Homework 2

Andrea Morghen, Maria Vittoria Cinquegrani, Valentina Giannotti

November 27, 2023

GitHub link: https://github.com/Andremorgh/RL_HW_02
GitHub link: https://github.com/MVCinquegrani/ROS_Homework2
GitHub link: https://github.com/ValentinaGiannotti/Homework2

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

1a `KDLPlanner` modifications

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:

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

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

Firstly, in the file `kdl_planner.h`, a new function named `trapezoidal_vel` has been declared within the `KDLPlanner` class. This declaration provides an interface that specifies the function signature, indicating that the `KDLPlanner` class contains a method with the following parameters: the current time `double t`, the acceleration duration `double tc`, and references to the curvilinear abscissa and its derivatives `double s`, `double s_d`, and `double s_dd`.

```
#ifndef KDLPlanner_H
#define KDLPlanner_H
...

class KDLPlanner
{

public:
    ...

    // New function to calculate the trapezoidal velocity profile
    void trapezoidal_vel(double t,double tc,double &s,double &s_d,double &s_dd
        );

private:
    ...
};
#endif
```

The `trapezoidal_vel` function is then implemented in `kdl_planner.cpp` as follows:

```
#include "kdl_ros_control/kdl_planner.h"
...

void KDLPlanner::trapezoidal_vel(double t,double tc,double &s,double &s_d,
    double &s_dd){

  double s_ddot_c = 5.0/(std::pow(trajDuration_,2));

  if(t <= tc)
  {
    s=0.5*s_ddot_c*std::pow(t,2);
    s_d = s_ddot_c*t;
    s_dd = s_ddot_c;
  }
  else if(t <= trajDuration_-tc)
  {
    s=s_ddot_c*tc*(t-tc/2);
    s_d = s_ddot_c*tc;
    s_dd = 0;
  }
  else
  {
    s=1-0.5*s_ddot_c*std::pow(trajDuration_-tc,2);
    s_d = s_ddot_c*(trajDuration_-t);
    s_dd = -s_ddot_c;
  }
}
...
```

The `trapezoidal_vel` function implements a trapezoidal velocity profile for trajectory planning reported in the equation 1. It calculates the maximum allowable acceleration based on the total trajectory duration t_f according to the equation:

$$|\ddot{s}_c| \geq \frac{4|q_f - q_i|}{{t_f}^2} = \frac{4|1-0|}{{t_f}^2} \geq \frac{4}{{t_f}^2} \tag{2}$$

in particular we have chosen

$$|\ddot{s}_c| = \frac{5}{{t_f}^2} \geq \frac{4}{{t_f}^2} \tag{3}$$

The process is divided into three main phases. During the acceleration phase, the curvilinear path follows a quadratic law, velocity is proportional to time and acceleration is constant. In the constant velocity phase, the curvilinear abscissa follows a linear law and the velocity remains constant. In the deceleration phase, the curvilinear abscissa follows an inverse parabolic law, the velocity is proportional to the remaining time and the acceleration is constant but negative. Due to pass by reference the function returns the values of the curvilinear abscissa, velocity and acceleration based on the current time, providing the mathematical model for precise trajectory planning in robotic systems.

1b `KDLPlanner::cubic_polinomial` creation

Create a function named `KDLPlanner::cubic_polynomial` *that creates the cubic polynomial curvilinear abscissa for your trajectory. The function takes as an argument a* `double` t *representing time and returns three* `double` $s, \dot{s}, \ddot{s}$ *that represent the curvilinear abscissa of your trajectory. Remember, a cubic polynomial is defined as follows:*

$$s(t) = a_3 t^3 + a_2 t^2 + a_1 t + a_0 \tag{4}$$

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 by calculating the time derivative of equation (4).*

Hence we declared the new function named `cubic_polinomial` within the `KDLPlanner` class.

```
#ifndef KDLPlanner_H
#define KDLPlanner_H
...

class KDLPlanner
{
public:
...
    // New function to calculate the trapezoidal velocity profile
    void trapezoidal_vel(double t,double tc,double &s,double &s_d,double &s_dd
        );
    // New function to calculate the cubic polynomial curvilinear abscissa
    void cubic_polinomial(double t,double &s,double &s_d,double &s_dd);
...
};
#endif
```

Then we implemented it as follow:

```
void KDLPlanner::cubic_polinomial(double t,double &s,double &s_d,double &s_dd)
    {
    double a_2=3/(std::pow(trajDuration_,2));
    double a_3=-2/(std::pow(trajDuration_,3));

    s=a_3*std::pow(t,3)+a_2*std::pow(t,2);
    s_d =3*a_3*std::pow(t,2)+2*a_2*t;
    s_dd = 6*a_3*t+2*a_2;
}
```

The `cubic_polynomial` function calculates the profile of a cubic polynomial for the curvilinear abscissa as a function of time t, returning three values: the curvilinear abscissa s, the velocity $\dot{s}$, and the acceleration $\ddot{s}$. The curvilinear abscissa is defined as a combination of cubic and quadratic terms of time, with coefficients a_2 and a_3 calculated by solving the equations:

$$\begin{cases} a_0 = q_i = 0 \\ a_1 = \dot{q_i} = 0 \\ a_3 t_f^3 + a_2 t_f^2 + a_1 t_f + a_0 = q_f = 1 \\ a_3 t_f^2 + 2 a_2 t_f + a_1 = \dot{q_f} = 0 \end{cases} \tag{5}$$

and thus obtaining the following coefficient values:

$$a_0 = 0, a_1 = 0, a_2 = 3/t_f^2, a_3 = -2/t_f^3 \tag{6}$$

The velocity and acceleration are then obtained by taking the derivatives of the resulting cubic polynomial s with respect to time. This function is useful for generating a controlled and smooth trajectory in motion planning contexts.

2 Create circular trajectories for your robot

2a Constructor KDLPlanner::KDLPlanner definition

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 stores them in the corresponding class variables (to be created in the `kdl_planner.h`).

```
#ifndef KDLPlanner_H
#define KDLPlanner_H
...

class KDLPlanner
{

public:
...

    // New function to calculate the trapezoidal velocity profile
    void trapezoidal_vel(double t,double tc,double &s,double &s_d,double &s_dd
        );
    // New function to calculate the cubic polynomial curvilinear abscissa
    void cubic_polinomial(double t,double &s,double &s_d,double &s_dd);
    // New constructor for the circular trajectories planner
    KDLPlanner(double _trajDuration, Eigen::Vector3d _trajInit, double
        _trajRadius);

private:
...

    double trajDuration_, accDuration_;
    Eigen::Vector3d trajInit_, trajEnd_;
    trajectory_point p;
    double trajRadius_;

};
#endif
```

The constructor for the `KDLPlanner` class is declared and takes the following parameters:

- `_trajDuration`: A `double` value representing the duration of the trajectory.
- `_trajInit`: A vector of type `Eigen::Vector3d` representing the initial position of the trajectory. `Eigen::Vector3d` is commonly used to represent three-dimensional vectors in Eigen, a C++ library for linear algebra.
- `trajRadius`: A `double` value representing the radius of the trajectory.

These parameters are passed to the constructor when creating a new object of the `KDLPlanner` class.

2b Circular trajectory implementation

The center of the trajectory must be in the vertical plane containing the end-effector. Create the positional path as a 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:

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

In this points, it is required to overwrite the `KDLPlanner::compute_trajectory` function, so we decided to create an auxiliary variable that would allow the user to choose which function to execute. Specifically, such an integer variable can be:
- less than 2, circular trajectory
- more equal than 2, rectilinear trajectory
In the already available function `KDLPlanner::compute_trajectory` we implemented the equation 7 and its derivatives to obtain trajectory position velocity and acceleration for each axis.

```
trajectory_point KDLPlanner::compute_trajectory(double time,int choice)
{
  double s,s_d,s_dd;
  cubic_polinomial(time,s,s_d,s_dd);
  //trapezoidal_vel(time,0.7,s,s_d,s_dd);

  trajectory_point traj;

  if(choice < 2)
  {
    // Create circular trajectory in the y-z plane
    traj.pos.x() = trajInit_.x();
    traj.pos.y() = trajInit_.y() - trajRadius_ * cos(2 * 3.14 * s)+trajRadius_
        ;
    traj.pos.z() = trajInit_.z() - trajRadius_ * sin(2 * 3.14 * s);

    // Set velocity and acceleration based on derivatives
    traj.vel.y() = trajRadius_ * 2 * 3.14 * s_d * sin(2 * 3.14 * s);
    traj.vel.z() = -trajRadius_ * 2 * 3.14 * s_d * cos(2 * 3.14 * s);

    traj.acc.y() = trajRadius_ * (2 * 3.14) * s_dd * sin(2 * 3.14 * s)+
        trajRadius_ * (2 * 3.14) *(2 * 3.14) * std::pow(s_d,2)* cos(2 * 3.14 *
         s);
    traj.acc.z() = trajRadius_ * (2 * 3.14) *(2 * 3.14) * std::pow(s_d,2) *
        sin(2 * 3.14 * s)-trajRadius_ * (2 * 3.14) * s_dd * cos(2 * 3.14 * s);
  }
  else
  {
    // Create linear trajectory
  }
   return traj;
}
```

The `KDLPlanner::compute_trajectory` function generates a trajectory based on the time given as an argument. First, it calculates the curvilinear abscissa and its derivatives according to the chosen velocity profile by calling the `trapezoidal_vel` or `cubic_polinomial` function. Then, it creates a circular trajectory in the $y - z$ plane using the initial coordinates x_i, y_i, z_i and the trajectory radius (r). Subsequently, it calculates the first and second derivatives of the circular trajectory and assigns the corresponding values to the `trajectory_point` structure representing position, velocity, and acceleration. The function then returns this structure, providing a desired trajectory based on the given time.

2c Linear trajectory implementation

Do the same for the linear trajectory.

For the linear trajectory we take into account the following parametric representation of a linear path:

$$p(s) = p_i + s(pf - pi) \tag{8}$$

Here the derivatives of the equation 8 that we used:

$$\dot{p}(s) = \dot{s}(pf - pi), \quad \ddot{p}(s) = \ddot{s}(pf - pi) \tag{9}$$

```
trajectory_point KDLPlanner::compute_trajectory(double time)
{
  double s,s_d,s_dd;
  cubic_polinomial(time,s,s_d,s_dd);
  // trapezoidal_vel ( time ,1.0 , s , s_d , s_dd )

  trajectory_point traj;

  traj.pos = trajInit_ + s*(trajEnd_-trajInit_);
  traj.vel = s_d*(trajEnd_-trajInit_);
  traj.acc = s_dd*(trajEnd_-trajInit_);

  return traj;
}
```

The `KDLPlanner::compute_trajectory` function calculates the linear trajectory based on the provided time. Using a cubic polynomial curvilinear abscissa, it computes the position, velocity, and acceleration of the linear trajectory. The calculated values are then assigned to the `trajectory_point` structure and returned.

3 Test the four trajectories

3a `kdl_robot_test.cpp` modification

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

We decided to improve the kinematic inversion as it was limited to space inversion only. We added velocity and acceleration inversion, this provided a great reduction in error.

```
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);
}
```

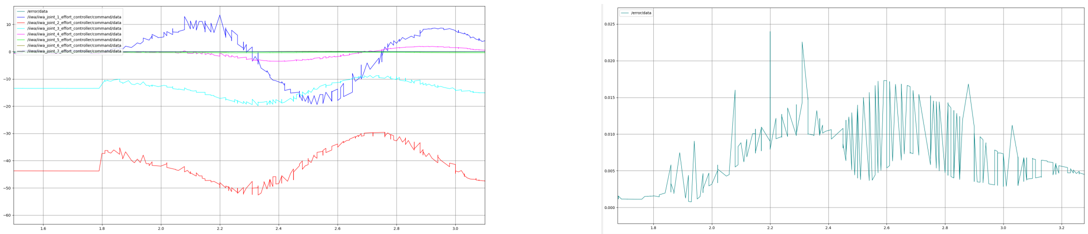
```
robot.getInverseKinematics(des_pose, des_cart_vel, des_cart_acc,qd,dqd,ddqd);
```

To test the things done so far, it is therefore necessary to start an instance of gazibo with the iiwa14 robot preloaded and then call up the kdl_robot_test programme modified earlier with the iiwa14.urdf file attached

```
roslaunch iiwa_gazebo iiwa_gazebo_effort.launch
rosrun kdl_ros_control kdl_robot_test /src/iiwa_stack/iiwa_description/urdf/
    iiwa14.urdf
```

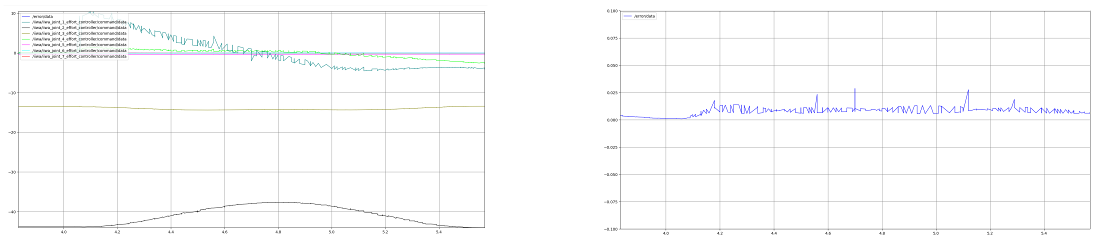
3b Control gains tuning

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.*



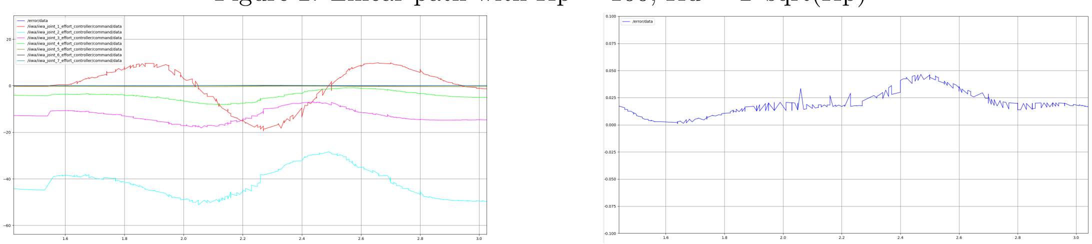
(a) Kp = 100, Kd = 2*sqrt(Kp) (b) error

Figure 1: Circular path with Kp = 100, Kd = 2*sqrt(Kp)



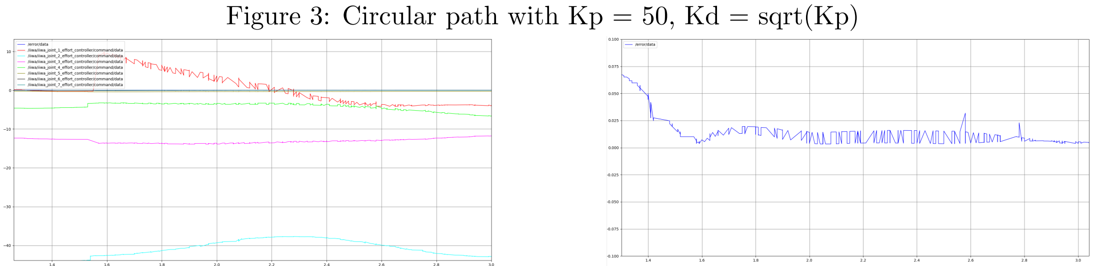
(a) Kp = 100, Kd = 2*sqrt(Kp) (b) error

Figure 2: Linear path with Kp = 100, Kd = 2*sqrt(Kp)



(a) Kp = 50, Kd = sqrt(Kp) (b) error

Figure 3: Circular path with Kp = 50, Kd = sqrt(Kp)



(a) Kp = 50, Kd = sqrt(Kp) (b) error

Figure 4: Linear path with Kp = 50, Kd = 2*sqrt(Kp)

Only the results of the cubic profile have been plotted. If you wish to view the other results, it is possible to comment out the code line for the cubic profile and uncomment the code line with the trapezoidal profile and switch between circular and linear trajectory via the traj_var variable in the kdl_robot_test program.

3c Optional:

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.

In order to accomplish this point, we ha to install the ROS toolbox in Matlab. We recorder the desired topics using the command

```
rosbag record /iiwa/iiwa_joint_1_effort_controller/command /iiwa/
    iiwa_joint_2_effort_controller/command /iiwa/
    iiwa_joint_3_effort_controller/command /iiwa/
    iiwa_joint_4_effort_controller/command /iiwa/
    iiwa_joint_5_effort_controller/command /iiwa/
    iiwa_joint_6_effort_controller/command /iiwa/
    iiwa_joint_7_effort_controller/command -o joint_torque.bag
```

This node registers to the node named after record and save the all in join_torque.bag

After that we sent this file in matlab where we extract the messages from each topics and plot them.

```
bag = rosbag('joint_torque_2.bag')
bagInfo = rosbag('info','joint_torque.bag')
rosbag info 'joint_torque.bag'

jt_1 = select(bag,'Topic','iiwa/iiwa_joint_1_effort_controller/command
    ');
msgStructs = readMessages(jt_1,'DataFormat','struct');
jt_1_double = cellfun(@(m) double(m.Data),msgStructs);
jt_2 = select(bag,'Topic','iiwa/iiwa_joint_2_effort_controller/command
    ');
msgStructs = readMessages(jt_2,'DataFormat','struct');
jt_2_double = cellfun(@(m) double(m.Data),msgStructs);
jt_3 = select(bag,'Topic','iiwa/iiwa_joint_3_effort_controller/command
    ');
msgStructs = readMessages(jt_3,'DataFormat','struct');
jt_3_double = cellfun(@(m) double(m.Data),msgStructs);
jt_4 = select(bag,'Topic','iiwa/iiwa_joint_4_effort_controller/command
    ');
msgStructs = readMessages(jt_4,'DataFormat','struct');
jt_4_double = cellfun(@(m) double(m.Data),msgStructs);
jt_5 = select(bag,'Topic','iiwa/iiwa_joint_5_effort_controller/command
    ');
msgStructs = readMessages(jt_5,'DataFormat','struct');
jt_5_double = cellfun(@(m) double(m.Data),msgStructs);
jt_6 = select(bag,'Topic','iiwa/iiwa_joint_6_effort_controller/command
    ');
msgStructs = readMessages(jt_6,'DataFormat','struct');
jt_6_double = cellfun(@(m) double(m.Data),msgStructs);
jt_7 = select(bag,'Topic','iiwa/iiwa_joint_7_effort_controller/command
    ');
msgStructs = readMessages(jt_7,'DataFormat','struct');
jt_7_double = cellfun(@(m) double(m.Data),msgStructs);

subplot (4,2,1)
plot(jt_1_double)
subplot (4,2,2)
plot(jt_2_double)
subplot (4,2,3)
plot(jt_3_double)
```

```
subplot (4,2,4)
plot(jt_4_double)
subplot (4,2,5)
plot(jt_5_double)
subplot (4,2,6)
plot(jt_6_double)
subplot (4,2,7)
plot(jt_7_double)
```

the results are shown below:

```
bag =
  BagSelection with properties:

           FilePath: 'C:\Users\ANDREA\Downloads\joint_torque_2.bag'
          StartTime: 0.4100
            EndTime: 2.9100
        NumMessages: 3514
    AvailableTopics: [7×3 table]
    AvailableFrames: {0×1 cell}
        MessageList: [3514×4 table]

bagInfo = struct with fields:

        Path: 'C:\Users\ANDREA\Downloads\joint_torque_2.bag'
     Version: '2.0'
    Duration: 2.5000
       Start: [1×1 struct]
         End: [1×1 struct]
        Size: 240685
    Messages: 3514
       Types: [1×1 struct]
      Topics: [7×1 struct]

Path:     C:\Users\ANDREA\Downloads\joint_torque_2.bag
Version:  2.0
Duration: 2.5s
Start:    gen 01 1970 01:00:00.41 (0.41)
End:      gen 01 1970 01:00:02.91 (2.91)
Size:     235.0 KB
Messages: 3514
Types:    std_msgs/Float64 [fdb28210bfa9d7c91146260178d9a584]
Topics:   /iiwa/iiwa_joint_1_effort_controller/command  502 msgs  : std_msgs/Float64
          /iiwa/iiwa_joint_2_effort_controller/command  502 msgs  : std_msgs/Float64
          /iiwa/iiwa_joint_3_effort_controller/command  502 msgs  : std_msgs/Float64
          /iiwa/iiwa_joint_4_effort_controller/command  502 msgs  : std_msgs/Float64
          /iiwa/iiwa_joint_5_effort_controller/command  502 msgs  : std_msgs/Float64
          /iiwa/iiwa_joint_6_effort_controller/command  502 msgs  : std_msgs/Float64
          /iiwa/iiwa_joint_7_effort_controller/command  502 msgs  : std_msgs/Float64
```

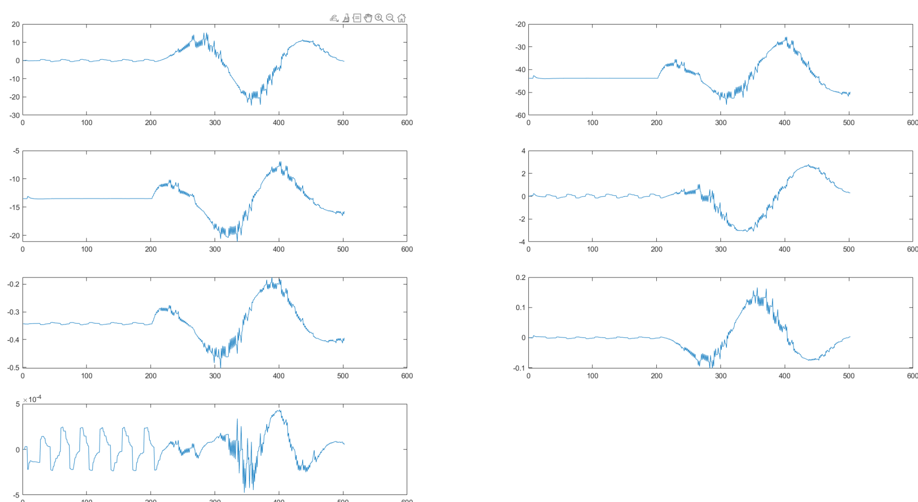


Figure 5: matlab result

4 Develop an inverse dynamics operational space controller

4a KDLController::idCntr implementation

Into the `kdl_contorl.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.*

Hence we declared the new function within the `KDLcontroller` class to create a controller. This function has the same name as the one running in the joint space `idCntr`, but having a different signature

```
#ifndef KDLControl
#define KDLControl
...
class KDLController
{

public:
...
   // New function for the operational space control
    Eigen::VectorXd idCntr(KDL::Frame &_desPos,
                           KDL::Twist &_desVel,
                           KDL::Twist &_desAcc,
                           double _Kpp,
                           double _Kpo,
                           double _Kdp,
                           double _Kdo,
                           double &error);
...
};
#endif
```

4b Creating a subscriber in the cpp file

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 endeffector parametrized pose x, velocity $\dot{x}$, and acceleration $\ddot{x}$, retrieve the current joint space inertia matrix M 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:

$$\tau = By + n, \quad y = J_A^{\dagger}(\ddot{x}_d + K_D\dot{\tilde{x}} + K_P\tilde{x} - \dot{J}_A\dot{q}). \tag{10}$$

We defined the structure of the function `idCntr` using the already existing functions of `utils.h` and `kdl_robot.h`

```
Eigen::VectorXd KDLController::idCntr(KDL::Frame &_desPos,
                                      KDL::Twist &_desVel,
                                      KDL::Twist &_desAcc,
                                      double _Kpp, double _Kpo,
                                      double _Kdp, double _Kdo, double &error)
{
    // calculate gain matrices
    Eigen::Matrix<double,6,6> Kp, Kd;
    // initializzation to 0 of the matrix Kp, Kd. On the wiki it is said it is
        not necessary but from our tests it result to be
```

```cpp
    Kp=Eigen::MatrixXd::Zero(6,6);
    Kd=Eigen::MatrixXd::Zero(6,6);
    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();

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

    // 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);

    // 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;
    error=x_tilde.norm();
    dot_x_tilde << dot_e_p, dot_e_o; //-omega_e;//dot_e_o;
    dot_dot_x_d << dot_dot_p_d, dot_dot_r_d;

    // null space control
    double cost;
    Eigen::VectorXd grad = gradientJointLimits(robot_->getJntValues(),robot_->
        getJntLimits(),cost);

   // 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)*(/*- 10*grad */- 1*robot_->
        getJntVelocities()))
             + robot_->getGravity() + robot_->getCoriolis();
}
```

In the kdl_robot_test file the following has been changed:

```
double Kp = 400;
double Ko = 400;
// Cartesian space inverse dynamics control
tau = controller_.idCntr(des_pose, des_cart_vel, des_cart_acc,Kp, Ko, 2*sqrt(
    Kp), 2*sqrt(Ko),Error);
```

The orientation error function has been modified in order to use the quaternions instead of the angle axis convention

```
inline Eigen::Matrix<double,3,1> computeOrientationError(const Eigen::Matrix<
    double,3,3> &_R_d, const Eigen::Matrix<double,3,3> &_R_e)
{
     Eigen::Matrix<double,3,1> e_o;
     Eigen::Quaterniond q_e(_R_e);
     Eigen::Quaterniond q_d(_R_d);
     Eigen::Quaterniond q = q_d*q_e.inverse();
     e_o << q.x(), q.y(), q.z();
     return e_o;
}
```

4c Test the controller

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

The controller was tested in the operating space tuning the ko and kp values. After several trials, values of ko=400 and kp=400 were selected.This choice of gains is aimed at minimizing errors and achieving the most precise trajectory, particularly when optimizing trajectories with a cubic profile.

To test the things done so far, it is therefore necessary to start an instance of gazibo with the iiwa14 robot preloaded and then call up the kdl_robot_test programme modified earlier with the iiwa14.urdf file attached

```
roslaunch iiwa_gazebo iiwa_gazebo_effort.launch
rosrun kdl_ros_control kdl_robot_test /src/iiwa_stack/iiwa_description/urdf/
    iiwa14.urdf
```