Writing open-loop motion scripts for Atlas

Wyatt Newman, September 24, 2013

In a previous document, “ROS and DRC v2”, 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 behavior server, and “playfile,” which are described next.

Low Level Joint Controller:

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 should incorporate 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 and then frozen. Users should not be able to change these gains by accident, since gain changes can result in instability and damage. For this reason, the low-level joint controller isolates the user from changing these gains. Gain changes must be made by editing and recompiling the low-level joint controller. While there is no danger in doing so for the simulator, users should be very cautious whenever making changes to this node to be run on the physical robot, as errors can result in hardware damage.

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. While some level of protection is provided by the low-level joint controller, it is still all too easy fpr the user to issue damaging commands to Atlas via this interface. In fact, the user should not talk directly to the low-level joint controller, but through yet a higher-level layer, the “behavior server.” Correspondingly, one would not typically run the low-level joint controller node independently. Instead, launch it together with the behavior server, using:

roslaunch behavior_server_v2 behavior_server.launch

(*note*: changes are anticipated to the behavior server, and the above launch command may be replaced in the near future).

On start-up, the lowLevelJointController sets appropriate (though not yet optimzed) values for joint feedback gains. (see the file "init_controller.cpp"). For the present, the low-level joint controller node merely passes along position and velocity commands from a higher level. This node is

currently under development for gain optimization, gain scheduling, gravity-load compensation, Coulomb friction compensation, acceleration feedforward commands, torque-sensing feedback, and compliant-motion control. However, the user interface to this node (via topic /lowLevelJntCmd) is not expected to change.

One complication is that the Atlas robot has modes in which all controls are taken over by Boston-Dynamics proprietary code. Mode switching to user control of joints is performed via another user interface.

The low_level_joint_controller package owns the include file "joint_names.h". Other packages should have a dependency on low_level_joint_controller in their manifests and should include the joint name file as: <low_level_joint_controller/joint_names.h> . This provides a single location for the definition of joint names (mnemonics) and their associated joint index numbers.

In the present context, commands sent to the low-level joint controller node (via topic /lowLevelJntCmd) come from the behavior server, which includes smooth trajectory profiling. Reactive behaviors, however (such balancing or hand-eye coordination) may communicate directly with the low-level joint controller, rather than passing through the behavior server.

SmoothMove library:
To prevent issuing dangerous, large jumps in joint commands to Atlas, a trajectory filter is used to lead the robot smoothly from an initial pose to some desired final pose. A means to do this is provided by a library in the package “smoothMove.” Note that teamHKU is anticipating upgrading this simple, linear interpolator with a cubic-spline interpolator via existing 3rd-party ROS packages. However, the intent is unchanged—it is desired to lead the robot smoothly towards a goal pose with fast, incremental updates to commands.

The smoothMove library incorporates smooth_joint_space_move.cpp, smooth_finger_joint_space_move.cpp, smooth_cartesian_space_move.cpp, lhand_joint_control.cpp, and rhand_joint_control.cpp. Packages using this library should include headers from this "include" directory. This package owns the file finger_joint_names.h (similar to joint_names.h in low_level_joint_controller). This library is used by the behavior server and by the crawl Jacobian (work in progress).

This library defines the class "smoothJointSpaceMove". An object of this class subscribes to "/atlas/joint_states" to get feedback from Atlas. It also publishes to /lowLevelJntCmd to communicate commands to the low_level_joint_controller.

To perform smooth move trajectory generation with appropriate time scale, the smoothMove object must be informed of the frequency at which it will be invoked. This is done with the member function: setTimeStep(loopFreq).

The main methods of this object are newMoveGoal() and update(). newMoveGoal(jointSelected, qGoals,moveDuration) specifies which joints are to be affected (by the boolean array jointSelected), what the joint-angle goals are (via the array qGoals) and how many seconds the coordinated move should take (via the moveDuration argument). After invoking this method to establish a new goal state (in joint space), the method "update()" should be invoked repeatedly at the prescribed frequency. Array arguments of "update()" get populated with incremented values for joint values and joint velocities. A third argument may be provided: jointSelected[] (a boolean array), in which case only the selected joints and joint velocities are updated. Updates are based on member variables retained within the smoothMove object. At present, updates are simply constant-velocity increments scaled such that all joints arrive at their goal destinations simultaneously at the chosen

arrival time (move duration). An anticipated improvement is to ramp velocity commands up and down, and to provide compatible acceleration commands as well. (as noted, a cubic-spline interpolator is anticipated).

Repeated calls to update() will compute updates to arrays q[] and qDot[]. However, these are not automatically communicated to the low_level_joint_controller. A separate step is required: invoking the member function publishToLowLevelJointController(). This has been separated out so that the behavior server can integrate commands from separate action clients and coordinate multiple trajectories with a single low-level joint controller. This is necessary so that tasks requiring subsets of joints can be executed by separate action clients (e.g., walk and chew gum, or turn a steering wheel and operate the brake pedal, or manipulate a tool while balancing).

publishToLowLevelJointController() comes in three options. With no arguments provided, the internal values of q[] and qDot[] (member variables of smoothMove object) are sent as commands to the Atlas controller. Alternatively, arrays qVec[] and qDotVec[] may be supplied, and these values will be published instead of the internal values. (This is useful if the higher level decides not to use the smoothMove interpolated values). Implicitly, both of these methods assume that the user has control over all of the robot's joints. In fact, use of these functions will change the controller status to enforce that all joints are under user control.

Alternatively, a third argument may be provided: an array of k_effort[]. In this case, the values in k_effort[] will establish if the respective joints are under Boston Dynamics control (value=0) or user control (value=255). In recent releases of the Atlas interface, this field is ignored by the Atlas robot, and one must change the mode via another interface. However, the simulator still requires use of this field to get user control of the joints.

Repeated calls to update() compute incremental updates of q[] and qDot[]. Continued calls to update() after the move is complete does no harm, since the goal values would simply be returned. To tell if a move is complete, invoke the method: moveIsDone(), which returns true (if the move is done) or false.

Since smoothMove subscribes to joint_states, a convenient member function has been provided to request the current state. The method: getSensedAngles(qVec[]) populates the provided array with the current sensed joint angles.

This library also defines the class SmoothFingerJointSpaceMove, which has counterparts of the smoothJointSpaceMove class, applied to Sandia-hand fingers. Since the Sandia-hand finger motions are relatively slow (even at top speed), it is not clear that trajectory profiling is necessary for the fingers.

For finger control, the counterpart of publishJntCmds() is contained within separate classes: rhand_joint_control and lhand_joint_control. Objects of these classes talk directly to the Sandia hands (in simulation or in hardware) without passing through a lowLevelJointController.

The class "smoothCartesianSpaceMove" is intended to enable Cartesian trajectory generation for Atlas's arms. This class is still under construction, but partially in (experimental) use. It mimics the functionality of smooth joint-space moves. A goal destination is provided in Cartesian space. The function "update()" takes arguments of selected joints, joint angles and joint velocities. The joint angle array is updated incrementally using Jacobians, and the corresponding joint velocity array is populated for the involved joints.

A smoothCartesianSpaceMove object should be instantiated with the argument RIGHT_HAND (1)

or LEFT_HAND (0) (and one would typically create an instance of each).

Various approaches to Cartesian-space updates have been initiated, including using subsets of joints for square Jacobians or using pseudo-inverses. This class needs work, particularly with incorporating target hand orientation.

A useful utility method is "get_current_wrist_pose()", which returns a PoseStamped object of the l_hand or r_hand frame (depending on which object was instantiated) with respect to the pelvis. This is useful for feedback when trying to reach a computed Cartesian pose.

The smoothCartesianSpaceMove depends on the Jacobians package.

Jacobians library:
This library defines the class JacobianComputer, which has methods for computing hand Jacobians.

The jacobianComputer has a transform listener, which it uses to obtain relationships among defined frames. Jacobians computed in this class are incomplete, but are already being used. It is also anticipated that this library may be replaced with the “KDL” (Kinematics and Dynamics Library) existing ROS package, but this is still under evaluation.

Notably, the method "computeRightHandJacobians()" populates the member arrays JLinRightHand[3][NJoints] and JRotRightHand[3][NJoints], which are both 3-by-28 Jacobians. The JLinRightHand array of values relates perturbations of each of 28 Atlas joints to perturbations of x,y,z of the r_hand frame with respect to the pelvis. The array JRotRightHand[3][28] contains the values mapping joint velocities to r_hand-frame angular velocity (with respect to the pelvis frame). These arrays are accessible using various member functions, e.g. getRightHandLinJacobian().

Similarly, computeLeftHandJacobians() populates JLinLeftHand[3][NJoints] and JRotLeftHand, which contain the values for the left-hand translational Jacobian and rotational Jacobian, respectively. Note, though, that what is defined in drcsim as a “hand” frame is actually a “wrist” frame. An additional transformation is required to compute the coordinates of a useful site on the palm of a Sandia hand.

Note that, with respect to the pelvis fraem, only 3 torso joints and 4 arm joints affect the 3-D coordinates of a wrist origin. The remaining 21 columns of JLin will contain all zeros. For the rotational Jacobians, 3 torso joints and 6 arm joints affect the orientation of a hand frame. Although sparse, the 3x28 representation is convenient, since all Jacobians fit this size and can be indexed by the defined joint-index names.

The above 2 Jacobian computation functions are believed to be correct, but they should be more thoroughly tested. Improvements, extensions and possible complete replacement are all anticipated.

Elbow and knee Jacobians were begun in this library. However, these are unfinished/untested. These were initiated with the intent of computations in support of crawling. However, these have been incorporated in a separate crawling Jacobian (crawl_jacobian package). It is also unclear if the Atlas robot will be capable of crawling.

To use the hand Jacobians computed here, include dependence in your manifest as: <depend package="jacobians"/>, and include the header file: <jacobians/JacobianComputer.h>

Behavior Server:
The package “behavior_server_v2” provides an “action server” (a concept defined in ROS, similar to “services”) for interfacing to the robot (simulated or physical). The behavior server is primarily useful for issuing pre-scripted motion commands, for issuing finger-motion commands, or for teleoperation of the arms. The behavior server is undergoing redesign, but the current functionality should remain.

The behavior server receives goal requests from action clients, typically resulting in joint motions (either Atlas body or Sandia fingers). The behavior server communicates incremental joint commands to the low-level joint controller. Thus the behavior server always requires that the low-level joint controller node is running. A launch file: *roslaunch behavior_server_v2 behavior_server.launch* will launch both the behavior server and the low-level joint controller.

An object of the behavior server class instantiates an action server, "hkuActionServer", which uses the actionServer goal, result and feedback messages defined in the "action" directory. The goal message is primarily designed to specify a joint-space destination (for the Atlas joints or for a right or left Sandia hand). As such, fields include a jointAngles array, a selectedJoints array, and a move duration. A PoseStamped may be specified for Cartesian-space goals. Additionally, stopEvents may be specified. These contain termination events detectable by an event listener. This capability has been used lightly to date, but it should be valuable in the future for incorporating sensor-based reactions (e.g. move until touch) as well as pause, abort and e-stop signals.

An important field of the goal message is a "behavior_type". These are simple #define codes contained in the /include file: behavior_codes.h. The most commonly used behavior code is: JOINT_SPACE_MOVE. Other useful behavior codes allow opening/closing right/left hands to specified finger poses, performing teleoperated right-hand/left-hand moves, performing left/right Cartesian-space hand (wrist) moves, and setting mappings for BDI vs user joint-control ownership.

The JOINT_SPACE_MOVE behavior expects the goal message to contain an array of selected joints, corresponding goal joint angles, a move duration, and (optionally) stopEvents. The "pose" field will be ignored. This behavior uses a smoothJointSpaceMove object to coordinate smooth joint motion to a specified pose in a specified time. The behavior server invokes updates at a fixed rate (currently 20Hz) and publishes the resulting updated values to the lowLevelJointController. Multiple joint-space goals can be specified to execute concurrently. For example, brake-pedal control may be peformed by right-leg selected joints while steering-wheel control may be performed by request of a separate action client controlling the arm and finger joints. Incrementaly-updated joint command values for Atlas' body joints are assembled in a common joint-command array, which is published to the low-level joint controller.

The behavior types "OPEN_RIGHT_HAND" and "OPEN_LEFT_HAND" do not require specifying any of the other goal parameters. These will cause all fingers of the indicated hand to go to their home angles (0 angles), thus opening the fingers.

The behavior types RIGHT_FINGERS_MOVE and LEFT_FINGERS_MOVE require specifying the goal angles for all 12 joints of the specified hand, as well as a move duration. The remaining fields (stop events, selectedJoints, and pose) do not have to be specified.

The behavior types CLOSE_RIGHT_HAND and CLOSE_LEFT_HAND are experimental special cases. These behaviors do not require specifying any other parameters. The effect is to close the fingers in a specific grasp shape (cylindrical) and to additionally impose maximum torque on the relevant (grasping) finger joints. Once one of these commands has been invoked, it would be necessary to reset the feedforward joint efforts to zeros to open the hand again. This is done by the

complementary OPEN_RIGHT_HAND or OPEN_LEFT_HAND behaviors.

The behavior type RIGHT_WRIST_TELEOP users smoothCartesianSpaceMove to incrementally update joints of the right arm based on a right-arm Jacobian and a specified, desired incremental Cartesian motion. This behavior uses the "pose" field of the goal message--but interprets it as a "twist" (incremental position and orientation commands, equivalent to velocity commands). In a variation, one may specify a subset of joints to be used for position control, and a complementary set to be controlled in joint space (which can use the goal joint angles array for input). The elbow angle should be kept bent, in this mode, to avoid Jacobian singularities. This behavior needs further development and testing.

The behavior RIGHT_WRIST_FOLLOW differs from TELEOP, in that it attempts to converge on a specified Cartesian goal--i.e., absolute position control vs. velocity control. This case was more difficult to implement, since it incorporated non-collocated feedback. The position of the right wrist is obtainable by the smoothCartsianSpaceMove member function "get_current_wrist_pose()". This may be compared to a goal pose, resulting in a Cartesian error computation, which may be used to compute a direction vector for wrist motion. However, since the wrist pose includes dynamics of an oscillating arm, for which the actual joint angles differ from the commanded joint angles, the closed-loop behavior is unstable. A Kalman-style filter was thus constructed with a feedforward model assuming ideal Jacobian incremental progress. The actual wrist pose is used in a low-bandwidth filter to extinguish accumulated error from integration of Jacobian updates. The constant "aFilt" may be adjusted to change the time constant of this filter. Additionally, the wrist velocity for convergence on a goal pose is saturated via the constant "maxDpInc", currently set to 20mm/iteration (0.4m/sec at 20Hz). These values were set empirically and should be better tuned.

LEFT_WRIST_FOLLOW has not been implemented; it should mimic RIGHT_WRIST_FOLLOW, pending further tests and tuning on RIGHT_WRIST_FOLLOW (TODO).

The behavior type SET_K_EFFORT_VALS is useful simply for remapping joint-control authority between BDI and user. Examples of how to use this mode are the action clients action_client_set_bdi_joints and action_client_set_all_user_joint_control in the package action_client_examples.

The behavior type GET_JOINTS_AND_POSES is a convenience. Since the behavior server has access to joint-state and transform data, it makes this information available to clients on request. Invoking this behavior only requires setting the behavior type; all other fields are ignored. The "result" message sent back to the client by the action server will contain all of the joint angles as well as a PoseStamped message for both the right wrist and the left wrist.

More behaviors are expected to be added to the behavior server. However, the main program should be more skeletal, and implementation should be moved to modules. The behavior server is currently over 1,000 lines of code and needs to be refactored (work in progress).

IMPORTANT: there are still issues with the behavior server “hanging”, which seem attributable to overwriting message queues when concurrent tasks conclude nearly simultaneously.

The behavior server should spin off hand control as separate action servers, which will alleviate some of the conflicts. Teleoperation invokes very brief actions repeatedly, which exacerbates the hang-up problem—and thus teleoperation perhaps should be invoked through topics instead of via action clients. Additionally, the source code for the action server should be modified to increase the message queue size to handle the maximum supported number of concurrent goals within the HKU behavior server. (TODO)

Motion scripts and the “playfile” action client:
A user communicates commands to Atlas (physical or drcsim) through the use of “action clients,” which interact with an action server (in our case, the HKU behavior server). Action clients send goal requests and wait for responses from the action server. A very useful action client example is "*playfile*"--recently upgraded to “hmi_playfile”. This node can be found in the “Examples” directory under “action_client_examples”. It can be compiled with: *rosmake action_client_examples.*

The action client “hmi_playfile” takes a command-line argument of the name of a motion file. For example, the directory "record_playback_scripts" contains a variety of useful tricks. To try out an example, do the following:

1) start up the Atlas simulator. In a terminal, enter:
VRC_CHEATS_ENABLED=1 roslaunch atlas_utils qual_task_2.launch
This will start up an interesting example with Atlas positioned in front of a table with an object (a drill). The option “VRC_CHEATS_ENABLED” will let us use a development mode in which the pelvis is “pinned”, thus preventing Atlas from falling down. This is not a mode available for the physical robot, but it is useful in letting us focus on arm motions without having to worry about maintaining balance.

Keep this terminal alive. You can minimize it, as we will not need to interact with it further.

2) pin the pelvis. In a second terminal, enter the command:
rostopic pub --once /atlas/mode std_msgs/String '{data: "pinned_with_gravity"}'
Although we started up the simulator with cheats enabled, the pelvis-pin cheat is not turned on until we say so, using this command. This command will run to conclusion in about 3 seconds, after which you can re-use this terminal.

3) start up the behavior server and low-level joint controller. If necessary, compile these first using the command:
rosmake robot_code
In a terminal (e.g., the terminal from step 2), start up the behavior server and low-level joint controller by entering the command:
roslaunch behavior_server_v2 behavior_server.launch

Tip: Some of these commands get pretty long. It is convenient to use “file completion” as a shortcut. For example, with the above command, you can type:
roslaunch be<tab>
that is, start typing the command, then hit the “tab” key. Linux will find and fill in part of the rest of your command entry for you. In the present case, it will respond with:
roslaunch behavior_server_v2

At this point, you can continue typing, entering the additional characters “be”, such that your command line now looks like:
roslaunch behavior_server_v2 be

again, hit the <tab> key, and Linux will fill in the rest of the typing for you, completing your entry as:
roslaunch behavior_server_v2 behavior_server.launch

Since this is correct, you can now simply hit the “Enter” key to invoke the command. You will

want to keep the behavior server and low-level joint controller nodes running, but you will no longer need to interact with them via the keyboard, so you can minimize this window.

Tip: We end up using quite a few terminals, and this can clutter your screen and frustrate you trying to find a window of interest. You can control some of the clutter by using tabs within a single terminal. E.g., from your first terminal, the top menu bar has a "file" drop-down option. Choose "open tab", and you will get a second terminal within the same window. You can move between these terminals by clicking on the tabs at the top.

4) execute a motion script. In a third terminal (or a new tabbed terminal), navigate to a directory containing motion scripts of interest. You can find some at:
roscd record_playback_scripts
cd trajectories

It is not strictly necessary to move to this directory first, but it will simplify input, as you will not have to specify a directory path as part of naming your file to be played.

From this directory, run the playfile action client. If necessary, first compile the code with:
rosmake action_client_examples

Then try running a motion script by entering the following command:
rosrun action_client_examples hmi_playfile prone.traj

This will run the node "hmi_playfile," which is an action client that is part of the "action_client_examples" package. This node will parse the text file "prone.traj," which is located in the current directory (record_playback_scripts/trajectories), and it will send motion requests to the behavior server. This particular example will cause Atlas to lower his arms to his sides. You can run the above command on an alternative script, "crucify.traj", which will cause Atlas to raise his arms up again in a cross pose (which corresponds to all angles at 0.0).

After all motion requests in a trajectory file have been executed, the hmi_playfile node concludes. For the above two examples, there is only a single destination pose for each. The behavior server accepts the goal poses, breaks up the motion into incremental subgoals, and sends these subgoals to the low-level joint controller fast enough that the motion is executed smoothly.

There are some interesting trajectory scripts in the trajectories directory, such as: kneel, roll_onto_belly, roll_onto_back, prone, and init_crawl_pose_from_prone. However, these were created for use with an earlier Atlas model, and they have not been updated to drcsim version 3 (and thus Atlas will fall down trying to execute some of these). Further, most of these commands will not be relevant with the pelvis pinned.

Motion scripting:
Primitive (but useful) motion commands can be specified by editing a text file. We have recently moved to "yaml" syntax (see, e.g., http://www.yaml.org). However, there are still legacy trajectory scripts in an older (but similar) format. The parser hmi_playfile can accept either style, but future scripts should be written in yaml style.

In addition to moving the arms, as described above with the prone.traj and crucify.traj files, one can control the fingers of the hands. An example is rfingers_prepare_grab_hose.traj, which moves the right-hand fingers to a claw-like pose in preparation for grabbing a firehose.

Scripting motion files requires filling in data to several fields. An example is given below.

#

##
##
Format (yaml compatible):
#
- 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.
0,0.0
]
there are 28 joints to set
unit:
1
0: degree, 1: radian
duration:
1.0
##
##
- hmi_msgs: init
  actionType:
          1
  actionCode:
          0
  selection: [
          #back and neck: b_lbz,b_mby,b_ubx,neck_ay
          1,   1,   1,   1,

          #left leg: l_uhz,l_mhx,l_lhy,l_kny,l_uay,l_lax,
          1,   1,   1,   1,   1,   1,

          #right leg: r_uhz,r_mhx,r_lhy,r_kny,r_uay,r_lax,
          1,   1,   1,   1,   1,   1,

          #left arm:l_usy,l_shx,l_ely,l_elx,l_uwy,l_mwx
          1,   1,   1,   1,   1,   1,

          #right arm: r_usy,r_shx,r_ely,r_elx,r_uwy,r_mwx
          1,   1,   1,   1,   1,   1
        ]

```
  angle: [
         #back and neck: b_lbz,b_mby,b_ubx,neck_ay
          0.00,-0.50,0.00,-0.00,

         #left leg: l_uhz,l_mhx,l_lhy,l_kny,l_uay,l_lax,
         0.00,0.00,-0.40,0.70,-0.20,0.00,

         #right leg: r_uhz,r_mhx,r_lhy,r_kny,r_uay,r_lax,
        -0.00,-0.00,-0.40,0.70,-0.20,-0.00,

         #left arm: l_usy,l_shx,l_ely,l_elx,l_uwy,l_mwx
        -1.00,-1.30,0.20,1.50,0.00,-0.00,

         #right arm: r_usy,r_shx,r_ely,r_elx,r_uwy,r_mwx
        -1.00,1.30,0.20,-1.50,0.00,0.00
       ]
  unit:
          1
  duration:
          0.40
##############################
- hmi_msgs: lean
  actionType:
          1
  actionCode:
          0
  selection: [
           #back and neck: b_lbz,b_mby,b_ubx,neck_ay
          1,   1,   1,   1,

          #left leg: l_uhz,l_mhx,l_lhy,l_kny,l_uay,l_lax,
          1,   1,   1,   1,   1,   1,

          #right leg: r_uhz,r_mhx,r_lhy,r_kny,r_uay,r_lax,
          1,   1,   1,   1,   1,   1,

          #left arm:l_usy,l_shx,l_ely,l_elx,l_uwy,l_mwx
          1,   1,   1,   1,   1,   1,

          #right arm: r_usy,r_shx,r_ely,r_elx,r_uwy,r_mwx
          1,   1,   1,   1,   1,   1
         ]
  angle: [
         #back and neck: b_lbz,b_mby,b_ubx,neck_ay
          0.00,-0.00,0.00,-0.00,

         #left leg: l_uhz,l_mhx,l_lhy,l_kny,l_uay,l_lax,
         0.00,0.00,-1.00,2.00,-0.70,0.00,

         #right leg: r_uhz,r_mhx,r_lhy,r_kny,r_uay,r_lax,
        -0.00,-0.00,-1.00,2.00,-0.70,-0.00,
```

```
        #left arm: l_usy,l_shx,l_ely,l_elx,l_uwy,l_mwx
       -1.90,-1.30,0.50,2.30,0.00,-0.00,

        #right arm: r_usy,r_shx,r_ely,r_elx,r_uwy,r_mwx
       -1.90,1.30,0.50,-2.30,0.00,0.00
      ]
  unit:
        1
  duration:
        0.40
##############################
- hmi_msgs: jump
  actionType:
        1
  actionCode:
        0

  selection: [
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
       ]
  angle: [
       0.00,-0.50,0.00,-0.00,0.00,0.00,0.00,0.00,0.60,0.00,-0.00,-0.00,0.00,0.00,0.60,-0.00,-1.90,-1.30,0.50,2.30,0.00,-0.00,-1.90,1.30,0.50,-2.30,0.00,0.00
      ]
  unit:
        1
  duration:
        0.10
##############################
- hmi_msgs: air
  actionType:
        1
  actionCode:
        0
  selection: [
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
       ]
  angle: [
       0.00,-0.00,0.00,-0.00,0.00,0.00,0.00,0.00,0.60,0.00,-0.00,-0.00,0.00,0.00,0.60,-0.00,-1.90,-1.30,0.50,2.30,0.00,-0.00,-1.90,1.30,0.50,-2.30,0.00,0.00
      ]
  unit:
        1
  duration:
        1.00
##############################
```

All of the lines beginning with the “#” symbol are comments. They are not needed in the yaml file, but they can be helpful to the reader.

Per convention, the yaml file begins with three dashes:

The first keyword, which starts with a single dash, is the message name:

- hmi_msgs: init

One must use the keyword “hmi_msgs:”, but the name following it should be chosen by the user to be something descriptive and meaningful.

The next two fields specify an action type and an action code.

```
 actionType:
        1
 actionCode:
        0
```

The actionType “1” is the most common, specifying coordinated joint-space moves of Atlas' body. Codes 4 and 5 specify right and left finger move commands, respectively. A list of command codes can be found in the include file “behavior_codes.h” within the behavior_server_v2 package.

The actionCode is almost always 0. (exceptions to be documented later).

The “selection” field start with the keyword:

```
    selection: [
```

followed by a list of ones or zeros. A “1” indicates that the corresponding joint should be moved, and a “0” implies the corresponding joint shoud be left at its current location. Most commonly, one uses all 1's and specifies all joint angles. However, it can be important sometimes to separate functions. For example, we may have one node responsible for turning a steering wheel and a separate node responsible for operating a brake or accelerator pedal. These are separate functions (steering vs speed control) that are conveniently subdivided into separate control nodes. One could use the selection choices to choose to control, e.g., the right arm while leaving other joints under the control of other nodes.

For the above move labeled “init”, the selection choices are grouped in 5 logical groups: the torso/neck, left leg, right leg, left arm and right arm. These are separated by newlines, a blank space and a comment listing the joint names. None of this formatting is required. Alternatively, the move “air” contains none of this formatting. It is the user's option whether to add such formmating and comments for readability.

The next field to be completed is:

```
 angle: [
```

followed by a list of 28 angles. (For hands, only the first 12 angles are used, and the trailing values are ignored, since there are only 12 finger joints per hand on the Sandia hands). The values are listed in the same order as the joint selections, which is the same order as the atlas_command data structure (labelled as per the figure provided in earlier documentation). The angles can be listed in units of degrees or radians—but the unit choice must be declared in the subsequent field.

```
 unit:
    1
```

The above implies units of radians. An entry of 0 implies degrees. This field is being change to accommodate entries of “rad” or “deg” to be more clear.

Finally, the field:

```
duration:
  1.0
```

Specifies that the robot should reach the goal pose with a move time of 1.0 seconds.

Existing *.traj files may be either in the current yaml format or in an older text-style format. If hmi_playfile attempts to operate on an older-style format, it will issue a warning, but it will nonetheless parse and execute the older format.

Other Action Client Examples:
The action_client_examples package contains some additional action-client examples for use with the hku behavior server. An action client assumes that Atlas (simulator or hardware) is running and that the behavior server and the low-level joint controller nodes are running.

Some simple examples invoke atomic (one-shot) goals, run to completion and return. The program action_client_close_right_hand.cpp demonstrates the experimental, special-case hand behaviors: CLOSE_RIGHT_HAND, CLOSE_LEFT_HAND--which form a cylindrical grasp and exert maximum joint torques, and the complementary behaviors OPEN_RIGHT_HAND, OPEN_LEFT_HAND.

The example *action_client_set_bdi_joints* and the complementary *action_client_set_all_user_joint_control* show how to invoke remapping of authority over the Atlas joints. These can be useful, e.g., upon start-up. If *atlas_utils keyboard_teleop.launch* is launched, this program will take control of the joints for walking/balancing. Upon launching *behavior_server_v2 behavior_server.launch*, the low-level joint controller node will take over control of all of the joints. To reinstate control to BDI, run a*ction_client_set_bdi_joints*, after which the keyboard teleop function will run again.

Action_client_nodder is a simple example of sending commands to a joint subset--in this case, causing the head to "nod" repetitively. This action client may be run simultaneously with other action clients, such as *lhand_grabber, action_client_close_right_hand*, etc, illustrating how multiple tasks can be run concurrently. Note, though that it is up to the user to make sure that there are not multiple action clients fighting for control over the same joints. The behavior server does not preclude this case, and the results are unpredictable.

Additional examples show how to respond to event triggers (see *action_client_example*, which uses the right-hand force/torque sensor to trigger a contact event, to which the arm motion responds).

This introduction only describes how to perform pre-scripted motions. Reactive or sensory-driven behaviors will be introduced in a later document.