Report – Homework 2

Student: **Nicola Caliendo**

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 tc as double arguments and returns three double variables s, $\dot{s}$ and $\ddot{s}$ that represent the curvilinear abscissa of your trajectory.

```
s_struct KDLPlanner::trapezoidal_vel(double time, double tc){

    s_struct s_abs;

    accDuration_= tc;

    double qc_ddot_ = 5/(std::pow(trajDuration_,2));

    if(time <= accDuration_)
    {
      s_abs.s_ = 0 + 0.5*qc_ddot_*std::pow(time,2);
      s_abs.s_dot_ = qc_ddot_*time;
      s_abs.s_ddot_ = qc_ddot_;
    }
    else if(time <= trajDuration_-accDuration_)
    {
      s_abs.s_ = 0 + qc_ddot_*accDuration_*(time-accDuration_/2);
      s_abs.s_dot_ = qc_ddot_*accDuration_;
      s_abs.s_ddot_ = 0;
    }
    else
    {
      s_abs.s_ = 1 - 0.5*qc_ddot_*std::pow(trajDuration_-time,2);
      s_abs.s_dot_ = qc_ddot_*(trajDuration_-time);
      s_abs.s_ddot_ = -qc_ddot_;
    }

    return s_abs;

}
```

```
struct s_struct{
    double s_=0;
    double s_dot_=0;
    double s_ddot_=0;
};
```

   b. Create a function named KDLPlanner::cubic_polinomial that creates the cubic polynomial curvilinear abscissa for your trajectory. The function takes as argument a double t representing time and returns three double s, $\dot{s}$ and $\ddot{s}$ that represent the curvilinear abscissa of your trajectory.

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

  s_struct s_abs;

  //offline value
  double a0 = 0;
  double a1 = 0;
  double a2 = 3/std::pow(trajDuration_,2);
  double a3 = -2/std::pow(trajDuration_,3);

  s_abs.s_ = a3*std::pow(time,3) + a2*std::pow(time,2) + a1*time + a0;
  s_abs.s_dot_ = 3*a3*std::pow(time,2) + 2*a2*time + a1;
  s_abs.s_ddot_ = 6*a3*time + 2*a2;

  return s_abs;

}
```

2. Create circular trajectories for your robot

   a. 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);
```

kdl_planner.cpp

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

   b. Create the positional path as function of s(t) directly in the function KDLPlanner::compute_trajectory: first, call the cubic_polinomial function to retrieve s and its derivatives from t; then fill in the trajectory_point fields traj.pos, traj.vel, and traj.acc.

   c. Do the same for the linear trajectory.

```
trajectory_point KDLPlanner::compute_trajectory(double time ,char* path, char* vel_prof)
{
  /* trapezoidal velocity profile with accDuration_ acceleration time period and trajDuration_ total duration.
     time = current time
     trajDuration_  = final time
     accDuration_   = acceleration time
     trajInit_ = trajectory initial point
     trajEnd_  = trajectory final point */

  trajectory_point traj;

  s_struct curv_abs_;

  if(strcmp(vel_prof,"cubic")==0)
    curv_abs_ = KDLPlanner::cubic_polinomial(time);
  else if(strcmp(vel_prof,"trapez")==0)
    curv_abs_ = KDLPlanner::trapezoidal_vel(time,accDuration_);
  else
    std::cout<<"ERRORE ABS"<<std::endl;

  if(strcmp(path,"circle")==0){
    traj.pos[0] = trajInit_[0];
    traj.pos[1] = trajInit_[1] - trajRadius_*cos(2*M_PI*curv_abs_.s_);
    traj.pos[2] = trajInit_[2] - trajRadius_*sin(2*M_PI*curv_abs_.s_);

    traj.vel[0] = 0;
    traj.vel[1] = 2*M_PI*trajRadius_*curv_abs_.s_dot_*sin(2*M_PI*curv_abs_.s_);
    traj.vel[2] = -2*M_PI*trajRadius_*curv_abs_.s_dot_*cos(2*M_PI*curv_abs_.s_);

    traj.acc[0] = 0;
    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_);
    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_);
  }
  else if(strcmp(path,"linear")==0){
    traj.pos = trajInit_ + (curv_abs_.s_ * (trajEnd_ - trajInit_));
    traj.vel = curv_abs_.s_dot_ * (trajEnd_ - trajInit_);
    traj.acc = curv_abs_.s_ddot_ * (trajEnd_ - trajInit_);
  }else{
    std::cout<<"ERRORE PATH"<<std::endl;
  }
  return traj;

}
```

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

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

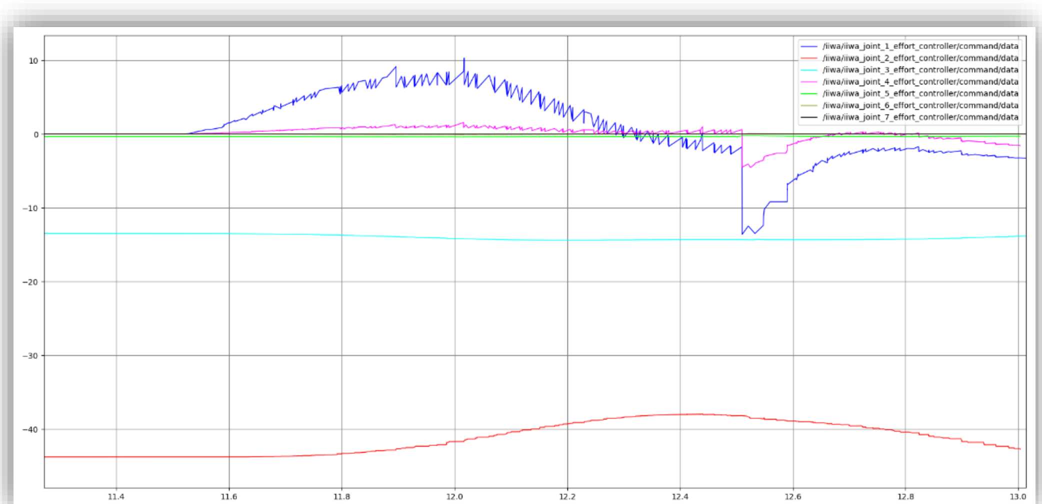
robot_test.cpp

```cpp
// 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 + circolar
    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 polinomial velocity profile + circ
    strcpy(path_type_,"circle");
    strcpy(vel_type_,"cubic");
}
```

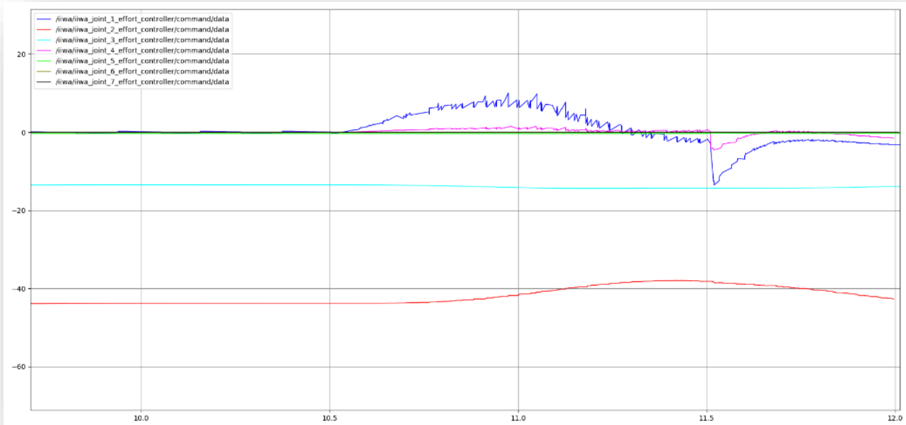
robot_test.cpp

```cpp
    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=75;
            Kd=sqrt(Kp);
            break;
        case 4:     //Circolare e cubica
            Kp=70;
            Kd=sqrt(Kp);
            break;
        default:
            Kp=50;
            Kd=sqrt(Kp);
            break;
    }
```

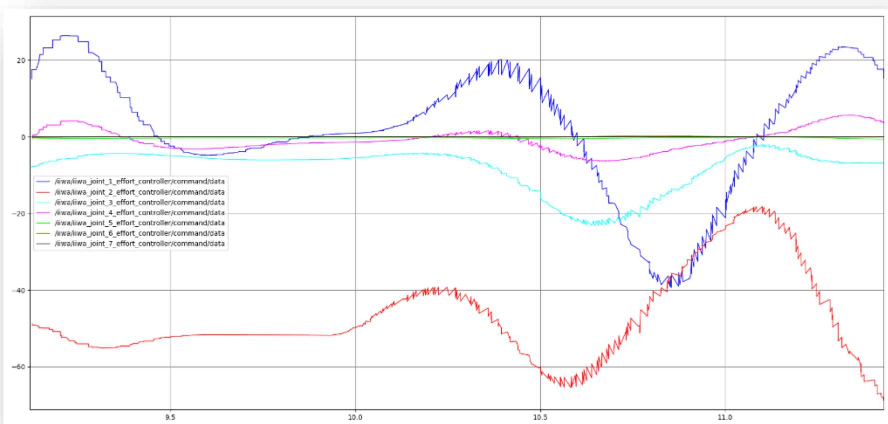
b. Plot the torques sent to the manipulator and tune appropriately the control gains Kp and Kd until you reach a satisfactorily smooth behavior.



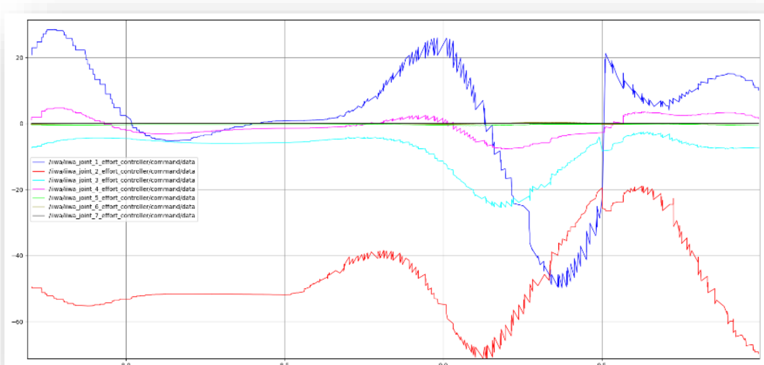
linear_cubic



linear_trapezoidal

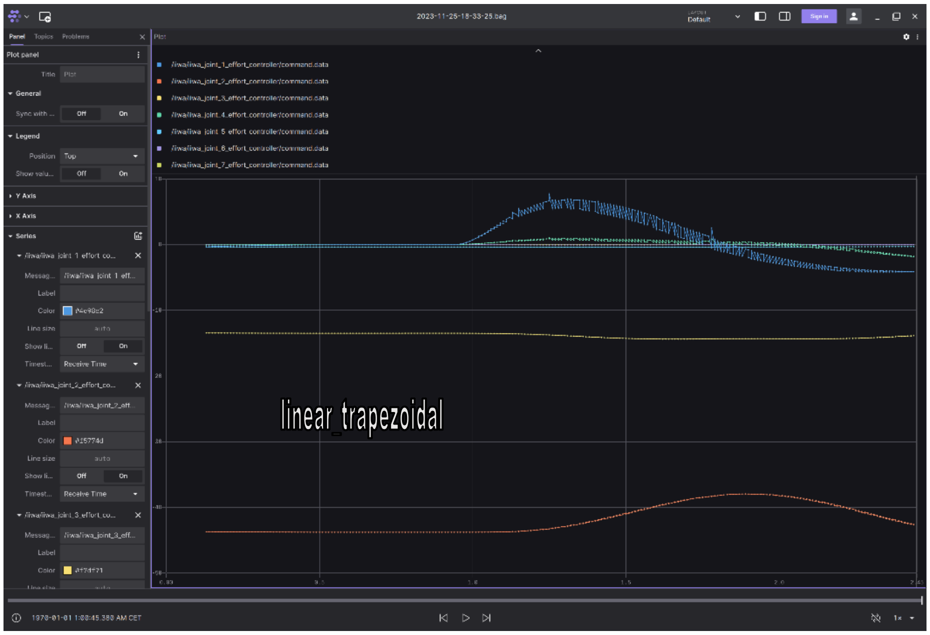


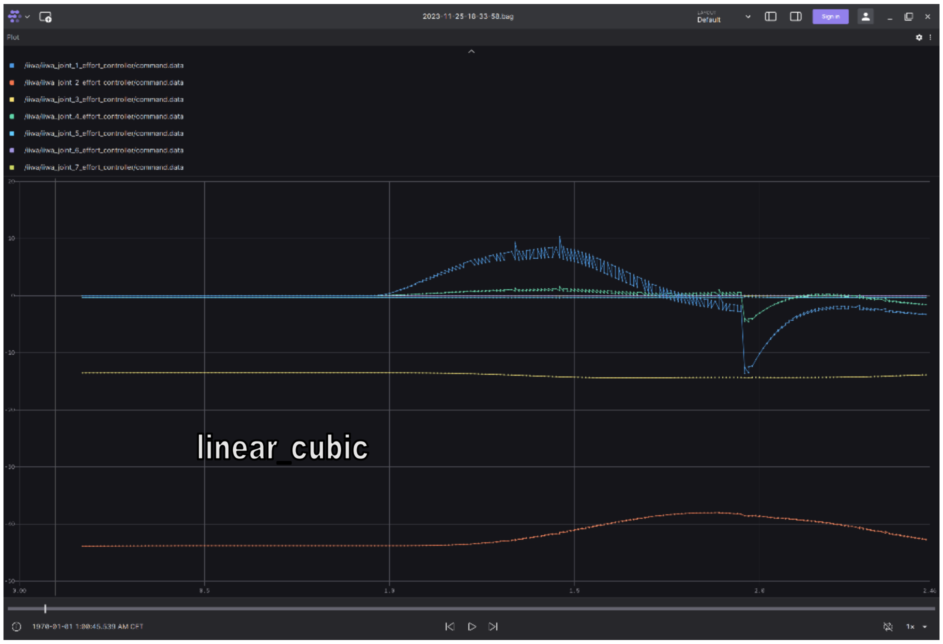
circular_cubic

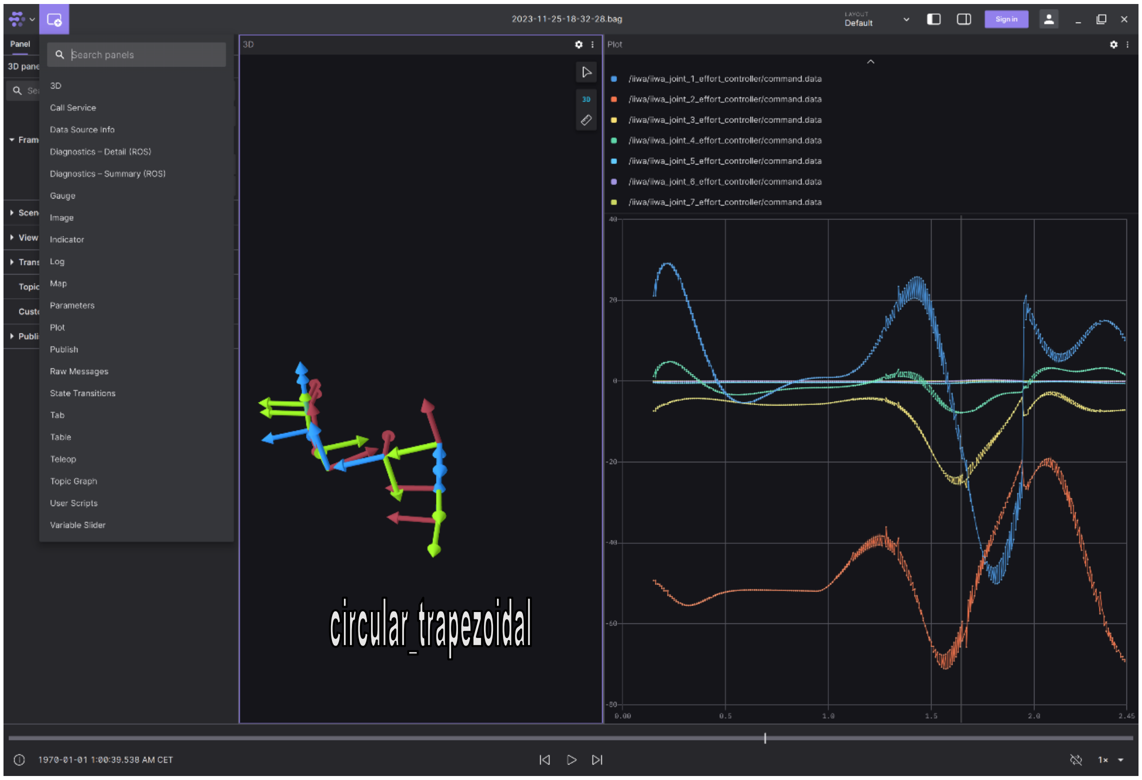


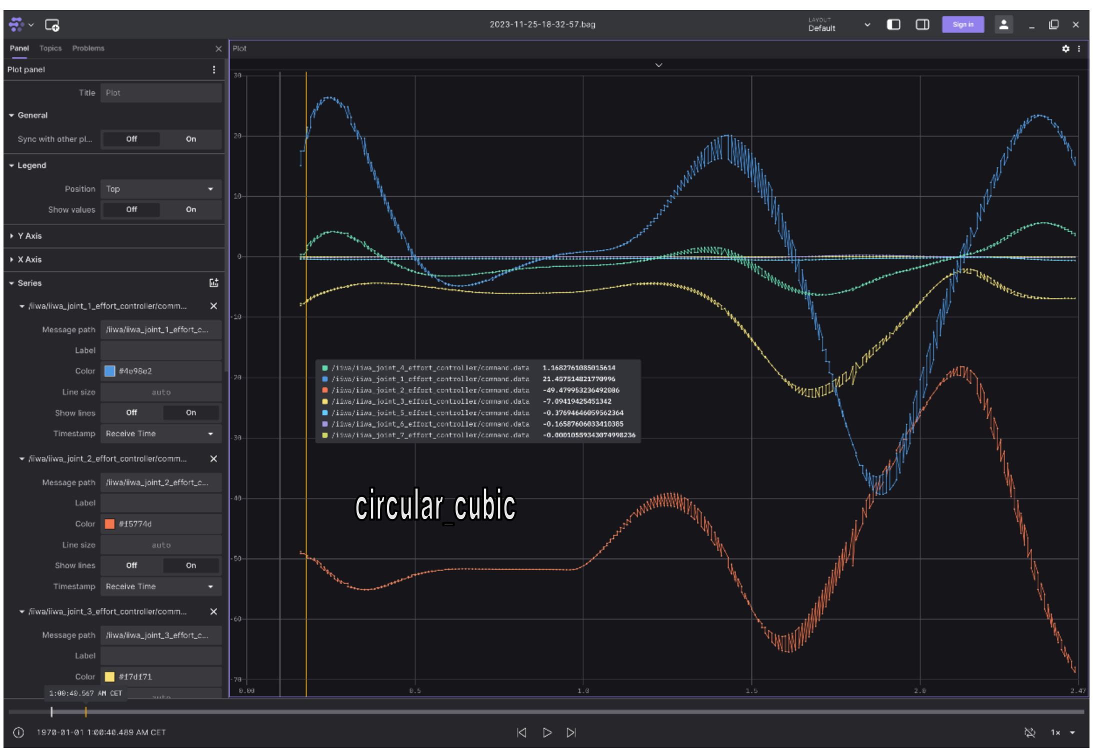
circular_trapezoidal

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





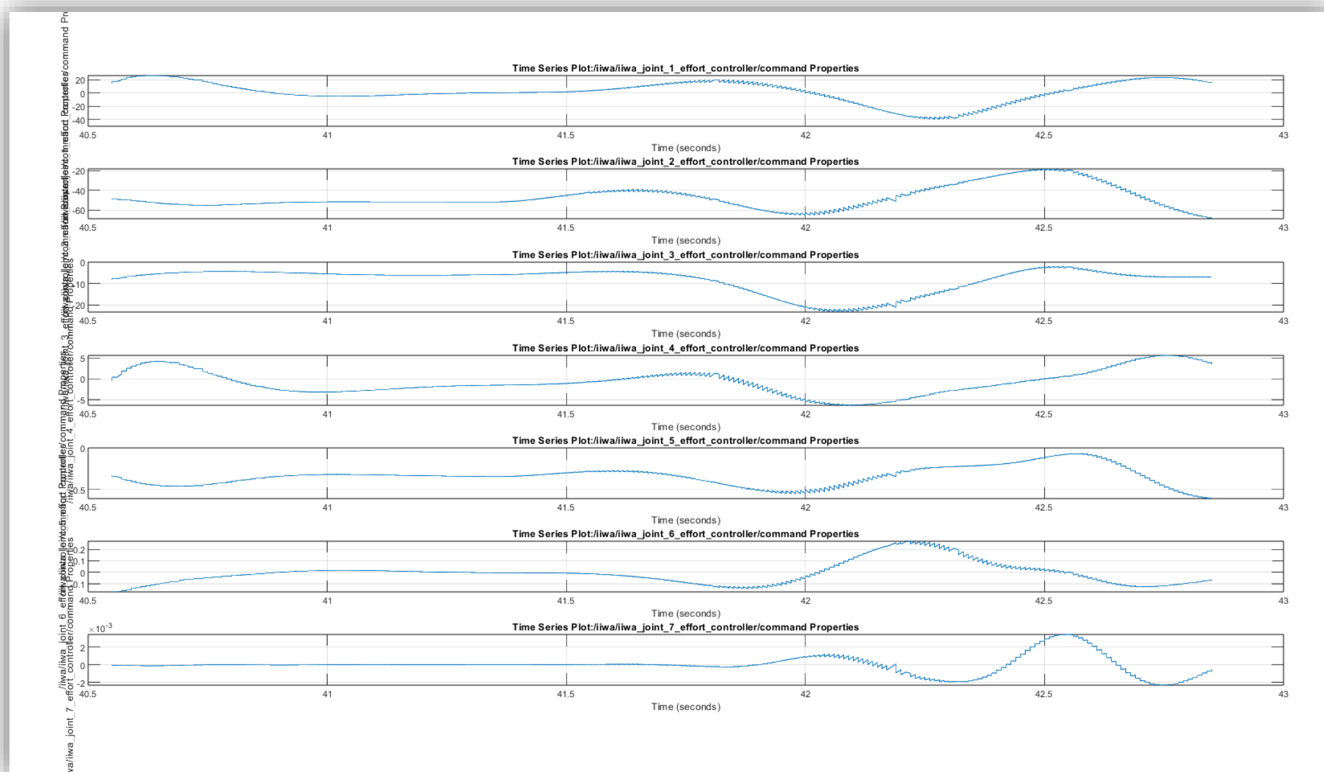

2023-11-25-18-32-28.bag
Search panels
3D
Call Service
Data Source Info
Diagnostics – Detail (ROS)
Diagnostics – Summary (ROS)
Gauge
Image
Indicator
Log
Map
Parameters
Plot
Publish
Raw Messages
State Transitions
Tab
Table
Teleop
Topic Graph
User Scripts
Variable Slider
/iiwa/iiwa_joint_1_effort_controller/command.data
/iiwa/iiwa_joint_2_effort_controller/command.data
/iiwa/iiwa_joint_3_effort_controller/command.data
/iiwa/iiwa_joint_4_effort_controller/command.data
/iiwa/iiwa_joint_5_effort_controller/command.data
/iiwa/iiwa_joint_6_effort_controller/command.data
/iiwa/iiwa_joint_7_effort_controller/command.data
circular_trapezoidal
1970-01-01 1:00:39.538 AM CET


2023-11-25-18-32-57.bag
Plot panel
Title
General
Legend
Position
Show values
Y Axis
X Axis
Series
/iiwa/iiwa_joint_1_effort_controller/comm...
/iiwa/iiwa_joint_2_effort_controller/comm...
/iiwa/iiwa_joint_3_effort_controller/comm...
circular_cubic
1970-01-01 1:00:40.489 AM CET

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

   a. 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::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.

   b. 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/utils.h file), finally compute your inverse dynamics control law following the equation

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

```cpp
Eigen::VectorXd KDLController::idCntr(KDL::Frame &_desPos,
                                      KDL::Twist &_desVel,
                                      KDL::Twist &_desAcc,
                                      double _Kpp, double _Kpo,
                                      double _Kdp, double _Kdo)
{
    // 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();
    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
     KDL::Jacobian JEE=robot_->getEEJacobian();
    Eigen::Matrix<double,6,7> J = toEigen(JEE);
    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
    KDL::Frame cart_pose = robot_->getEEFrame();
    Eigen::Vector3d p_d(_desPos.p.data);
    Eigen::Vector3d p_e(cart_pose.p.data);
    Eigen::Matrix<double,3,3,Eigen::RowMajor> R_d(_desPos.M.data);
    Eigen::Matrix<double,3,3,Eigen::RowMajor> R_e(cart_pose.M.data);
    R_d = matrixOrthonormalization(R_d);
    R_e = matrixOrthonormalization(R_e);
```

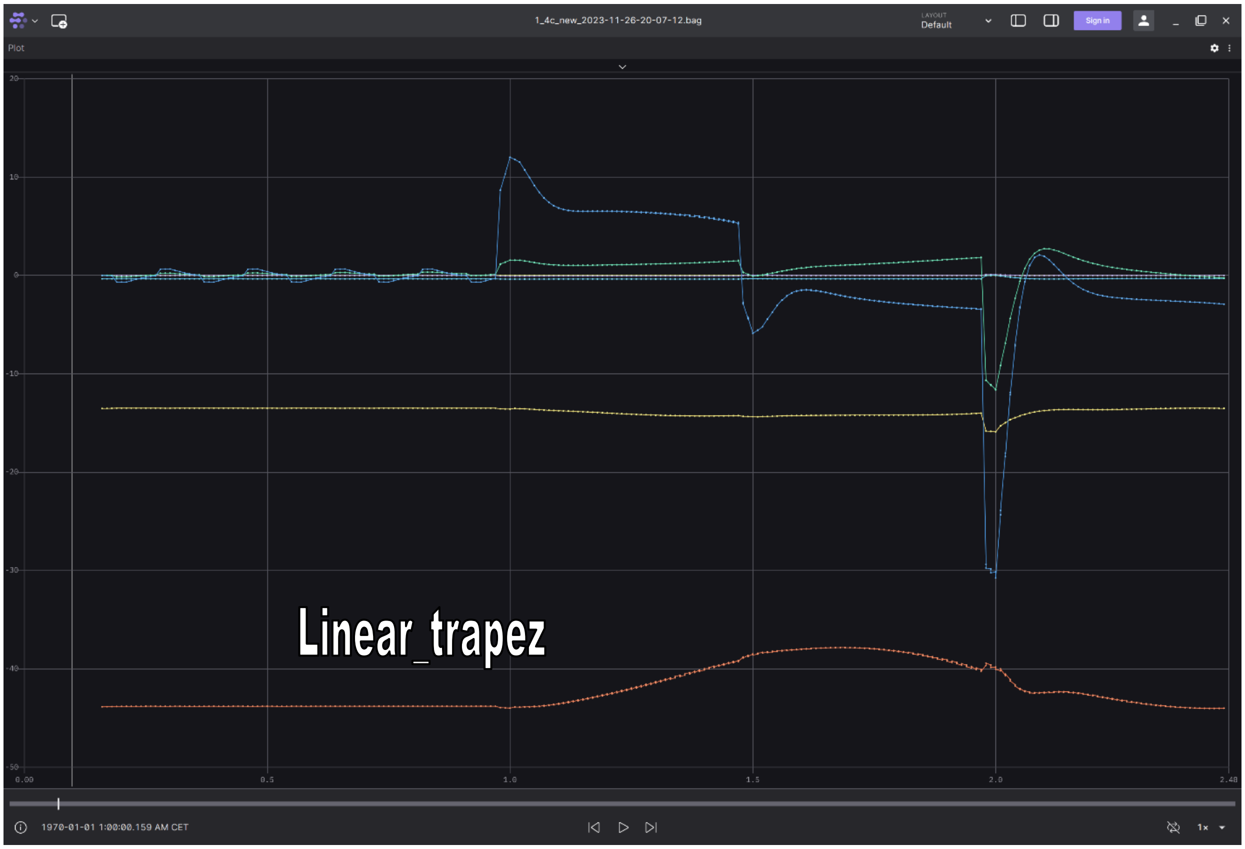
```cpp
switch (exit){
    case 1:      //lineare e trapezoidale
        Kp=100;
        Ko=100;
        break;
    case 2:      //lineare e Cubica
        Kp=100;
        Ko=100;
        break;
    case 3:      //Circolare e trapezoidale
        Kp=60;
        Ko=20;
        break;
    case 4:      //Circolare e cubica
        Kp=30;
        Ko=10;
        break;
    default:
        Kp=1000;
        Ko=1000;
        break;
}
```

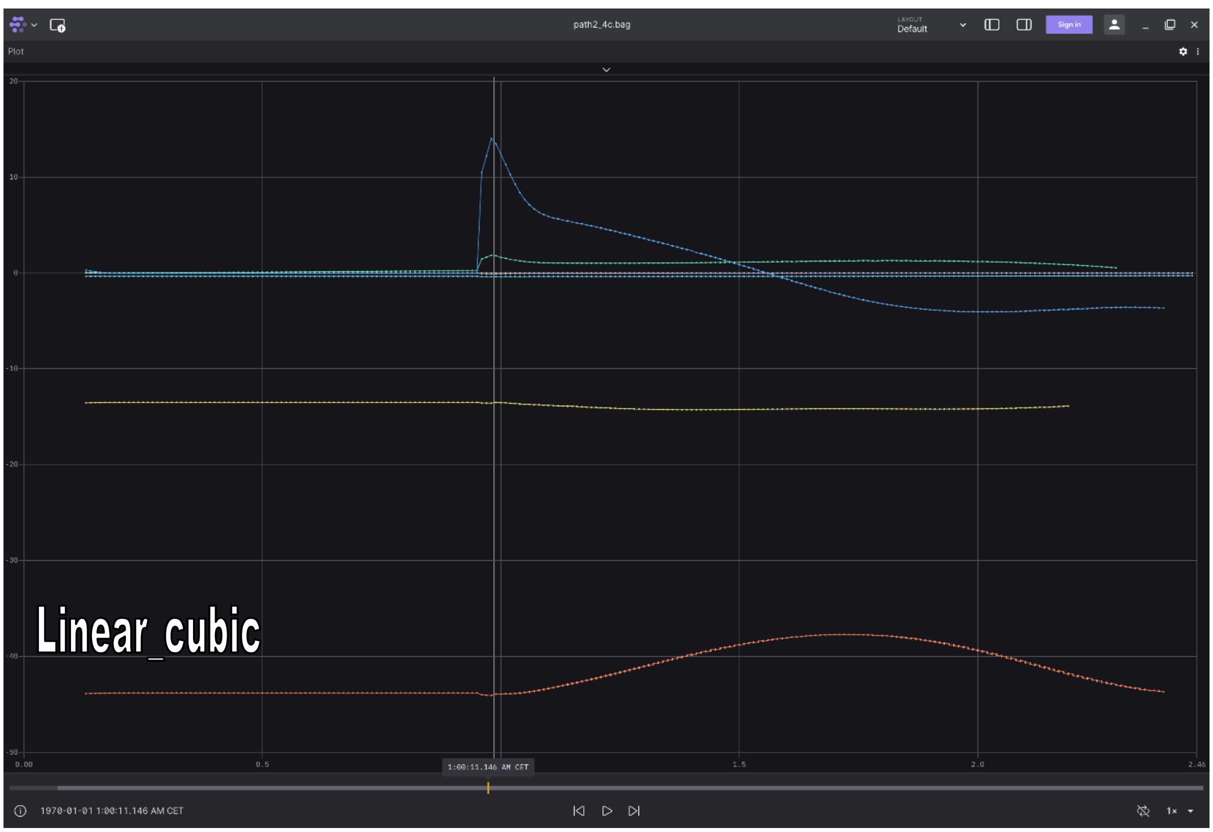
```cpp
    // velocity
    KDL::Twist cart_twist = robot_->getEEVelocity();
    Eigen::Vector3d dot_p_d(_desVel.vel.data);
    Eigen::Vector3d dot_p_e(cart_twist.vel.data);
    Eigen::Vector3d omega_d(_desVel.rot.data);
    Eigen::Vector3d omega_e(cart_twist.rot.data);

    // 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);

    // shared control
    // Eigen::Vector3d lin_acc;
    // lin_acc << _desAcc.vel.x(), _desAcc.vel.y(), _desAcc.vel.z(); //use desired acceleration
    // lin_acc << dot_dot_p_d + _Kdp*(dot_e_p) + _Kpp*(e_p); // assuming no friction no loads
    // Eigen::Matrix<double,3,3> R_sh = shCntr(lin_acc);

    // 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;
    dot_x_tilde << dot_e_p, -omega_e;//dot_e_o;
    dot_dot_x_d << dot_dot_p_d, dot_dot_r_d;
```

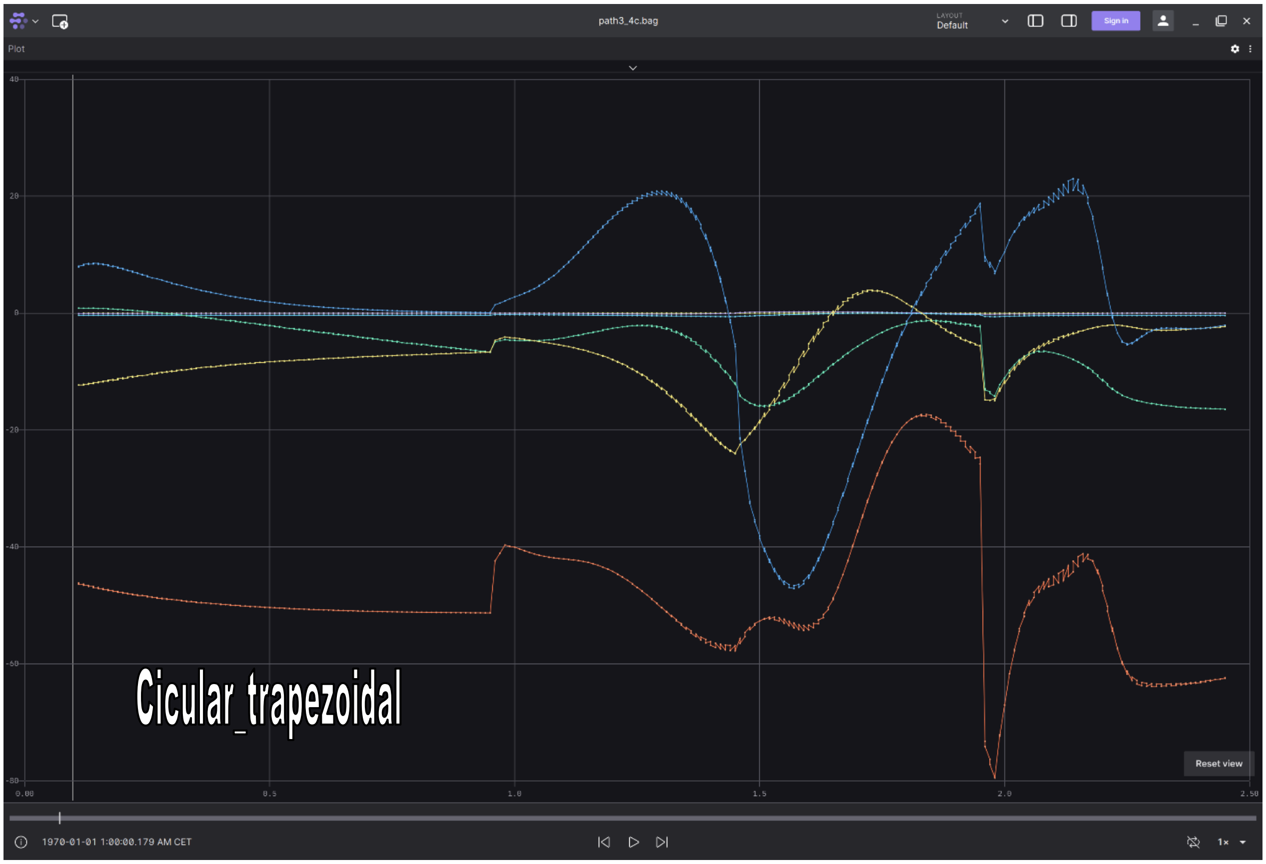
```cpp
    // null space control
    double cost;
    Eigen::VectorXd grad = gradientJointLimits(robot_->getJntValues(),robot_->getJntLimits(),cost);

//    std::cout << "--------------------" << std::endl;
//    std::cout << "p_d: " << std::endl << p_d << std::endl;
//    std::cout << "p_e: " << std::endl << p_e << std::endl;
//    std::cout << "dot_p_d: " << std::endl << dot_p_d << std::endl;
//    std::cout << "dot_p_e: " << std::endl << dot_p_e << std::endl;
//    std::cout << "R_sh*R_d: " << std::endl << R_d << std::endl;
//    std::cout << "R_e: " << std::endl << R_e << std::endl;
//    std::cout << "omega_d: " << std::endl << omega_d << std::endl;
//    std::cout << "omega_e: " << std::endl << omega_e << std::endl;
//    std::cout << "x_tilde: " << std::endl << x_tilde << std::endl;
//    std::cout << "dot_x_tilde: " << std::endl << dot_x_tilde << std::endl;
//    std::cout << "jacobian: " << std::endl << toEigen(robot_->getEEJacobian()) << std::endl;
//    std::cout << "jpinv: " << std::endl << Jpinv << std::endl;
//    std::cout << "jsim: " << std::endl << robot_->getJsim() << std::endl;
//    std::cout << "c: " << std::endl << robot_->getCoriolis().transpose() << std::endl;
//    std::cout << "g: " << std::endl << robot_->getGravity().transpose() << std::endl;
//    std::cout << "q: " << std::endl << robot_->getJntValues() << std::endl;
//    std::cout << "Jac Dot qDot: " << std::endl << toEigen(robot_->getEEJacDotqDot()) << std::endl;
//    std::cout << "qdot: " << std::endl << robot_->getJntVelocities() << std::endl;
//    std::cout << "Jnt lmt cost: " << std::endl << cost << std::endl;
//    std::cout << "Jnt lmt gradient: " << std::endl << grad.transpose() << std::endl;
//    std::cout << "--------------------" << std::endl;

    // inverse dynamics
    Eigen::Matrix<double,6,1> y;
    y << dot_dot_x_d - toEigen(robot_->getEEJacDotqDot())*robot_->getJntVelocities() + Kd*dot_x_tilde + Kp*x_tilde;

    return M * (Jpinv*y + (I-Jpinv*J)*(/*- 10*grad*/ - 1*robot_->getJntVelocities()))
            + robot_->getGravity() + robot_->getCoriolis();
}
```

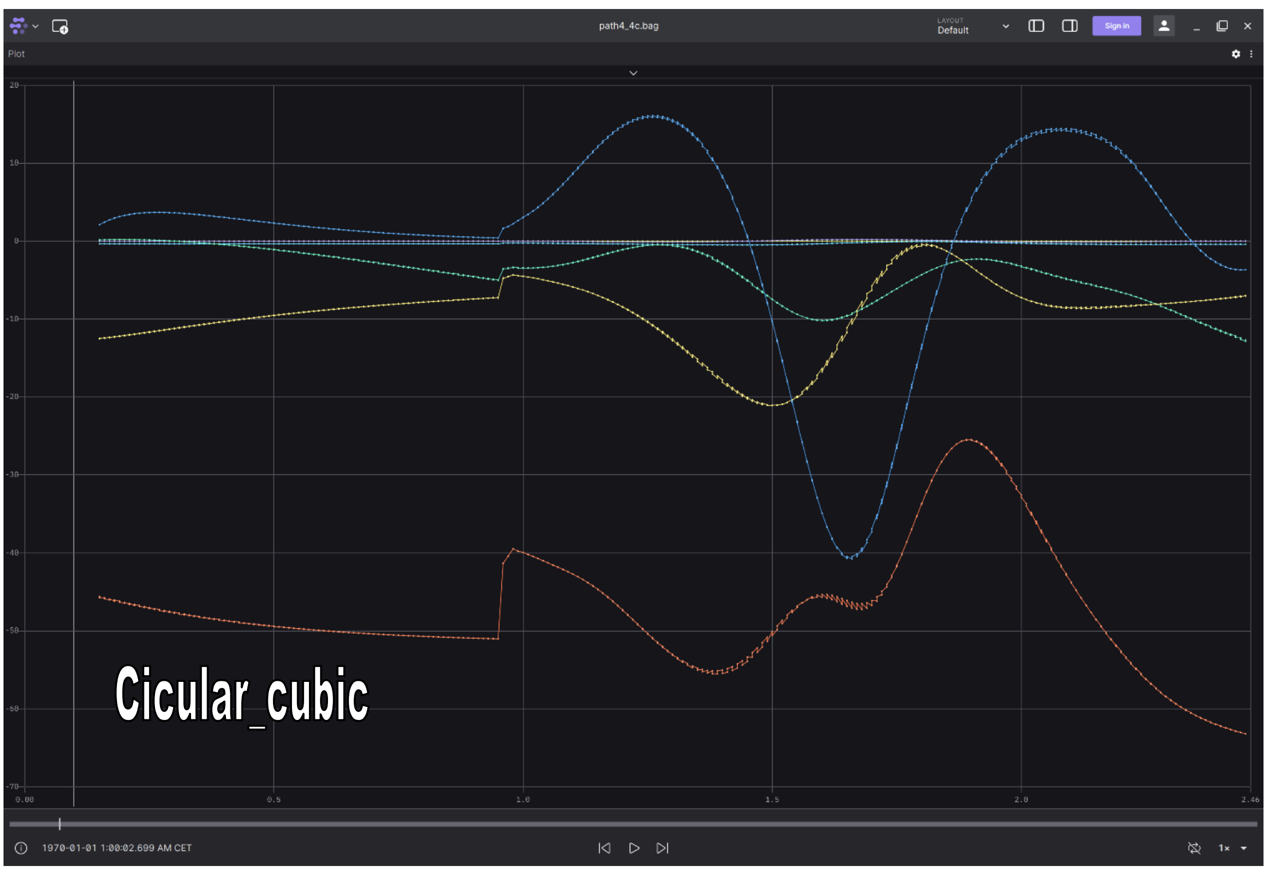
```cpp
// Eigen::VectorXd KDLRobot::getEEJacDotqDot()
// {
//     return s_J_dot_ee_.data;
// }

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

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






path3_4c.bag
Cicular_trapezoidal
Reset view
1970-01-01 1:00:00.179 AM CET


path4_4c.bag
Cicular_cubic
1970-01-01 1:00:02.699 AM CET