Commanding Joint Trajectories

Wyatt Newman
October, 2015

As described within a URDF file, a robot model consists of a collection (a tree) of robot links, connected pair-wise via joints. Each joint allows a single degree of freedom of motion—either a rotation or a translation. Either type of motion may be referred to generically as a "displacement." A low-level servo controller (similar to the minimal_joint_controller example) exerts a torque or force (generically referred to as an "effort") to attempt to achieve a desired "state". The desired state is comprised of a desired position, possibly augmented by desired velocity, possibly further augmented by desired acceleration. The controller compares the desired state to the actual (measured) state to derive a control effort, which is to be imposed by the respective actuator.

If the desired state is too far from the actual state, the controller will try to exert unreasonable efforts, resulting in effort saturation, leading to poor following of or convergence to desired states. The resulting motion can be unpredictable and dangerous.

In order for the robot to achieve a desirable motion, the motion commands should be evaluated in advance to make sure they are achievable within the constraints of:

- the robot should not hit objects in its environment
- the path in joint space should conform to the min and max range of motion of the joints
- the joint velocities should remain within the velocity limits of the respective actuators
- the required joint efforts should remain within the effort limits of the actuators

In general, meeting these requirements can be difficult to solve. A planned motion may be simulated in advance, to test for collisions or violations of joint ranges, joint velocities or joint torques. One condition to avoid is a "step" command to any joint. If a joint is commanded to move instantaneously from A to B, the motion will be physically impossible to achieve, and the resulting behavior may be undesirable.

The Joint Trajectory Message:

A necessary (but not sufficient) requirement of a safe robot command is that the commands should be updated frequently (e.g. 100Hz or faster is typical), and the commands should form an approximation of a smooth, continuous stream for all joints.

Since this is a common requirement, ROS includes a message type for this style of command: the trajectory_msgs::JointTrajectory. Invoking:

```
rosmsg show trajectory_msgs/JointTrajectory
```

shows that this message is comprised of the following fields:

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string[] joint_names
trajectory_msgs/JointTrajectoryPoint[] points
  float64[] positions
  float64[] velocities
  float64[] accelerations
  float64[] effort
```

duration time_from_start

In the header, the frame_id is not meaningful, since the commands are in joint space (desired state of each joint).

The vector of strings “joint_names” should be populated with text names assigned to the joints. For a serial-chain robot, joints are conventionally known by integers, starting from joint 1, the joint closest to ground (most “proximal” joint), and progressing sequentially out to the most distal joint. However, robots with multiple arms and/or legs are not so easily described, and thus names are introduces.

In specifying a vector of desired joint displacements, one must associate the joint commands with the corresponding joint names. Generically, there is no requirement to specify the joint commands in any specific order. Further, there is generally no requirement that one must specify all joint commands on every iteration. For example, one might command a neck rotation only in one instance, then follow that by a separate command to a subset of joints of the right arm, etc. However, some packages require that all joint states be specified in every command (whether or not it is desired to move all joints). Other packages receiving trajectory messages might implicitly depend on specifying joint commands in a specific, fixed order, ignoring the “joint_names” field (although this is not preferred).

The bulk of the trajectory message is a vector of type “trajectory_msgs/JointTrajectoryPoint”. This type contains 4 variable-length vectors and a “duration.” A trajectory command can use as few as one of these fields and as many as all four. One common minimal usage is to specify only the joint displacements in the “positions” vector. This can be adequate, particularly for low-speed motions controlled by joint position feedback. Alternatively, the trajectory command might communicate with a velocity controller, e.g. for speed control of wheels. A more sophisticated motion plan communicating with a more sophisticated joint controller would include multiple fields (specifying both positions and velocities is common).

To command coordinated motion of all 7 joints of a 7-DOF arm, for example, one would populate (at least) the “positions” vector for each of N “points” to visit, starting from the current pose and ending at some desired pose. Preferably, these points would be spaced relatively close together in space (i.e. with relatively small changes in any one joint displacement command between sequential “point” specifications). Further, it is desirable that each point include specification of consistent joint velocities (although it is legal to specify a trajectory without specifying the joint velocities).

The each joint-space point to be visited must also specify a “time_from_start.” These time specifications must be monotonically increasing for sequential points. Further, these time specifications should be consistent with the velocity specifications. It is the user's responsibility to evaluate that the specified joint displacements, joint velocities and point arrival times are all self consistent and achievable within the robot's limitations.

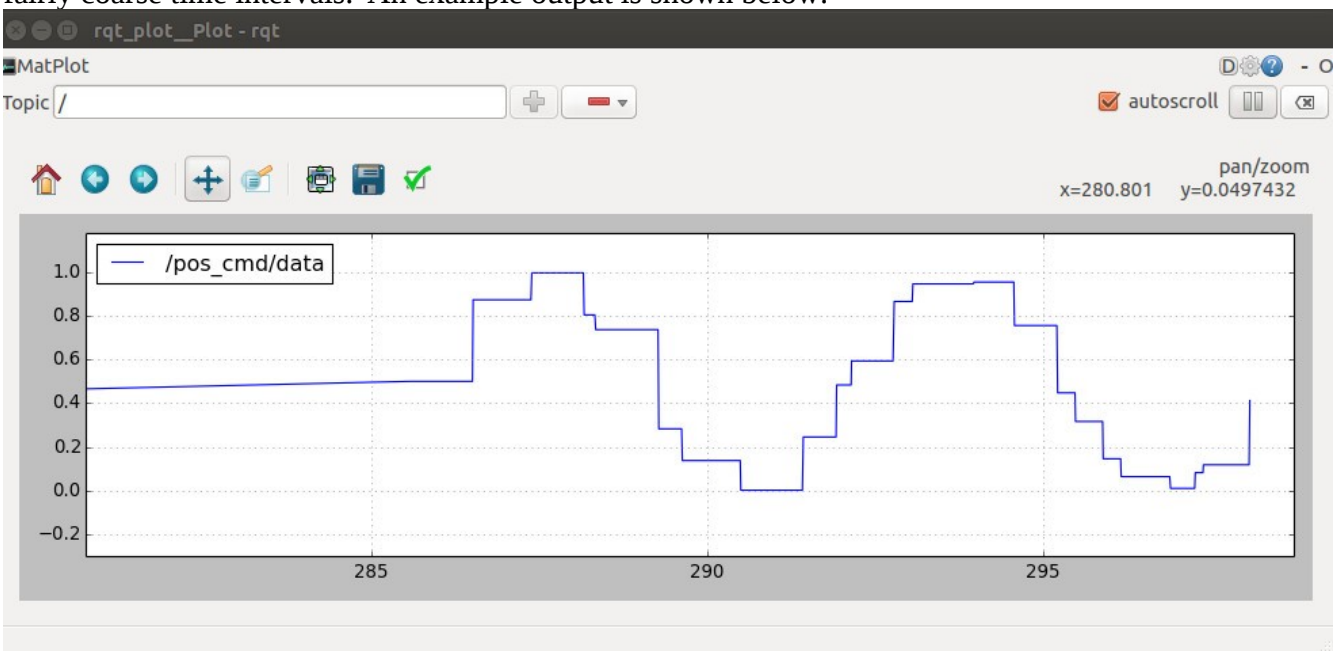
The “time_from_start” value, specified for each joint-space point to be visited, distinguishes a “path” from a “trajectory.” If one were to specify only the sequence of poses to be realized, this would comprise a path description. By augmenting space (path) information with time, the result is a trajectory.

Example use of a Trajectory Message:

The package “example_trajectory” illustrates how to use trajectory messages with an action server. In this package, an action message is defined: “TrajAction.action”. The “goal” field of this action messages contains: trajectory_msgs/JointTrajectory trajectory. This action message is used by a

trajectory client to send goals to a trajectory action server.

Two illustrative nodes are the “example_trajectory_action_client” and “example_trajectory_action_server.” The client computes a desired trajectory—in this case, consisting of samples of a sinusoidal motion of “joint1”. (This could easily be extended to N joints, but this is sufficient for testing on the “minimal_robot.”) The samples are deliberately taken at irregular and fairly coarse time intervals. An example output is shown below:



The values of position command (radians) from the above figure show that the originating sinusoidal function is sampled coarsely and irregularly. This was done to illustrate the generality of the trajectory message.

The joint-command samples are packaged into a trajectory message along with associated arrival times, and this message is transmitted to the trajectory action server within the action goal.

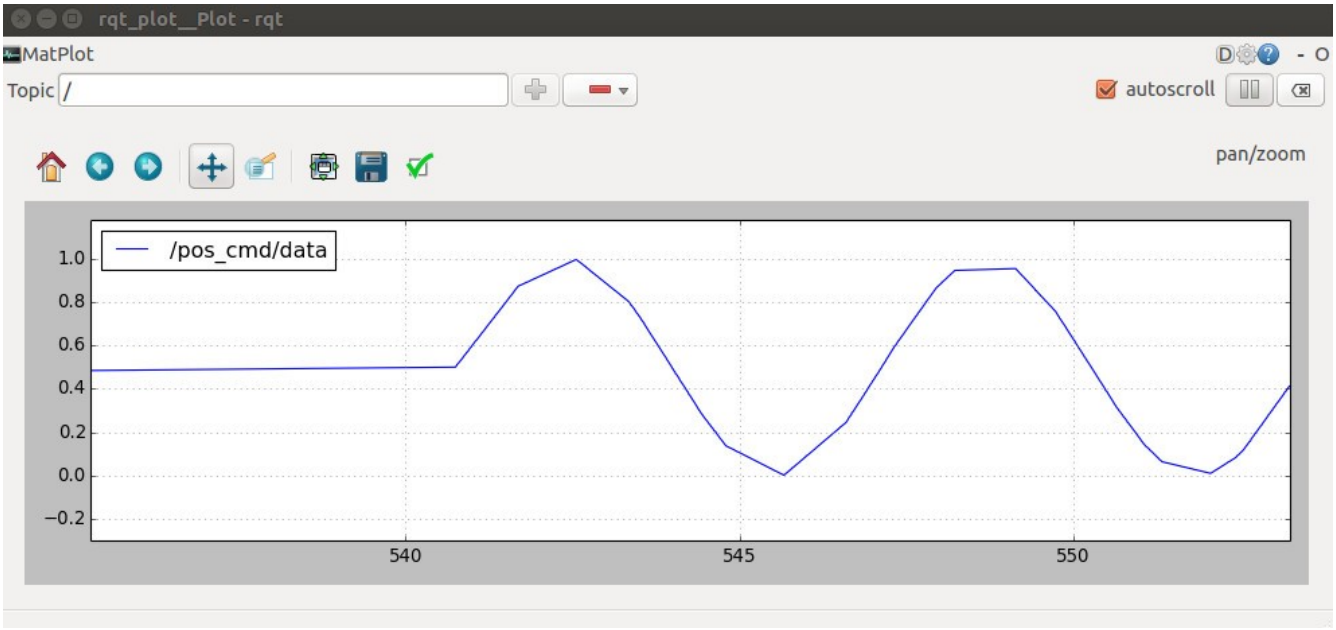
The example_trajectory_action_server receives the goal message and interpolates linearly between specified points, resulting in the following profile (for another randomly-sampled sinusoid). The resulting profile is piecewise linear, but sufficiently smooth that the minimal_robot can follow this command reasonably well, yielding a smooth motion.

The example is run with:

roslaunch minimal_robot_description minimal_robot.launch (to bring up the one-DOF robot and its controller)

rosrun example_trajectory example_trajectory_action_server (to start the trajectory action server, compatible with the one-DOF robot)

rosrun example_trajectory example_trajectory_action_client (to generate a coarse trajectory and send it to the action server within a goal message).



The one-DOF robot displays acceptable motion.

For the example trajectory action server, the velocities specified by the client in the goal message are ignored. One could do better by including velocity commands as part of the command message sent to the servo controller. In the one-DOF robot case, the minimal joint controller does not accept velocity feedforward commands.

It would also be desirable to have a smoother interpolator, e.g. cubic splines. (See e.g. http://sdk.rethinkrobotics.com/wiki/Joint_Trajectory_Action_Server for a joint trajectory action server for the Baxter robot, written in Python, which uses cubic spline interpolation).

It is a design decision as to how intelligent the trajectory action server should be. Optimizing a trajectory with considerations of speed, precision and collision avoidance is a computationally difficult problem. Further, the optimization criteria could change from one instance to another. The trajectory specification—even if it is coarse—must still take into consideration issues of collisions and speed and effort constraints. Rather than perform this optimization twice (for construction of a viable trajectory, then further smoothing and optimization of that trajectory by the action server), one could perform trajectory optimization as part of the trajectory specification. This could yield a dense sampling of points in the trajectory specification, along with compatible velocities, accelerations and gravity-load compensation. If this approach is taken, then linear interpolation of the dense trajectory commands should be adequate in general—although the trajectory action server should also pass along the corresponding velocity and effort values contained within these otpimized trajectories.

Trying to make the trajectory action server more intelligent than mere linear interpolation is thus not warranted—and may, in fact, interfere with optimization of pre-computed trajectories. Thus, this simple example is adequate.

This discussion of trajectory messages applies directly to ROS-Industrial (see http://rosindustrial.org/). To bridge ROS to existing industrial robots, one can write the equivalent of the example_trajectory_action_server in the native language of the target robot. One must write custom communications software for the industrial robot to receive packets containing the equivalent

information of trajectory messages, and the robot must also gather and transmit joint state data.