# Cartesian Trajectory Messages

The need to verify by plotting the trajectory profile of the gotraj package was desired. So, first a Cartesian trajectory message set with acceleration and jerk was scoured on the Internet, with no luck. There is some ROS effort to establish a set of Cartesian trajectories (<http://wiki.ros.org/robot_mechanism_controllers/Reviews/Cartesian%20Trajectory%20Proposal%20API%20Review>) but nothing permanent has been established. Instead, the draft message constructs were used and a “cartesian\_trajectory\_msg” package was built that simply compiled the msg files into C++ header files. This was the only purpose of the cartesian\_trajectory\_msg package. Five message files were coded:

CartesianTrajectoryResult.msg

CartesianTrajectoryGoal.msg

CartesianTolerance.msg

CartesianTrajectoryPoint.msg

CartesianTrajectoryError.msg

The primary message file of interest was the CartesianTrajectoryPoint.msg since it contained velocity, acceleration and jerk fields that could be published and read by rqt\_plot:

#

# CartesianTrajectoryPoint.msg

#

std\_msgs/Duration time\_from\_start

geometry\_msgs/Pose pose

geometry\_msgs/Twist twist

std\_msgs/Float64[] posture

std\_msgs/Float64 velocity

std\_msgs/Float64 acceleration

std\_msgs/Float64 jerk

So, this message was filled out by the controller, based on the current and last robot motion state and the results were published.

The ROS command rostopic can be used to verify that the NIST controller publishes a Cartesian status topic for the Fanuc and the Motoman robots. You should see the name of the robot with an “\_” prefixed to to the “cartesian\_status” topic name. If you have ROS running you can do a rostopic list to verify this.

> rostopic list

/clicked\_point

/crcl\_command

/crcl\_status

/fanuc\_cartesian\_status

/fanuc\_crcl\_command

/fanuc\_crcl\_status

/initialpose

/joint\_states

/motoman\_cartesian\_status

/motoman\_crcl\_command

/motoman\_crcl\_status

/move\_base\_simple/goal

/nist\_controller/robot/joint\_states

/rosout

/rosout\_agg

/tf

/tf\_static

/visualization\_marker

/visualization\_marker\_array

Once the topic fanuc\_cartesian\_status has been verified as existing, you can monitor the topic for traffic to verify that the controller is updating the Cartesian velocity, acceleration and jerk statu within the topic. At the command line, with ROS and the controller running in another terminal, run “rostopic echo fanuc\_cartesian\_status” as a command and you should see a stream of Cartesian status. You are now ready to plot the Cartesian status data.

>rostopic echo fanuc\_cartesian\_status

. . .

velocity:

data: 0.000134053950859

acceleration:

data: 6.7024994282e-05

jerk:

data: -4.51721361763e-05

rqt\_plot displays a scrolling time plot of the data published on topics. The controller was modified - it added cartesian trajectory messages so that topics were available to publish to. Once an end-effector Cartesian motion was available to publish topics, the velocity, acceleration and jerk of fanuc and motoman robots were published. To calculate these values, the current and last pose were used. Velocity was calculated as the distance traveled divided by two. If no previous pose was availabe, zero was used. Likewise acceleration was calculated by taking the current velicty minus the last velocity and dividing the result by two. Similarly, jerk was calculated.

A circular buffer of size one was kept to maintain the last vel, acc, and jerk using the boost templated implementation http://www.boost.org/doc/libs/1\_55\_0/doc/html/circular\_buffer/example.html. Both the current status and the last status have the pose, but this only helps in calculating the velocity. So a circular buffer

boost::circular\_buffer<cartesian\_trajectory\_msg::CartesianTrajectoryPoint> profiles;

was declared of size 1 (initialized at the Controller constructor). Below is the code to compute vel, acc and jerk. Each of these were computed since the go traj motion planner supports smoothing based on jerk. It was not evident in the plots, but the motions were relatively short compared to the maximum vel distance, so the smooth ramping was not evident.

tf::Pose &lastpose(laststatus.currentpose);

lastpose = status.currentpose;

status.currentpose = Kinematics()->FK(status.currentjoints.position); /\*\*< current robot pose \*/

// compute ee cartesian vel, acc, jerk

cartesian\_trajectory\_msg::CartesianTrajectoryPoint profile;

// this doesn't include angular velocity calculation - assume scale is almost same as linear

profile.velocity.data = (status.currentpose.getOrigin().distance(lastpose.getOrigin())) / 2.0;

double lastvel = (profiles.size() > 0) ? profiles[0].velocity.data : 0.0;

double lastacc = (profiles.size() > 0) ? profiles[0].acceleration.data : 0.0;

profile.acceleration.data = (fabs(profile.velocity.data) - fabs(lastvel)) / 2.0;

profile.jerk.data = (fabs(profile.acceleration.data) - fabs(lastacc)) / 2.0;

profiles.push\_back(profile);

cartesian\_status.publish(profile);

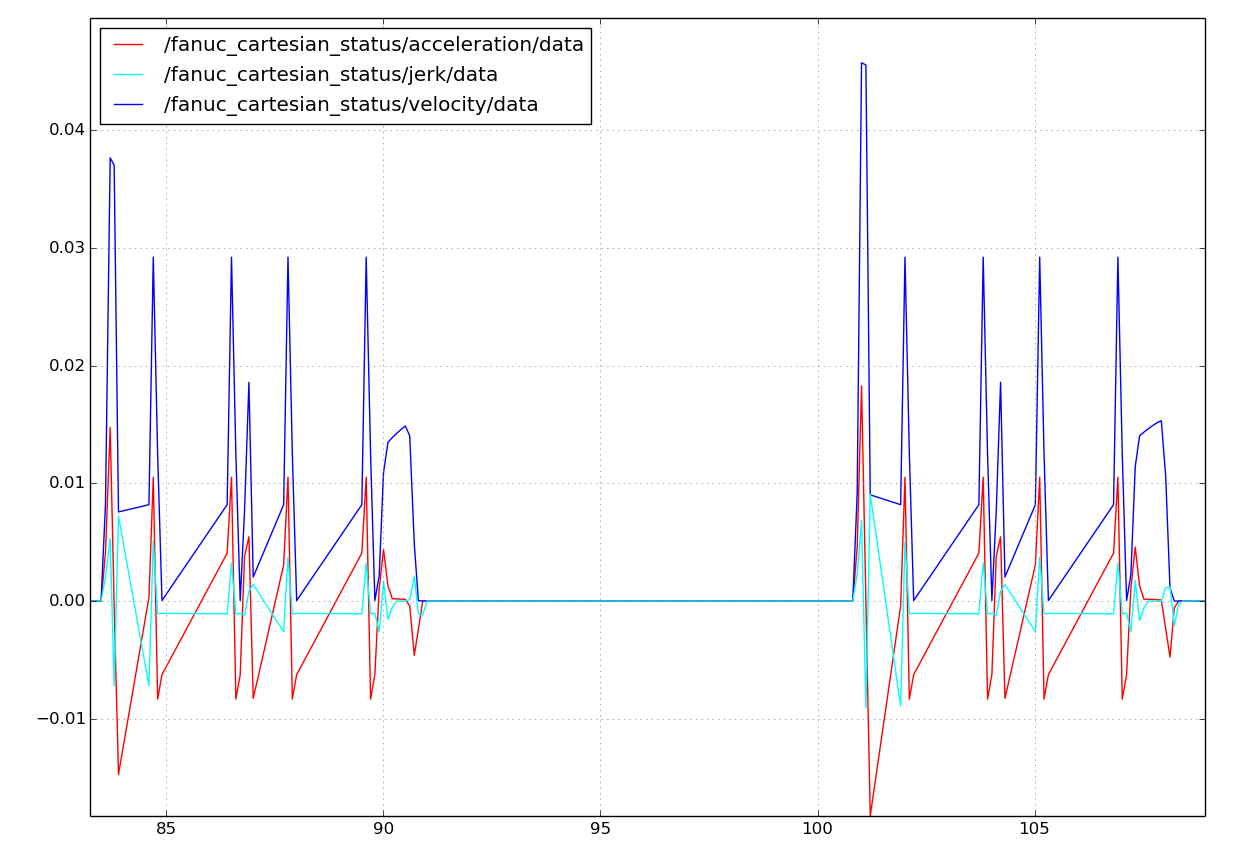
Of note, the std\_msgs::Float64 is not a C++ data primitive. Instead. you must assign a double to its data field, e.g., profile.acceleration.data= (double) 1.1;

To run the rqt\_plot at a terminal console with ROS and the NIST conttroller running type:

> rqt\_plot /fanuc\_cartesian\_status/velocity:acceleration:jerk

which listens to the published vel/acc/jerk values published by the fanuc lr mate controller. You have to play with the display to see anything, since the vel/acc/jerk are small. It is suggested to use the last button on the right (the check mark) and change x to run from 0 to 1000.

Then you can use the fourth button (zoom) which allows you to draw crosshairs that can select a range of the plot to zoom in on (from 80-110) or 30 time units. The x axis is the time sequence. This can be done since the profile will be a visible blip, but might be small. Just zoom in, and you should see the vel/acc/jerk profile. The checkers sequence is evident in the plot: first, the robot approaches the hecker offset, then the robot descends to the part, grasps the checker, and retracts to a safe distance, and then makes the checker move: approach, descend, release the checker, and depart. Finally, a coordinated joint move to a programmed “safe” robot position. The last joint move has slower max profile so the plot is flatter with an actual hump.



Then, the plot is zoomed again, 110-130, that is 20 total time units.

