Writing open-loop motion scripts for Atlas

Wyatt Newman, September, 2014

In a previous document, "ROS and DRC", it was shown how one can send joint-angle commands to Atlas by publishing to the "atlas/atlas_command" topic. However, sending angle commands that have large jumps from previous command to new command gives a poor response. Joints given large jumps in commands will jerk violently, likely causing damage to the robot or the environment, or causing the robot to fall down. Instead, we need to send streams of commands with incremental updates leading to a goal pose. The present notes describe a means to do this using some primitive motion scripting. Motions produced by this means are "closed loop" in the sense that Atlas performs actuator feedback from joint angles and joint velocities to try to follow joint-angle commands. However, such motions are "open loop" in the sense that poses recorded in a motion script do not use information from the sensing head, the ankle or wrist force/torque sensors, nor the pelvis-mounted inertial measurement unit (IMU). Nonethless, motion scripts provide a useful functionality in many instances, and this is a good place to start in learning to control Atlas.

Three nodes are used to control Atlas to execute motion scripts: the low-level joint controller, the joint-trajectory behavior, and "play_file," which are described next.

Low Level Joint Controller (LLJC):

The package "low_level_joint_controller" defines a node that is responsible for direct communications with the Atlas robot. It publishes to the topic /atlas/atlas_command. This node is the *only* node that should talk directly to Atlas. Any other interface should communicate with Atlas via this gateway, and this node incorporates protections to prevent execution of dangerous commands (e.g., joint commands out of range or excessive velocities).

The low-level joint controller node is also responsible for setting feedback gains (which are part of the AtlasCommand message). Such gains should be tuned carefully. Users should not be able to change these gains by accident, since gain changes can result in instability and damage. Details of the true actuators--which are electrohydraulic--differ considerably from the simulated actuators--which are modeled as torque sources. Appropriate gains for the physical robot are very different from gains appropriate for the simulated robot. The low-level joint controller must be informed of whether it will be controlling the simulated robot or the physical robot, and it consults different parameter files for each.

To help protect Atlas from casual user errors, there is a more restrictive interface to the low-level joint controller. The low-level joint controller node subscribes to the (hku) topic /lowLevelJntCmd, which receives messages of type hku_msgs::lowLevelJointController, as published by a higher level. This message includes fields to command all 28 joint angles, joint-angle velocities, and joint-angle accelerations. Higher-level nodes attempting to send motion commands to the LLJC must first register with the LLJC. As part of this registration, the higher-level node must declare which joints it would like to "own." The LLJC is responsible for coordinating commands from multiple higher-level nodes, combining commands from subsets of joints into a common atlas/atlas_command message. This partitioning is useful for specialization of higher-level controls. For example, one node may be responsible for controlling operation of a steering wheel using the arms while a separate node is responsible for operating an accelerator or brake pedal using the legs. Similarly, one node may be responsible for controlling arm motions during a task while a separate node is responsible for controlling the legs to maintain balance. When a higher-level node no longer needs its "owned" joints, it should release these joints back to the LLJC.

The low-level joint controller can (optionally) incorporate gravity compensation to reduce servo errors. Also, it can (optionally) interact with a lower-level force-feedback node that enables Atlas to emulate torque-controlled joints.

The LLJC can be launched independently, but it is usually launched together with additional nodes via a launch script. Importantly, once the LLJC is running, the user must inform LLJC whether it will be controlling physical Atlas or simulated Atlas (discussed later).

Joint Trajectory Behavior:
The package "joint_traj_behavior" defines an "action server" (see http://wiki.ros.org/actionlib). Action servers are nodes, but they behave somewhat differently from publish/subscribe nodes. An action server communicates directly with an "action client" using assured reliable communications. An action client connects with an action server and transmits a "goal" message. The action server confirms receipt of the goal request, optionally provides interim feedback updates, and ultimately returns a "result" (after which the client and server disconnect, and the server waits patiently in the background for new client requests). Action servers that are launched typically consume negligible resources until/unless they are called upon to service goals. Thus, many action servers can be launched at start-up, even if it is unlikely their services will be needed.

Upon being requested to perform a service, the action server may simply return the result of some computation (e.g., return coordinates from sensory processing in the "result" message, or return a navigation plan), or it may interact with the rest of the system to achieve a specified physical goal, such as moving joints. Action servers are non-blocking, so code invoking an action client can proceed with parallel computations while the action server is working on a goal.

The Joint Trajectory Behavior action server receives goals containing specifications of desired trajectories, expressed in joint space. These trajectory messages specify which joints are involved, their desired resulting joint angles, and the duration over which the motion should be performed. The joint trajectory action server breaks up the goals into incremental commands and sends these commands to the LLJC in 3msec updates, which is fast enough that these streams of commands behave as though they are smooth and continuous. All joints to be moved are scheduled to arrive at their specified destinations simultaneously by the end of the specified move duration.

The joint trajectory behavior has a variety of options, specified in terms of three codes: *actionType*, *actionCode* and joint *selection*.

The currently defined actionType codes are:
1 - JOINT_SPACE_MOVE
4 - RIGHT_FINGERS_MOVE
5 - LEFT_FINGERS_MOVE
6 - OPEN_RIGHT_HAND
7 - CLOSE_RIGHT_HAND
13 - OPEN_LEFT_HAND
14 - CLOSE_LEFT_HAND

Code "1" specifies that the angle commands refer to Atlas' body joints. Codes 4 and 5 refer to specifying angles for the right and left Sandia hand fingers, respectively. Codes 6, 7, 13 and 14 are shortcuts for common operations of fully opening or fully closing all fingers of a specified hand.

The remaining options refer to variations on the joint-space move actionType. The actionCode options are currently:
1 - Trapezoidal velocity profile (Stop at each way point)
2 - Via point, linear interpolation between way points (Pass through)

Code "1" implies that the robot's joint angles should come to a full halt at the specified angles. In breaking up the move into small incremental commands to LLJC, the joint trajectory behavior ramps up the joint velocities at the start of the move and ramps down the velocities before the end of the move (a trapezoidal velocity profile). This results in relatively graceful moves from initial angles to final angles.

Alternatively, a move can be specified in terms of multiple segments that are to be blended together without halting after each segment. For this case, the actionCode is set to “2.”

For joint “selection”, one specifies a code for each joint to be commanded. The options are:
0 - Not selected
1 - Position Control
2 - Natural Admittance Control (NAC), compliant motion

The joint-selection code must be specified for all 28 joints of Atlas' body. If “0” is specified for a given joint, then this joint is not claimed for “ownership” by the joint-trajectory behavior (and thus this joint is available for control by another behavior node). Specifying “1” for a joint selection implies that the corresponding joint is to be controlled under position control, using the gains within the LLJC.

The third option, selection=2, is relatively new and only lightly tested. This option allows the user to specify soft servo gains, emulating “Natural Admittance Control” that is being implemented on Atlas' electro-hydraulic joints. Although this implementation on (physical) Atlas is complex, the result emulates a behavior that is modeled simply by a low-friction, low-stiffness set of joint servos. In some cases, using a deliberately soft set of feedback gains can be useful, e.g. to perform manipulation of delicate objects or to accommodate kinematic constraints, such as door hinges. If this option is chosen, then one can also (optionally) specify the proportional gains to be used. (Else a default, soft set of gains will be assumed).

The joint selection codes can be mixed within a single goal specification, e.g. specifying some joints to be left unclaimed, some to be controlled under stiff position control, and others to be controlled under soft, compliant motion.

To interact with the joint trajectory behavior, one needs a complementary action client, as described next.

Motion scripts and the “play_file” action client:
A user communicates goals to the Joint Trajectory Behavior action server through the use of an “action client.” Action clients send goal requests and wait for responses from the action server. A very useful action client that interacts with the joint-trajectory behavior is “play_file.”

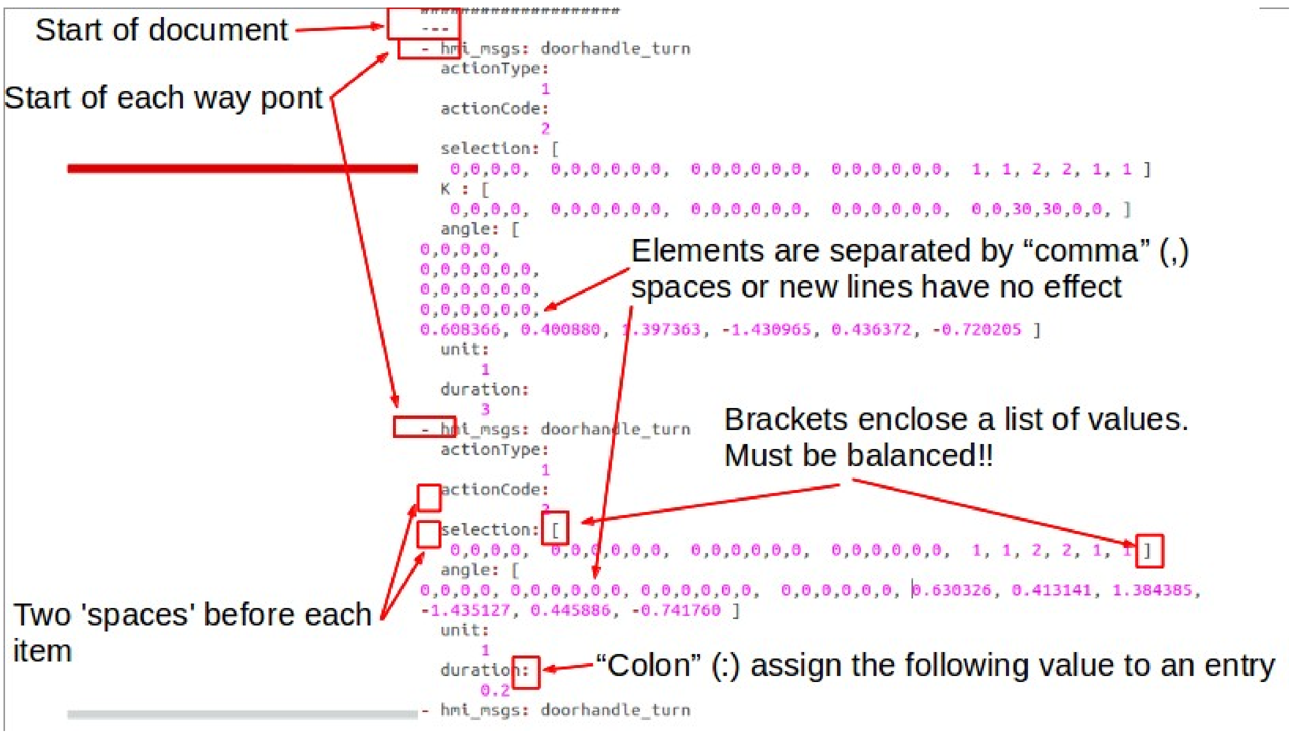
The action client “play_file” takes a command-line argument of the name of a motion file (a *.traj file, defined by TeamHKU). Some example *.traj playfile scripts are contained within the directory “traj_files” of the team code. Team HKU trajectory scripts are written in YAML syntax (see wikipedia.org/wiki/YAML).

The “play_file” node parses the specified *.traj file, extracts the data to populate a “goal” message, connects to the joint-trajectory behavior, and transmits the goal message. If the motion script contains multiple moves, each move is sent to the joint-trajectory behavior in sequence.

The format of the TeamHKU trajectory script is shown below. One may embed comments using the “#” symbol. Often, trajectory scripts will contain comments in the header reminding the user of the format, e.g.:

```
##########################################################################################
#   Format:
#
#   - hmi_msgs: msg_name
#     ### msg_name is the message name, preferrably to be unique
#     actionType:
#               1
#     ### actionType corresponds to the behavior type defined
#     actionCode:
#               0
#     ### actionCode is used to set sub behavior types
#     selection: [
#                0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
#                ]
#     ### 0: not selected, 1: selected - 28 joints to select
#     angle: [
#            0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
#            ]
#     ### there are 28 joints to set
#     unit:
#               1
#     ### 0: degree, 1: radian
#     duration:
#               1.0
##########################################################################################
```

An example trajectory script is:



Start of document
Start of each way pont
Elements are separated by "comma" (,) spaces or new lines have no effect
Brackets enclose a list of values. Must be balanced!!
Two 'spaces' before each item
"Colon" (:) assign the following value to an entry

```
####################
---
- hmi_msgs: doorhandle_turn
  actionType:
          1
  actionCode:
          2
  selection: [
   0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  1, 1, 2, 2, 1, 1 ]
  K : [
   0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  0,0,30,30,0,0, ]
  angle: [
0,0,0,0,
0,0,0,0,0,0,
0,0,0,0,0,0,
0,0,0,0,0,0,
0.608366, 0.400880, 1.397363, -1.430965, 0.436372, -0.720205 ]
  unit:
      1
  duration:
      3
- hmi_msgs: doorhandle_turn
  actionType:
          1
  actionCode:
          2
  selection: [
   0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  1, 1, 2, 2, 1, 1 ]
  angle: [
0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0,  0,0,0,0,0,0, 0.630326, 0.413141, 1.384385,
-1.435127, 0.445886, -0.741760 ]
  unit:
      1
  duration:
      0.2
- hmi_msgs: doorhandle_turn
```

Illustration 1: Example YAML syntax for trajectory script

The YAML document starts with the key "---". A single hyphen indicates start of a new group. The group contains multiple fields, indented, each with a label and a colon. Fields may be populated with lists, bounded by balanced [ ] brackets.

The field "hmi_msgs:" is used to name a move. This is for human readability and is not used by the trajectory behavior. The actionType and actionCode are specified next, with codes as noted above.

The “selection:” field must contain 28 integers corresponding to unclaimed, position controlled, or compliantly-controlled joints. For readability, one may insert spaces in this list for natural groupings. The numbering corresponds to the Atlas joint definitions. The first group of 4 corresponds to the 3 torso joints and the neck. The subsequent group of 6 corresponds to the left leg. The next group of 6 is the right leg. The next group of 6 is the left arm, and the final group of 6 is the right arm. (For hand control, the fingers are also specified with 28 joints, although only the first 12 joints correspond to actual finger phalanges).

In the above example, only the right-arm joints are claimed. All other joints are unspecified by this script. They will either be commanded to remain at their current angles, or they will be commanded by some other behavior node. For the 6 joints claimed by this trajectory script, 4 of them are specified to be under position control and 2 are specified to be under compliant-motion control.

Following the joint selection, the set of stiffnesses is specified. Often, it is unnecessary to specify a “K:” field, e.g. if all joints are under position control (and thus use the default LLJC gains) or if the default compliant stiffness gains (15 N-m/rad) are acceptable. If the K: field is included, then values must be entered for all 28 joints. However, the numerical values will be ignored for all but the joints selected to be under compliant control. In the above example, these gains are all set to 0 (place-holder values that will be ignored), except for joints 24 and 25 (right-arm humerus and elbow, respectively), which are specified to have torsional stiffnesses of 30 N-m/rad.

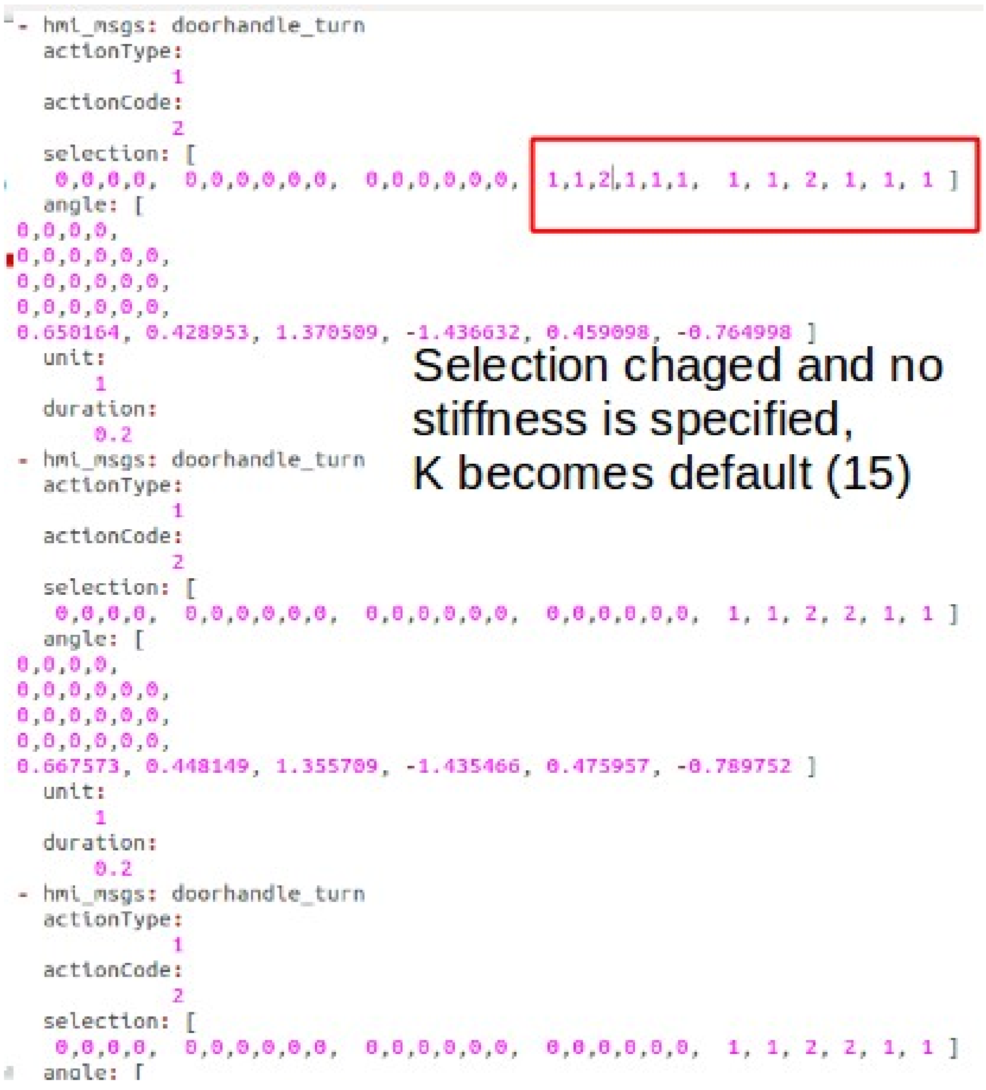
Following the (optional) “K:” field, one specifies the desired joint angles for all 28 joints. Again, these may be (optionally) formatted for readability into logical groups of joints. In the above example, place-holder values (all zeros, in this case) are filled in for the unclaimed joints (and these values will have no effect). If joint-angle goals are specified outside the achievable range, the LLJC will note an error, and it will not attempt to move the robot to the unreachable angles.

The next field, “unit:” is set to “1”, specifying that the joint-angle commands are in units of radians. (alternatively, unit code “0” implies “degrees”).

The “duration:” field specifies the number of seconds for the move. This may be a floating-point number and it may be less than unity. Specification of this move time does not guarantee that the robot will accomplish the move within this time. If the move is impossibly fast, the LLJC will object, and the move will not be completed. Alternatively, if the robot is already very close to the commanded angles, the move will be reported as completed early.

The trajectory script may (and typically will) contain multiple, sequential moves. If the moves are to be performed as via points (without stopping between move segments), this would be specified in the actionCode. In the above example, the actionCode is “2”, specifying that the first move should blend into the second move without stopping between move segments.

Continuation of the example trajectory script is shown below. For the next move, two joints are selected to be compliant, but no K: field is specified. Consequently, the default value of K=15 Nm/rad is used for these two joints.



```
- hmi_msgs: doorhandle_turn
  actionType:
            1
  actionCode:
            2
  selection: [
   0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  1,1,2,1,1,1,  1, 1, 2, 1, 1, 1 ]
  angle: [
0,0,0,0,
0,0,0,0,0,0,
0,0,0,0,0,0,
0,0,0,0,0,0,
0.650164, 0.428953, 1.370509, -1.436632, 0.459098, -0.764998 ]
  unit:
      1
  duration:
      0.2
- hmi_msgs: doorhandle_turn
  actionType:
            1
  actionCode:
            2
  selection: [
   0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  1, 1, 2, 2, 1, 1 ]
  angle: [
0,0,0,0,
0,0,0,0,0,0,
0,0,0,0,0,0,
0,0,0,0,0,0,
0.667573, 0.448149, 1.355709, -1.435466, 0.475957, -0.789752 ]
  unit:
      1
  duration:
      0.2
- hmi_msgs: doorhandle_turn
  actionType:
            1
  actionCode:
            2
  selection: [
   0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  0,0,0,0,0,0,  1, 1, 2, 2, 1, 1 ]
  angle: [
```

Selection chaged and no
stiffness is specified,
K becomes default (15)

Launching nodes for executing play-file scripts:
To run a scripted trajectory, perform the following operation, in order:

1) launch Gazebo/drcsim. Within a terminal, enter:
   `roslaunch hku_worlds drc_practice_task_4.launch`
   (alternative task scenarios may be used).
2) Wait for Atlas to drop down and stand. Then, in a second terminal, launch LLJC and joint-trajectory behavior (along with several additional, useful nodes) using:
   `roslaunch fc_bringup sim_task_4.launch`
3) In a third terminal, inform LLJC that it will be interacting with the drcsim simulator. Enter:
   `rosrun sim_controller USER`
   Wait for 3 seconds for this to conclude. If you forget to do this step and subsequently try to run “play_file”, you will get errors, and you will have to start over with launching Gazebo.
4) Run a trajectory script. First, navigate to a directory that contains your desired script (e.g. ~/ros_workspace/hku_drc_class_wsn/traj_files). Run the desired script by specifying its name in the command line. E.g., to run “stand_on_right_foot.traj”, enter:
   `rosrun play_file play_file stand_on_right_foot.traj`

The “play_file” node will run to completion and then return. However, the joint trajectory behavior server remains alive (though idle). “play_file” (step 4) may be run again, and the joint-trajectory server will continue to service goal requests.

Alternative interfaces:
The node “play_file” parses trajectory scripts that use the format described herein, and play_file sends corresponding goals to the joint trajectory server (which, in turn, sends a stream of incremental goals to the LLJC). This is useful for routine, stereotyped moves. However, specifying moves in a hard-coded trajectory script is limited in its capabilities, since it does not respond to any sensory information.

Although the play_file node is limited to parsing YAML scripts, the joint-trajectory behavior is useful more generally. It will respond to alternative clients, as long as those clients successfully connect to the server and conform to the expected goal syntax. Thus, the joint-trajectory behavior can be used more generally for moves that are computed dynamically under program control. For example, a desired object may be identified by a vision system and corresponding grasp coordinates may be computed. If the desired grasp pose is reachable, corresponding arm joint angles may be computed. A move sequence may start with a safe approach from above, ending with the desired joint angles for successful grasp. This sequence of joint-space moves may be communicated to the joint-trajectory behavior as a goal message. The joint-trajectory behavior would then execute the sequence of moves smoothly.

The joint-trajectory behavior is not well suited for all types of control, however. For example, to maintain balance requires rapid response to inertial and force/torque signals. Attempting to specify such responses through goal angles and via an interface with uncertain latency would be unlikely to succeed. Rather, such nodes should communicate directly with the LLJC.