Robotics Lab course

Homework 2
Control a manipulator to follow a trajectory

Student:
Vincenzo Palomba
P38000180

Teacher:
Prof. Mario Selvaggio
Assistants:
Andrea Capuozzo
Claudio Chiariello

GitHub repo

It is possible to find all the code used in this report at: https://github.com/vincip99/RoboticsLabHW2.

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

- 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

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

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

  First of all the new function prototype must be added into the header file of the `KDLplanner` class:

kdl_planner.h

```
void trapezoidal_vel(double t, double t_c, double &s, double &s_dot, double &
    s_ddot);
```

Notice that the function itself is void so it does not return anything. However the values of s, $\dot{s}$ and $\ddot{s}$ are passed by reference, so they are correctly modified. Next, the function definition is written:

kdl_planner.cpp

```
/**
 * @brief [trapezoidal velocity profile function].
 * @param [t] [current time at which the curvilinear abscissa is computed].
 * @param [t_c] [cruise time].
 * @param [s] [curvilinear abscissa]
 * @param [s_dot] [velocity]
 * @param [s_ddot] [acceleration]
 */
void KDLPlanner::trapezoidal_vel(double t, double t_c, double &s, double &
    s_dot, double &s_ddot)
{
  // Compute s_ddot_c
  double s_ddot_c = -1.0 / (std::pow(t_c),2 - trajDuration_*t_c);

  // Compute s, s_dot, s_ddot at current time t
  if (t >= 0 && t <= t_c){
    s = 0.5*s_ddot_c*std::pow(t,2);
    s_dot = s_ddot_c*t;
    s_ddot = s_ddot_c;
  } else if (t > t_c && t <= (trajDuration_ - t_c)){
    s = s_ddot_c*t_c*(t - t_c/2);
    s_dot = s_ddot_c*t_c;
    s_ddot = 0;
  } else{
    s = 1 - 0.5*s_ddot_c*(trajDuration_ - t)^2;
    s_dot = -s_ddot_c*(trajDuration_ - t);
    s_ddot = -s_ddot_c;
  }
}
```

- 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

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

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 Eq.(2).

As before, the function prototype was declared into the `KDLPlanner` class:

kdl_planner.h

```
void cubic_polynomial(double t, double &s, double &s_dot, double &s_ddot);
```

while for the function implementation, the polynomial coefficient were obtained by solving the following system of equations:

$$\begin{cases} a_0 = q_i \\ a_1 = \dot{q}_i \\ a_3 t_f^3 + a_2 t_f^2 + a_1 t_f + a_0 = q_f \\ 3a_3 t^2 + 2a_2 t + a_1 = \dot{q}_f \end{cases} \tag{3}$$

By imposing the initial and final conditions, $q_i = 0$, $\dot{q}_i = 0$, $q_f = 1, \dot{q}_f = 0$, the following parameters are obtained:

$$\begin{cases} a_0 = 0 \\ a_1 = 0 \\ a_2 = \frac{3}{t_f^2} \\ a_3 = -\frac{2}{t_f^3} \end{cases} \tag{4}$$

The function definition is shown below:

kdl_planner.cpp

```
/**
 * @brief [cubic polynomial function]
 *
 * @param [t] [current time at which the curvilinear abscissa is computed].
 * @param [s] [curvilinear abscissa]
 * @param [s_dot] [velocity at the current time]
 * @param [s_ddot] [acceleration at the current time]
 */
void KDLPlanner::cubic_polynomial(double t, double &s, double &s_dot, double
    &s_ddot)
{
  // Polynomial coefficient
  double a_0 = 0;
  double a_1 = 0;
  double a_2 = 3 / std::pow(trajDuration_, 2);
  double a_3 = -2 / std::pow(trajDuration_,3);

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

Create circular trajectories for your robot

- 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`).

The constructor prototype is defined within the `KDLPlanner` class, in which a default value of `_accDuration` is added.

kdl_planner.h

```
KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double
    _trajRadius, double _accDuration = 0.5);
```

The implementation of the above constructor function is

kdl_planner.cpp

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

- 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_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:

$$x = x_i, \quad y = y_i - rcos(2\pi s), \quad z = z_i - rsin(2\pi s) \tag{5}$$

A modular approach has been chosen, both to improve overall readability and allows more flexibility for future implementations. The circular path function:

kdl_planner.cpp

```
/**
 * @brief [circular trajectory function]
 * @param [s] [curvilinear abscissa]
 * @param [s_dot] [velocity at the current time]
 * @param [s_ddot] [acceleration at the current time]
 */
trajectory_point KDLPlanner::compute_circular_trajectory(double s, double
    s_dot, double s_ddot)
{
  trajectory_point traj;

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

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

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

  return traj;
}
```

while the trajectory functions are implemented as follows, specifying both the time law and the path:

kdl_planner.cpp

```
trajectory_point KDLPlanner::circular_traj_cubic(const double t)
{
  double s, s_dot, s_ddot;
  cubic_polynomial(t, s, s_dot, s_ddot);
  return compute_circular_trajectory(s, s_dot, s_ddot);
}

trajectory_point KDLPlanner::circular_traj_trapezoidal(const double t)
{
  double s, s_dot, s_ddot;
  trapezoidal_vel(t, accDuration_, s, s_dot, s_ddot);
  return compute_circular_trajectory(s, s_dot, s_ddot);
}
```

The circular trajectory is described in terms of position, velocity, and acceleration.

- The position along the x-axis of the circular trajectory is constant and equal to the initial position.
- positions along the y-axis and z-axis are calculated with the given formulas.
- The velocity and acceleration along the y and z axes are calculated using the expressions of the derivatives of circular motion.

- Do the same for the linear trajectory.

  Similar to the circular trajectory case, the linear trajectory function was implemented following the formulas:

$$p = p_i + s(p_f - p_i), \quad \dot{p}_i = \dot{s}(p_f - p_i), \quad \ddot{p}_i = \ddot{s}(p_f - p_i)$$

kdl_planner.cpp

```
/**
 * @brief [linear trajectory function]
 *
 * @param [s] [curvilinear abscissa]
 * @param [s_dot] [velocity at the current time]
 * @param [s_ddot] [acceleration at the current time]
 */
trajectory_point KDLPlanner::compute_linear_trajectory(double s, double s_dot
    , double s_ddot)
{
  trajectory_point traj;

  Eigen::Vector3d delta_p = trajEnd_ - trajInit_;

  traj.pos = trajInit_ + s*delta_p;
  traj.vel = s_dot*delta_p;
  traj.acc = s_ddot*delta_p;

  return traj;
}
```

kdl_planner.cpp

```
trajectory_point KDLPlanner::linear_traj_cubic(const double t)
{
  double s, s_dot, s_ddot;
  cubic_polynomial(t, s, s_dot, s_ddot);
  return compute_linear_trajectory(s, s_dot, s_ddot);
}

trajectory_point KDLPlanner::linear_traj_trapezoidal(const double t)
{
  double s, s_dot, s_ddot;
  trapezoidal_vel(t, accDuration_, s, s_dot, s_ddot);
  return compute_linear_trajectory(s, s_dot, s_ddot);
}
```

Test the four trajectories

- At this point, you can create both linear and circular trajectories, each with trapezoidal velocity or cubic polynomial curvilinear abscissa. Modify your main file `ros2_kdl_node.cpp` and test the four trajectories with the provided joint space inverse dynamics controller.

  To correctly use the **joint space inverse dynamics controller**, it is necessary to obtain the desired joint acceleration, velocity and position from the trajectory, that can be used as a reference for the control algorithm. To do that the following code was implemented in the `cmd_publisher` callback in `ros2_kdl_node.cpp`.

  Initially, The current end-effector position and velocity are obtained from the robot model. The desired position, velocity, and acceleration are defined based on the input trajectory:

  ros2_kdl_node.cpp

  ```
  // Compute EE frame acctual position and velocity
  KDL::Frame cartpos = robot_->getEEFrame();
  KDL::Twist cartvel = robot_->getEEVelocity();

  // Compute desired Frame position, velocity and acceleration at current p
  KDL::Frame desPos; desPos.M = cartpos.M; desPos.p = toKDL(p.pos);   // x_des
  KDL::Twist desVel; desVel.rot = cartvel.rot; desVel.vel = toKDL(p.vel);   //
      x_dot_des
  KDL::Twist desAcc; desAcc = KDL::Twist(KDL::Vector(p.acc[0], p.acc[1], p.acc
      [2]),KDL::Vector::Zero());   // X_ddot_des
  ```

  Linear and rotational errors are computed between the desired and actual states:

  $$e = x_{\text{des}} - x, \qquad \dot{e} = \dot{x}_{\text{des}} - \dot{x} \tag{6}$$

  in which, for the rotational part rotation matrix are used to compute the errors. R_{des} and R_e represent the desired and actual rotation matrices.

ros2_kdl_node.cpp

```
// compute errors
Eigen::Vector3d error = computeLinearError(p.pos, Eigen::Vector3d(cartpos.p.
    data));
Eigen::Vector3d o_error = computeOrientationError(toEigen(init_cart_pose_.M),
     toEigen(cartpos.M));

// Ensure proper orientation matrices for desired and current rotations
Eigen::Matrix<double,3,3,Eigen::RowMajor> R_des(init_cart_pose_.M.data);
Eigen::Matrix<double,3,3,Eigen::RowMajor> R_e(robot_->getEEFrame().M.data);
R_des = matrixOrthonormalization(R_des);
R_e = matrixOrthonormalization(R_e);

// Compute angular velocity errors
Eigen::Matrix<double,3,1> omega_des(init_cart_vel_.rot.data);
Eigen::Matrix<double,3,1> omega_e(robot_->getEEVelocity().rot.data);

// Compute velocity errors (linear and rotational)
Eigen::Vector3d error_dot = computeLinearError(p.vel, Eigen::Vector3d(cartvel
    .vel.data));
Eigen::Vector3d o_error_dot = computeOrientationVelocityError(omega_des,
    omega_e, R_des, R_e);
```

The desired joint accelerations are computed using a second order **CLIK** algorithm, as follows:

$$\ddot{q} = J^{\dagger}\left(\ddot{x}_{\text{des}} + K_d\dot{e} + K_p e - \dot{J}\dot{q}\right), \tag{7}$$

where $J^{\dagger}$ is the pseudo-inverse of the Jacobian, $\dot{J}$ accounts for Jacobian rate changes, and $\dot{q}$ is the joint velocity. K_d and K_p are proportional and derivative gains. The joint velocities and positions are updated using numerical integration, ensuring continuous trajectory tracking.

ros2_kdl_node.cpp

```
// Compute joint accelerations using the Jacobian pseudo-inverse
Vector6d cartacc; cartacc << p.acc + 5*error_dot + 10*error, 5*o_error_dot +
    10*o_error;
joint_accelerations_.data = pseudoinverse(robot_->getEEJacobian().data) * (
    cartacc
                    - robot_->getEEJacDotqDot()*joint_velocities_.data);

// Integrate acceleration to obtain velocity and position
joint_velocities_.data = joint_velocities_.data + joint_accelerations_.data*
    dt;
joint_positions_.data = joint_positions_.data + joint_velocities_.data*dt;
```

Finally, inverse dynamics is used to calculate the joint torques required to achieve the computed accelerations:

$$\tau = B(q)y + n(q,\dot{q}) \tag{8}$$

where τ represents the joint torques.

ros2_kdl_node.cpp

```
// Inverse dynamics to compute control torques
joint_torques_.data = controller_.idCntr(joint_positions_, joint_velocities_,
     joint_accelerations_, Kp, Kd);
```

For the purpose of testing the rectilinear trajectories, the planner was instanciated as follow:

ros2_kdl_node.cpp

```
if(traj_type_ == "linear"){
    planner_ = KDLPlanner(traj_duration, acc_duration, init_position,
    end_position);
    if(s_type_ == "trapezoidal")
    {
        p = planner_.linear_traj_trapezoidal(t_);
    }else if(s_type_ == "cubic")
    {
        p = planner_.linear_traj_cubic(t_);
    }
}
else if(traj_type_ == "circular")
{
    planner_ = KDLPlanner(traj_duration, init_position, traj_radius,
    acc_duration);
    if(s_type_ == "trapezoidal")
    {
        p = planner_.circular_traj_trapezoidal(t_);
    }else if(s_type_ == "cubic")
    {
        p = planner_.circular_traj_cubic(t_);
    }
}
```

To send trajectory point to the logic and using the effort controller, the following code was implemented:

ros2_kdl_node.cpp

```
// Retrieve the trajectory point
trajectory_point p;
if(traj_type_ == "linear"){
    if(s_type_ == "trapezoidal")
    {
        p = planner_.linear_traj_trapezoidal(t_);
    }else if(s_type_ == "cubic")
    {
        p = planner_.linear_traj_cubic(t_);
    }
}
else if(traj_type_ == "circular")
{
    if(s_type_ == "trapezoidal")
    {
        p = planner_.circular_traj_trapezoidal(t_);
    }else if(s_type_ == "cubic")
    {
        p = planner_.circular_traj_cubic(t_);
    }
}
```

Finally when the trajectory ends, the desired torques commands are the ones calculated taking as reference the last point of the trajectory, in order to maintain the manipulator at the final position.

- 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.

  In the Linux terminal is possible to open a `plotjuggler` window with the command

```
$ ros2 run plotjuggler plotjuggler
```

that run a gui interface that easilly allows to plot and retrieve data from ros topics.

Tuning with a trial-and-error approach, a good trade-off for proportional and derivative gain, result to be $K_p = 25$ and $K_d = \sqrt{K_p}$, that ensure smooth torques and limited peak, while mantaining a good tracking of the reference.

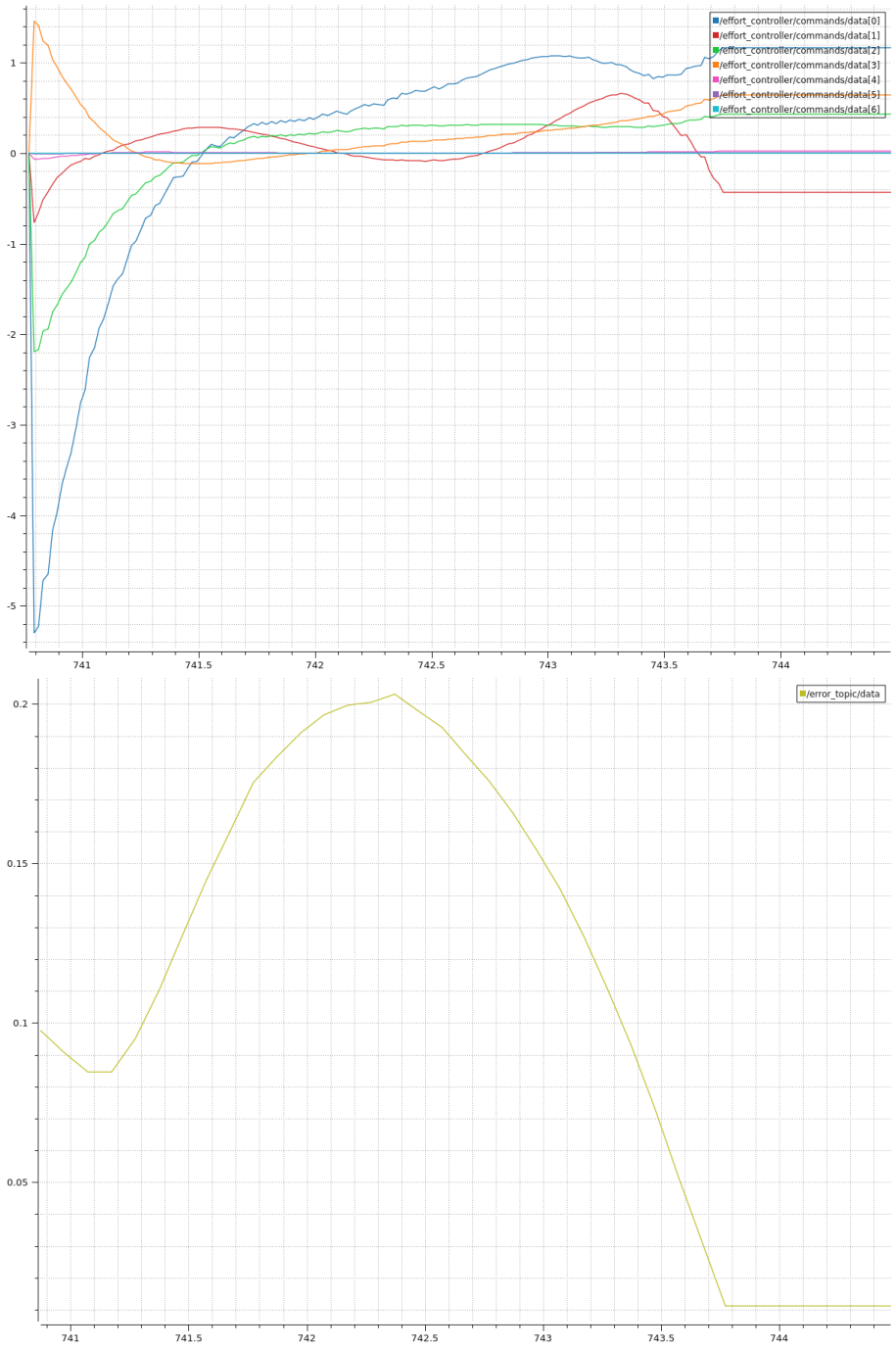


Table 1: Rectilinear trajectory with cubic profile

- **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.

```
$ ros2 bag record /effort_controller/commands -o joint_torque
```

Develop an inverse dynamics operational space controller

- 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.

  Initially, two proportional gain matrices, K_p and K_d, are computed to regulate position and orientation. These are represented as block-diagonal matrices:

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 proportional and derivative gains
    Eigen::Matrix<double,6,6> Kp = Eigen::Matrix<double,6,6>::Zero();
    Eigen::Matrix<double,6,6> Kd = Eigen::Matrix<double,6,6>::Zero();

    // Set gains for position and orientation
    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();
```

  where K_{pp}, K_{po}, K_{dp}, and K_{do} are the scalar gains for position and orientation, and `Eigen::Matrix3d::Identity()` is the 3×3 identity matrix.

  Subsequentely, robot dynamic matrices and jacobians are updated, while other variables are initialized:

kdl_control.cpp

```
// Update the robot's state using its Jacobian and dynamics
    Eigen::Matrix<double,6,7> J = robot_->getEEJacobian().data;
    Eigen::Matrix<double,7,6> Jpinv = pseudoinverse(J);
    Eigen::Matrix<double,7,7> B = robot_->getJsim();

    // Compute desired and actual end-effector positions and orientations
    Eigen::Vector3d x_des(_desPos.p.data);
    Eigen::Vector3d x_e(robot_->getEEFrame().p.data);
    Eigen::Matrix<double,3,3,Eigen::RowMajor> R_des(_desPos.M.data);
    Eigen::Matrix<double,3,3,Eigen::RowMajor> R_e(robot_->getEEFrame().M.data
    );
    R_des = matrixOrthonormalization(R_des);
    R_e = matrixOrthonormalization(R_e);

    // Compute desired and actual velocities
    Eigen::Vector3d x_dot_des(_desVel.vel.data);
    Eigen::Vector3d x_dot_e(robot_->getEEVelocity().vel.data);
    Eigen::Matrix<double,3,1> omega_des(_desVel.rot.data);
    Eigen::Matrix<double,3,1> omega_e(robot_->getEEVelocity().rot.data);

    // Compute desired accelerations (linear and rotational)
    Eigen::Matrix<double,6,1> x_ddot_des = Eigen::Matrix<double,6,1>::Zero();
    Eigen::Matrix<double,3,1> alpha_des(_desAcc.vel.data);
    Eigen::Matrix<double,3,1> alpha_r_des(_desAcc.rot.data);
```

  Later in the controller functions the following errors are computed:

  - **Position Error:** $e_p = x_{\text{des}} - x_e$, where x_{des} and x_e are the desired and actual positions.

- **Velocity Error:** $e_{\dot{p}} = \dot{x}_{\text{des}} - \dot{x}_e$.
- **Orientation Error:** e_o is derived from the difference between the desired and actual orientations using rotation matrix representations.
- **Angular Velocity Error:** $e_{\dot{o}}$ accounts for discrepancies in angular velocity.

kdl_control.cpp

```
// Compute linear errors
Eigen::Matrix<double,3,1> e_p = computeLinearError(x_des, x_e);
Eigen::Matrix<double,3,1> e_dot_p = computeLinearError(x_dot_des, x_dot_e
);
// Compute Rotational errors
Eigen::Matrix<double,3,1> e_o = computeOrientationError(R_des, R_e);
Eigen::Matrix<double,3,1> e_dot_o = computeOrientationVelocityError(
omega_des, omega_e, R_des, R_e);
```

- The logic behind the implementation of your controller is sketched within the function: you must calculate the gain matrices, read the current Cartesian state of your manipulator in terms of end-effector parametrized pose x, velocity $\dot{x}$, and acceleration $\ddot{x}$, retrieve the current joint space inertia matrix B 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

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

The control law combines the proportional and derivative errors with the desired accelerations:

$$y = J^{\dagger}(\ddot{x}_{\text{des}} + K_d\dot{\tilde{x}} + K_p\tilde{x} - J\dot{q}) \tag{9}$$

where $J^{\dagger}$ is the pseudo-inverse of the Jacobian, $\tilde{x}$ and $\dot{\tilde{x}}$ are the position and velocity errors, and $J\dot{q}$ accounts for the Jacobian velocity effects.

At the end, desired joint torques are computed using:

$$\tau = By + n \tag{10}$$

where B is the joint-space inertia matrix and C represents Coriolis effects. Optionally, gravity compensation can be added, if the simulation environments accounts for it.

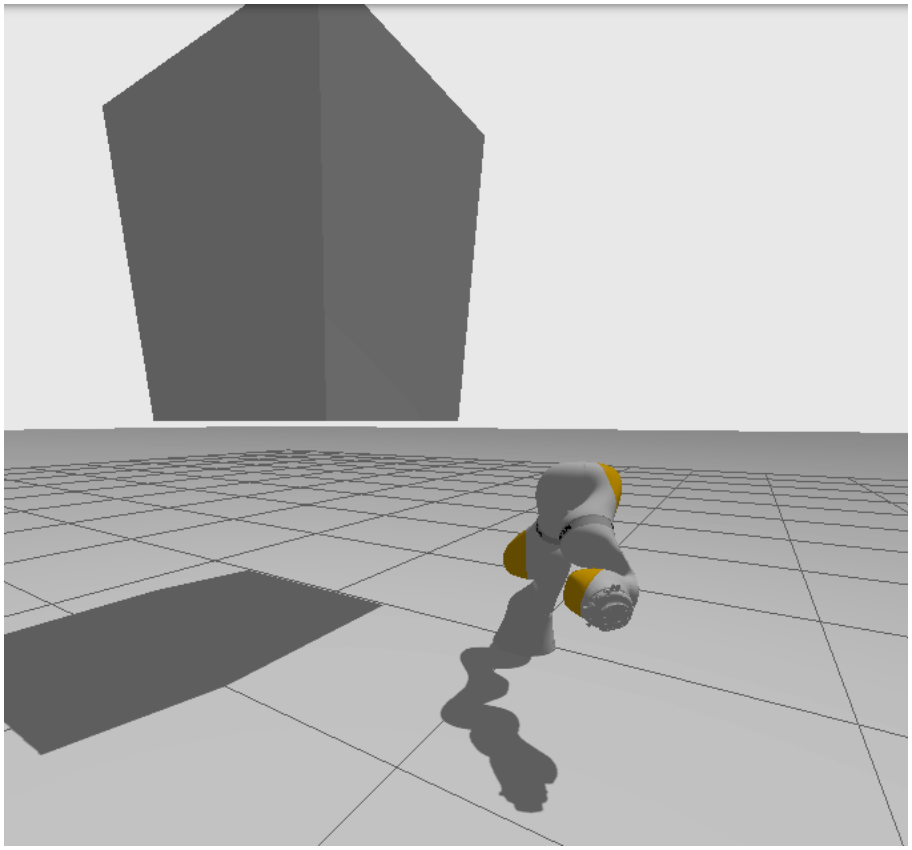
kdl_control.cpp

```
// Combine position and orientation errors into a single vector
Eigen::Matrix<double,6,1> x_tilde = Eigen::Matrix<double,6,1>::Zero();
Eigen::Matrix<double,6,1> x_dot_tilde = Eigen::Matrix<double,6,1>::Zero()
;
x_tilde << e_p, e_o;
x_dot_tilde << e_dot_p, e_dot_o;
x_ddot_des << alpha_des, alpha_r_des;

// Inverse dynamics
Eigen::Matrix<double,6,1> y = Eigen::Matrix<double,6,1>::Zero();
Eigen::Matrix<double,6,1> J_dot_q_dot = robot_->getEEJacDotqDot()*robot_
->getJntVelocities();
y << (x_ddot_des + Kd*x_dot_tilde + Kp*x_tilde - J_dot_q_dot);

// Return the computed joint torques (Optionally include gravity
compensation)
return B*(Jpinv*y) + robot_->getCoriolis(); //+ robot_->getGravity();
```

- Test the controller along the planned trajectories and plot the corresponding joint torque commands.

The operational space controller was tested across all the planned trajectories to evaluate its performance and the resulting joint torque commands. The results are presented showcasing both the torque outputs and the trajectory tracking errors.

- **Smoothness vs. Accuracy:**
  - The cubic profile consistently yielded smoother torque commands and lower tracking errors, suitable for applications prioritizing precision and efficiency.
  - The trapezoidal profile showed higher control efforts and tracking errors, which could be beneficial in time-critical tasks but at the cost of smoothness.
- **Trajectory Complexity:**
  - Rectilinear paths were easier to follow than circular paths, as expected. The latter introduced rotational dynamics that challenged the controller, especially with acceleration changes.
- **Controller Robustness:**
  - Across all scenarios, the controller demonstrated strong adaptability, maintaining acceptable torque levels and small tracking errors, even in more demanding trajectories.



Finally remember to start gazebo in paused mode and without gravity, to correctly use the effort controller.

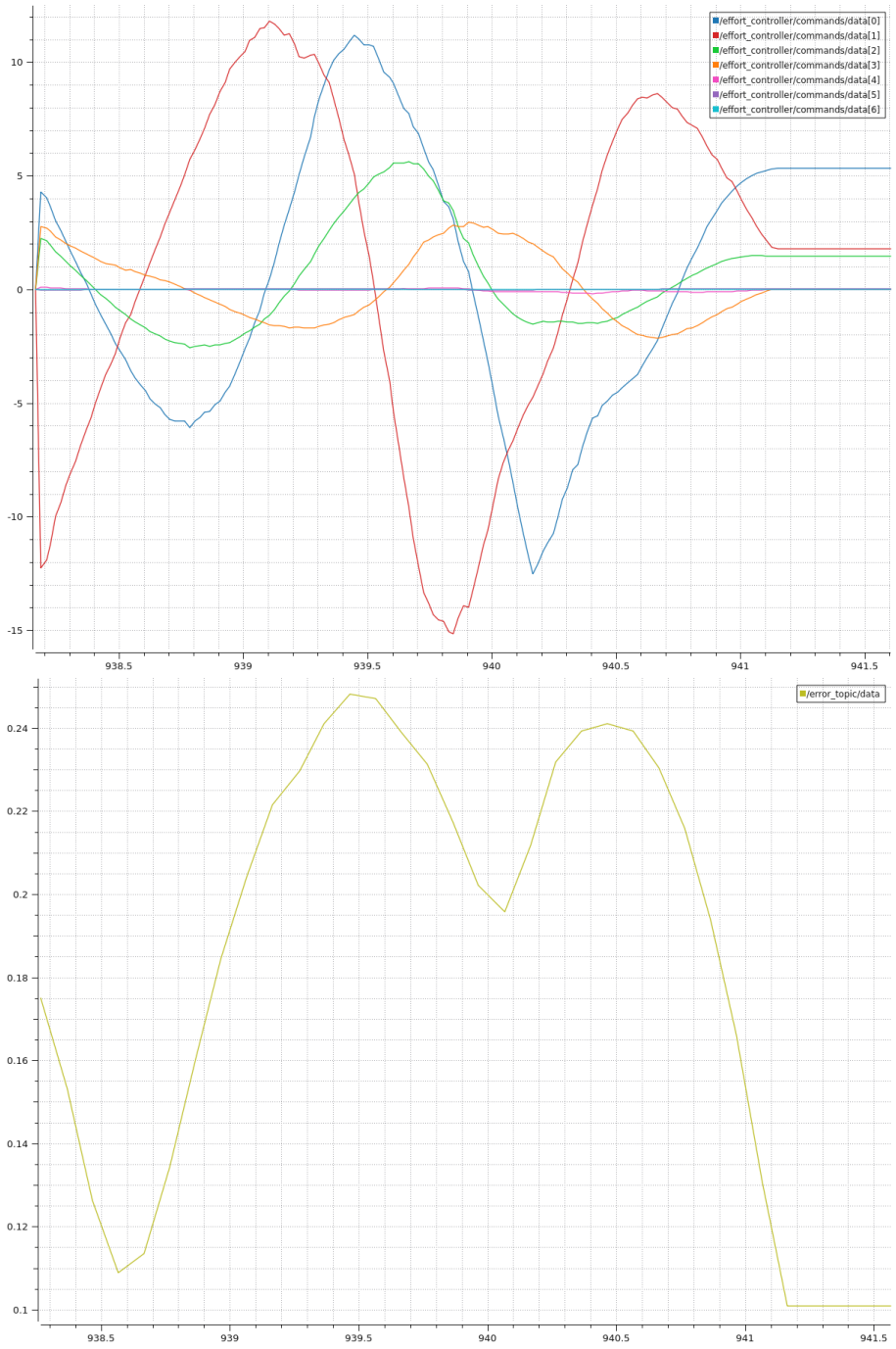


Table 2: Circular trajectory with cubic profile

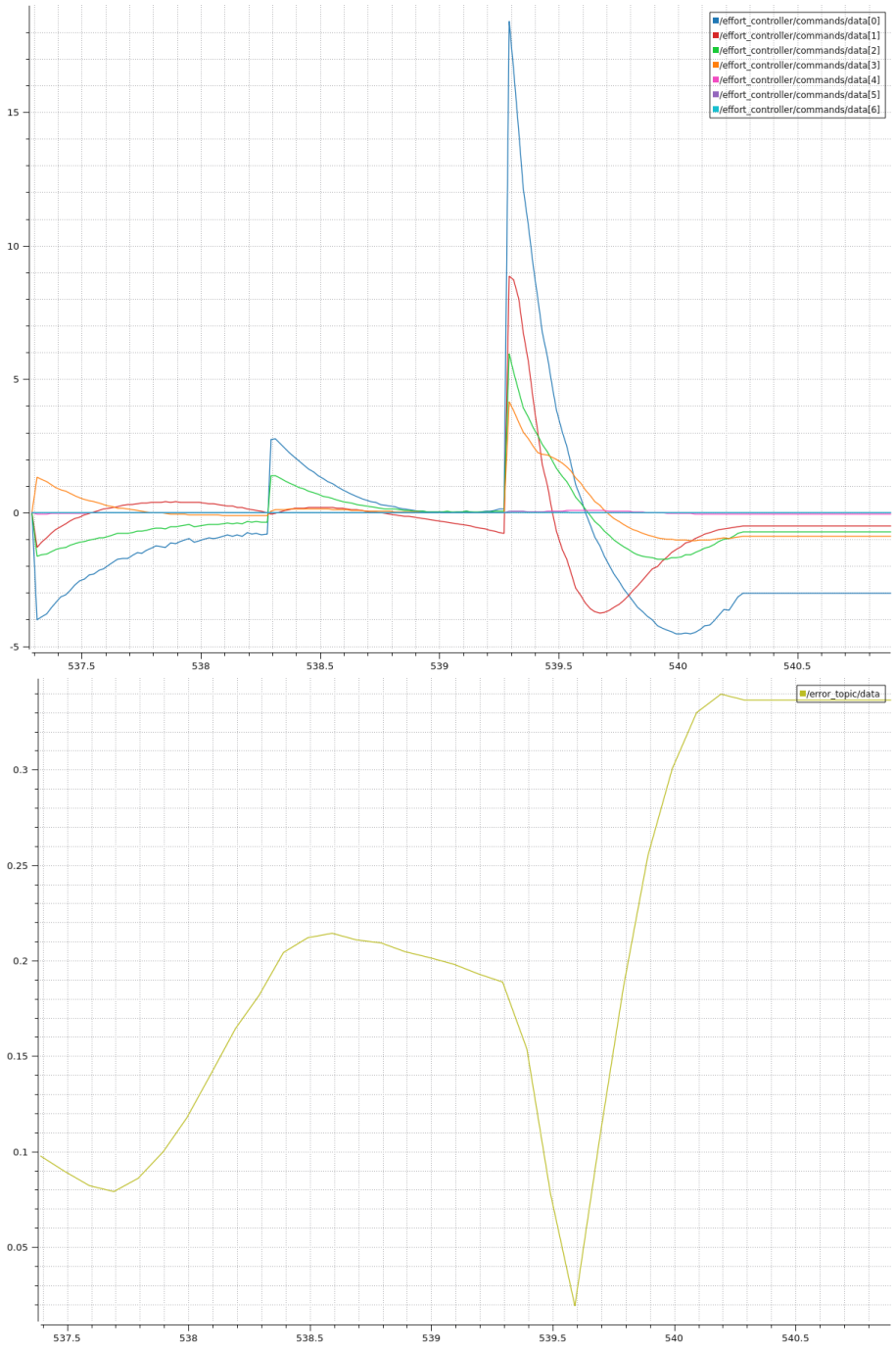


Table 3: Rectilinear trajectory with cubic profile

Table 4: Rectilinear trajectory with cubic profile

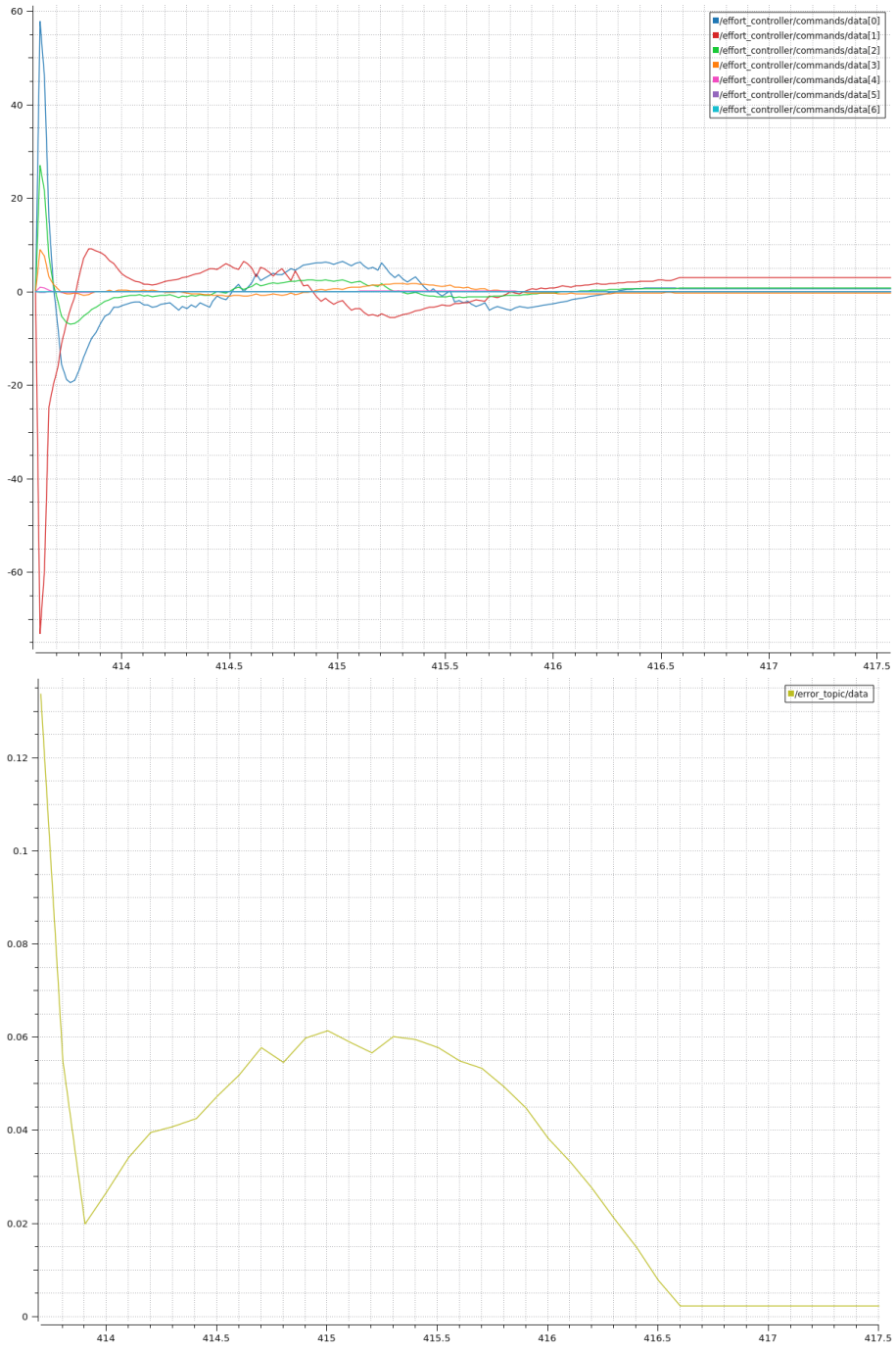


Table 5: Circular trajectory with cubic profile

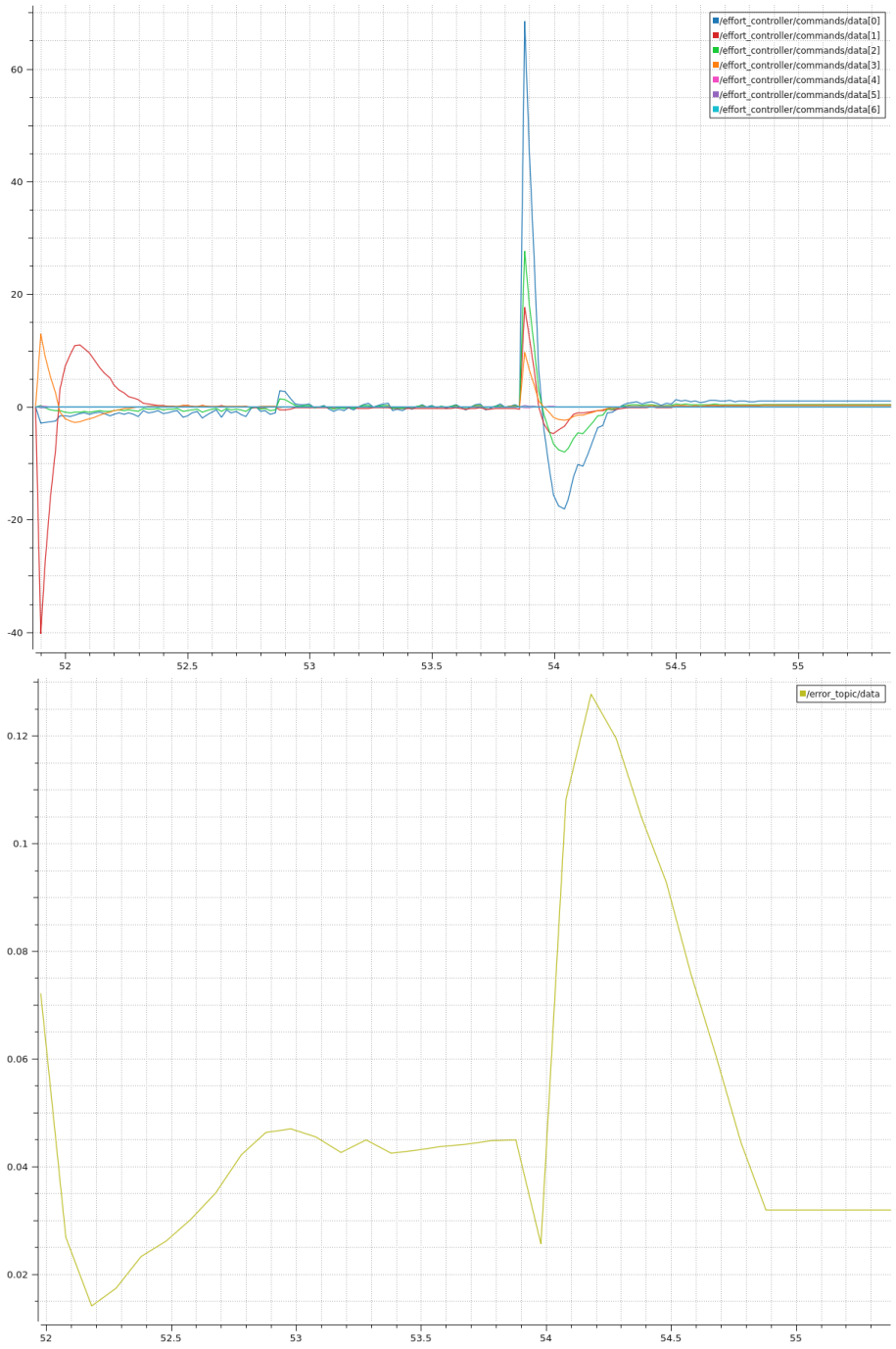


Table 6: Rectilinear trajectory with cubic profile

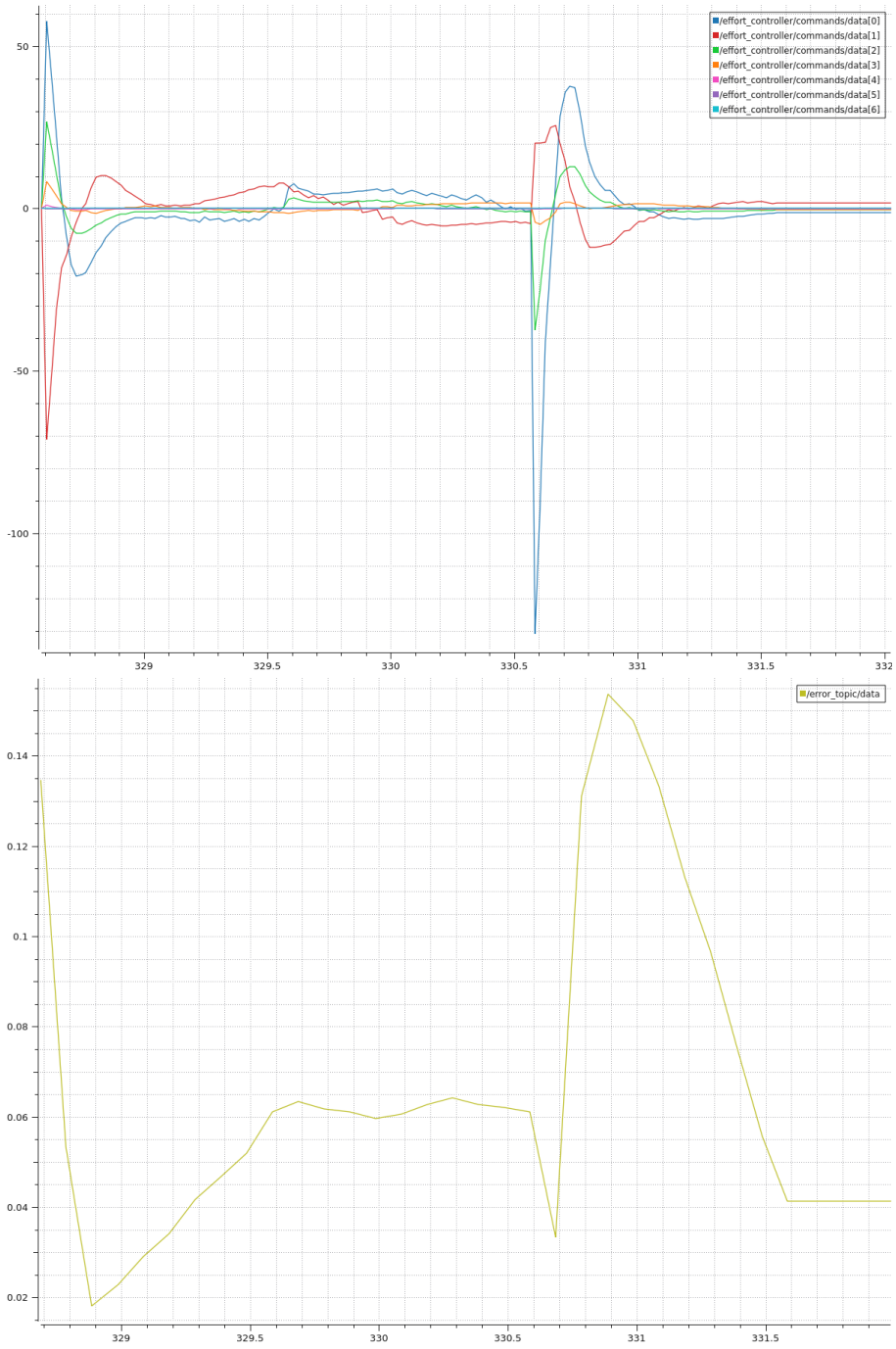


Table 7: Circular trajectory with trapezoidal profile