Readme for Gotraj in ROS

Michaloski, John L. (Fed)

Tuesday, March 28, 2017

This document discusses Robot Operating System (ROS) and the creation and use of Gomotion package to generate trajectory motion.

## Gomotion Trajectory Planning Algorithms

As background, relevant trajectory terms will be given. A trajectory is the path of a robot arm through space. It is desirable that the motion is smooth and minimizes the wear and tear on the robot servos. A trajectory is based on a combination of dynamical properties to define the motion. These properties include (but are not limited to) velocity, acceleration, and jerk. Velocity describes the rate of change of the position (and orientation) of the robot, with respect to time. While acceleration is the rate of change of the velocity of the robot, again with respect to time. Jerk is the rate of change of acceleration with respect to time, and as such is the second derivative of velocity, or the third derivative of position. Of note, robot velocity, acceleration and jerk can be categorized as pertaining to position, orientation, or a combination of both. Cruise is the case when the robot has zero acceleration and is travelling at a constant velocity.

Trajectory planning functions abbreviations used include:

* CV means constant velocity, CA means constant acceleration,
* CJ means constant jerk.

The Go Motion trajectory planning algorithms are based on smooth velocity profiling with bounded speed, acceleration and jerk, called "constant jerk" or "S-curve" velocity profiling. This gives smoother control than "trapezoidal" velocity profiling, which transitions instantaneously between acceleration and no acceleration and incurs spikes in unbounded jerk.

Constant-jerk (CJ) profiling is shown in Figure 1, where a plot of the speed versus time is given. There are 7 phases to the motion- designated d1 through d7. The phases motion behavior follows.

* Phase 1 (or d1) is a jerk phase, where the acceleration varies smoothly from 0 at time 0 to a1 at time t1 following the jerk (change in acceleration per unit time) j0.
* Phase 2 (or d2) is an acceleration phase, with constant acceleration a1 throughout.
* Phase 3 (or d3) is a jerk phase (or de-jerk phase) with constant (negative) jerk slowing down the acceleration from a1 to 0.
* Phase 4 (or d4) is a constant speed phase at speed v3.
* Phase 5 (or d5) is a constant-jerk counterpart to phase 3, where the deceleration varies smoothly from 0 to -a1.
* Phase 6 (or d6) is a constant-acceleration counterpart to phase 2.
* Phase 7 (or d7) is a constant-jerk counterpart to phase 1, where the deceleration varies smoothly from -a1 to 0 and motion stops.

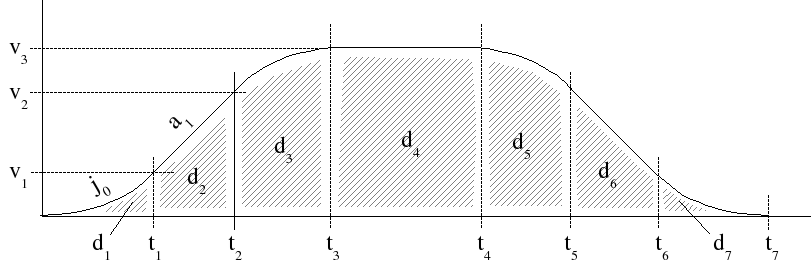


Figure Constant jerk velocity profiling.

However, often in robot motion planning, the robot never achieves the all stages of motion. The next sections will cover the trajectory calculations.

### No Cruise or Acceleration

Let's assume the case where no cruise or acceleration phases apply and only a deceleration phase is required.



Figure Deceleration only trajectory phase

Where t is the current time, and t3 is three clock cycles in the future. d3 is phase. The term amax is clearly the acceleration maximum while vmax is the velocity maximum.

If then no acceleration or cruise, and the following equation determines the distance traveled (d3):

[1]

Which is equivalent to:

[2]

The equation [2] is over constrained with d, v, a so relax a, which gives



Figure Deceleration only trajectory phase relaxed parameters

So

Given

### No Cruise

In this case, the robot may accelerate to attain the maximum velocity, but never coasts along at a constant velocity. The robot is either accelerating or decelerating.



Figure No cruise trajectory

The case where no cruise trajectory occurs when as shown in Figure 4. The calculations for this trajectory case are:

|  |  |
| --- | --- |
|  |  |

If then no cruise is required in the trajectory. The trajectory can then be recalculated and pictorially given by:



giving:

|  |  |
| --- | --- |
|  |  |

Otherwise there is a coast but the trajectory never attains the maximum velocity, and follows the following trajectory profile, outlined in the figure below:



### Cruise but not attaining maximum velocity

And leads to the following calculations:

# Cartesian Trajectory Messages

To verify the trajectory profile of the gotraj package, plotting was desired. Unfortunately, Cartesian trajectory messages with position, velocity, acceleration and jerk specified were not available in ROS. So, first a Cartesian trajectory message set with acceleration and jerk was required. There is some ROS activity 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 created. 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 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 status 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 available, zero was used. Likewise, acceleration was calculated by taking the current velocity 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 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 gotraj 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);

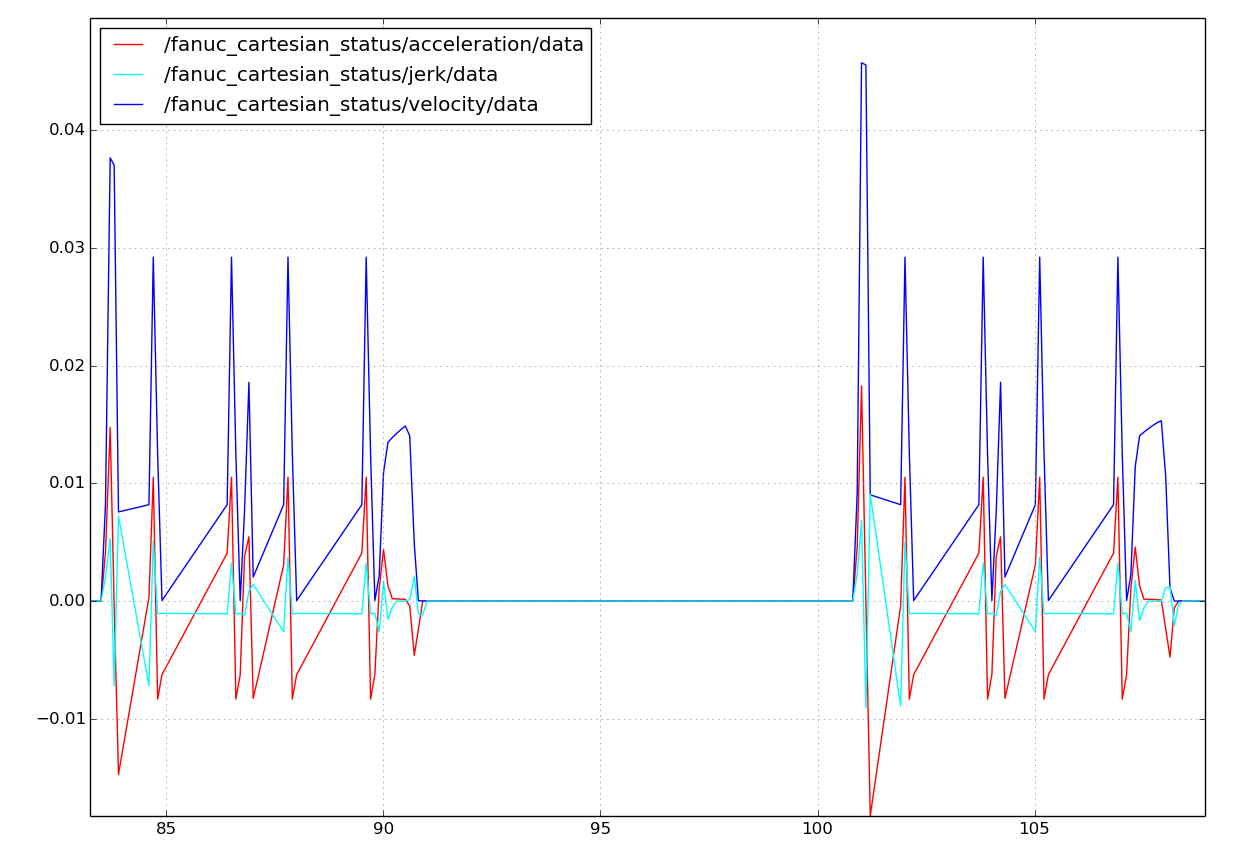
Of note, the std\_msgs::Float64 is not a C++ data primitive. Instead, you must assign a double to its data field, e.g., profile.acceleration.data= (double) 1.1;

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

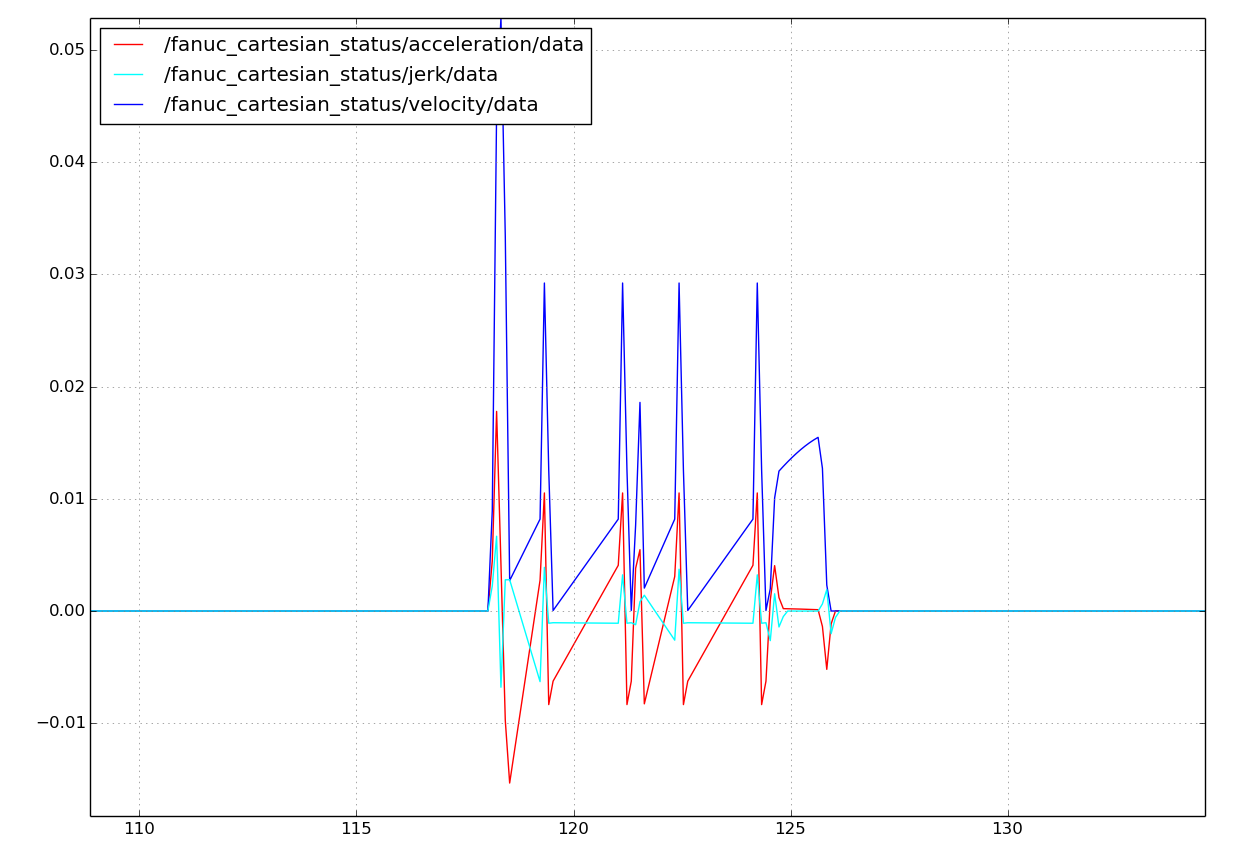
> rqt\_plot /fanuc\_cartesian\_status/velocity:acceleration:jerk

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 checker 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.



## Adjusting the Cartesian Trajectory Parameters

The configuration of the Cartesian trajectory profile encompasses setting the maximum linear velocity, acceleration and jerk as well as the maximum rotational velocity, acceleration and jerk. This capability can be adjusted easily within the ini file associated with the controllers. For each controller section tag, it must have a parameter tag that defines an std::vector of three values.

The Cartesian trajectory parameters are set in an ini file that is read upon startup by the controller each time. The ini file has several sections which are lines that have the section enclosed in brackets, e.g.., [Section]. The file location of the configuration ini file is under the config folder of the nist\_robotsnc ROS package under the nistfanuc\_ws workspace (i.e., .../nistfanuc\_ws/src/nist\_robotsnc/config/nist\_robotsnc.ini). Inside the ini file, is a section {fanuc\_] that describes the Fanuc LR Mate 200 id robot. Included in this are the "linearmax" and the"rotationmax" tags which control the ramping profile of the gotraj motion trajectory.

[fanuc\_]

longname=FanucLRMate200iD

prefix=fanuc\_

. . .

**linearmax=0.01, 0.1, 1**

**rotationmax=.1, 1.0, 10.0**

In the previous section, the linear max trajectory parameters were

linearmax=1.0,10., 100.0

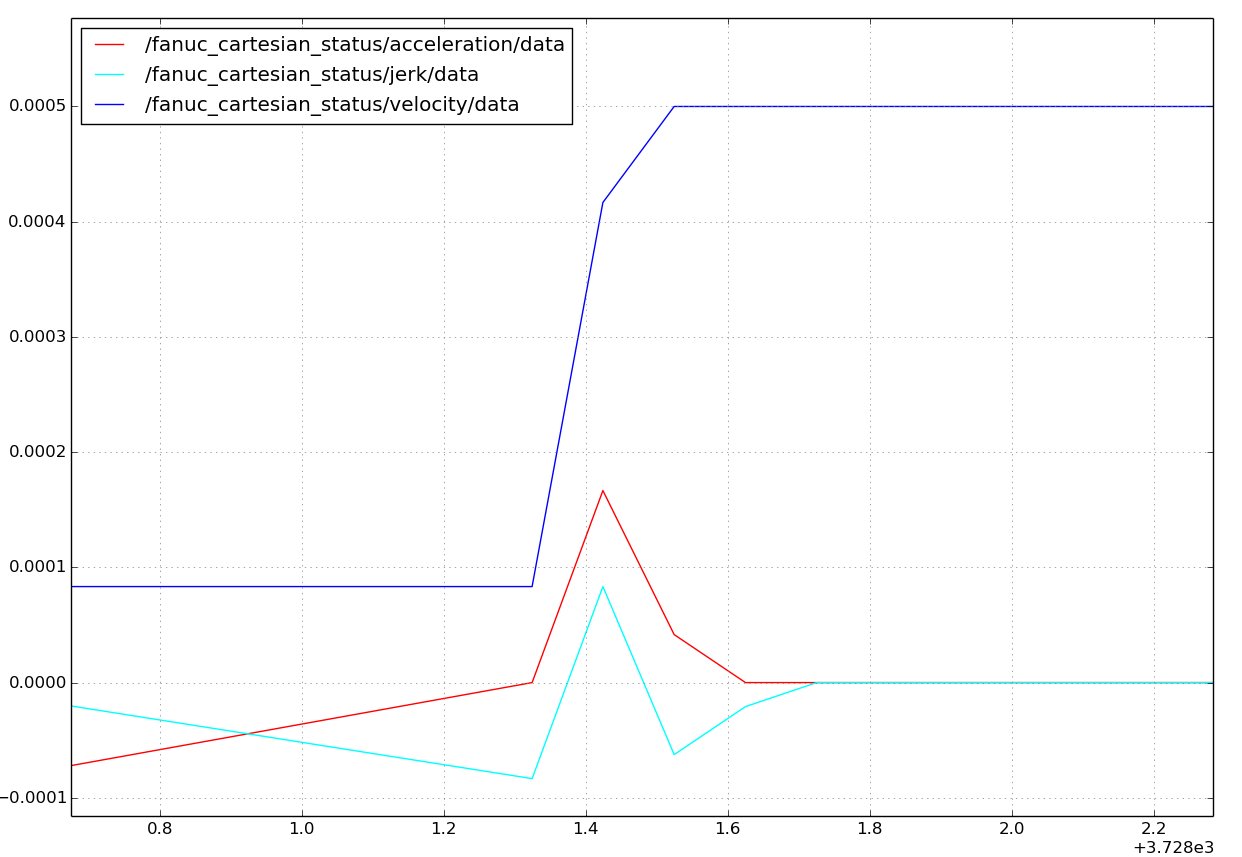
rotationmax=.1, 1.0, 10.0

and this gave a very with no coasting, and not reaching the velocity maximum. Using the slower trajectory profile parameters

linearmax=0.01, 0.1, 1

rotationmax=.1, 1.0, 10.0

one can see a slower ramping acceleration and deceleration, in fact, the ramping is so slow that an entire coasting profile cannot be graphed, and instead a portion where the transition activity is shown:



The ini file parameters are read by the robot control system (RCS) interpreter, which queries the gotraj routines for path updates. When there are no more path updates on the gotraj motion queue, the motion has completed. Below is the code that when if finds a new Cartesian move command, initializes the gotraj pose generator with the current pose (curpose) and goal pose (finalpose) and supplies the motion parameters as supplied by the ini file: e.g., \_nc->linearmax ()[0], where the maximum velocity is in array position 0, acceleration is in array position 1, and jerk is in array position 2.

if (cmd.CommandNum() != \_lastcmdid) {

\_lastcmdid = cmd.CommandNum();

\_go->InitPose(curpose,

finalpose,

gomotion::GoTrajParams(\_nc->linearmax()[0],

\_nc->linearmax()[1],

\_nc->linearmax()[2]),

gomotion::GoTrajParams(\_nc->rotationmax()[0],

\_nc->rotationmax()[1],

\_nc->rotationmax()[2])); // 1 meter/sec

}

Then each cycle (including the first one that did the InitPose above, a new pose is calculated:

tf::Pose gopose = \_go->NextPose() ;

tf::Pose nextpose = \_nc->invBasePose() \* gopose \* \_nc->invGripperPose();

Then the gotraj library is queried to see if it is done with this trajectory sequence (could have several waypoints appended to the motion queue). The IsDone() method is calls the gotraj motion queue, and if it's done returns true.

if (\_go->IsDone())

return CanonStatusType::CANON\_DONE;

else

return CanonStatusType::CANON\_WORKING;

The gotraj trajectory generator can generate code for joint or Cartesian motion. However, it assumes that one user per instance is commanding the gotraj routine. That is, unpredictable behavior would result if multiple threads attempted to perform a NextPose on the same gotraj object.

# Robot Kinematic Chain

The robot path is specified I terms of a "kinematic chain " made up of a series of homogeneous matrix transforms relation the manipulator to the task.



Figure Position Equation

This kinematic chain is evaluated many times a second, each time providing a new set of joint angles positions for the manipulator to follow. This kinematic chain will execute a function each sample period that returns a 4x4 Homogeneous Transform that defines the position and orientation of that element. The Trajectory Generator will use the values in the kinematic chain to solve the kinematics for a robot. Sensor integration is accomplished with the same mechanism; transforms are determined in real time based on sensor input instead of by a static transform.