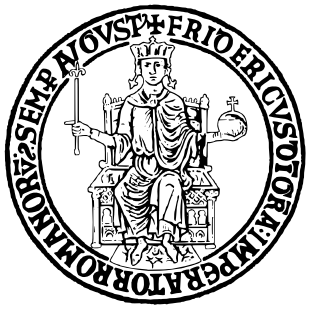
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Robotics Lab

Homework 2: Dynamic Control of a 7-DOF Robotic Manipulator

Components
William Notaro
Chiara Panagrosso
Salvatore Piccolo
Roberto Rocco

Anno Accademico 2024–2025

Contents

The goal of this homework is to develop a ROS package to dynamically control a 7-degrees-of-freedom robotic manipulator arm into the Gazebo environment. At the beginning of each chapter, a brief explanation of its purpose will be provided, outlining the steps that will be followed to complete the task.

The code used for this project can be viewed and downloaded from the following GitHub repositories:

- William Notaro: `https://github.com/well-iam/RL24-Homework2`
- Chiara Panagrosso: `https://github.com/chiarapanagrosso/RL24-Homework2.git`
- Salvatore Piccolo: `https://github.com/SalvatorePiccolo/RL24_Homework2`
- Roberto Rocco:`https://github.com/Robertorocco/RL-24-Homework2.git`

It is possible to view the videos of the simulation results of the effort controller in the joint space on YouTube at the following links:

- Linear trajectory with cubic polynomial profile: `https://www.youtube.com/watch?v=y3k_JMHPOs0`
- Linear trajectory with trapezoidal velocity profile: `https://www.youtube.com/watch?v=i31O8PMZzUs`
- Circular trajectory with cubic polynomial profile: `https://www.youtube.com/watch?v=BLOtAiTAEUg`
- Circular trajectory with trapezoidal velocity profile:`https://www.youtube.com/watch?v=C4SFhiw1xZo`

Chapter 1

Trajectory Planning

The aim of this chapter is to modify the implementation of the KDLPlanner class to include two new methods, providing the user with the option to select between two types of curvilinear abscissa for trajectory computation: one based on a trapezoidal velocity profile and the other using a cubic polynomial profile. The following sections detail the implementation of these methods.

1.1 Trapezoidal Velocity Profile

To begin, the `KDLPlanner` class has been modified to provide an interface for creating a trajectory with a trapezoidal velocity profile. For this purpose, both `kdl_planner.h` and `kdl_planner.cpp` were updated. A new method, `KDLPlanner::trapezoidal_vel`, was defined: this method takes as input the current time `t` and an acceleration time `tc` (assumed to be one-third of the total trajectory time) and outputs the variables `s`, `sdot`, and `sddot`. These variables represent the position, velocity, and acceleration profiles that the end-effector must follow at time `t` during the trajectory.

```
//(1a)
void KDLPlanner::trapezoidal_vel(double time, double _accDuration, double&
    s, double& sdot, double& sddot) {
    accDuration_ = _accDuration;
    double ddot_s_c = -1.0 / (std::pow(accDuration_, 2) - trajDuration_ *
   accDuration_);
    if (time <= accDuration_) {
        s = 0.5 * ddot_s_c * std::pow(time, 2);
        sdot = ddot_s_c * time;
        sddot = ddot_s_c;
    }
    else if (time <= trajDuration_ - accDuration_) {
        s = ddot_s_c * accDuration_ * (time - accDuration_ / 2);
        sdot = ddot_s_c * accDuration_;
        sddot = 0;
    }
    else {
        s = 1 - 0.5 * ddot_s_c * std::pow(trajDuration_ - time, 2);
        sdot = ddot_s_c * (trajDuration_ - time);
        sddot = -ddot_s_c;
    }
}
```

Listing 1.1: piece of kdl_planner.cpp

The equations used to compute s is as follows:

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

After, $\dot{s}$ and $\ddot{s}$ were obtained by differentiating the expression for s.

1.2 Cubic Polynomial curvilinear abscissa

In this section, the problem of generating a cubic polynomial curvilinear abscissa for the trajectory is addressed. Specifically, a new function, `KDLPlanner::cubic_polinomial`, was developed. This function takes the current time `t` as an input argument and returns the variables `s`, `sdot`, and `sddot`, which represent the position, velocity, and acceleration profiles, respectively, that the end-effector must follow at time `t` along the trajectory.

The curvilinear abscissa is modeled as a generic third-order polynomial of the form:

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

This coefficients can be calculated offline: the formula of each one can be computed imposing the boundary conditions on position and speed: on one hand `s` and `sdot` must be zero for `t=0` , on the other hand `s` must be one (normalized length of path) and `sdot` zero for `t=1` This conditions give out a closed formula for all the four coefficient, which will be initialized in the constructor through the call of the `calculateCoefficients` method.

```
//(1b)
void KDLPlanner::cubic_polinomial(double time, double& s, double& sdot,
    double& sddot) {
     s = a3 * std::pow(time, 3) + a2 * std::pow(time, 2) + a1 * time + a0;
     sdot = 3 * a3 * std::pow(time, 2) + 2 * a2 * time + a1;
     sddot = 6 * a3 * time + 2 * a2;
}

void KDLPlanner::calculateCoefficients(double _trajDuration){
     a0=0;
     a1=0;
     a2=3.0/std::pow(_trajDuration,2);
     a3= -2.0/std::pow(_trajDuration,3);
}
```

Listing 1.2: piece of kdl_planner.cpp

As can be seen the coefficients depend only on the time chosen for the trajectory. The value of sdot and sddot are computed as the derivative of s.

Chapter 2

Circular Trajectory

The objective of the second chapter is to implement the constructors and functions required to enable the creation of both circular and linear trajectories for the end-effector.

2.1 Circular trajectory constructor

To generate a circular trajectory, an additional `double` type field named `trajRadius_` has been added to the `KDLPlanner` class. Its purpose is to store information about the radius length of the circle along which the user wants the end-effector to move. Consequently, a constructor has been implemented to initialize this new field, in addition to the already existing fields `trajDuration_`, `accDuration_` and `trajInit_`. This constructor initializes the `KDLPlanner` object in such a way as to achieve a trapezoidal velocity profile along the curvilinear abscissa with a circular trajectory.

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

Listing 2.1: constructor in kdl_planner.cpp

If you want to use instead a cubic polynomial curvilinear abscissa a different constructor must be used. This constructor initializes exclusively the fields `trajDuration_`, `trajInit_`, and `trajRadius_`, and calls the `calculateCoefficients` function to inizialize the object field relative to the cubic polinomial coefficients.

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

Listing 2.2: Constructor in kdl_planner.cpp

2.2 End Effector trajectory equation for circular path

The circular trajectory consists of a circular arc, with the radius size configurable by the user, lying in the vertical plane containing the end-effector. From the provided equations, it is evident that the circular arc is traced in the y-z plane. Additionally, the equations for the center of the circle can be derived by inverting the equations and evaluating them at $s = 0$.

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

Where the signs have been assigned accordingly to the desired orientation of the circular arc. In the code, two additional methods were implemented to compute the position, velocity, and acceleration of the end-effector at a given time t, ensuring the trajectory to lie on a circular arc with the previously described properties. The method can be invoked in two different ways (overloading), allowing the creation of either a trapezoidal or cubic profile for the curvilinear abscissa, as shown in Lst. 2.3.

The choice of the profile is determined by the parameters passed to the method: one interface takes only the time instance t as input and generates a cubic profile, while the other interface also requires the acceleration duration `_accDuration` and produces a trapezoidal profile. In both cases, the velocity and acceleration of the end-effector are calculated by differentiating the equation of the circle in the previously defined plane.

Finally, a constant α has been introduced for code clarity: By modifying the parameter, it is possible to define circular arcs of varying lengths. For our purposes, we will use an α equal to half of the value provided in the original equations, ensuring that the trajectory corresponds to a semicircular arc. By appropriately setting the radius, both the linear and circular trajectories will share the same goal point.

```
1  //(2b): Version for Cubic and circular
2  trajectory_point KDLPlanner::compute_trajectory_circular(double time) {
3      trajectory_point traj;
4      Eigen::Vector3d trajCenter(trajInit_ + Eigen::Vector3d(0, trajRadius_,
       0));
5      double s, sdot, sddot;
6      const double _alpha_ = 1 * 3.14;
7      cubic_polinomial(time, s, sdot, sddot);
8
9     // std::cout << "Circuling: " << std::endl;
10     traj.pos << trajCenter[0],
11         trajCenter[1] - trajRadius_ * cos(_alpha_ * s),
12         trajCenter[2] + trajRadius_ * sin(_alpha_ * s);
13
14     traj.vel << 0,
15         _alpha_* trajRadius_* sin(_alpha_ * s)* sdot,
16         +_alpha_ * trajRadius_ * cos(_alpha_ * s) * sdot;
17
18     traj.acc << 0,
19         std::pow(_alpha_, 2)* trajRadius_* cos(_alpha_ * s)* std::pow(sdot
       , 2) + _alpha_*trajRadius_*sin(_alpha_*s)*sddot,
20         -std::pow(_alpha_, 2)* trajRadius_* sin(_alpha_ * s)* std::pow(
       sdot, 2) + _alpha_*trajRadius_*cos(_alpha_*s)*sddot;
21
22     return traj;
23 }
24 //(2b): Version for Traps and circular
25 trajectory_point KDLPlanner::compute_trajectory_circular(double time,
       double _accDuration) {
```

```
26      trajectory_point traj;
27      Eigen::Vector3d trajCenter(trajInit_ + Eigen::Vector3d(0, trajRadius_,
        0));
28      double s, sdot, sddot;
29      const double _alpha_ = 1 * 3.14;
30      trapezoidal_vel(time, _accDuration, s, sdot, sddot);
31
32      traj.pos << trajCenter[0],
33          trajCenter[1] - trajRadius_ * cos(_alpha_ * s),
34          trajCenter[2] + trajRadius_ * sin(_alpha_ * s);
35
36      traj.vel << 0,
37          _alpha_* trajRadius_* sin(_alpha_ * s)* sdot,
38          _alpha_ * trajRadius_ * cos(_alpha_ * s) * sdot;
39
40      traj.acc << 0,
41          std::pow(_alpha_, 2)*trajRadius_*cos(_alpha_*s)*std::pow(sdot, 2)
       + _alpha_*trajRadius_*sin(_alpha_*s)*sddot,
42          -std::pow(_alpha_, 2)*trajRadius_*sin(_alpha_*s)*std::pow(sdot, 2)
        + _alpha_*trajRadius_*cos(_alpha_*s)*sddot;
43      return traj;
44  }
```

Listing 2.3: Methods in kdl_planner.cpp

2.3 End Effector trajectory equation for Linear path

As done the previous chapter, additional methods have been implemented to compute the position, velocity, and acceleration of the end-effector at a given time t, as shown in Lst. 2.4. Remember that the abscissa given as output by the called `cubic_polinomial` function is in the range [0,1]. For the linear path with trapezoidal velocity profile it's used the one already implemented by the package.

```
1   //(2c)
2   trajectory_point KDLPlanner::compute_trajectory_linear(double time) {
3       trajectory_point traj;
4       Eigen::Vector3d distance = trajEnd_ - trajInit_;
5
6       double s, sdot, sddot;
7       cubic_polinomial(time, s, sdot, sddot);
8
9       traj.pos = trajInit_ + s * distance;
10      //std::cout << "Traj pos: " << traj.s[0] << ' ' << traj.s[1] << ' ' <<
        traj.s[2] << std::endl;
11      traj.vel = sdot * distance;
12      traj.acc = sddot * distance;
13
14      return traj;
15  }
```

Listing 2.4: Method in kdl_planner.cpp

2.4 Testing Trajectories with position controller

All the trajectories presented so far were tested using the position controller provided by the base package of the project. Each combination was evaluated: linear paths with both cubic and trapezoidal profiles, as well as circular paths with both cubic and trapezoidal profiles.

The simulations yielded positive outcomes; however, the official test results will be discussed in the next chapter, where the effort-based controller is utilized.

Chapter 3

Joint Space Inverse Dynamic Controller

This chapter is dedicated to the validation of the previously projected trajectories, to the development of a joint space inverse dynamics controller and to the plot of the torque commands. In particular, the validation of the trajectories has been performed firstly from a pure kinematical viewpoint, using the 'position_controller', and then also from a dynamical viewpoint, through the use of the 'effort_controller'. Furthermore, the plot of the torque commands will be performed once in the `rqt_plot` interface and once in a MATLAB figure, after havind read and converted the data.

3.1 Controller Implementation

To begin, we present the control algorithm implemented for inverse dynamic control. The function `idCntr` is the core of this algorithm: it takes as input the desired values for joint position, velocity, and acceleration and outputs the vector of commands to be applied to each joint in order to achieve the desired values.

Internally, the algorithm uses the joint position and velocity measurements obtained from the sensors through the `robot_` object, which is appropriately updated in the code using the `update()` method. This ensures a closed-loop inverse dynamics algorithm, where the convergence dynamics of the error are governed by the constants `_Kp` and `_Kd`.

Finally, the vector of desired commands will be retrieved into an appropriate variable in the main program and published by the `cmd_publisher` to the corresponding topic. The script is shown in Lst. ??.

```
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;
```

```
15
16      return robot_->getJsim() * (ddqd + _Kd*de + _Kp*e)
17              + robot_->getCoriolis() + robot_->getGravity();
18  }
```

3.2 Computation of joint variables references

In this section we show how the reference joint variables have been computed. The reference trajectory is expressed in operational space variables, so needs to be transformed in the corresponding joint values. From a programming point of view, this section has made use of the provided `KDLRobot` class and also of some solvers that belong to the KDL (Kinematics and Dynamics Library).

3.2.1 Desired joint position

The desired joint position is expressed as a `KDL::Frame` data type, which is composed of two fields: the position, which is equal to the desired position obtained by the planner; the orientation, which is kept equal to the initial orientation of the robot. The corresponding joint values are calculated exploiting an existing Inverse Kinematics solver provided by the KDL library, belonging to the `KDL::ChainIkSolverPos_NR_JL` class. The code snippet responsible for executing and monitoring the inverse kinematics is shown in Lst. 3.1.

```
1 int ret = robot_->ikSol_->CartToJnt(qd_prev_,eeFrame_d,qd_);
2 if(ret != 0) {std::cout << robot_->ikSol_->strError(ret) << std::endl;};
```

Listing 3.1: Desired joint positions computation

3.2.2 Desired joint velocity

Analogously to the position part, the desired velocity is expressed in operational space variables and is represented by the `KDL::Twist` data type, also composed of two parts: linear velocity, retrieved from the planner; angular velocity, which is kept to be zero. The corresponding joint speeds are calculated exploiting another Inverse Kinematics solver provided by the KDL library, that belongs to the `KDL::ChainIkSolverVel_pinv` class. The code snippet responsible for executing and monitoring the velocity inverse kinematics is shown in Lst. 3.2.

```
1 KDL::Twist eeVelTwist_d(toKDL(p.vel),KDL::Vector::Zero());
2 robot_->getInverseKinematicsVel(eeVelTwist_d, dqd_);
```

Listing 3.2: Desired joint speeds computation

3.2.3 Desired joint acceleration

The computation of the desired joint acceleration has been obtained differentiating the desired joint velocities. To this purpose a variable containing the previous desired joint velocity has been created. The same variables is updated at the end of the acceleration computation as show in the following Lst. 3.3.

```
1 ddqd_.data = (dqd_.data - dqd_prev_.data)/dt;
2 dqd_prev_ = dqd_;
```

Listing 3.3: Desired joint accelerations computation

3.3 Testing Trajectories with effort controller

In this section we show the capabilities of the robot to track the desired trajectory.

First, it has been necessary to tune the controller parameters, i.e. the Kp and Kd gains. The tuning process has been performed in the following way: first both gains have been put to zero, and we observed the robot behavior, it just fell to the ground. Then we started increasing the proportional gain while keeping the derivative gain to zero until a good-looking tracking behavior started to show. Then we increased the derivative gain until we noticed a good smoothness in the robot movements.

The final trajectory performances can be observed in the dedicated videos that can be found in the project repository and at the following links to the YouTube platform:

- 'linear_cubs' trajectory, i.e. a linear segment with a cubic time law:
  `https://www.youtube.com/watch?v=y3k_JMHPOs0`
- 'linear_traps' trajectory, i.e. a linear segment with a trapezoidal velocity profile:
  `https://www.youtube.com/watch?v=i31O8PMZzUs`
- 'circular_cubs' trajectory, i.e. a semi-circumference arc with a cubic time law:
  `https://www.youtube.com/watch?v=BLOtAiTAEUg`
- 'circular_traps' trajectory, i.e. a semi-circumference arc with a trapezoidal velocity profile:
  `https://www.youtube.com/watch?v=C4SFhiw1xZo`

3.4 Torques Plot

In this section we plot the torque commands sent to the robot's actuators.

The process of saving the 7 torque values has been accomplished through the use of **bag files**. A bag file is a format used in ROS to record and store messages exchanged on topics. The interesting part in them is that you can also replay data from a running ROS system, for example using `rqt_plot`, or plot it in MATLAB.

The first step to do is to identify the topic, which in our case corresponds to `/effort_controller/commands`. Then, once the simulation has started and the controller began sending the commands over the appropriate topic, the torque values can be recorded using the following command:

```
$ ros2 bag record -o "path_to_file" /effort_controller/commands
```

Having obtained the bag file, first we show an example on how to plot its contents on a `rqt_plot` and then the procedure to plot it in MATLAB.

3.4.1 RQT Plot

Launch the `rqt` interface through and find the `plot` plugin in "Plugins¿Visualization¿Plot".
In the topic box write the name of the desired topic to plot, in our case '/effort_controller/commands'. In particular, to plot each component of the topic, you must access the data field of the structure and then press the Add button. The procedure must be repeated 7 times, once for each joint, and the full text to write is '/effort_controller/commands/data[i]' where 'i' represents the i-th joint.

The example rqt plot for the linear_cubs trajectory is shown in Fig. 3.1.

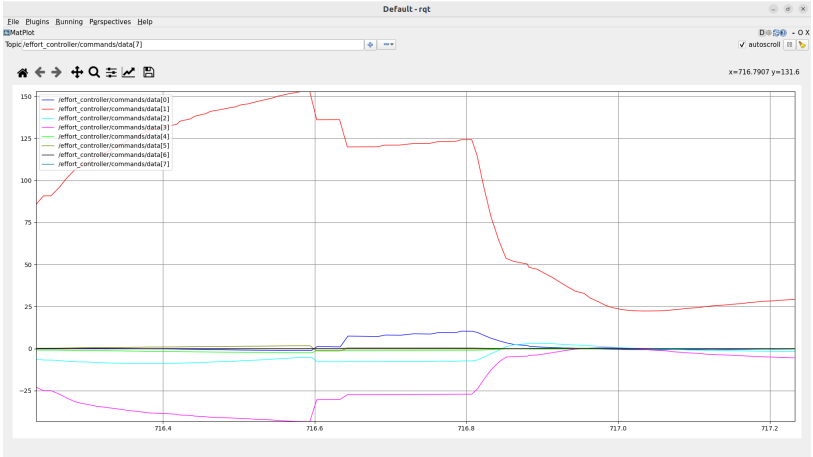


Figure 3.1: Torque commands sent by the effort command interface plot using `rqt_plot`

3.4.2 MATLAB

The plots obtained using MATLAB are shown in Figs. 3.2-3.5 and have been obtained through the following script Lst. 3.4.

```
folderName = "rosbagCircularCubs";
bag = ros2bag(folderName);
topics = bag.AvailableTopics;
disp(topics);
topic = '/effort_controller/commands';

% Select messages from the desired topic
msgs = readMessages(bag);

% Extract data (assuming std_msgs/Float64MultiArray)
torques = cellfun(@(msg) msg.data', msgs, 'UniformOutput', false);

% Convert to a matrix
torque_matrix = cell2mat(torques);

% Create a time vector (assumes constant frequency)
num_samples = size(torque_matrix, 1);
time = linspace(0, num_samples / 100, num_samples); % Adjust frequency
    (100 Hz example)

% Plot each joint torque
figure;
plot(time, torque_matrix, 'LineWidth', 1)
xlabel('Time (s)', 'FontSize', 18);
ylabel('Torque (Nm)', 'FontSize', 18);
grid on;
legend('Joint 1', 'Joint 2', 'Joint 3', 'Joint 4', 'Joint 5', 'Joint 6', '
    Joint 7', 'FontSize', 16);
title('Joint Torque Commands Over Time', 'FontSize', 20);

% Save image
saveas(gcf, strcat(folderName,'.png'));
```

Listing 3.4: MATLAB script for reading ros bag file and plot its contents.

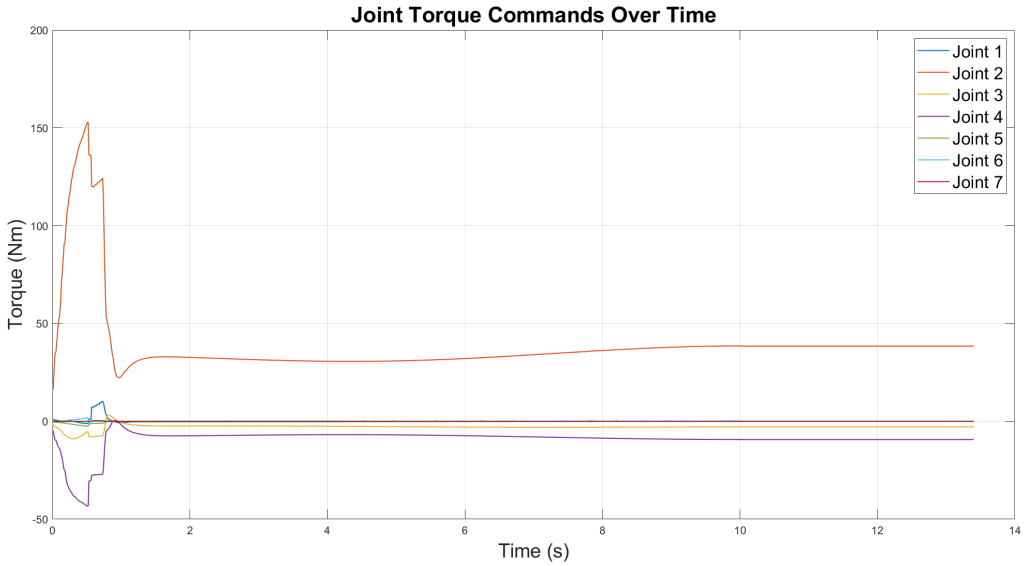


Figure 3.2: Torque commands sent by the effort command interface during the linear_cubs trajectory

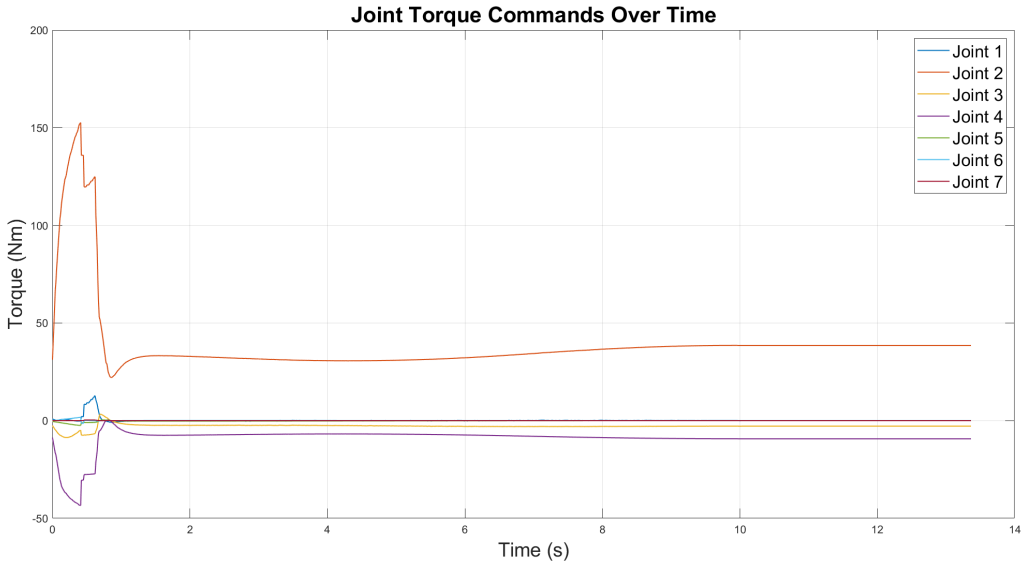


Figure 3.3: Torque commands sent by the effort command interface during the linear_traps trajectory

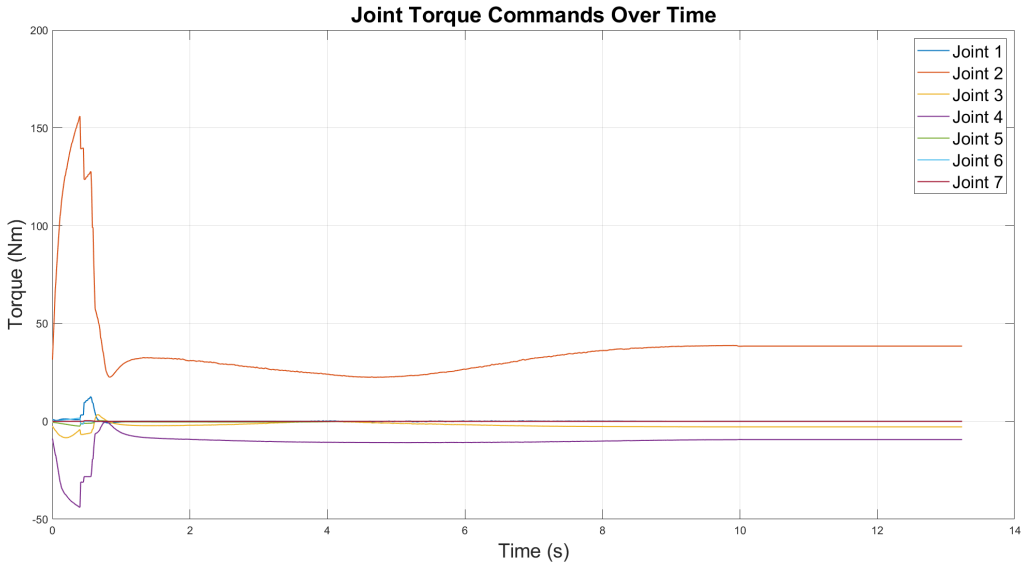


Figure 3.4: Torque commands sent by the effort command interface during the circular_cubs trajectory

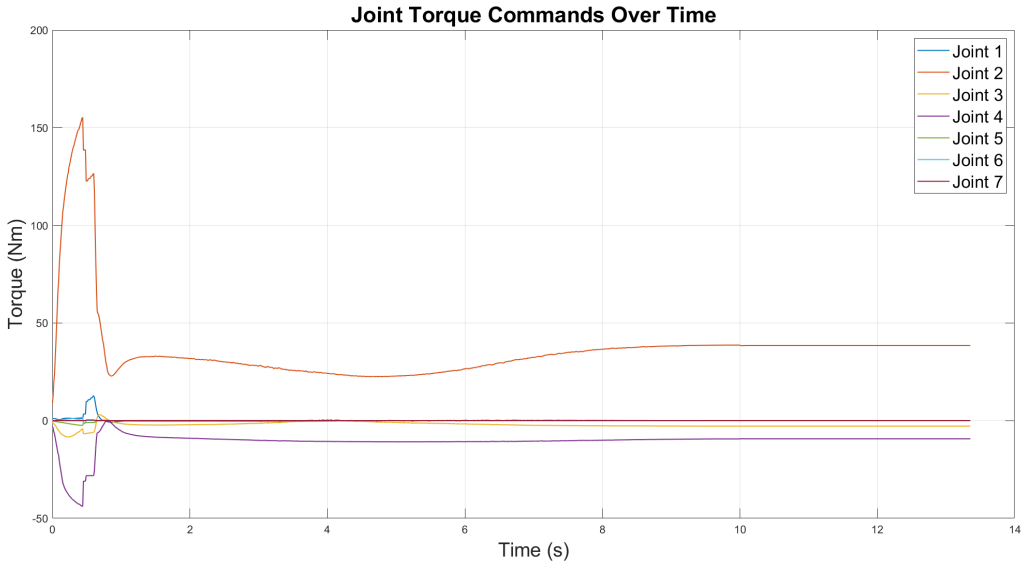


Figure 3.5: Torque commands sent by the effort command interface during the circular_traps trajectory

Chapter 4

Operational Space Inverse Dynamic Controller

In this chapter, we will illustrate how an additional controller inverse dynamic controller operating in operational space was implemented, and also demonstrate its performance on the previously planned trajectories.

4.1 Controller Implementation

Unlike the previous controller, the inverse dynamics controller in operational space computes errors directly in the operational space. Specifically, the reference is expressed directly in operational variables, namely the desired position and orientation of the end-effector, and the measured joint variables are transformed into operational space variable simply by computing the direct kinematics. Based on the manipulator dynamic model:

$$B(\mathbf{q})\ddot{\mathbf{q}} + n(\mathbf{q}, \dot{\mathbf{q}}) = \tau$$

the new controller is designed to provide the torque τ that enables tracking of the desired trajectory, selecting the following error dynamics:

$$\ddot{\tilde{x}} + K_D\dot{\tilde{x}} + K_P\tilde{x} = 0$$

Where the error is expressed directly into the operational space. To retrieve it into joint space, the operational space controller is expected to use the analytical jacobian, which is difficult to compute. However, if the orientation error is expressed via an angle and axis representation then the geometric jacobian can be used, therefore the latter is used in this project to avoid repercussions from possible representation singularities.

The control input is returned by the following function

$$u = n(\mathbf{q}, \dot{\mathbf{q}}) + B(\mathbf{q})\left(J(\mathbf{q})^{-1}\left(\ddot{\mathbf{x}}_d + K_p\tilde{\mathbf{x}} + K_d\dot{\tilde{\mathbf{x}}} - \dot{J}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}}\right)\right)$$

The implementation code of the controller is obtained by overloading the `idCntr` function, taking as parameters `KDL::Frame` type object for the position reference and two `KDL::Twist` type object for the desired acceleration and velocity. Lastly it takes four `double` type, two proportional `_Kpp, _Kpo` and two derivative `_Kdp, _Kdo` gain parameters applied to the position and the orientation error.

```
Eigen::Matrix<double,6,6> Kp, Kd;
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.block(2,2,1,1) = 5*Kp.block(2,2,1,1);
Kd.block(2,2,1,1) = 2*Kd.block(2,2,1,1);
```

Listing 4.1: Gains definitions

The implementation of the controller starts by reading the current state of the end-effector, so the inertia matrix, the geometric jacobian and its pseudoinverse must be computed:

```
Eigen::Matrix<double,6,7> J = robot_->getEEJacobian().data;

Eigen::Matrix<double,7,7> M = robot_->getJsim();

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

Listing 4.2: Computation of the inertia matrix

In order to compute the control action the desired positions and linear velocities are to be extracted as well as the desired and the current rotation matrices.

```
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);
```

Listing 4.3: Desired position and rotation matrix extraction

```
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);
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, dot_e_o;
dot_dot_x_d << dot_dot_p_d, dot_dot_r_d;
```

Listing 4.4: Computation of errors

The control action is computed as follows:

```
Eigen::Matrix<double,7,1> y;
Eigen::Matrix<double,7,1> n;

y << Jpinv * (dot_dot_x_d - Jdotqdot + Kd*dot_x_tilde + Kp*x_tilde);
n << robot_->getCoriolis() + robot_->getGravity();

return M * y + n;
```

Listing 4.5: Control action

Where n is the sum of the Coriolis and gravity terms.

4.1.1 Switching Effort Controller

As will be shown in the following section, the controller demonstrates good performance in trajectory tracking but fails to maintain the manipulator's stability at the end of the trajectory execution. In light of this, the code has been modified to switch to the previously implemented effort controller based on the value of the variable `t_` , defined within the `cmd_publisher` function, which represents the time variable for calculating the references and influences the control action. The switch occurs when the `total_time`, i.e., the duration of the executed trajectory, is exceeded. When this happens, the effort controller is called, which operates in joint space, receiving as references the `qd_` obtained by applying inverse kinematics to the desired reference in operational space. In the script, this is contained in an object of type `KDL::Frame` called `desFrame`, with `dqd_` and `ddqd_` set to zero, similar to the previous case.

```
KDL::Frame eeFrame_d = desFrame;
KDL::Twist eeVelTwist_d(KDL::Vector::Zero(),KDL::Vector::Zero());
KDL::Twist eeAccTwist_d(KDL::Vector::Zero(),KDL::Vector::Zero());

//SWITCHING CONTROLLER
robot_->getInverseKinematics(desFrame, qd_);

// GET desired qdot
dqd_.data.setZero();
// GET desired qddot
ddqd_.data.setZero();
desired_commands_ = toStdVector(controller_.idCntr(qd_, dqd_, ddqd_, Kp_,
    Kd_));
}
```

Listing 4.6: Inverse Dynamic Operational Space Switch

4.2 Testing Trajectories with effort controller

In the same fashion of the old controller K_p and K_d gains were tuned to obtain both good tracking and smooth movements.

In the same fashion of the controller in the joint space, the values of the matrix gains K_p and K_d were tuned by assigning different weights to the elements of the diagonal gain matrices. Specifically, greater emphasis was placed on both the proportional and derivative actions of the third operational variable, which corresponds to the z-axis, representing the distance of the end-effector from the ground. Considering that gravity acts along this direction, the objective was to achieve a smoother trajectory and stronger control action in the z-direction.

Both linear and circular trajectories were tested with this controller using both a cubic polynomial profile and a trapezoidal velocity profile and all gave great results.