# Robotics Lab: Homework 2
# Control a manipulator to follow a trajectory

Ludovica Ruggiero P38000207

repository: ludruggiero/RL/homework2
collaborators: Luigi Catello, Stefano Covone, Cristiana Punzo

## Contents

The aim of this homework is to control a manipulator to follow a trajectory, using the `KDL` library. In Section 1 functions to compute curvilinear abscissa are defined; functions to compute an operational space trajectory are shown in Section 2; Section 3 is about testing the trajectories with a Joint Space Inverse Dynamics Controller, whereas in Section 4 an Operational Space Inverse Dynamics Controller is defined.

# 1 Curvilinear abscissa computation

## 1.a Trapezoidal velocity function

Before defining the function `KDLPlanner::trapezoidal_vel`, a new struct named `curvilinearAbscissa` was introduced in the file `kdl_planner.h`. This struct contains the curvilinear abscissa and its first and second derivatives:

```
struct curvilinearAbscissa{
    double s;
    double sdot;
    double sddot;
};
```

The purpose of the function `KDLPlanner::trapezoidal_vel` is to compute the curvilinear abscissa $s$ and its derivatives, assuming a trapezoidal profile is assigned to the trajectory, with $s$ that takes values in the range $[0, 1]$. The relations implemented in the code are:

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

At first the magnitude of acceleration during the parabolic blends is computed:

```
curvilinearAbscissa KDLPlanner::trapezoidal_vel(double time){
    curvilinearAbscissa abscissa;
    double scddot = -1.0/(std::pow(accDuration_ ,2)-
                          trajDuration_*accDuration_);
```

Then, the curvilinear abscissa is computed at each of the three segments. The trajectory starts with a parabolic segment with constant acceleration:

```
    if (time >= 0 && time <= accDuration_) {
        abscissa.s = 0.5*scddot*std::pow(time,2);
        abscissa.sdot = scddot*time;
        abscissa.sddot = scddot;
```

Then, once the cruise velocity is reached, it continues with constant velocity for a time $t_f - 2t_c$, where $t_c$ is the acceleration duration variable:

```
    } else if (time > accDuration_ &&
                time <= trajDuration_-accDuration_) {
        abscissa.s = 0.5*scddot*(time-accDuration_/2);
        abscissa.sdot = 0.5*scddot;
        abscissa.sddot = 0;
```

Finally, a parabolic segment with negative acceleration drives the abscissa to its final value:

```
    } else if (time > (trajDuration_-accDuration_) &&
                                time <= trajDuration_) {
        abscissa.s = 1 - 0.5*scddot*std::pow(trajDuration_-time,2);
        abscissa.sdot = scddot*(trajDuration_-time);
        abscissa.sddot = -scddot;
    }

    return abscissa;
}
```

## 1.b Cubic polinomial abscissa

An alternative way to assign the curvilinear abscissa consists of using a cubic profile for $s(t)$, which allows the minimization of energy dissipation. In particular, if a profile

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

is assigned to che curvilinear abscissa, it is possible to compute the four parameters imposing the initial and final values of $s(t)$ and $\dot{s}(t)$, respectively:

$$a_0 = s_i \tag{3}$$
$$a_1 = \dot{s}_i \tag{4}$$
$$a3t_f^3 + a_2 t_f^2 + a_1 t_f + a_0 = s_f \tag{5}$$
$$3a3t_f^2 + 2a_2 t_f + a_1 = \dot{s}_f \tag{6}$$

where $s_i = \dot{s}_i = \dot{s}_f = 0$ and $s_f = 1$.

In the `KDLPlanner` object a `KDLPlanner::cubic_polinomial` function has been created which, only at the first iteration, solves the system of equations (3), using the `colPivHouseholderQr` solver from the `Eigen` C++ library:

```
curvilinearAbscissa KDLPlanner::
            cubic_polinomial(double time) {

    curvilinearAbscissa abscissa;

    static bool coeffsComputed = false;
    static Eigen::Matrix4d CoeffsMat;
    static Eigen::Vector4d Boundaries;
    static Eigen::Vector4d coeffs;
```

```
    CoeffsMat << 1,0,0,0,
                 0,1,0,0,
                 1,trajDuration_,pow(trajDuration_,2),
                                 pow(trajDuration_,3),
                 0,1,2*trajDuration_,3*pow(trajDuration_,2),
    Boundaries << 0,0,1,0;

    if (!coeffsComputed) {
        coeffs =
            CoeffsMat.colPivHouseholderQr().solve(Boundaries);
        coeffsComputed = true;
    }
```

Then, at each iteration of the function the curvilinear abscissa $s(t)$ and its derivatives $\dot{s}(t)$ and $\ddot{s}(t)$ are computed as follows:

```
    if (time >= 0 && time <= trajDuration_) {
        abscissa.s = coeffs(3)*pow(time,3) +
                         coeffs(2)*pow(time,2) +
                         coeffs(1)*time + coeffs(0);
        abscissa.sdot = 3*coeffs(3)*pow(time,2) +
                         2*coeffs(2)*time + coeffs(1);
        abscissa.sddot = 6*coeffs(3)*time + 2*coeffs(2);
    }
    return abscissa;
}
```

# 2 Operational space trajectories

## 2.a New onstructors definition

The only constructor present in the code was the one for calculating linear trajectories with a trapezoidal velocity profile.

Despite the prompt requesting the implementation of only one constructor for circular trajectories, it was decided to also implement constructors for cases where it is used to assign a cubic velocity profile.

Four constructors have been implemented:

- Constructor to compute linear trajectory with trapezoidal velocity profile

```
KDLPlanner(double _trajDuration, double _accDuration,
                                 Eigen::Vector3d _trajInit,
                                 Eigen::Vector3d _trajEnd);
```

- Constructor to compute circular trajectory with trapezoidal velocity profile

```
KDLPlanner(double _trajDuration, double _accDuration,
                                 Eigen::Vector3d _trajInit,
```

```
                                    double _trajRadius);
```

- Constructor to compute linear trajectory with cubic polinomial profile

```
KDLPlanner(double _trajDuration,
           Eigen::Vector3d _trajInit,
           Eigen::Vector3d _trajEnd);
```

- Constructor to compute circular trajectory with cubic polinomial profile

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

As an example, the constructor for calculating a circular trajectory with a cubic velocity profile is presented:

```
KDLPlanner::KDLPlanner(double _trajDuration,
    Eigen::Vector3d _trajInit, double _trajRadius) {
            trajDuration_ = _trajDuration;
            accDuration_ = -1;
            trajInit_ = _trajInit;
            trajEnd_ = _trajInit;
            trajRadius_ = _trajRadius;
    }
```

The constructors for linear trajectories initialize the radius variable to $-1$. Similarly, the constructors for cubic profile initialize the `acc.duration` variable to $-1$. These values are used as flags to distinguish the use of one trajectory over the other.

## 2.b Computation of circular trajectory

The function `KDLPlanner::compute_circle_trajectory` checks the value of `acc.duration` and decides accordingly whether to calculate the curvilinear abscissa using a trapezoidal or cubic profile.

```
    curvilinearAbscissa abscissa;
    if (accDuration_ < 0) {
        abscissa = cubic_polinomial(time);
    } else {
        abscissa = trapezoidal_vel(time);
    }
```

It is desired to implement a circular trajectory with the following equations:

$$x = x_i \qquad y = y_i - r\cos 2\pi s \qquad z = z_i - r\sin 2\pi s \tag{7}$$

where $x_i$, $y_i$ and $z_i$ represent the coordinates of the center of the circular trajectory.

The center of the trajectory is located in the vertical plane containing the end-effector, therefore it was calculated by adding to the `y-component` of the initial point an amount equal to the radius.

```
    trajectory_point traj;
    Eigen::Vector3d circleCenter = trajInit_;
    circleCenter(1) += trajRadius_;
```

Then, given the coordinates of the just calculated center, it was possible to compute the sequence of points of the circular trajectory as follows:

```
Eigen::Vector3d circle;
circle << circleCenter(0),
        circleCenter(1)-trajRadius_*cos(2*M_PI*abscissa.s),
        circleCenter(2)-trajRadius_*sin(2*M_PI*abscissa.s);
```

By taking the derivatives of the position expressions, velocities and accelerations are similarly obtained:

```
Eigen::Vector3d circledot;
circledot << 0,
    2*M_PI*trajRadius_*abscissa.sdot*sin(2*M_PI*abscissa.s),
    -2*M_PI*trajRadius_*abscissa.sdot*cos(2*M_PI*abscissa.s);

Eigen::Vector3d circleddot;
circleddot << 0,
              4*pow(M_PI,2)*trajRadius_*std::pow(abscissa.sdot,2)*
              cos(2*M_PI*abscissa.s) +
              2*M_PI*trajRadius_*pow(abscissa.sdot,2)*
              sin(2*M_PI*abscissa.s),
              4*pow(M_PI,2)*trajRadius_*std::pow(abscissa.sdot,2)*
              sin(2*M_PI*abscissa.s) -
              2*M_PI*trajRadius_*pow(abscissa.sdot,2)*
              cos(2*M_PI*abscissa.s);
```

## 2.c Computation of linear trajectory

For the function `KDLPlanner::compute_linear_trajectory`, a code similar to the one above has been implemented. First, the flag indicating whether the assigned velocity profile is trapezoidal or cubic is analyzed.

```
curvilinearAbscissa abscissa;
if (accDuration_ < 0) {
  abscissa = cubic_polinomial(time);
} else {
  abscissa = trapezoidal_vel(time);
}
```

Unlike the function `KDLPlanner::compute_circle_trajectory`, the point of the trajectory is added directly to the `traj.pos` variable:

```
traj.pos = trajInit_ + abscissa.s * (trajEnd_ - trajInit_);
```

The velocity and acceleration were then calculated by taking the derivatives of the position:

```
traj.vel = abscissa.sdot * (trajEnd_ - trajInit_);
traj.acc = abscissa.sddot * (trajEnd_ - trajInit_);
```

# 3 Joint Space Inverse Dynamics Controller

After having implemented the four different trajectories, it is possibile to test them using the provided Joint Space Inverse Dynamics Controller. It can be noticed how in the original file this controller is provided with uncomplete references: in particular, the function `getInvKin()` only provides a position inversion, whereas the controller needs references of velocity and acceleration, too.

To solve this inconvenience, the function `getInverseKinematics()`, already defined as a member function of the class `KDLRobot`, is implemented to compute the second order inverse kinematics:

```
void KDLRobot::getInverseKinematics(KDL::Frame &f,
                            KDL::Twist &twist,
                            KDL::Twist &acc,
                            KDL::JntArray &q,
                            KDL::JntArray &dq,
                            KDL::JntArray &ddq)
{
    q = getInvKin(q,f);
    ikVelSol_->CartToJnt(q,twist,dq);

    Eigen::Matrix<double,6,7> J = toEigen(getEEJacobian());
    Eigen::VectorXd x_ddot = toEigen(acc);
    Eigen::VectorXd Jdot_qdot = getEEJacDotqDot();
    Eigen::Matrix<double,7,6> Jpinv = pseudoinverse(J);

    ddq.data = Jpinv*(x_ddot - Jdot_qdot);
}
```

The position inversion is implemented using the function `getInvKin()` already provided by the class `KDLRobot`, whereas the velocity inversion is carried out by the KDL standard solver for inverse kinematics `KDL::ChainIkSolverVel_wdls` using its member function `CartToJnt()`. The function takes as inputs an initial guess on joint positions and the robot twist and saves the joint velocities into a variable `KDL::JntArray` provided by reference.

As regards the inversion of accelerations, KDL doesn't implement a dedicated solver, so it is computed through an implementation of the Second Order Inverse Differential Kinematics:

$$\ddot{q} = J^{\dagger}(q)(\ddot{x}_e + \dot{J}(q,\dot{q})\dot{q}) \tag{8}$$

It is important to note that this implementation provides an open loop kinematic inversion of acceleration, therefore the solution may be affected by a numerical drift caused by the discrete integration method.

### 3.a Trajectory testing

After fixing the controller, it is now possible to test the computed trajectories. The main file `kdl_robot_test.cpp` is modified in order to allow the selection of the desired trajectory at execution time:

```
int trajFlag;
std::cout << "Choose desired trajectory:" << std::endl <<
             "1. linear (trapezoidal profile)\n" <<
             "2. linear (cubic profile)\n" <<
             "3. circular (trapezoidal profile)\n" <<
             "4. circular (cubic profile)\n";
std::cout << "Insert number: ";
std::cin >> trajFlag;
```

Then, a function `chooseTrajectory` is defined, which takes as input the given flag and the trajectory parameters and consequentially invokes the right `KDLPlanner` constructor:

```
KDLPlanner chooseTrajectory(int trajFlag,
                            Eigen::Vector3d init_position,
                            Eigen::Vector3d end_position,
                            double traj_duration,
                            double acc_duration,
                            double t,
                            double init_time_slot,
                            double traj_radius) {
    switch(trajFlag) {
        case 1:
            return KDLPlanner(traj_duration, acc_duration,
                              init_position, end_position);
            break;
        ...
        default:
            return KDLPlanner(traj_duration, acc_duration,
                              init_position, end_position);
    }
}
```

which is then called inside the main:

```
KDLPlanner planner = chooseTrajectory(trajFlag,
                                      init_position,
                                      end_position,
                                      traj_duration,
                                      acc_duration,
                                      t,
                                      init_time_slot,
                                      traj_radius);
```

The result at execution time is shown in Figure 3.1

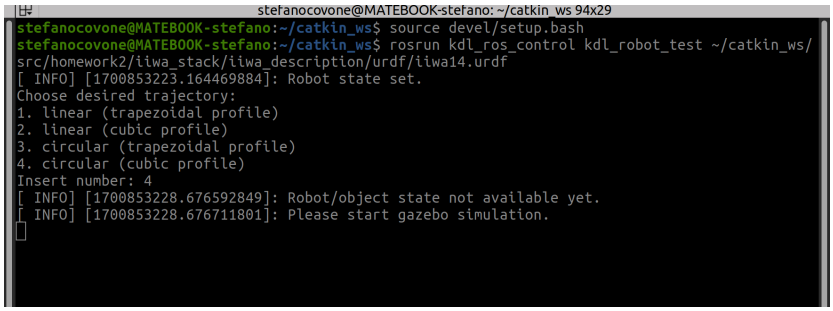



Figure 3.1: Trajectory choice at execution time

## 3.b Controller tuning and Results visualization

The control gains were chosen by trial and error. The best trade-off between error norm and actuators effort was obtained with:

$$K_P = 100 \qquad K_D = 20 \tag{9}$$

To evaluate the performance, trying to achieve low error and reduced chattering, `rqt_plot` is executed from the terminal.

To obtain a better evaluation of control performances, it not only subscribes to the torque command topics, but it was also created a new topic which delivers information about the norm of the error.

```
e_norm = e.norm();
```

The `iiwa/error` topic is implemented analogously to the already defined torque commands topic, but the controller `KDLController::idCntr` was modified in order to output the norm of the error, computed via the member function `Eigen::VectorXd::norm()`.

So, `rqt_plot` subscribes to the `/iiwa/error` topic and plots the norm of the error published by the `kdl_robot_test` node. An example of the plot is shown in Figure 3.2

## 3.c Plotting from bagfile using MATLAB

To obtain cleaner plots, it is possible to record all the messages sent on a set of topics in a bagfile, and use MATLAB to create the figures. Before launching the simulation, `rosbag` is set up listing all the topics that are going to be recorded:

```
$ rosbag record -o data.bag /iiwa/error /iiwa/iiwa_joint_...
```

The MATLAB function used to plot the error and the torques is the following:

```
function plotErrorAndTorques(bagname)
    % load ROS bag
```

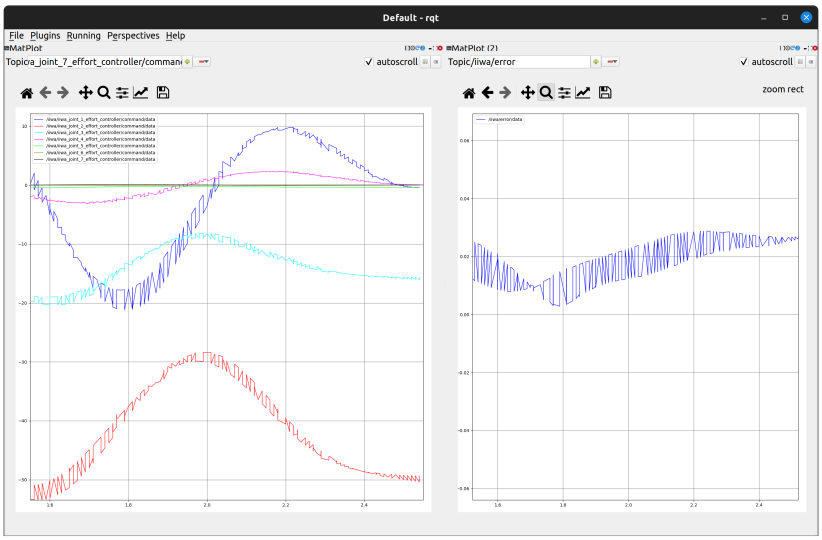



Figure 3.2: Plotting the error using rqt_plot

```
bag = rosbag(bagname);

% error plotting
err = bag.select("Topic","/iiwa/error");
errTs = timeseries(err);
figure("Name","Error")
plot(errTs)
grid on
ylabel("")
title("Error norm")

% torques
fig_torques = figure("Name","Torques");
leg = cell(7,1);
for i = 1:7
    topicName = "/iiwa/iiwa_joint_" + string(i) +
        "_effort_controller/command";
    torqueTs = timeseries(bag.select("Topic",
        topicName));
    figure(fig_torques);
    plot(torqueTs)
    hold on
    leg{i} = "Joint " + i + " torque";
end
hold off
```

```
    title("Torques")
    legend(leg,"location","sw")
    ylabel("")
    grid on
end
```

The obtained plots for each possible combination of abscissa and trajectory are shown in Table 1.

**Results analysis** Without the implementation of the second order differential inverse kinematics, the error norm was of about 0.3 m, so the assignment of the correct references provided significative improvements in the control performance. Anyway, the error norm still has a mean value of some centimeters, which is not an acceptable value for a good tracking of trajectory.

Tuning the PD gains was not effective to reduce both chattering and error norm: in particular, larger gains, despite lowering the error norm, would have increased the chattering of the torque commands.

This behaviour is probably due to numerical error in the simulation or to an imperfect feedback linearization of the robot dynamics: specifically, the friction torques are not modelled within the controller. It was tried to include such effects, as can be shown in the code, but it caused the manipulator to become unstable. Some further investigation could include the reduction of sampling time or the adoption of a closed loop algorithm to compute joints acceleration references.

# 4 Inverse Dynamics Operational Space Controller

## 4.a Inverse Dynamics Controller

To implement an operational space dynamics controller, the `KDLController::idCntr` function was overloaded. It returns the control torques required by the joints for the end-effector to follow the desired trajectory and the norm of the error (to be published via the `iiwa/error` topic defined above). Differently from joint space inverse dynamics controller, the operational space controller computes the errors in Cartesian space.

## 4.b Implementation

The controller logic was already defined inside the function, but some errors had to be fixed to make it work properly. The function computes the gain matrices $K_p$ and $K_d$, given the input `_Kpp`, `_Kpo`, `_Kdp`, `_Kdo` as follows:

```
// calculate gain matrices
Eigen::Matrix<double,6,6> Kp, Kd;
Kp = Eigen::MatrixXd::Zero(6,6);
Kd = Eigen::MatrixXd::Zero(6,6);
Kp.block(0,0,3,3) = _Kpp*Eigen::Matrix3d::Identity();
```

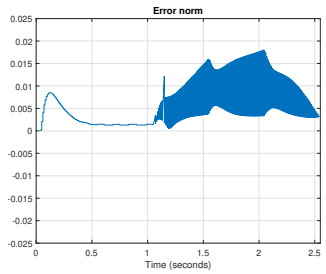
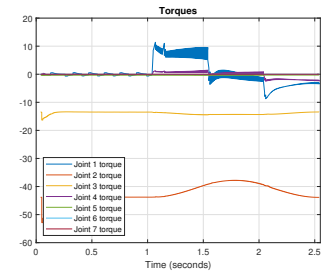
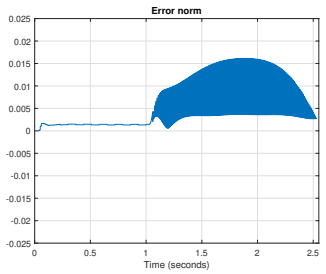
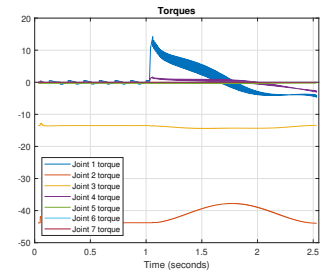
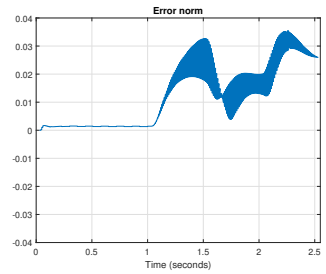
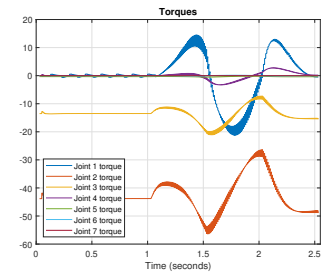
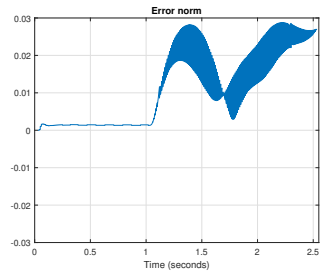
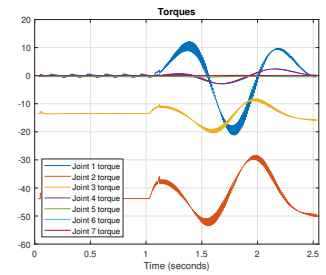
| Velocity profile: | trapezoidal | cubic |
|---|---|---|
| Linear trajectory |   |   |
| Circular trajectory |   |   |

Table 1: Error and torques plots

```
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();
```

Unlike in the original file, a zero initialization of the matrices has been added, to set to zero also the off-diagonal elements.

The objective of the function is to compute inverse dynamics operational space control based on the equation:

$$\tau = By + n \quad n = J_A^{\dagger}(\ddot{x}_d + K_D\dot{\tilde{x}} + K_P\tilde{x}) \tag{10}$$

In order to achieve this, it is necessary to retrieve the inertia matrix $B$, the Jacobian $J$, the pseudoinverse matrix $J^{\dagger}$, its time derivative, the linear $e_p$ and the angular $e_o$ errors directly from the `KDLRobot` class or by utilizing functions within the `utils.h` file. Here, the only change with respect to the original controller regards Jacobian computation, where the original code was modified in order to obtain a matrix of type `Eigen::Matrix`:

```
// read current state
Eigen::Matrix<double,6,7> J = toEigen(robot_->getEEJacobian());
Eigen::Matrix<double,7,7> I =
                        Eigen::Matrix<double,7,7>::Identity();
Eigen::Matrix<double,7,7> M = robot_->getJsim();
Eigen::Matrix<double,7,6> Jpinv = weightedPseudoInverse(M,J);
```

Then, the robot current and desired pose, velocity and accelerations are defined. To obtain cartesian space pose, the function `getEEFrame()` was used. Similiarly, velocities were obtained via the `getEEVelocity()` function:

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

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

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

Then, the position and orientation errors are computed. In particular, since the `getEEJcobian()` function returns the geometric jacobian, the orientation

error must be computed using angle and axis non-minimal representation of orientation. Finally, the computation of the operational space error norm was added:

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

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

Finally, once that all the term of the equation (10) are available, it is possible to compute the joint control torques $\tau$.

```
// inverse dynamics
Eigen::Matrix<double,6,1> y;
y << dot_dot_x_d - robot_->getEEJacDotqDot() +
                            Kd*dot_x_tilde + Kp*x_tilde;
return M * (Jpinv*y + (I-Jpinv*J)*
                        (- 1*robot_->getJntVelocities())) +
                                    robot_->getGravity() +
                                    robot_->getCoriolis();
```

The `getEEJacDotqDot()` function, originally returning only $\dot{J}$, was fixed to return the proper quantity:

```
Eigen::VectorXd KDLRobot::getEEJacDotqDot() {
    return s_J_dot_ee_.data*jntVel_.data;
}
```

## 4.c Results

The controller was tested in simulation for all four possible combinations of trajectory and velocity profile, yielding satisfactory results at the expense of some fine tuning of the gains. The errors and torques for every case are shown in Table 2. The plots were obtained using the MATLAB function defined in Section 3.c.

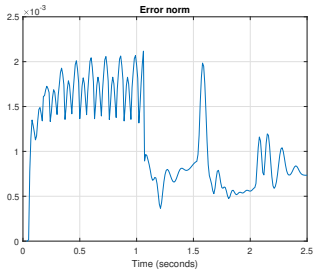
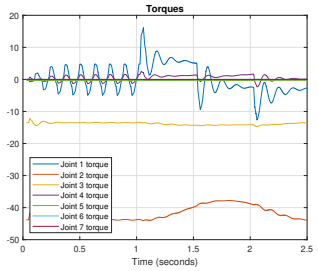
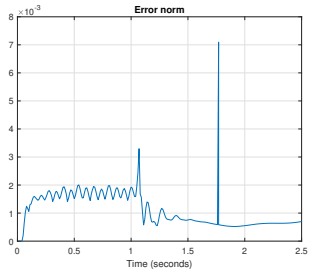
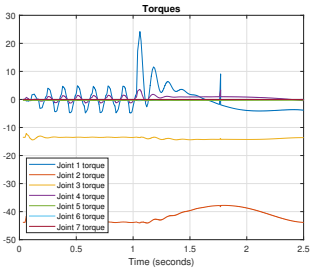
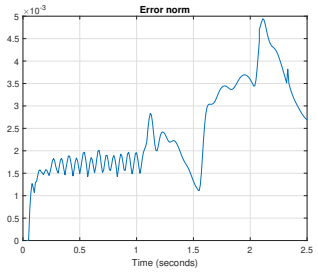
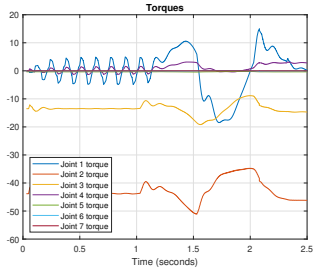
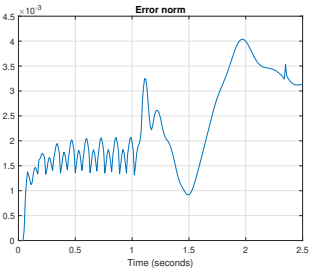
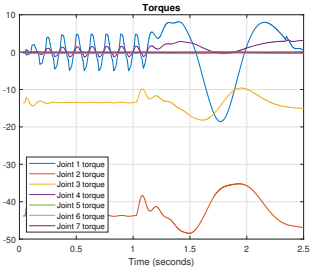
| Velocity profile: | trapezoidal | cubic |
|---|---|---|
| Linear trajectory |  <br>  |  <br>  |
| Circular trajectory |  <br>  |  <br>  |

Table 2: Error and torques plots

**Results analysis** The performances of the operational space controller are much better than that obtained with he joint space controller. The results shown in Table 2 were obtained with gains:

$$K_{P_p} = 400 \qquad K_{D_p} = 80$$
$$K_{P_p} = 400 \qquad K_{D_o} = 80$$

The error norm, for both linear and circular trajectory, has a mean value $< 0.002$ mm and the chattering is far less than the joint space case.

Similiarly to the joint space one, this controller is affected by imperfect compensation of nonlinear terms, due to the unmodelled friction torques. This controller does not require a kinematic inversion, which is already included in the controller logic, so its better performance coud justify the larger error of the joint space controller with a numerical error in the kinematic inversion.