Update on DRCsim 2.5 code changes

Wyatt Newman, May 18, 2013

With the release of drcsim 2.4, the previous interface to “atlas” was changed, making our earlier code obsolete. Part of the motivation for this was to allow use of the Boston Dynamics walking/balancing action server. Code revisions to accommodate these changes also prompted general reorganization of files. A summary of this reorganization follows.

Low Level Joint Controller:

The package "low_level_joint_controller" defines a node that is responsible for direct communications with the atlas robot (or simulator). It currently runs at 100Hz, publishing position and velocity commands to /atlas/atlas_command.

Typically, a user would not communicate directly with this node. Rather, commands to this node should come from the hku behavior server. Correspondingly, one would not typically run this node independently. Instead, launch it together with the behavior server, using:
roslaunch behavior_server_v2 behavior_server.launch

The low_level_joint_controller node subscribes to /atlas/atlas_state to get sensor feedback. This sensor feedback, however, is currently only used on start-up. The robot's current joint angles (and efforts) are used to specify initial values for joint commands. For a "bumpless" transition to low-level joint control, the joint sensor values are averaged for 3 seconds, as well as the joint efforts. The joint commands for bumpless transition are computed as:

theta_cmd = theta_sensed + effort/Kp.

As a result, the low-level joint controller can assume command without gravity droop.

This node subscribes to the (hku) topic /lowLevelJntCmd, which receives messages of type hku_msgs::lowLevelJointController, as published by a higher level. This message now includes the field k_effort[], which specifies if a joint is to be controlled by BDI or by the user.

It may be useful, in the future, to extend this message to allow specification of feedforward joint efforts (torques) as well. (TODO)

It may also be desirable to modulate joint gains in the future from a higher level. This would come from a separate topic. (TODO)

On start-up, the lowLevelJointController sets values for kp_position, kp_velocity and kd_position. (see the file "init_controller.cpp"). The gains kp_position and kd_position are estimated from analyzing bag files while under BDI control (although a few of these gains are still uncertain).

The values of kp_velocity are more mysterious. Per OSRF, these values should affect inherent joint damping (viscous friction). It is not clear what these values should be.

Also, velocities from Atlas_state are quite noisy, apparently estimated from position differences. Large values of kp_velocity or kd_position will amplify this noise. It might be useful to put a Kalman filter in the lowLevelJointController and perform additional damping in from this node. (TODO)

For the present, this node merely passes along position and velocity commands (and, optionally, k_effort selections) from a higher level. It is anticipated that high-speed (servo-level) algorithms will get included in this node: e.g. compliant motion control/force-feedback and active balancing.

(TODO)

On start-up, this controller sets ALL joints to user control. If using with the BDI controller, the order of start-ups may matter.

Reassigning joints for control by BDI or by the user should be done via action clients. There are two illustrative examples in the package "action_client_examples":
action_client_set_all_user_joint_control.cpp and *action_client_set_bdi_joints.cpp*.
Either of these may be edited to reassign joints as desired, or the code may be emulated for inclusion in an action_client application for mixed BDI/user joint control.

The low_level_joint_controller package also 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>

Typically, commands sent to this node (via topic /lowLevelJntCmd) would come from the behavior server, which includes smooth trajectory profiling. An exception is the "crawl jacobian", which is under development. It is unclear if future crawling will communicate directly with the lowLevelJointController node, or if it will get integrated within the behavior server.

ODDITY: in roslaunch atlas_utils vrc_task_2.launch, Atlas is initialized standing stably in the starting pen. However, as soon as low_level_joint_controller is launched, Atlas begins to slip around, as though the surface were ice. The reason for this is currently unknown, but problematic!

SmoothMove library:
This package contains code for smooth trajectory profiling. It has been extracted from work formerly coupled with original Atlas control code and subsequent behavior-server code. This is now a library, incorporating 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.

This library defines the class "smoothJointSpaceMove". An object of this class subscribes to "/atlas/joint_states" to get feedback from Atlas. (TODO: update this to /atlas/atlas_state?). 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 improvement would be to ramp velocities up and down (TODO).

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 BDI control (value=0) or user control (value=255). Blended control (values between 0 and 255) are also possible, but this has not been tested to date. Note that this function can be used to simply to remap ownership of the joints. However, if using the function for this purpose, it is still necessary to specify arrays of joint angles and joint velocities. These should be set to the current joint-angle commands and zero velocities, so the effect will be to reassign ownership, not move the joints.

Repeated calls to update() compute incremental updates of q[] and qDot[]. Continued calls to update() after the move is complete does not 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. This class does not need to consider mixed BDI/user control. At present, the simulated maximum finger velocities are very slow, and thus simply commanding a set of goal angles without interpolated trajectories is adequate. Trajectory profiling may be more important with the actual hardware.

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. If there is a need to perform high-speed feedback of the Sandia-hand fingers, a lowLevelHandController node may be introduced.

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. (TODO).

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 (TODO), but are already being used.

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 that only 3 torso joints and 4 arm joints affect the location of the origin of a hand frame; 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.

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).

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” has been reorganized. Trajectory generation is now its own library (smoothMove), as well as Jacobians, and low-level interfacing to Atlas is now via the separate node lowLevelJointControl. Former action-client examples now reside in a separate package: action_client_examples. New cases have been added to the behavior server.

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. Although the low-level joint controller can be useful on its own, the behavior server assumes availability of a low-level joint controller node. 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 should be valuable 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 specify 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. The LEFT_WRIST_TELEOP has not been implemented. (TODO)

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 (TODO).

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 (TODO).

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)

Action Client Examples:
This package contains some 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.

The example *action_client_elbow_test.cpp* illustrates partial user control. Two joints of the right arm are enabled for user control, and the rest are delegated to BDI. The action client repeatedly moves the right arm back and forth while the BDI walking program is still functional. (Note: having trouble getting robot to walk in vrc_task_2; try this in vrc_task_1 instead). Example start-up process:
roslaunch atlas_utils vrc_task_1.launch
roslaunch behavior_server_v2 behavior_server.launch
rosrun action_client_examples action_client_set_bdi_joints
roslaunch atlas_utils keyboard_teleop.launch

The above will assign joint control to the user (low-level joint control), but the action client will reassign joint control to BDI. The keyboard teleop program will then allow sending walking commands to the robot. Running *rosrun action_client_examples action_client_elbow_test* will then assign two right-arm joints to the user and cause the arm to swing while BDI balancing/walking is being performed.

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).

A very useful action client utility is "*playfile*". This action client takes a command-line argument of a motion file. For example, the directory "record_playback_scripts" contains a variety of useful tricks. From this directory, run:
rosrun action_client_examples playfile ./home.txt
This will run the "playfile" action client, which parses the text file "home.txt" to obtain joint selections, goal angles and move duration for a smooth joint-space move. Playfile uses this data to

send a corresponding goal to the behavior server, then "playfile" concludes. Multiple, sequential moves may be specified in such text files. Examples include: kneel, roll_onto_belly, roll_onto_back, prone, and init_crawl_pose_from_prone.

At present, "playfile" only handles body joints as a smoothJointSpaceMove behavior.

The file format for these script files follows from Zongwei's GUI programmer text format. Although not tested yet, "playfile" should work on programs composed using the GUI programming interface (TODO).

Further details on the motion scripts can be found in the record_playback_scripts directory.

Motion scripts:
This directory contains text files describing Atlas body motions. The format for these scripts follows Zongwei's GUI record/playback format. The parser within the action client example "playfile" is adapted from Zongwei's broker code for playing GUI-programmed robot actions.

The playfile utility is an action client. It requires the behavior server and the low-level joint controller to be running. As an example, from the current directory, enter the command line:
rosrun action_client_examples playfile ./home.txt

This will send a single joint-space move command to the behavior server. The contents of home.txt are:
hmi_msgs:home
actionType:
1
#0;1;2;3; 4;5;6;7;8;9; 10;11;12;13;14;15; 16;17;18;19;20;21; 22;23;24;25;26;27
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;1

#0;1;2;3; 4;5;6;7;8;9; 10;11;12;13;14;15; 16;17;18;19;20;21; 22;23;24;25;26;27
Angle:
0;0;0;0; 0;0;0;0;0;0; 0;0;0;0;0;0; 0;-1.5;0;0;0;0; 0;1.5;0;0;0;0
Duration
5

The parser looks for the keywords hmi_msgs:, actionType:, selection:, Angle:, and Duration

It is helpful, but not required, to put a mnemonic hame with hmi_msgs: (e.g. "home" was used above).

The actionType is specified as "1", corresponding to a smooth joint-space move. Consistent with the GUI text output format, an actionType of "0" also implies smooth joint-space move, by default. At present, the playfile action client ONLY understands smooth joint-space moves. (Even finger motions are not yet handled).

Extra lines may be inserted as comments. In the above, the line:
#0;1;2;3; 4;5;6;7;8;9; 10;11;12;13;14;15; 16;17;18;19;20;21; 22;23;24;25;26;27
is merely to help the programmer keep track of the joint index numbers when assigning joint selections.

Following the keyword "selection" is a list of which joints are to be moved (1 for true, 0 for false).

There are 28 of these; the extra spaces help to group the joints: 4 back/neck; 6 left leg; 6 right leg; 6 left arm; 6 right arm.

The next line,
  #0;1;2;3; 4;5;6;7;8;9; 10;11;12;13;14;15; 16;17;18;19;20;21; 22;23;24;25;26;27
is again a comment, intended to help the user keep track of joint numbering when specifying angle goals.

After the keyword "Angle" is a list of the desired goal angles. This list must include 28 values, although only the selected joints will actually be affected. One may thus enter dummy values, e.g. 0's for non-selected joints. Extra spaces may be inserted to help readability. In the above example, the specifications are grouped into 5 groups (torso/head, two legs, two arms).

Finally, the keyword "Duration" is used to set the move duration, in seconds. In this example, the move time is specified as 5 seconds (which must appear on the line following the "Duration" keyword). In Zongwei's format, the move duration is repeated 28 times. This does not appear to be necessary, but compatibility should be tested. (TODO).

One can "comment out" an entire move by putting a #-sign in front of "Duration"

In a more interesting example, the file "kneel.txt" specifies 7 sequential, coordinated motions. It starts with the robot standing upright, and slowly descends to a squat with wrists forward. In this move, the pelvis is kept vertical to be compatible with Gazebo's "reset model poses" menu option, which insists on resetting Atlas's pelvis to a vertical orientation. The robot squats while maintaining its c.g. over its ankles and its pelvis vertical. After reaching maximum flex of the hips, the robot leans forward and lands on its wrists. The fall at the end (onto the wrists) does not seem to trip the DARPA "damaging fall" penalty, but it does seem to induce some VRCScoringPlugin LogRecord response.

Some other useful scripts in this directory are: roll_onto_belly; roll_onto_back; and init_crawl_pose_from_prone.

Center of Mass Computations:
The package “pub_atlas_com” contains code written by Kenneth Tjoeng (modified slightly by wsn). This package is almost identical to the package “CenterOfMass”, except that pub_atlas_com also publishes its results to a topic (CenterOfMass).

This node accesses the Atlas IMU to get orientation and uses a transform_listener to compute the contributions of all links to the center of mass of the Atlas robot. Mass-property information obtained from the atlas URDF file is hard-coded into this program. The URDF should be re-evaluated for possible changes. (TODO). More generally, the applicable URDF could be parsed for the required information (though this may be hard to generalize).

The result of the COM computation is published to topic CenterOfMass as a geometry_msgs, PointStamped. Publications are updated at 10Hz. Results are also printed using ROS_INFO().

Wsn added a variation, outputting COM coordinates relative to the right ankle.

Wsn did some testing of this while writing the "kneel.txt" script. The COM computation seems to be slightly off (perhaps comparable to BDI walking program errors?). Slight joint motions while balancing upright were made while monitoring the ankle force/torque sensors using:

rxplot /atlas/atlas_state/r_foot/torque/y,/atlas/atlas_state/l_foot/torque/y

It was found that ~zero ankle torque did not quite correspond to computed COM above the ankles. During kneeling (with ~zero ankle torque), the COM computation was sometimes forward and sometimes behind the actual balance point—with errors on the order of 1cm.

TODO: need to find the discrepancy to use this program for balance control.

Crawl Jacobian:
This package contains code under development for the purpose of producing crawling motions. It is intended for use with contacts at the wrists and knees. Crawling on elbows and knees was originally considered, but it produced a spine angle that severely restricted the view of the sensor head. Further, using the wrists as contact points provides more mobility, as the elbow angles can be employed.

A crawl Jacobian for no-slip, swaying motions: theory
As a precursor to crawling, it is desired to be able to maintain 4-point contact (wrists and knees) with no slip between ground and the contact points, but still be able to displace the pelvis. It is kinematically possible to rock forward/backward, side-to-side, and even rotate heading (yaw) while maintaining no-slip conditions at the contact points. In fact, this can be achieved (within joint range limitations) using only 11 of the 28 body joints.

With elbow and knee contacts, it is clear that the knee angles and the two ankle angles are irrelevant (as long as they are kept bent and out of the way). Similarly, the wrist joints (l_arm_mwx and r_arm_mwx) should be kept maximally flexed to try to keep the hands from touching the ground. The neck should be bent backwards maximally to permit the best forward viewpoint for the sensor head. This eliminates 9 of the 28 joints from consideration. Forearm rotation (l_arm_uwy and r_arm_uwy) can be used to help achieve desirable contact orientations of the wrists, but these degrees of freedom do not affect the wrist origins. Further, rotations of the upper arms (about the equivalent of the humerus bone major axes), r_arm_ely and l_arm_ely, are primarily useful for helping to place the wrist locations at about shoulder width. As an approximation, these joint angles can be frozen at a useful angle. Although it may be useful in the future to employ these angles to help place the wrists at attractive contact sites, at present it is adequate to freeze these joint angles.

For the legs, three joints affect the position of each knee: (right and left) leg_uhz, leg_mhx andleg_lhy. However, position of the knee with respect to the hip is approximately constrained to the surface of a sphere, which depends on only two degrees of freedom (e.g. azimuth and elevation at constant radius). Intuitively, rotation about the (equivalent of) the femur does not change the position of the knee origin. When in a crawling pose (for which the leg_lhy is flexed, e.g. drawing the knees towards the chest), then the motions of l_leg_mhx and r_leg_mhx have little influence on the knee origin positions, as these primarily produce rotations about the femur. It is thus an adequate approximation to restrict consideration to two hip joints (leg_uhz and leg_lhy) on each side, while freezing l_leg_mhx and r_leg_mhx to useful values.

The above arguments reduce consideration to of knee and wrist placements to three arm joints (each side) and 2 hip joints (each side)--plus up to three torso motions. It is less obvious that it is adequate to freeze two of the three torso degrees of freedom (back_mby and back_ubx) and still achieve rocking, swaying and yawing while maintaining no-slip conditions at the wrists and knees. It may be desirable to utilize these degrees of freedom, but a simple place to start is to freeze them at their home angles (0 angles). It is necessary, however, to retain use of the back_lbz joint, i.e. rotation about the spine (with the back kept straight). This results in 11 joints used for swaying in place.

We may visualize the problem as follows. Consider some initial pose at which we have 4-point contact (wrists and knees), with 17 of the joints locked at prespecified angles (the joints listed above as being held fixed during crawling motions). Consider the "utorso" frame, which has its z-axis pointing along the spine, from pelvis to neck. At the initial pose considered, the utorso frame may be described in the world frame through 6 values (origin coordinates and orientation angles). Additonally, the four contact points each have 3-D coordinates in the world frame. Now, consider a perturbation of the utorso frame—in general, in 6-D (three translations and 3 rotations). It is desired that, in doing so, all four of the contact points remain stationary (no-slip condition).

However, this scenario corresponds to trying to satisfy 12 constraints (4, 3-D contact points) with only 11 variables (the 11 selected joints). Typically, there will be no solution, as the problem is over-constrained. Instead, remove one of the constraints: specifically, let the z-coordinate (elevation) of the utorso origin be variable, but impose two horizontal translations and 3 rotations. It is still desired that the 4 contact points retain their same x,y values, but the z constraint is that all 4 contact points should have the same z value (corresponding to 3 constraints). The result is a problem in 11 variables with 11 constraints, which is prospectively solvable.

This proposed problem may be solved by breaking it up into two sub-problems: first solve for the 4 hip angles, then solve for the arm angles. The hip sub-problem may be addressed as follows:

Crawl Jacobian hip and back-joint sub-problem solution:
Consider the sub-system consisting of the utorso frame, from which branches the pelvis (via the back_lbz joint) and right and left knees (via right and left leg_uhz and leg_lhy joints). The following notation definitions will be used.

R_u/w is the orientation matrix of the utorso frame with respect to the world frame
O_u/w is the origin (3-D point) of the utorso frame with respect to the world frame
r_rk/w is the origin of the right knee (3-D point) with respect to the world frame
r_lk/w is the origin of the left knee with respect to the world frame
r_rk/u is the origin of the right knee with respect to the utorso frame
r_lk/u is the origin of the left knee with respect to the utorso frame
O_dot_u/w is the velocity of the utorso-frame origin w/rt the world frame
Rdot_u/w is the time derivative of R_u/w, d/dt(R_u/w)
omega_utorso is the angular velocity vector of the utorso frame w/rt the world frame
S(omega_utorso) is the skew-symmetric matrix of omega_utorso, equivalent to omega cross
then Rdot_u/w = S(omega_utorso)*R_u/w
v_fwd_vec is a unit direction vector defined (by the user) as the "forward" direction
v_lat_vec is a unit directon vector defined as lateral (sideways) motion
v_elev_vec is an elevation direction vector

The three defined direction vectors must be orthogonal, with (v_fwd)x(v_lat) = v_elev
It would be natural to define the "elevation" vector as the surface normal of the local tangent plane of the ground, and the forward vector as parallel to the ground, e.g. the projection of the utorso z-axis (the robot's spine) onto the local ground tangent plane. If the ground is horizontal, the elevation vector will be perpendicular to gravity.

The three defined direction vectors may be organized as columns in a 3x3 matrix as:
V_dir = [v_fwd_vec | v_lat_vec | v_elev_vec]

In the present approach, the user specifies the desired angular velocity of the utorso frame. A logical choice is (0,0,0), which will keep the shoulders parallel to the ground and will keep the angle of the spine fixed with respect to the ground. This would be useful for helping to minimize

motion blur from the sensor head.

The user also specifies the forward and lateral velocities: scalars v_fwd and v_lat. However, the elevation velocity, v_elev, must be left free to be solved as a consequence of satisfying the 5 utorso specifications while achieving no slip at the four contact points.

With the above definitions, we can state the coordinate-frame transform relationship as two equivalent ways of specifying the position of the right and left knees in the world:
r_rk/w = O_u/w + R_u/w * r_rk/u
r_lk/w = O_u/w + R_u/w * r_lk/u

It is desired that, in the world frame, the two knee contact points remain stationary (no-slip). Thus, we can impose the condition:
0vec= d/dt (r_rk/w) and
0vec= d/dt (r_lk/w)
where 0vec is defined as a 3x1 column vector of zeros.

But differentiation of the coordinate-frame transform relation yields the following 6 equations:
d/dt (r_rk/w) = O_dot_u/w + Rdot_u/w * r_rk/u + R_u/w* (d/dt r_rk/u)
and
d/dt (r_lk/w) = O_dot_u/w + Rdot_u/w * r_lk/u + R_u/w* (d/dt r_lk/u)

Considering motions of only the back rotation back_lbz and two hip motions of the right leg, r_leg_uhz and r_leg_lhy, one can compute a Jacobian for the motion of the right-knee frame origin with respect to the utorso frame corresponding to:
d/dt r_rk/u = J_rk/u * qdot_r
where qdot_r is a 3x1 column vector consisting of the joint velocities of r_leg_uhz, r_leg_lhy and back_lbz.

Similarly, a left-knee Jacobian relates qdot_l, consisting of l_leg_uhz, l_leg_lhy and back_lbz, to velocity of the left-knee origin with respect to the utorso frame:
d/dt r_lk/u = J_lk/u * qdot_l

At any instant, all joint angles of the robot are known from sensor values. The following quantities are all computable from available information at any robot pose:
r_rk/w , r_rk/u, r_lk/w, r_lk/u, J_rk/u, J_lk/u

We can also say something about O_u/w and R_u/w. From the IMU of the pelvis, we have an estimate of the pelvis orientation in the world, R_p/w. Coordinate transforms are also available (from the ROS transform listener) such that we know R_u/p (orientation of the utorso frame with respect to the pelvis frame), from which it follows: R_u/w = R_u/p * R_p/w.

Assuming the user-defined direction vectors have been expressed with respect to the world, we can say that:
O_u/w = v_vwd * v_fwd_vec + v_lat* v_lat_vec + v_elev*v_elev_vec
The first two terms on the right-hand side are know (by specification), but the scalar v_elev is unknown.

Given specification of the desired utorso angular velocity (desirably, [0;0;0], but may be specified to other values), the skew-symmetric matrix S(omega_utorso) follows. Consequently, Rdot_u/w follows from:
Rdot_u/w = S(omega_utorso)*R_u/w

Combining the above, and rearranging such that known quantities are on the left and unknown quantities are on the right yields:
-v_vwd * v_fwd_vec - v_lat* v_lat_vec - Rdot_u/w * r_rk/u =
v_elev*v_elev_vec + R_u/w*J_rk/u*qdot_r

and:
-v_vwd * v_fwd_vec - v_lat* v_lat_vec - Rdot_u/w * r_lk/u =
v_elev*v_elev_vec + R_u/w*J_lk/u*qdot_l

For convenience, define the known quantities:
b_r = -v_vwd * v_fwd_vec - v_lat* v_lat_vec - Rdot_u/w * r_rk/u
b_l = -v_vwd * v_fwd_vec - v_lat* v_lat_vec - Rdot_u/w * r_lk/u

Also, name the columns of the knee Jacobians as:
[j_rk_uhz | j_rk_lhy | j_rk_lbz] = J_rk/u
That is, the column j_rk_uhz is a 3x1 vector of influence from hip-joint r_leg_uhz velocity on right-knee frame origin velocity, 3x1 vector. Similarly, the influence vectors for r_leg_lhy and back_lbz are defined.

For the left knee, name the 3 columns of the knee Jacobian as:
[j_lk_uhz | j_lk_lhy | j_lk_lbz] = J_lk/u

With these definitions, set up the 6x6 matrix equation:
b_vec = M*x_vec
where: b_vec = [b_r; b_l] (interpreted as the 6x1 vector with b_r stacked on top of b_l)
x_vec = [qdot_r_uhz; qdot_r_lhy; qdot_l_uhz; qdot_l_lhy; qdot_back_lbz; v_elev]
which is a 6x1 velocity vector consisting of the 4 involved hip joints, the back_lbz back joint, and the unspecified vertical velocity of the utorso origin.

The 6x6 M matrix is then:
[j_rk_uhz, j_rk_lhy, 0vec, 0vec, j_rk_lbz, v_elev_vec]
[0vec, 0vec, j_lk_uhz, j_lk_lhy, j_lk_lbz, v_elev_vec]
using the above definitions for 3x1 column vectors, and defining 0vec as a 3x1 vector of zeros. All of the quantities in M are computable at a given pose, or specified (v_elev_vec).

The result is an equation of the form: b_vec = M * x_vec.

This equation may be solved for x_vec, yielding the 4 hip-joint velocities, the back-rotation velocity and the utorso-frame vertical velocity. The method used in the crawl_Jacobian_class_v3 code is the Eigen linear-algebra method “partialPivLu()”; other options are available as well.

It is thus shown that, for the problem specification (direction vectors, specified forward and lateral velocities, and specified utorso-frame angular velocity vector), corresponding hip and back-joint velocities can be computed. A side effect is determination of the utorso-frame vertical velocity consistent with the specifications and no-slip condition of the knees. With this information, the utorso frame velocity, both translation and rotation, is fully specified. This information may be used to solve for the corresponding, consistent arm-joint velocities that satisfy the no-slip condition for the wrist contact points. (nb: the present analysis actually considers no-slip with respect to the frame origins of the knees and wrists. The actual contact points are displaced from these origins, and this variation should be considered, but is not treated here).

Crawl Jacobian arm-joint sub-problem solution:
Having solved the hip/back-joint sub-problem, the velocity of the utorso frame (both translational and rotational) is fully specified. It is desired to find arm-joint velocities (three right-arm joint velocities and three left-arm joint velocities) consistent with the specified utorso-frame motion and the no-slip condition for wrist contact points with respect to the world frame. The analysis here follows the logic of the previous (hip-joint) section.

The coordinate-frame transformation relationships for the left and right wrist are:
r_rw/w = O_u/w + R_u/w * r_rw/u
r_lw/w = O_u/w + R_u/w * r_lw/u

Where the left and right wrist vectors above are defined as:
r_rw/w is the origin of the right wrist (3-D point) with respect to the world frame
r_lw/w is the origin of the left wrist with respect to the world frame
r_rw/u is the origin of the right wrist with respect to the utorso frame
r_lw/u is the origin of the left wrist with respect to the utorso frame

It is desired that, in the world frame, the two wrist contact points remain stationary (no-slip). Thus, we can impose the condition:
0vec= d/dt (r_rw/w) and
0vec= d/dt (r_lw/w)
where 0vec is defined as a 3x1 column vector of zeros.

Differentiation of the coordinate-frame transform relation yields the following 6 equations:
d/dt (r_rkw/w) = O_dot_u/w + Rdot_u/w * r_rw/u + R_u/w* (d/dt r_rw/u)
and
d/dt (r_lw/w) = O_dot_u/w + Rdot_u/w * r_lw/u + R_u/w* (d/dt r_lw/u)

Considering motions of three right-arm joints, r_arm_usy, r_arm_shx and r_arm_elx (two shoulder joints and the elbow joint of the right arm) and three left-arm joints, rl_arm_usy, l_arm_shx and l_arm_elx, we can define two (left and right) 3x1 joint-velocity vectors, qdot_r_arm, qdot_l_arm. The velocity of the wrists with respect to the utorso frame follows from a pair of 3x3 arm Jacobians:
d/dt r_rw/u = J_rw/u * qdot_r_arm
d/dt r_lw/u = J_lw/u * qdot_l_arm

Substituting in the above Jacobians results in two, independent, 3x3 matrix equations:
-O_dot_u/w - Rdot_u/w * r_rw/u = (R_u/w* J_rw/u)*qdot_r_arm
and:
-O_dot_u/w - Rdot_u/w * r_lw/u = (R_u/w* J_lw/u)* qdot_l_arm

both of which are of the form:
b_vec = M*x_vec,
where b_vec is a known 3x1 vector, M is a 3x3 matrix of known values, and it is desired to solve for the 3x1 vector qdot. This can be done with a 3x3 matrix inversion, thus yielding the consistent three right-arm joint velocities and 3 left-arm joint velocities.

Implementation:
The above theory is implemented in the class definition “crawl_Jacobian_class_v3.cpp”. An object of this class has a transform listener, which provides all of the required relative vectors and orientation matrices. The parent function must provide R_pelvis_wrt_world from IMU

measurements—or from an alternative world-orientation means (e.g., IMU supplemented with visual odometry and a feedforward model of body motion from joint velocities). The necessary knee Jacobians are computed with member functions compute_JRKnee_wrt_utorso and compute_JLKnee_wrt_utorso. Wrist Jacobians are computed via the member functions compute_JRWrist_wrt_torso() and comptue_JLWrist_wrt_torso(). These computations also yield the necessary vectors r_wrist_origin_wrt_utorso_, etc. The method computeKneeJacobians() assembles and solves the 6x6 matrix equation for hip-joint velocities and back-joint velocity (and utorso vertical velocity) from user-specified utorso-frame angular velocity vector and specification of 2 out of 3 utorso origin translational velocity components (forward and lateral desired velocities).

The nominal utorso angular velocity is (0,0,0), which would prevent the sensor head from tilting. However, specifycation of a "yaw" angular velocity would be appropriate for turning. Further, a pitch velocity may be needed if the robot needs to raise up (e.g. to clear an obstacle or get a better view) or tilt down (e.g. to avoid singularities from fully extended arms).

The method computeWristQdots() solves for the consistent arm-joint velocities, given a fully-specified utorso origin velocity and angular velocity. As such, this function must be invoked *after* having solved for the hip/back joint velocities, from which the utorso vertical velocity follows.

Crawling with satisfaction of no-slip conditions is constrained by joint-angle range limitations. The function jointCmdsInRange() tests specified angles of a specified set of joints for validity. If any of the joints of interest have violated upper or lower angle constraints, the function returns "false" and the vector of booleans, saturatedJoints[], indicates which joints are out of range. When the crawl Jacobian is used for no-slip body motions, this function should be invoked to test for joint limitations. When a joint limit is reached, it is no longer possible to move as prescribed while satisfying no-slip contact conditions. In such cases, a higher-level function must decide what to do next (e.g., to lift a limb and reposition it, or to shift the body in a different, viable direction).

Test harness:
The program "crawl_Jacobian_v3_test_main.cpp" is a test program that also illustrates use of the crawl Jacobian. This module gets compiled with crawl_Jacobian_class_v3.cpp and executes as a node. It requires that a "low_level_joint_controller" node is running. (A behavior server may be running as well, but is not needed for this test routine).

Using the crawl Jacobian requires specifying 5 velocity components, which appear near the top of the test main program:

```
  double v_fwd= 0.01; //0.01; //specify desired fwd vel;
  double v_lateral= -0.01; //-0.01; // specify desired lateral velocity
  // specify desired rotation rate of utorso frame:
  double omega_yaw = 0.0;
  double omega_pitch = 0.0;
  double omega_roll = 0.0;
```

A smoothJointSpaceMove object is instantiated, although this object is only used to access joint state data and to send commands to the low-level joint controller. The crawl Jacobian computes incremental motions, so joint-space profiling is not needed from smoothJointSpaceMove.

The test main program subscribes to Atlas's IMU data to establish the (presumed) orientation of the pelvis with respect to the world frame.

Desired motion direction vectors are defined as follows. The elevation direction is [0;0;1], with the assumption that the ground is horizontal. The "forward" direction is defined as the projection of the

robot's spine (the z-axis of the utorso frame) onto the horizontal plane. The lateral direction follows from the cross product of elevation vector into forward vector, comprising a consistent right-hand coordinate system.

In the main loop, joint velocities of the 11 involved joints are computed, consistent with no-slip contact conditions and the desired utorso-frame velocity. Corresponding incremental joint commands are published to the low_level_joint_controller. Joint-range limits are tested each iteration. User input/output is in place for single stepping through iterations, though this may be commented out. The test program terminates when a joint limit violation is detected.

Testing:
Only cursory testing has been performed. The robot is placed into a viable initial pose for crawling (see e.g. init_crawl_pose.txt script file, excecutable by “playfile”, described above). The crawl_Jacobian_v3_test_main program is run and the knee/wrist positions are monitored. In simulation, gravity may be set to a low value (e.g. -0.2) so that joint deflections from gravity droop to not affect the test results. The current test parameters call for a velocity of 1cm/sec with an update rate of 10Hz. Thus, the robot should move by 1mm each iteration. Imperfect transform lookups (e.g., not all transforms at the same time stamp) and imperfect joint servoing result in motion errors comparable to the 1mm desired increments. However, repeating the iterations for enough (e.g. 50) iterations produces net motions that should be consistent (e.g., 5cm motion over 50 iterations).

Ideally, these motions would be interpreted in world-frame coordinates, in which case all four contact-point displacement vectors should be identical. Unfortunately, the Atlas robot in the Gazebo simulator “skates” around the floor, and thus the heading vector keeps changing. Wrist and knee motions may be interpreted instead in the utorso frame. In this frame, all four contact point displacements (cumulative over N iterations) should be identical to each other. From the utorso frame perspective, the ground seems to move and rotate relative, but the four contact poitns should move together as a solid body.

Forward and lateral motion seem to work, at least for defined zero angular velocity of the utorso frame, but more testing should be performed. Code to accommodate arbitrary pelvis rotation vectors is in place, but non-zero rotations have not been tested yet.

Needed Extensions:
The present solution seems theoretically consistent. However, joint-angle limitations occur within relatively small net displacements. Once a joint limit is reached, it is no longer possible to satisfy the prescribed velocity values under no-slip conditions. A necessary next step is to address how to respond to this, presumably with lifting one wrist or knee and relocating the contact point.

Before lifting a limb, the center of mass should be checked to see if the robot can remain stable under 3-point contact. A strategy for sequencing limb placements and lifts needs to be devised.

It is unknown whether the back-joint constraints (ignoring 2 out of 3 torso joints) imposes too severe of a penalty in terms of joint-range limitations. While yaw motions (implemented, but untested) can be achieved while maintaining 4-point contact with no slip, joint limitations likely limit such turning to impractically small angular displacements. Use of additional back joints may be desired for better turning capability.

Additionally, it may be useful to deliberately tilt the torso frame, with or without use of additional torso joints. Such motion may help to accommodate joint-angle limitations, as well as elevate or lower the spine to climb over obstacles, achieve better viewpoints, or avoid elbow-angle arm

singularities.

It should be noted that the crawl Jacobian does not depend on the contact points being co-planar. Any initial pose is allowed and subsequent incremental motions will produce the specified utorso frame velocity—within joint-range limitations. The definitions of direction vectors should evolve to fit local conditions (e.g. a ramp). Also, new contact points should be planned intelligently or, alternatively, invoked reactively. At present, there is no means for detecting new contacts after free swing of a limb.

An expected limitation of this code is that it assumes that the “elevation” direction is, roughly speaking, normal to the plane containing the contact points. (The contact points do not have to be coplanar, but a best-fit plane through these 4 points may be used for reference). That is, the degree of freedom left open for this solution approach assumes that the robot can make adjustments towards or away from the contact surface. In contrast, if the robot is climbing a vertical ladder, and if the direction of gravity is defined as the vertical direction, the problem statement may be ill posed and no solution may be achievable. This may be visualized equivalently by considering the robot in 4-point contact on a plane and attempting to request that the utorso origin move vertically while the contact points maintain no slip. i.e., if the robot is lifted up, the knees cannot maintain contact with no slip, since they cannot extend their length. This degree of freedom must be left available in the solution.