocity profile

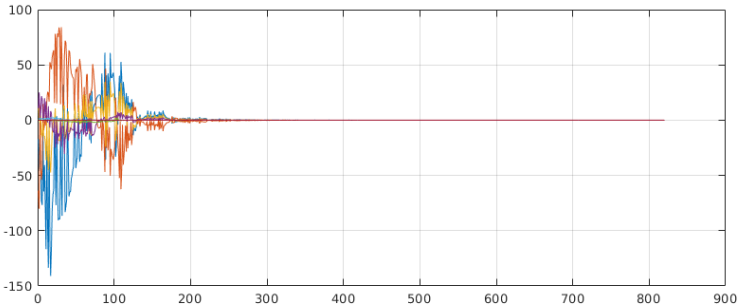

Figure 4.5: Torque commands - Circular trajectory, polynomial velocity profile