Robotics Lab

Report Homework 2

Federico Trenti P38000263
Matteo Russo P38000247
Alessandra Del Sorbo P38000289
Giulio Acampora P38000258

2024/2025

GitHub Links

- Link Trenti
- Link Russo
- Link Del Sorbo
- Link Acampora

Contents

Chapter 1

1.1 Substitution of the current trapezoidal velocity profile with a cubic polinomial linear trajectory

First of all we downloaded the "ros2_kdl_package" package from the Github repository.

```
$ git clone https://github.com/RoboticsLab2024/
    ros2_kdl_package.git
```

After that we modified appropriately the KDLPlanner class, inside the files "kdl_planner.h" and "kdl_planner.cpp", in order to 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 our trajectory. Considering that a trapezoidal velocity profile for a curvilinear abscissa $s \in [0, 1]$ is defined as:

$$s(t) = \begin{cases} \frac{1}{2}s_c t^2 & \text{se } 0 \leq t \leq t_c, \\ s_c t_c (t - t_c/2) & \text{se } t_c < t \leq t_f - t_c, \\ 1 - \frac{1}{2}s_c (t_f - t)^2 & \text{se } t_f - t_c < t \leq t_f. \end{cases}$$

```cpp
//Definition of the prototype in kdl_planner.h
void trapezoidal_vel(double t, double tc, double& s,
    double& s_dot, double& s_ddot);
```

```cpp
//Implementation in kdl_planner.cpp
void KDLPlanner::trapezoidal_vel(double time, double
    accDuration, double& s, double& s_dot, double&
    s_ddot){
```

```cpp
  /* trapezoidal velocity profile with accDuration acceleration time period and trajDuration_ total duration.
     time = current time
     trajDuration_  = final time
     accDuration    = acceleration time
     trajInit_ = trajectory initial point
     trajEnd_  = trajectory final point */

  double sc_ddot=-1.0/(std::pow(accDuration,2)-trajDuration_*accDuration);
  if(time <= accDuration)
  {
    s = 0.5*sc_ddot*std::pow(time,2);
    s_dot = sc_ddot*time;
    s_ddot = sc_ddot;
  }
  else if(time <= trajDuration_-accDuration)
  {
    s = sc_ddot*accDuration*(time-(accDuration/2));
    s_dot = sc_ddot*accDuration;
    s_ddot = 0.0;
  }
  else
  {
    s = 1 - 0.5*sc_ddot*std::pow(trajDuration_-time,2);
    s_dot = sc_ddot*(trajDuration_-time);
    s_ddot = -sc_ddot;
  }
}
```

From there, we had to create another function named `KDLPlanner::cubic_polynomial` that creates the cubic polynomial curvilinear abscissa for our trajectory. In this case, we considered the formula:

$$s(t) = a_3 t^3 + a_2 t^2 + a_1 t + a_0$$

So we added in both previous files the prototype and its implementation.

```cpp
//Definition of the prototype in kdl_planner.h
void cubic_polynomial(double t, double& s, double& s_dot, double& s_ddot);
```

```cpp
//Implementation in kdl_planner.cpp
void KDLPlanner::cubic_polynomial(double t, double& s, double& s_dot, double& s_ddot){
```

```
    double a0, a1, a2, a3, s0, sDot0, sDotf, sf; // Coefficienti del polinomio cubico
    //condizioni iniziali
    s0 = 0;
    sDot0 = 0;
    sDotf = 0;
    sf = 1;
    // Calcola i coefficienti del polinomio cubico
    a0 = s0;
    a1 = sDot0;
    a2 = (3 * (sf - s0) / (trajDuration_ * trajDuration_)) - ((2 * sDot0 + sDotf) / trajDuration_);
    a3 = (-2 * (sf - s0) / (trajDuration_ * trajDuration_ * trajDuration_)) + ((sDot0 + sDotf) / (trajDuration_ * trajDuration_));

    // Calcola s(t), s'(t) e s''(t) usando i coefficienti a0, a1, a2, a3
    s = a3 * t * t * t + a2 * t * t + a1 * t + a0;
    s_dot = 3 * a3 * t * t + 2 * a2 * t + a1;
    s_ddot = 6 * a3 * t + 2 * a2;
}
```

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 our trajectory.

Chapter 2

2.1 Creation of circular trajectories for the robot

We began by creating a new constructor `KDLPlanner::KDLPlanner` that takes as arguments the time duration, the starting point and the radius of our trajectory and store them in the corresponding class variables.
As before, we updated "kdl_planner.h" and "kdl_planner.cpp".

```
//Definition in kdl_planner.h
KDLPlanner(double _trajDuration, Eigen::Vector3d
    _trajInit, double _trajRadius);
```

```
//Implementation in kdl_planner.cpp
KDLPlanner::KDLPlanner(double _trajDuration, Eigen::
    Vector3d _trajInit, double _trajRadius)
{
    trajDuration_ = _trajDuration;
    trajInit_ = _trajInit;
    trajRadius_ = _trajRadius;
    trajEnd_ = Eigen::Vector3d::Zero();
}
```

In order to obtain the desidered trajectories, we implemented the positional path directly in the function `KDLPlanner::compute_trajectory` and in `KDLPlanner::compute_trajectory_circ`.

```
//Computing trajectories for rectilinear path and
    trapezoidal
trajectory_point KDLPlanner::compute_trajectory(double
    time, double accDuration)
{
  double s,s_dot,s_ddot;
  this->trapezoidal_vel(time, accDuration, s, s_dot,
     s_ddot);
```

```
  trajectory_point traj;

  traj.pos = trajInit_ + s*(trajEnd_-trajInit_);
  traj.vel = s_dot*(trajEnd_-trajInit_);
  traj.acc = s_ddot*(trajEnd_-trajInit_);
  return traj;

}

//Computing trajectories for rectilinear path and
   cubic_polynomial
trajectory_point KDLPlanner::compute_trajectory(double
   time)
{
  double s,s_dot,s_ddot;
  this->cubic_polynomial(time, s, s_dot, s_ddot);
  trajectory_point traj;

  traj.pos = trajInit_ + s*(trajEnd_-trajInit_);
  traj.vel = s_dot*(trajEnd_-trajInit_);
  traj.acc = s_ddot*(trajEnd_-trajInit_);
  return traj;

}
```

```
//Computing trajectories for circular path and
   trapezoidal
trajectory_point KDLPlanner::compute_trajectory_circ(
   double time, double accDuration)
{
  double s,s_dot,s_ddot;
  this->trapezoidal_vel(time, accDuration, s, s_dot,
     s_ddot);
  trajectory_point traj;

  traj.pos.x() = trajInit_.x();
  traj.pos.y() = trajInit_.y() - trajRadius_*cos(2*M_PI
     *s);
  traj.pos.z() = trajInit_.z() - trajRadius_*sin(2*M_PI
     *s);
  traj.vel.x() = 0;  //s_dot*(trajEnd_-trajInit_);
  traj.vel.y() = trajRadius_*sin(2*M_PI*s)*2*M_PI*s_dot
     ;
  traj.vel.z() = -trajRadius_*cos(2*M_PI*s)*2*M_PI*
     s_dot;
  traj.acc.x() = 0;
```

```
  traj.acc.y() = -trajRadius_*2*M_PI*(-2*M_PI*sin(2*M_PI*s))*pow(s_dot,2)+cos(2*M_PI*s)*s_ddot;
  traj.acc.z() = -trajRadius_*2*M_PI*(-trajRadius_*sin(2*M_PI*s)*pow(s_dot,2)+cos(2*M_PI*s)*s_ddot);

  //traj.acc = s_ddot*(trajEnd_-trajInit_);
  return traj;

}


//Computing trajectories for circular path and trapezoidal
trajectory_point KDLPlanner::compute_trajectory_circ(double time)
{
  double s,s_dot,s_ddot;
  this->cubic_polynomial(time, s, s_dot, s_ddot);
  trajectory_point traj;

  traj.pos.x() = trajInit_.x();
  traj.pos.y() = trajInit_.y() - trajRadius_*cos(2*M_PI*s);
  traj.pos.z() = trajInit_.z() - trajRadius_*sin(2*M_PI*s);
  traj.vel.x() = 0;  //s_dot*(trajEnd_-trajInit_);
  traj.vel.y() = trajRadius_*sin(2*M_PI*s)*2*M_PI*s_dot;
  traj.vel.z() = -trajRadius_*cos(2*M_PI*s)*2*M_PI*s_dot;
  traj.acc.x() = 0;
  traj.acc.y() = -trajRadius_*2*M_PI*(-2*M_PI*sin(2*M_PI*s))*pow(s_dot,2)+cos(2*M_PI*s)*s_ddot;
  traj.acc.z() = -trajRadius_*2*M_PI*(-trajRadius_*sin(2*M_PI*s)*pow(s_dot,2)+cos(2*M_PI*s)*s_ddot);

  //traj.acc = s_ddot*(trajEnd_-trajInit_);
  return traj;

}
```

Chapter 3

3.1 Test of the trajectories

Considering the trajectories obtained in the previous chapter, we tested them with the joint space inverse dynamic controller. Before plotting, the torques sent to the manipulator, we tuned appropriately the gains K_p and K_d to reach a smooth behavior.
The function that return the torques by taking as inputs the desire joints position,velocity and acceleration:

```
    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() /*friction compensation?*/; // gravity has been set to 0 in the .world
}
```

As follow, the node's implementation is:

```
    //  JOINT SPACE INVERSE DYNAMICS CONTROL

                    // Compute differential IK

                    joint_velocities_old_.data = joint_velocities_.data;

                    Vector6d cartvel; cartvel << p.vel + 5*error, o_error;
                    joint_velocities_.data = pseudoinverse(robot_->getEEJacobian().data)*cartvel;
                    joint_positions_.data = joint_positions_.data + joint_velocities_.data*dt;
                    joint_accelerations_d_.data = (joint_velocities_.data - joint_velocities_old_.data)/dt;


                    KDLController controller_(*robot_);
                    joint_efforts_.data = controller_.idCntr(joint_positions_, joint_velocities_, joint_accelerations_d_, 50.0, 5.0);

                    KDL::Frame frame_final = robot_->getEEFrame();
                    KDL::Twist velocities_final; velocities_final.vel=KDL::Vector::Zero(); velocities_final.rot=KDL::Vector::Zero();
                    KDL::Twist acceleration_final; acceleration_final.vel=KDL::Vector::Zero(); acceleration_final.rot=KDL::Vector::Zero();
```

Moreover, to ensure Gazebo starts at launch the default value of "use_sim" has been set to true in both the "iiwa.launch.py" file and the "iiwa.config.xacro" file.

```
<xacro:arg name="use_sim" default="true" />
```

```
declared_arguments.append(
        DeclareLaunchArgument(
            'use_sim',
            default_value='true',
            description='Start robot in Gazebo
               simulation.',
        )
    )
```

To run our world, it has been added in "iiwa.launch.py" the following code:

```
declared_arguments.append(
        DeclareLaunchArgument(
            'gz_args',
            default_value=[iiwa_simulation_world, ' -r
               -v 1'],
            description='Arguments for gz_sim'
        )
    )
```

The effort_controller has added inside the file "iiwa_controllers.yaml":

```
    effort_controller:
      type: effort_controllers/
         JointGroupEffortController
```

```
effort_controller:
  ros__parameters:
    command_interfaces:
      - effort
    state_interfaces:
      - position
      - velocity
      - effort
    joints:
      - joint_a1
      - joint_a2
      - joint_a3
      - joint_a4
      - joint_a5
      - joint_a6
      - joint_a7

    state_publish_rate: 200.0
    action_monitor_rate: 20.0
```

In the file "empty.world" has been added two links from internet:

```
    <include>
      <uri>https://fuel.gazebosim.org/1.0/OpenRobotics/
         models/Ground Plane</uri>
    </include>
    <include>
      <uri>https://fuel.gazebosim.org/1.0/OpenRobotics/
         models/Sun</uri>
    </include>
```

To visualize them it's possible to use the command:

```
$ ros2 run rqt_plot rqt_plot
```

Then, we selected `/effort_controller/commands/data[i]` for all the seven joints.

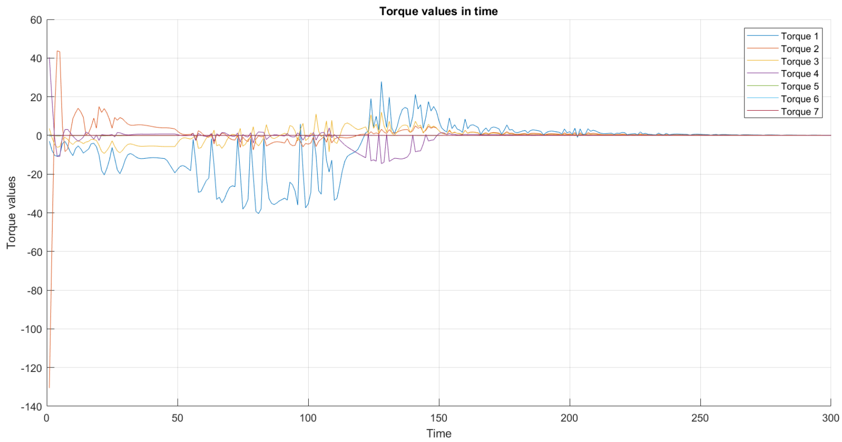

Figure 3.1: Torques values in joints space for a linear trapezoidal trajectory

After that, we created a script in MATLAB which receive a bag file that contains the informations about the torques of each joint. In order to create this bag file we used the command:

```
$ ros2 bag record effort_controller/commands/data -
  o my_bag
```

This bag file was sent to a MATLAB script to retrieve the desidered plot of the torques. This file is available in the repository.

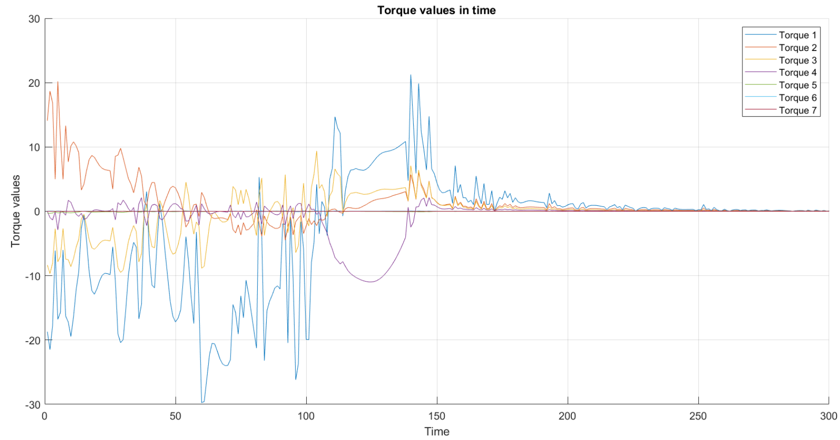

Figure 3.2: Torques values in joints space for a linear cubic_polynomial trajectory

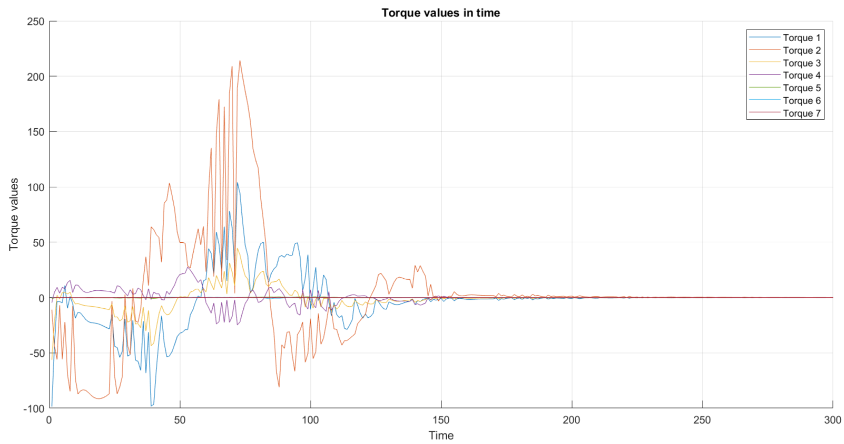

Figure 3.3: Torques values in joints space for a circular trapezoidal trajectory

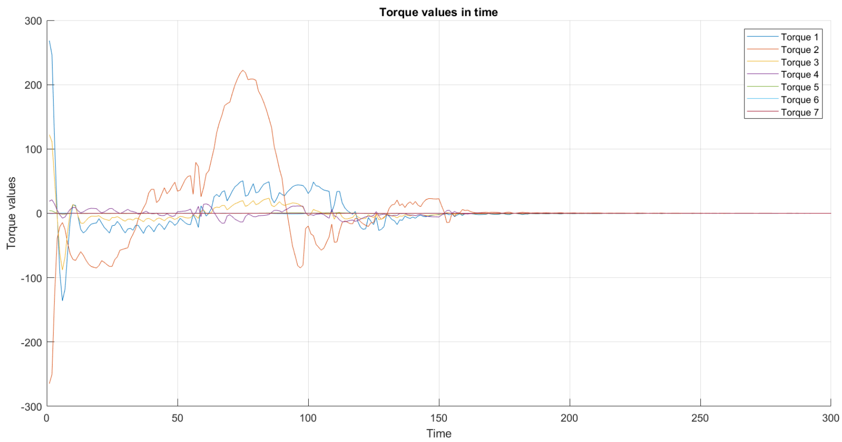

Figure 3.4: Torques values in joints space for a circular cubic_polynomial trajectory

Chapter 4

4.1 Develop an inverse dynamics operational space controller

Then the four trajectories have been implemented with the operational space inverse dynamic controller tuning appropriately the gains Kpp, Kpo, Kdp, Kdo to reach a smooth behaviour and making sure the manipulator stays at rest after completing the trajectory. The operational space controller function takes as inputs the desired frame, velocity and acceleration and the four gains and returns the torque. The operational space controller has been tested on the four trajectories and the joint torque commands has been plot. The code is :

```
    Eigen::VectorXd KDLController::idCntr(KDL::Frame &_desPos,
                                          KDL::Twist &_desVel,
                                          KDL::Twist &_desAcc,
                                          double _Kpp, double _Kpo,
                                          double _Kdp, double _Kdo)
{
    // read current state
    KDL::Frame x = robot_->getEEFrame();
    KDL::Twist dx = robot_->getEEVelocity();

    Vector6d x_tilde, dot_x_tilde;

    Eigen::Matrix<double,6,6> Kp;
    Eigen::Matrix<double,6,6> Kd;
    Kp.setZero();
    Kd.setZero();

```

```
    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();

    computeErrors(_desPos,x,_desVel,dx,x_tilde,dot_x_tilde);

    for (unsigned int i=0;i<3;i++){x_tilde(i)=_Kpp*x_tilde(i);dot_x_tilde(i)=_Kdp*dot_x_tilde(i);}
    for (unsigned int i=3;i<6;i++){x_tilde(i)=_Kpo*x_tilde(i);dot_x_tilde(i)=_Kdo*dot_x_tilde(i);}

    Eigen::Matrix<double,7,1> y;
    y = pseudoinverse(robot_->getEEJacobian().data)*(toEigen(_desAcc)+x_tilde+dot_x_tilde-robot_->getEEJacDotqDot());

    return  robot_->getJsim()*y + robot_->getCoriolis(); //+ robot_->getGravity();      // gravity has been set to 0 in the .world
}
```

```
// OPERATIONAL SPACE INVERSE DYNAMICS CONTROL

                    Kpp = 250.0;
                    Kpo = 250.0;
                    Kdp = 50.0;
                    Kdo = 50.0;

                    KDL::Frame desired_frame;
                    desired_frame.p = toKDL(p.pos);
                    desired_frame.M = cartpos.M;

                    KDL::Twist desired_vel;
                    desired_vel.vel = toKDL(p.vel);

                    KDL::Twist desired_accel;
                    desired_accel.vel = toKDL(p.acc);

                    robot_->getInverseKinematics(desired_frame,
```

```
                    des_joint_positions_);
                    robot_->getInverseKinematicsVel(desired_vel, des_joint_velocities_);
                    robot_->getInverseKinematicsAcc(desired_accel, joint_accelerations_d_);  // defined in utils.h

                    joint_efforts_.data=controller_.idCntr(desired_frame,desired_vel,desired_accel,Kpp, Kpo, Kdp, Kdo);
                }
```

To stop the manipulator, the code is:

```
    if(cmd_interface_ =="effort")
                {
                    KDLController controller_(*robot_);
                    des_joint_velocities_.data=Eigen::VectorXd::Zero(7,1);
                    des_joint_accelerations_.data=Eigen::VectorXd::Zero(7,1);

                    joint_efforts_.data=controller_.KDLController::idCntr(joint_positions_, des_joint_velocities_, des_joint_accelerations_, 50.0, 5.0);

                    robot_->update(toStdVector(joint_positions_.data),toStdVector(joint_velocities_.data));

                    for (long int i = 0; i < joint_velocities_.data.size(); ++i) {
                        desired_commands_[i] = joint_efforts_.data[i];
                    }
                }
```

4.2 Plot of the torques in the operation space for the given trajectories

As follow, the plot of the torques in the operational space:

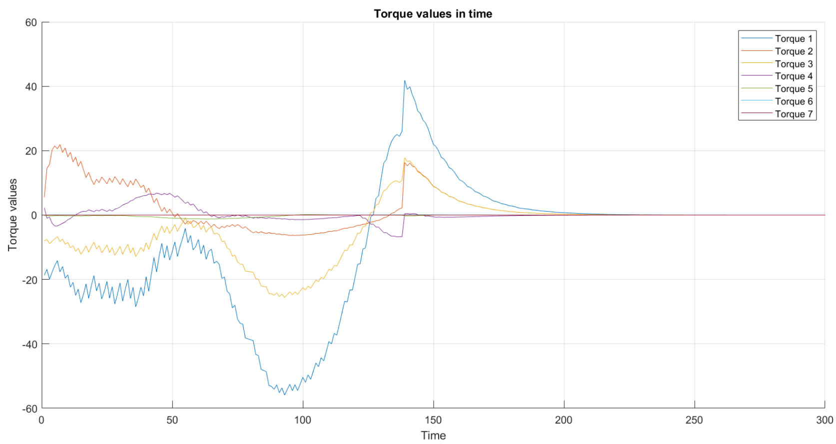

Figure 4.1: Torques values in operational space for a linear trapezoidal trajectory

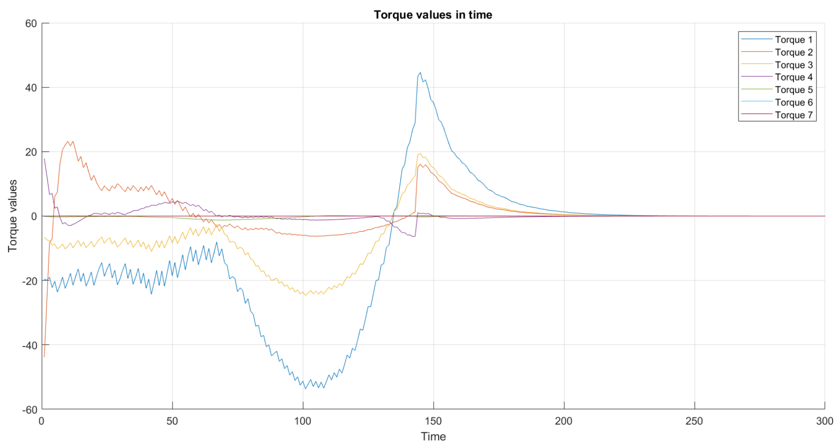

Figure 4.2: Torques values in operational space for a linear cubic_polynomial trajectory

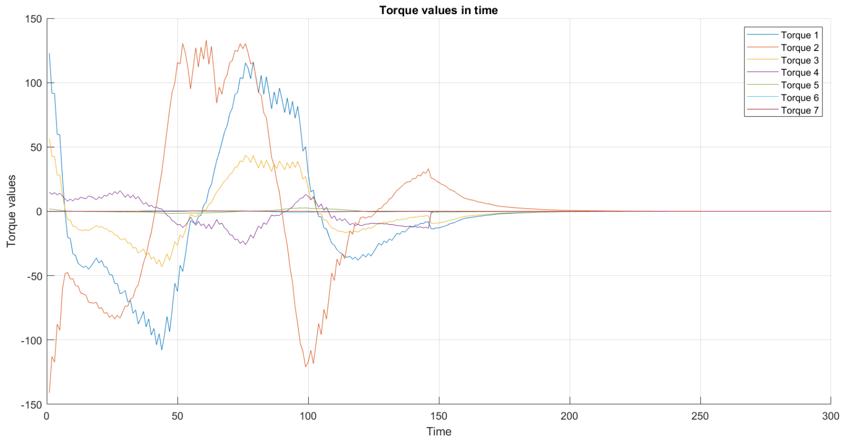

Figure 4.3: Torques values in operational space for a circular trapezoidal trajectory

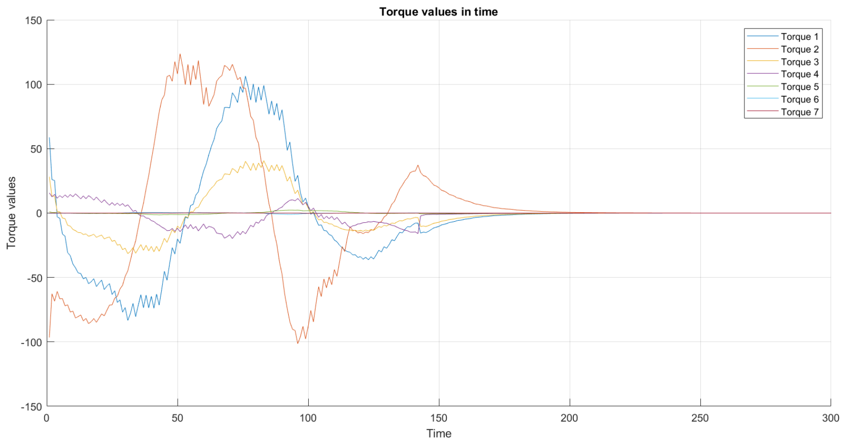

Figure 4.4: Torques values in operational space for a circular cubic_polynomial trajectory