

Report – Homework 2
Student: Raffaele Freschini

Control a manipulator to follow a trajectory

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

```
139  s_struct KDLPlanner::trapezoidal_vel(double time, double tc){
140
141      s_struct s_abs;
142
143      accDuration_ = tc;
144
145      double qc_ddot_ = 5/(std::pow(trajDuration_,2));
146
147      if(time <= accDuration_)
148      {
149          s_abs.s_ = 0 + 0.5*qc_ddot_*std::pow(time,2);
150          s_abs.s_dot_ = qc_ddot_*time;
151          s_abs.s_ddot_ = qc_ddot_;
152      }
153      else if(time <= trajDuration_-accDuration_)
154      {
155          s_abs.s_ = 0 + qc_ddot_*accDuration_*(time-accDuration_/2);
156          s_abs.s_dot_ = qc_ddot_*accDuration_;
157          s_abs.s_ddot_ = 0;
158      }
159      else
160      {
161          s_abs.s_ = 1 - 0.5*qc_ddot_*std::pow(trajDuration_-time,2);
162          s_abs.s_dot_ = qc_ddot_*(trajDuration_-time);
163          s_abs.s_ddot_ = -qc_ddot_;
164      }
165
166      return s_abs;
167
168  }
```

```
24  struct s_struct{
25      double s_=0;
26      double s_dot_=0;
27      double s_ddot_=0;
28  };
29
```

- b. Create a function named KDLPlanner::cubic_polynomial 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.

```
170  s_struct KDLPlanner::cubic_polynomial(double time){
171
172      s_struct s_abs;
173
174      //offline value
175      double a0 = 0;
176      double a1 = 0;
177      double a2 = 3/std::pow(trajDuration_,2);
178      double a3 = -2/std::pow(trajDuration_,3);
179
180      s_abs.s_ = a3*std::pow(time,3) + a2*std::pow(time,2) + a1*time + a0;
181      s_abs.s_dot_ = 3*a3*std::pow(time,2) + 2*a2*time + a1;
182      s_abs.s_ddot_ = 6*a3*time + 2*a2;
183
184      return s_abs;
185
186  }
187
```

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.

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

16 KDLPlanner::KDLPlanner(double _trajDuration, double _trajRadius, Eigen::Vector3d _trajInit)
17 {
18     trajDuration_ = _trajDuration;
19     trajInit_ = _trajInit;
20     trajRadius_ = _trajRadius;
21     trajEnd_ = _trajInit;
22 }
```

kdl_planner.cpp

- 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`.
- Do the same for the linear trajectory.

```
94 trajectory_point KDLPlanner::compute_trajectory(double time, char* path, char* vel_prof)
95 {
96     /* trapezoidal velocity profile with accDuration, acceleration time period and trajDuration, total duration.
97     time = current time
98     trajDuration_ = final time
99     accDuration_ = acceleration time
100     trajInit_ = trajectory initial point
101     trajEnd_ = trajectory final point */
102     trajectory_point traj;
103     s_struct curv_abs;
104     if(strcmp(vel_prof, "cubic")==0)
105     {
106         curv_abs = KDLPlanner::cubic_polynomial(time);
107     }
108     else if(strcmp(vel_prof, "trapez")==0)
109     {
110         curv_abs = KDLPlanner::trapezoidal_vel(time, accDuration_);
111     }
112     else
113     {
114         std::cout<<"ERRORE ABS"<<std::endl;
115     }
116     if(strcmp(path, "circle")==0)
117     {
118         traj.pos[0] = trajInit_[0];
119         traj.pos[1] = trajInit_[1] - trajRadius_*cos(2*M_PI*curv_abs.s);
120         traj.pos[2] = trajInit_[2] - trajRadius_*sin(2*M_PI*curv_abs.s);
121         traj.vel[0] = 0;
122         traj.vel[1] = 2*M_PI*trajRadius_*curv_abs.s_dot*sin(2*M_PI*curv_abs.s);
123         traj.vel[2] = -2*M_PI*trajRadius_*curv_abs.s_dot*cos(2*M_PI*curv_abs.s);
124         traj.acc[0] = 0;
125         traj.acc[1] = 2*M_PI*trajRadius_*curv_abs.s_ddot*sin(2*M_PI*curv_abs.s)+4*M_PI*M_PI*trajRadius_*std::pow(curv_abs.s_dot,2)*cos(2*M_PI*curv_abs.s);
126         traj.acc[2] = -2*M_PI*trajRadius_*curv_abs.s_ddot*cos(2*M_PI*curv_abs.s)+4*M_PI*M_PI*trajRadius_*std::pow(curv_abs.s_dot,2)*sin(2*M_PI*curv_abs.s);
127     }
128     else if(strcmp(path, "linear")==0)
129     {
130         traj.pos = trajInit_ + (curv_abs.s * (trajEnd_ - trajInit_));
131         traj.vel = curv_abs.s_dot * (trajEnd_ - trajInit_);
132         traj.acc = curv_abs.s_ddot * (trajEnd_ - trajInit_);
133     }
134     else
135     {
136         std::cout<<"ERRORE PATH"<<std::endl;
137     }
138     return traj;
139 }
```

For easier testing I decided to use only one `compute_trajectory` function and I modified the `robot_test.cpp` file so that I can decide what type of trajectory and velocity profile I want to use at runtime.

3. Test the four trajectories

- Modify your main file `kdl_robot_test.cpp` and test the four trajectories with the provided joint space inverse dynamics controller.

```
// Retrieve the first trajectory point
char path_type[7] = "test";
char vel_type[7] = "test";

// Plan trajectory
double traj_duration = 0.0, acc_duration = 0.0, t = 0.0, init_time_slot = 0, rad=0;
KDLPlanner planner(traj_duration,acc_duration,rad,init_position,end_position);

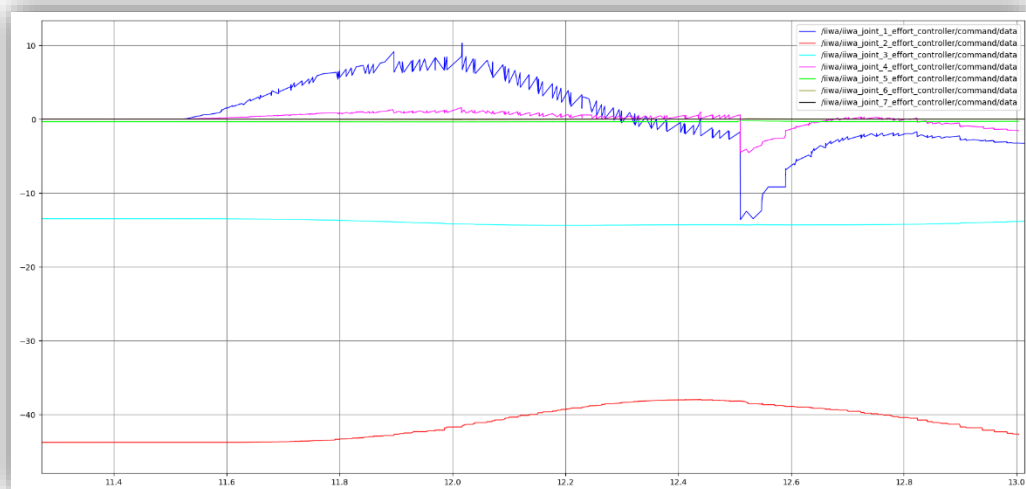
if(exit==1){
    traj_duration = 1.5; acc_duration = 0.5; t = 0.0; init_time_slot = 1.0;
    planner.set_all(traj_duration, acc_duration,0.0,init_position, end_position); // currently using trapezoidal velocity profile
    strcpy(path_type,"linear");
    strcpy(vel_type,"trapez");
}else if(exit==2){
    traj_duration = 1.5; acc_duration = 0.5; t = 0.0; init_time_slot = 1.0;
    planner.set_all(traj_duration, acc_duration,0.0,init_position, end_position); // currently using trapezoidal velocity profile
    strcpy(path_type,"linear");
    strcpy(vel_type,"cubic");
}else if(exit==3){
    traj_duration = 1.5; acc_duration = 0.5; rad = 0.2; t = 0.0; init_time_slot = 1.0;
    planner.set_all(traj_duration, acc_duration, rad, init_position, init_position); // currently using trapezoidal velocity profile + circular
    strcpy(path_type,"circle");
    strcpy(vel_type,"trapez");
}else if(exit==4){
    traj_duration = 1.5; rad = 0.2; t = 0.0; init_time_slot = 1.0;
    planner.set_all(traj_duration, 0.0,rad, init_position, init_position); // currently using cubic polynomial velocity profile + circ
    strcpy(path_type,"circle");
    strcpy(vel_type,"cubic");
}
```

```
trajectory_point p = planner.compute_trajectory(t,path_type,vel_type);

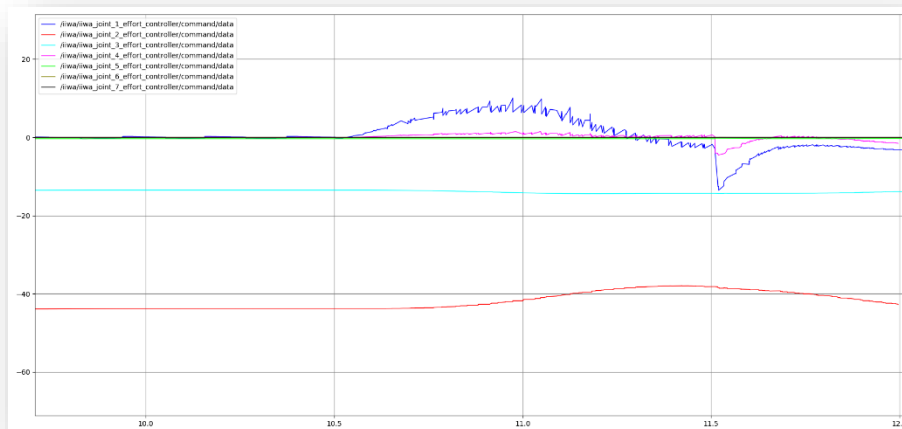
// Gains
double Kp = 50, Kd = sqrt(Kp), Ko=Kp;
//double Kp = 50, Kd = 10;

switch (exit){
    case 1: //Lineare e trapezoidale
        Kp=50;
        Kd=10;
        break;
    case 2: //Lineare e Cubica
        Kp=50;
        Kd=10;
        break;
    case 3: //Circolare e trapezoidale
        Kp=70;
        Kd=sqrt(Kp);
        break;
    case 4: //Circolare e cubica
        Kp=70;
        Kd=sqrt(Kp);
        break;
    default:
        Kp=50;
        Kd=sqrt(Kp);
        break;
}
```

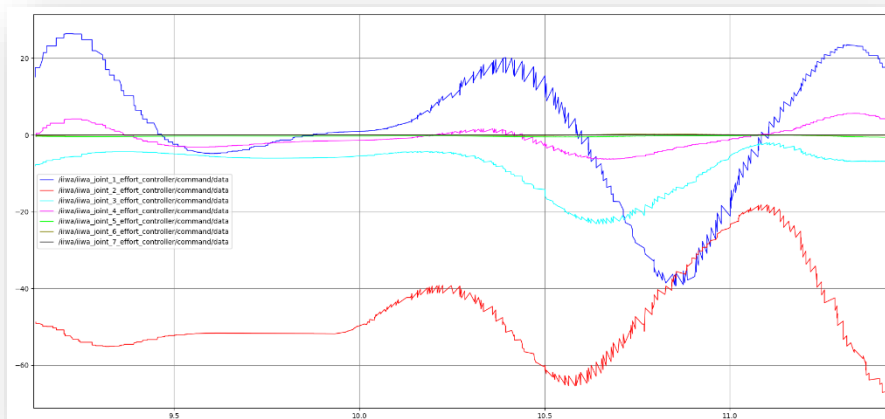
- Plot the torques sent to the manipulator and tune appropriately the control gains K_p and K_d until you reach a satisfactorily smooth behavior.



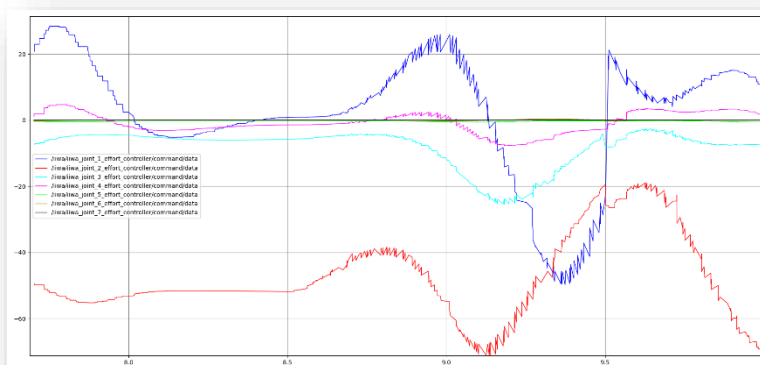
linear_cubic



linear_trapezoidal

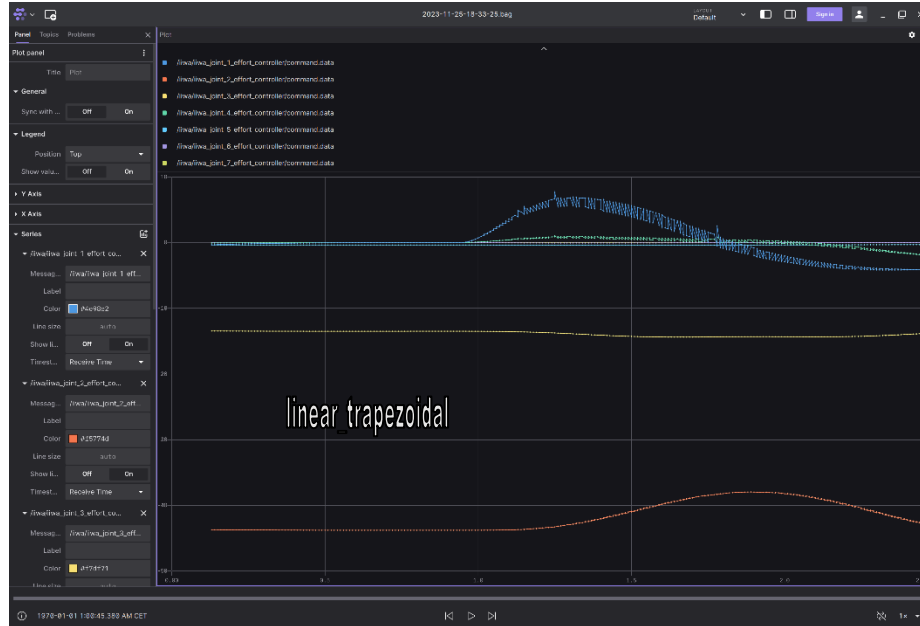


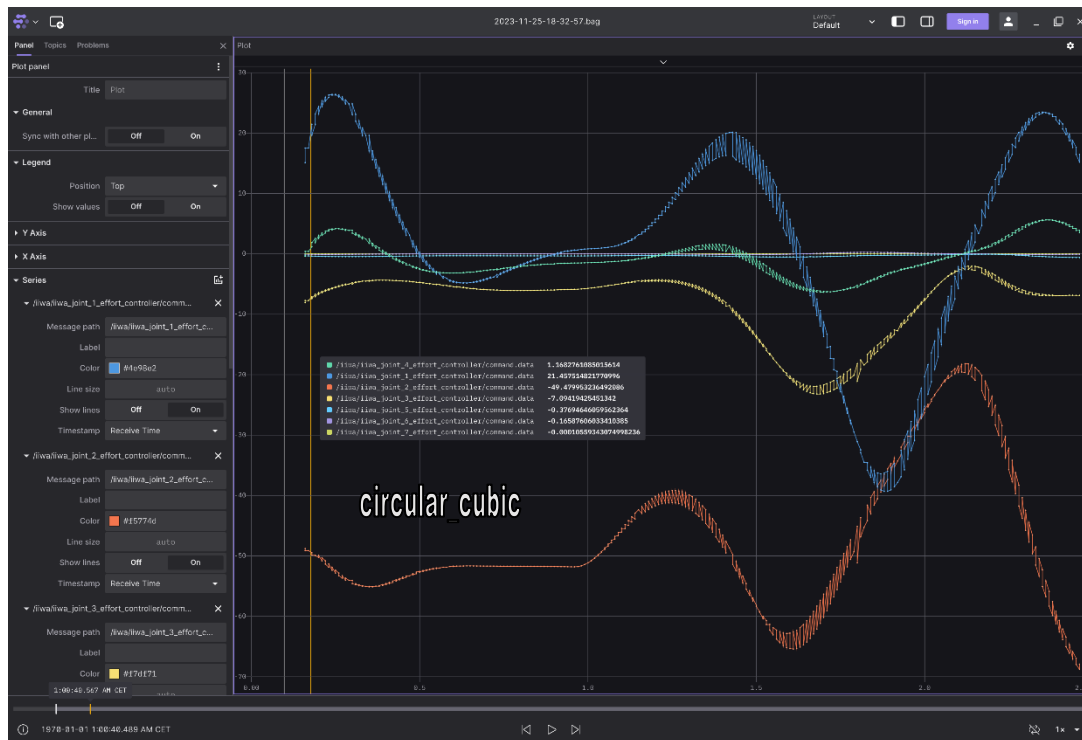
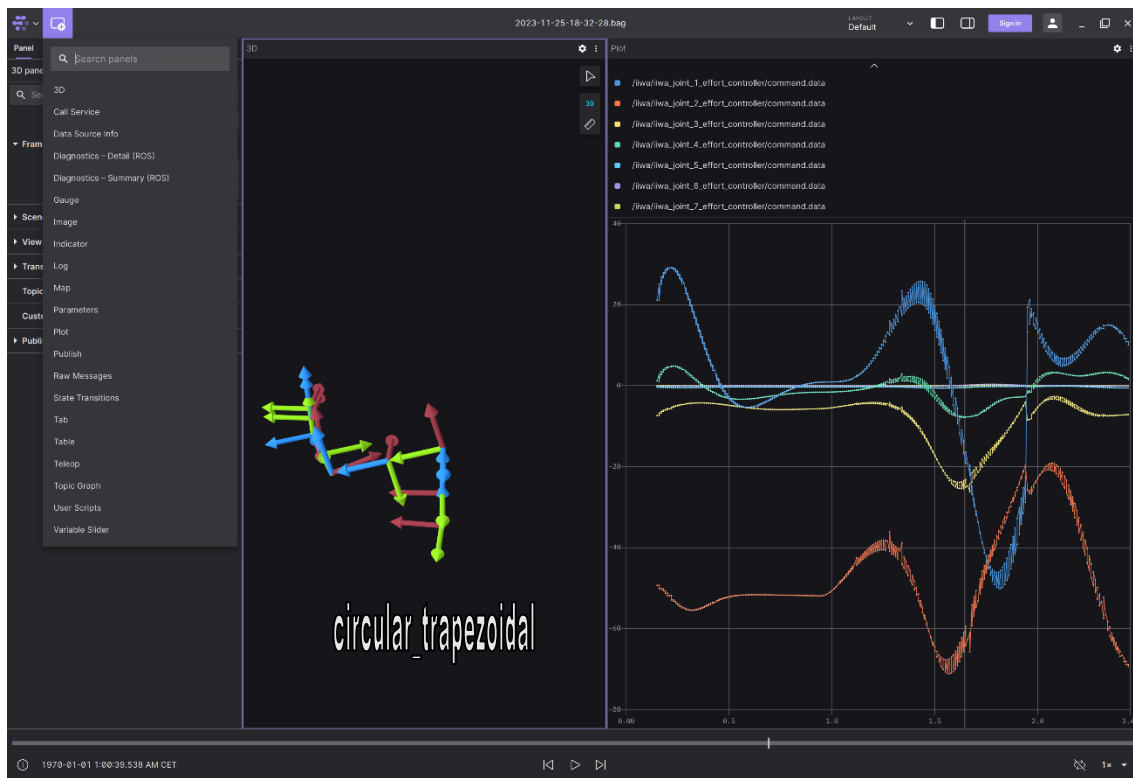
circular_cubic



circular_trapezoidal

Since the visualization capabilities of rqt are quite limited, we also plotted the torques using Foxglove.





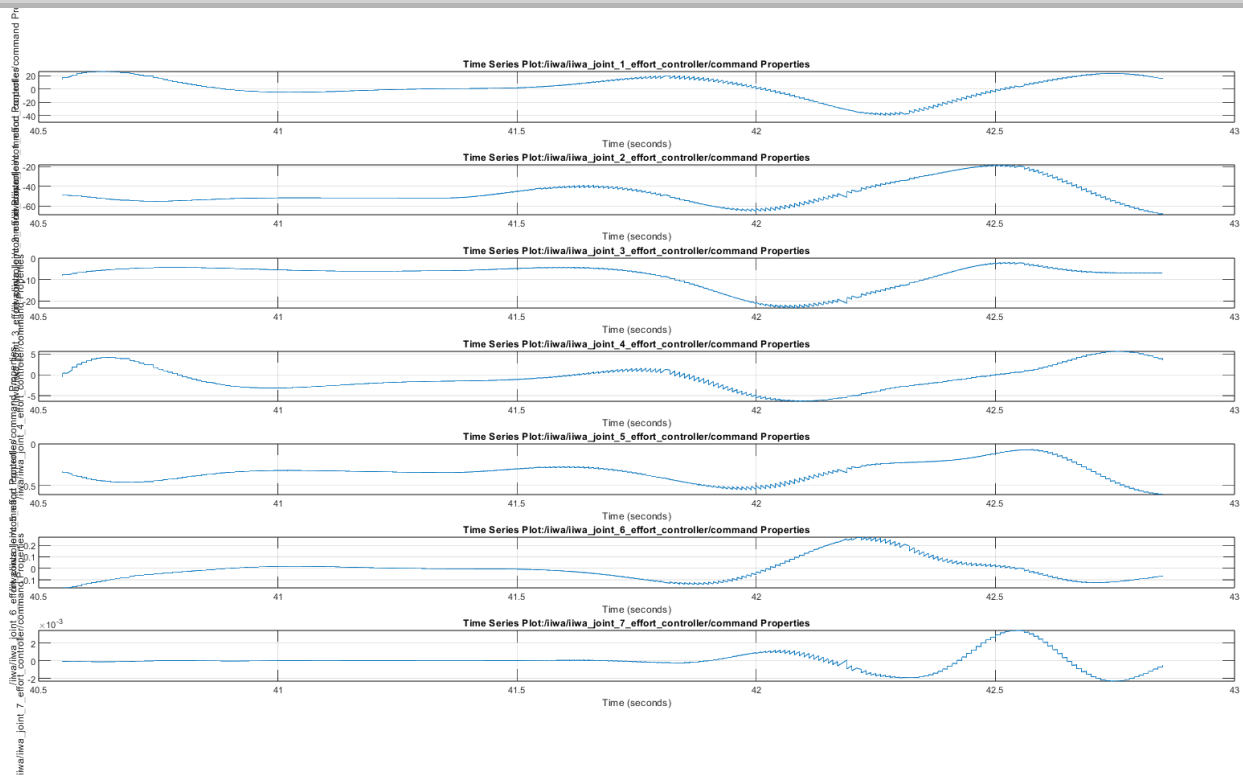
- c. **Optional:** Save the joint torque command topics in a bag file and plot it using MATLAB.

```
bagMsgs = rosbagreader("4.bag")

bagSelection1 = select(bagMsgs,'Topic','/iiwa/iiwa_joint_1_effort_controller/command');
bagSelection2 = select(bagMsgs,'Topic','/iiwa/iiwa_joint_2_effort_controller/command');
bagSelection3 = select(bagMsgs,'Topic','/iiwa/iiwa_joint_3_effort_controller/command');
bagSelection4 = select(bagMsgs,'Topic','/iiwa/iiwa_joint_4_effort_controller/command');
bagSelection5 = select(bagMsgs,'Topic','/iiwa/iiwa_joint_5_effort_controller/command');
bagSelection6 = select(bagMsgs,'Topic','/iiwa/iiwa_joint_6_effort_controller/command');
bagSelection7 = select(bagMsgs,'Topic','/iiwa/iiwa_joint_7_effort_controller/command');

ts1 = timeseries(bagSelection1);
ts2 = timeseries(bagSelection2);
ts3 = timeseries(bagSelection3);
ts4 = timeseries(bagSelection4);
ts5 = timeseries(bagSelection5);
ts6 = timeseries(bagSelection6);
ts7 = timeseries(bagSelection7);
```

```
subplot(7,1,1)
plot(ts1)
grid on
subplot(7,1,2)
plot(ts2)
grid on
subplot(7,1,3)
plot(ts3)
grid on
subplot(7,1,4)
plot(ts4)
grid on
subplot(7,1,5)
plot(ts5)
grid on
subplot(7,1,6)
plot(ts6)
grid on
subplot(7,1,7)
plot(ts7)
grid on
```



4. Develop an inverse dynamics operational space controller

- Into the `kdl_contorl.cpp` file, fill the empty overlaid `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::Framepose`, 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.
- 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 M and the Jacobian and its time derivative, compute the linear e_p and the angular e_o errors (some functions are provided into the `include/utlis.h` file), finally compute your inverse dynamics control law following the equation

$$\tau = By + n, \quad y = J^{\dagger}(\ddot{x}_d + K_d \dot{\tilde{x}} + K_p \tilde{x} - \dot{J}_A \dot{q})$$

```

27  Eigen::VectorXd KDLController::idCntr(KDL::Frame &_desPos,
28                                          KDL::Twist &_desVel,
29                                          KDL::Twist &_desAcc,
30                                          double _Kpp, double _Kpo,
31                                          double _Kdp, double _Kdo)
32  {
33      // calculate gain matrices
34      Eigen::Matrix<double,6,6> Kp, Kd;
35      Kp=Eigen::MatrixXd::Zero(6,6);
36      Kd=Eigen::MatrixXd::Zero(6,6);
37      Kp.block(0,0,3,3) = _Kpp*Eigen::Matrix3d::Identity();
38      Kp.block(3,3,3,3) = _Kpo*Eigen::Matrix3d::Identity();
39      Kd.block(0,0,3,3) = _Kdp*Eigen::Matrix3d::Identity();
40      Kd.block(3,3,3,3) = _Kdo*Eigen::Matrix3d::Identity();
41
42      // read current state
43      KDL::Jacobian JEE=robot->getEEJacobian();
44      Eigen::Matrix<double,6,7> J = toEigen(JEE);
45      Eigen::Matrix<double,7,7> I = Eigen::Matrix<double,7,7>::Identity();
46      Eigen::Matrix<double,7,7> M = robot->getJsim();
47      Eigen::Matrix<double,7,6> Jpinv = weightedPseudoInverse(M,J);
48      //Eigen::Matrix<double,7,6> Jpinv = pseudoInverse(J);
49
50      // position
51      KDL::Frame cart_pose = robot->getEEFrame();
52      Eigen::Vector3d p_d(_desPos.p.data);
53      Eigen::Vector3d p_e(cart_pose.p.data);
54      Eigen::Matrix<double,3,3,Eigen::RowMajor> R_d(_desPos.M.data);
55      Eigen::Matrix<double,3,3,Eigen::RowMajor> R_e(cart_pose.M.data);
56      R_d = matrixOrthonormalization(R_d);
57      R_e = matrixOrthonormalization(R_e);
58
59      switch (exit){
60      case 1: //lineare e trapezoidale
61          Kp=100;
62          Ko=100;
63          break;
64      case 2: //lineare e cubica
65          Kp=100;
66          Ko=100;
67          break;
68      case 3: //Circolare e trapezoidale
69          Kp=60;
70          Ko=20;
71          break;
72      case 4: //Circolare e cubica
73          Kp=30;
74          Ko=10;
75          break;
76      default:
77          Kp=1000;
78          Ko=1000;
79          break;
80      }

```

```

58
59 // velocity
60 KDL::Twist cart_twist = robot_->getEEVelocity();
61 Eigen::Vector3d dot_p_d(_desVel.vel.data);
62 Eigen::Vector3d dot_p_e(cart_twist.vel.data);
63 Eigen::Vector3d omega_d(_desVel.rot.data);
64 Eigen::Vector3d omega_e(cart_twist.rot.data);
65
66 // acceleration
67 Eigen::Matrix<double,6,1> dot_dot_x_d;
68 Eigen::Matrix<double,3,1> dot_dot_p_d(_desAcc.vel.data);
69 Eigen::Matrix<double,3,1> dot_dot_r_d(_desAcc.rot.data);
70
71 // compute linear errors
72 Eigen::Matrix<double,3,1> e_p = computeLinearError(p_d,p_e);
73 Eigen::Matrix<double,3,1> dot_e_p = computeLinearError(dot_p_d,dot_p_e);
74
75 // shared control
76 // Eigen::Vector3d lin_acc;
77 // lin_acc << _desAcc.vel.x(), _desAcc.vel.y(), _desAcc.vel.z(); //use desired acceleration
78 // lin_acc << dot_dot_p_d + _Kdp*(dot_e_p) + _Kpp*(e_p); // assuming no friction no loads
79 // Eigen::Matrix<double,3,3> R_sh = shCntr(lin_acc);
80
81 // compute orientation errors
82 Eigen::Matrix<double,3,1> e_o = computeOrientationError(R_d,R_e);
83 Eigen::Matrix<double,3,1> dot_e_o = computeOrientationVelocityError(omega_d,
84                                                                    omega_e,
85                                                                    R_d,
86                                                                    R_e);
87 Eigen::Matrix<double,6,1> x_tilde;
88 Eigen::Matrix<double,6,1> dot_x_tilde;
89 x_tilde << e_p, e_o;
90 dot_x_tilde << dot_e_p, -omega_e;//dot_e_o;
91 dot_dot_x_d << dot_dot_p_d, dot_dot_r_d;

```

```

93 // null space control
94 double cost;
95 Eigen::VectorXd grad = gradientJointLimits(robot_->getJntValues(),robot_->getJntLimits(),cost);
96
97 // std::cout << "-----" << std::endl;
98 // std::cout << "p_d: " << std::endl << p_d << std::endl;
99 // std::cout << "p_e: " << std::endl << p_e << std::endl;
100 // std::cout << "dot_p_d: " << std::endl << dot_p_d << std::endl;
101 // std::cout << "dot_p_e: " << std::endl << dot_p_e << std::endl;
102 // std::cout << "R_sh*R_d: " << std::endl << R_d << std::endl;
103 // std::cout << "R_e: " << std::endl << R_e << std::endl;
104 // std::cout << "omega_d: " << std::endl << omega_d << std::endl;
105 // std::cout << "omega_e: " << std::endl << omega_e << std::endl;
106 // std::cout << "x_tilde: " << std::endl << x_tilde << std::endl;
107 // std::cout << "dot_x_tilde: " << std::endl << dot_x_tilde << std::endl;
108 // std::cout << "jacobian: " << std::endl << toEigen(robot_->getEEJacobian()) << std::endl;
109 // std::cout << "jpinv: " << std::endl << Jpinv << std::endl;
110 // std::cout << "jsim: " << std::endl << robot_->getJsim() << std::endl;
111 // std::cout << "c: " << std::endl << robot_->getCoriolis().transpose() << std::endl;
112 // std::cout << "g: " << std::endl << robot_->getGravity().transpose() << std::endl;
113 // std::cout << "q: " << std::endl << robot_->getJntValues() << std::endl;
114 // std::cout << "Jac Dot qDot: " << std::endl << toEigen(robot_->getEEJacDotqDot()) << std::endl;
115 // std::cout << "qdot: " << std::endl << robot_->getJntVelocities() << std::endl;
116 // std::cout << "Jnt lmt cost: " << std::endl << cost << std::endl;
117 // std::cout << "Jnt lmt gradient: " << std::endl << grad.transpose() << std::endl;
118 // std::cout << "-----" << std::endl;
119
120 // inverse dynamics
121 Eigen::Matrix<double,6,1> y;
122 y << dot_dot_x_d - toEigen(robot_->getEEJacDotqDot())*robot_->getJntVelocities() + Kd*dot_x_tilde + Kp*x_tilde;
123
124 return M * (Jpinv*y + (I-Jpinv*J)*(- 10*grad/ - 1*robot_->getJntVelocities()))
125         + robot_->getGravity() + robot_->getCoriolis();
126 }

```

```

// Eigen::VectorXd KDLRobot::getEEJacDotqDot()
// {
//     return s_J_dot_ee_.data;
// }

KDL::Jacobian KDLRobot::getEEJacDotqDot()
{
    return s_J_dot_ee_;
}

```

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

