TouchIt package documentation
Wyatt Newman Jan 28, 2014

The package “touchit_action_server”, currently under rosbuild/elec7078_newman, is undergoing refactoring. This document is a checkpoint, detailing the theory of operation and identifying future work to further develop this package.

The intent of this package is to create an action server that accepts Cartesian goals and smoothly moves a hand to these goals. The behavior should either converge on the specified goal or should halt upon some other condition, including out of reach, approaching a singularity, approaching joint limits, or detecting contact force.

The TouchIt package creates action servers separately for the right and left arms. These action servers have much in common, including the modules: jog_behavior_server.h, jog_behavior_server_methods.cpp, jog_increment_methods.cpp, helper_methods.cpp, and set_magic_nums.cpp. The right and left arms have separate main functions and separate constructors for the class JacobianMoveServer.

The TouchIt package relies on input from additional nodes, including a force/torque filter node, an interactive marker node, and a teach-pendant style jog GUI. Output of the TouchIt package communicates relatively high frequency, incremental joint commands to the low-level joint controller (LLJC).

Theory of operation overview: This discussion will refer to control of the right hand, for simplicity. Left-hand control is analogous.

Motion commands via the touchit_action_server package use incremental commands from Jacobian inverses. The Jacobians are expressed in terms of the (right-hand) grasp frame with respect to the robot's pelvis frame. The hand grasp frame is a Team-HKU definition for the Sandia hand, illustrated below.

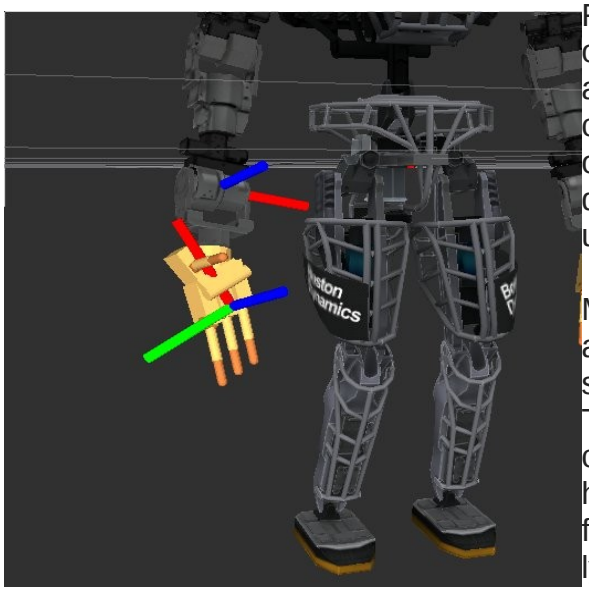

Illustration 1: rhand frame (at wrist) vs rhand_grasp_frame (on palm)

For the hand grasp frame, the z-axis (blue axis) is oriented normal to the palm. The y-axis (green axis) runs parallel to the “palm cylinder” (a cylindrical hump that crosses the palm). The origin is offset from the palm cylinder by a distance that is a good contact point for picking up cylindrical objects roughly 8 cm in diameter.

Many objects can be approximated by cylinders: a railing, a firehose, a drill handle, a segment of a steering wheel, a segment of a valve wheel, etc. Two common applications are to pick up a cylinder that is oriented vertically (e.g., the handle of a drill sitting upright on a table, or a firehose hanging vertically) or a cylinder that is lying on its side on a table surface (e.g., a drill on its side or a firehose on a table). For the former case, the robot should approach the object with the y axis pointing vertically (parallel or antiparallel to gravity) and the z (normal) axis pointing towards the object. For the latter case, the grasp approach should be with the z axis pointing down (parallel to gravity) with

approach from above, and with the y axis oriented parallel to the major axis of the cylinder.

One control option is to command motions along grasp-frame axes. It is frequently desirable to approach an object along the direction of the grasp-frame z-axis (the palm normal), in preparation for grasping the object. Additionally, a grasp plan may include first orienting the palm appropriately (e.g., aligning the grasp-frame y-axis parallel to the object's major axis), then approaching the object along the palm-normal direction while maintaining hand orientation. There are a variety of options in the touchIt package to support this.

Performing arm moves to a target 6-DOF gripper position and orientation is a common requirement in robotics. Surprisingly, it was found experimentally that the arm degrees of freedom of Atlas are highly constraining. Although Atlas' arms each have 6 degrees of freedom (the theoretical minimum number of joints required to reach an arbitrary 6-DOF pose), Atlas typically has difficulty reaching target poses. Attempting to solve inverse kinematics for a 6-DOF target pose typically results in no solution, due to joint range limitations. (This seems to be due to the fact that Atlas has only 2 wrist degrees of freedom).

To achieve successful grasps, it is important to pre-position Atlas relative to an object of interest to allow for viable kinematic solutions. Additionally, exploitation of null-space motions can be essential for achieving viable kinematic solutions that conform to joint-range limitations. Identification of null space motions is task dependent. In some instances, full specification of the grasp orientation is not required. For example, when using a hook hand (or simply a pole extending from the wrist) to turn a valve or door handle, it may be important to achieve a tip position, and to satisfy an approximate orientation of a major axis, but spin about this axis (e.g. pole) may be irrelevant. Similarly, in grasping a steering wheel, one may wish to close a hand's fingers around a segment of the wheel with the grasp-frame y axis tangent to wheel—but achieving a specific orientation of the palm normal is not required (as long as it points radially inward with respect to the cylinder segment grasped). Similarly, in grabbing a fire hose, the grasp-frame y axis should be parallel to the hose fitting's cylindrical axis, but this still leaves an infinity of equivalent solutions (any palm normal pointing radially inward towards the hose's major axis). In other cases, a specified hand pose might need to be achieved only approximately. For example, when cutting a wall with a power tool, the bit should be roughly perpendicular to the wall, but a slight tilt of the tool away from nominal is acceptable. Exploiting such options can be essential in finding an achievable inverse kinematic solution.

Use of the torso joints adds more degrees of freedom that can help to achieve viable grasp poses. It was found, however, that bending of the waist should be avoided, as these joints are weak. If Atlas bends over too far—particularly with arms outstretched—he will collapse at the waist and be unable to return to standing upright (until the mass distribution is re-altered, e.g. by moving the arms backwards).

Rotation of joint zero (rotation of the torso about vertical) does not suffer from this limitation, and thus this degree of freedom can be utilized. Some of the current touchIt options exploit 7-dof control that includes torso rotation. If only one hand is involved in a task—perhaps including use of that hand's cameras--then 7-DOF control for manipulation is desirable.

Seven-DOF arm control for grasping can be a problem, however, if one hand is used for

manipulation while the other hand is used for pan/tilt/zoom hand-camera control—a feature that may be highly useful. When using joint zero for grasp approach, the view from the opposite hand is also affected, undesirably rotating it away from the target view. One approach to addressing use of torso joints is to: pre-plan a torso rotation to enable a viable grasp pose; move the torso to this position; then control the two arms independently with only 6-DOF each. Some of the necessary 6-DOF code already exists within the current touchIt version, and this approach is being re-examined within current code refactoring.

Pre-positioning Atlas' torso to achieve viable grasp poses extends to pre-positioning the pelvis more generally. One can command Atlas to step to a location that supports achieving a grasp pose of interest. Additionally, Atlas may be commanded to squat by a specified amount to help bring his shoulders to a more favorable position for satisfying a kinematic solution for a desired grasp pose. Given Atlas's severe constraints on inverse-kinematic solutions, planning how to pre-position Atlas (including feet positions, pelvis height and torso rotation) is an important issue for successful grasp. Such planning is not part of the touchIt package. Rather, successful touchIt behavior depends on Atlas being pre-positioned such that an intended grasp pose is possible.

TouchIt and Jacobians: TouchIt attempts to achieve Cartesian (task-space) goals via hand motions that are straight-line in Cartesian space. TouchIt does not attempt to do more general motion planning. Instead, touchIt commands Jacobian-based incremental motions. This presents some difficulties normally addressed by more general motion-planning algorithms. One such difficulty is trying to assure that the resulting Jacobian-based motion will avoid collisions—whether between the robot and the environment, self collisions, or hitting joint-motion mechanical limits. Approach planning (through stepping, squatting and rotation of the torso) can help satisfy the prerequisites for a viable solution to a desired grasp pose. However, this does not guarantee that touchIt will produce an acceptable motion (including collision avoidance).

The intent of touchIt is to control relatively short motions, with the expectation that nearby goals will not require sophisticated planning. Consider for example a case for which: 1) the desired grasp pose is known to be reachable (e.g. achieved through planning and pre-positioning Atlas' pelvis and torso); 2) the palm normal (hand grasp frame z-axis) is pre-positioned to be colinear with the z-axis of the desired pose for grasping an object; 3) the grasp-frame y-axis is pre-positioned to be aligned parallel to the y-axis of the desired grasp pose; and 4) the grasp-frame origin is close to (e.g. within roughly 20cm) the desired grasp pose origin. Under these conditions, an implicit motion plan is to converge on the desired grasp pose by moving in a straight line along the hand normal while preserving hand orientation. With this implicit default motion plan, the operator can easily visualize if there would be a collision between the hand and the environment—or between the hand and the robot—before invoking the Jacobian move. Further, if the initial (preparatory) hand pose and the desired (target) hand pose are relatively close in Cartesian space, and if both of these poses are known to be achievable (including collision free), then it is likely that the entire (short) path from approach to finish is also safe and achievable within the robot's joint-limit and collision-avoidance constraints. This is not mathematically guaranteed, but it tends to be true for short moves. Safe, achievable approach paths are also more likely if both the initial pose and the target pose are achieved with joint angles that are sufficiently far from the joint-angle limits.

As noted above, avoiding the surprisingly harsh constraints of joint-angle limits can be aided through use of null-space motions. If the target pose has fewer than 6 requirements,

and further if additional (e.g. torso rotation) degrees of freedom are recruited, then null-space motions exist. That is, the target specifications can be achieved through a variety of joint-space solutions. Some of these solutions are better than others—particularly with respect to avoiding joint limits. TouchIt makes an attempt to utilize null-space degrees of freedom to help avoid joint-space limits. However, this also presents potential problems with local minima and with null-space "wind-up."

Local minima problems occur because Jacobian-based moves consider only local gradients and do not anticipate that moves along gradients can lead to joint-limit barriers, although viable solutions may well exist.

Null-space wind-up is a consequence of uncontrolled drift in null space, potentially leading to problems with self collisions or joint-limit violations. Consider, for example motion of a human arm with the wrist tracing out a circle (e.g., like window washing, or writing on a blackboard). The desired wrist motion can be achieved using different "orbit" angles of the elbow (e.g., with the elbow low, or with the elbow elevated). The elbow "orbit" degree of freedom is unconstrained by a Jacobian move. Consequently, one may start the example motion with the elbow low, and the elbow may gradually drift upwards until joint-limit problems are encountered. Similarly, when Atlas is controlled by touchIt to increment along Cartesian directions, a sequence of such moves can result in arm wind-up to contorted angles presenting problems with potential self collisions and/or joint-limit violations. TouchIt attempts to avoid wind-up problems by virtually repelling joints away from joint limits (while conforming to Cartesian move constraints). However, the bumper repulsion algorithm needs improvement to eliminate observed chattering near joint limits. Further, the joint-limit repulsion technique is inadequate to fully prevent null-space wind-up. For this reason, there is a command to "unfreeze" an arm that has moved to a pose that is highly constraining. The intent of "unfreeze" is to perform a null-space move, preserving end-effector constraints while reorienting the arm to move towards a more mobile pose. Current implementations are partially successful but should be improved.

Another consequence of Jacobian-based moves is end-effector drift. The derivative of hand motion with respect to joint perturbations (a Jacobian) is used to compute joint increments that correspond to a desirable Cartesian incremental motion of the hand. However, such increments accumulate error, due both to servo imprecision and to mathematical extrapolation of gradient-based motions in non-linear kinematics. For long moves, as well as for sequences of short moves, the hand pose can drift significantly from the intent. This is particularly noticeable in terms of orientation. For example, if the hand is commanded incrementally to move around a circular path while preserving orientation, after completing the path, the final orientation will have drifted away from the initial orientation. To correct for such cumulative errors, touchIt has a command for "reorient." The intent of "reorient" is to perform an arm motion that results in re-aligning the hand orientation to the target orientation while holding the hand's grasp-frame origin fixed. In some cases, the target orientation refers to only a single axis, which permits more null-space mobility. Nonetheless, the desired reorientation may or may not be achievable. Current implementations of the "reorient" command are partially successful but should be improved.

TouchIt Code Organization Overview: The executable "rhand_jog_server" instantiates a ROS action server through an object of the class JacobianMoveServer. Two separate modules exist that implement the constructor of the JacobianMoveServer: one for the right hand (rhand_jog_move_constructor.cpp) and the other for the left hand (lhand_jog_move_constructor.cpp). These constructors have only minor differences,

including establishing the names of the respective servers (e.g. “rhand_jog_jacobian_move_server”) and the respective hand frames for computing Jacobians (e.g. “right_grasp_frame”). Additionally, there are two separate “main” programs for the two jog servers: rhand_jog_server_main.cpp and lhand_jog_server_main.cpp. (These main programs currently include “test” in the names, which should be removed). These two main programs are also largely identical, except for setting up different node names, registering with the Low-Level Joint Controller (LLJC) with distinct names, subscribing to right vs left wrist force/torque sensor signals, monitoring poses of right vs left grasp frames, and instantiating separate (right vs left) JacobianMoveServer objects. All other methods in the additional 4 modules used are identical for left and right arm code.

TouchIt Code, main program: The main program is responsible primarily for communications and for timing. It relies on an object of type JacobianMoveServer to perform all necessary kinematic computations. The main program subscribes to: filtered force/torque signals; LLJC last command; LLJC heartbeat; and AtlasState. The force/torque signals to which touchIt subscribes are published by a separate node (rhand_wrench_filter2, for the right hand). The main program detects if this node is publishing, and if not, wrist force/torque signals are assumed to be zero (and thus the wrist force/torque signals are subsequently ignored). The main program also instantiates a tf listener, which is set to listen for transforms of the (right) grasp frame with respect to the pelvis.

The main program registers with the LLJC, and the main program is subsequently responsible for publishing all joint commands to the LLJC.

After initializations (including subscriptions with callbacks), the main program enters a loop that enforces a fixed update rate, currently set in jog_behavior_server.h to 50Hz. Within this loop, main informs the JacobianMoveServer object of data updates, as received via main's subscriptions and from main's tf_listener. Such updates include: current hand orientation, current robot joint angles, and current wrist force/torque sensor values.

On each iteration of its timed loop, the main program first tests if a new goal has been requested of the action server object. If so, main performs some initialization computations through the JacobianMoveServer member function “initMoveIncrement()” before starting a loop to achieve the goal.

In the remainder of the loop, the main program inquires of the JacobianMoveServer object if the current goal is complete. If not, main invokes the object's member function “move Increment()”, which causes the object to compute an incremental update to all 28 joints of Atlas—with method determined by a move code within the received goal message. The main program then gets the computed joint angles and joint velocities from the JacobianMoveServer object and publishes these to the LLJC.

If the JacobianMoveServer object reports that the current goal is concluded, then main instructs the object to inform its action client that the goal is concluded. (Main does this via the member function “setSucceeded()”). The message sent by the move server object to its client will contain a code describing the status of completion—which may be that the move is concluded due to an error condition, or due to successful accomplishment.

The main program is relatively brief. It does not attempt to perform any kinematic computations, but leaves this up to the JacobianMoveServer. The JacobianMoveServer

has many different options for computing kinematic updates, and these are specified via a move code within the goal request from the current action client. With respect to the main program, the JacobianMoveServer is responsible for computing updated values of 28 joint angles and 28 joint velocities each time it is invoked, as well as keeping track of whether a requested goal is still in progress or not.

TouchIt Code, JacobianMoveServer constructor: As noted, the JacobianMoveServer class is responsible for kinematic computations within the touchIt package. Its member variables and methods are declared within "jog_behavior_server.h". Separate constructors exist for right-hand vs left-hand jog servers.

The constructor establishes dimensions for many member variables and initializes them to valid values. In performing initializations, it is assisted by a call to the function "setMagicNums()."

The constructor also names the action server that it creates, and it binds the member function "goalCallback" to respond to goal messages sent by action clients. (A "cancelCallback" function is also bound, but this behavior is not well tested). The action-server messages for the jacobianMove are defined within jacobianMove.action, in the folder touchit_action_server/action (as is convention for action servers). The goal message contains a PoseStamped pose (a goal pose for the respective hand grasp frame) and an integer "mode" (implying a particular method to use to approach the goal). The response (result) message contains an integer return code and a PoseStamped pose. The pose returned is the computed grasp-frame origin based on the achieved joint angles. At present, the hand-origin position values returned are correct, but the orientation of this pose is not correct. The values returned for orientation are merely a copy of the commanded orientation. (TODO: fix this).

The constructor also instantiates an object of type TaskVariableXYZRPY. This class was created by Vladimir Ivan (teamHKU collaborator at U. Edinburgh) to simplify use of "KDL" (Kinematics and Dynamics Library). The object instantiated (called "vlads_fancy_solver" in this package) is initialized by interrogating the parameter server regarding URDF (universal robot definition file) parameters that define the kinematic tree (robot) of interest. Conveniently, when changes are made to the robot description (including link lengths, joint transforms, mass distributions, and hand-mount position/orientation variations), these are programmatically incorporated into the KDL-based solver. It is thus unnecessary to edit touchIt when the robot model is changed.

The specific KDL-based object that is instantiated is set to compute properties of the right_grasp_frame with respect to the pelvis. The constructor for the left-hand jog server specifies a solver with respect to the left grasp frame. Member functions within the TaskVariableXYZRPY compute forward kinematics and Jacobians of the target frame with respect to the pelvis.

The constructor anticipates possible use of multiple, alternative Jacobians, including 7 involved joints (1 arm plus torso joint-0 rotation) or 6 involved joints (arm joints only), as well as separate, 3-D translational and rotational Jacobians, plus full 6-D rotational-and-translational Jacobians. Values for these Jacobians are populated within member functions using results from the TaskVariableXYZRPY kinematic solver object.

TouchIt Code, Jog Behavior Methods: The module "jog_behavior_server_methods.cpp" contains implementation of some of the methods within the JacobianMoveServer class.

These are higher-level methods, which it turn invoke lower-level methods implemented within the module “jog_increment_methods.cpp.”

The action-server goalCallback, cancelCallback and setSucceeded methods are implemented in the “jog_behavior_server_methods.cpp” module. The goalCallback function is responsible for receiving and setting pose goals as well as motion parameters. At present (for obsolete, historical reasons) the move mode is unnecessarily obscure, and this will be changed. Nonetheless, the current message interpretation is described here.

The “mode” field within the goal message is comprised of up to 3 values, concatenated in a base-10 number. For straight-line, incremental moves, the mode is a number greater than 999. The leading digits (1000 and above) specify a move duration, in seconds. The next digit (100's digit) specifies a move distance (1 through 9 centimeters). The final two digits specify a frame, direction and technique for making the Cartesian-space move. If the move mode is 999 or less (i.e. a move time of 0), then the remaining digits are interpreted to be a special case (e.g., “unfreeze” or “reorient”).

The above method of packing 3 distinct pieces of information into a single integer is unnecessarily complex. Instead, the action message should be modified to contain 3 separate fields for these values. (TODO). Nonetheless, the goal message describes the information needed to perform a useful variety of move types.

The goalCallback function copies the received pose data into member variables, for use by member methods. Quaternion targets are converted to 3x3 rotation matrix orientation specification. A vector is computed from current hand position to goal position (3-D translation). This vector is also normalized to define a direction of motion to the target.

For specified Cartesian “jog” moves, the move is assumed to be performed over “n_iters_max” iterations, which is derived from the goal-specified move time and the update rate of the action server (currently set to RATE=50Hz in the header file).

Aside from implementing the action-server goalCallback function, the key methods implemented within jog_behavior_server_methods.cpp are: computeJacobian(), initMoveIncrement() and moveIncrement().

The computeJacobian() method comes in two flavors. If no arguments are provided, the Jacobian is computed based on the robot's current joint angles (as supplied by the main program to the JacobianMoveServer object). Alternatively, if joint angles are provided as an argument to this method, the Jacobian is computed based on the provided angles. The latter is more general, allowing for “what-if” planning vs. computations based solely on current state. The latter can also be more stable, as it can be based on incremental commands instead of sensor data; the sensed joint angles can introduce complications of noise, latency and servo dynamics effects.

Computing the Jacobian relies on the simplified KDL solver instantiated by the JacobianMoveServer constructor. Given a set of joint angles, the solver computes the forward kinematics and the Jacobian of the desired hand grasp frame with respect to the pelvis. The resulting Jacobian is 6x9, since it includes all joints in the chain from the pelvis to the desired hand. The 6x9 Jacobian is parsed to populate 8 variations on the 6x9 Jacobian. These include subspace 3xN Jacobians for translations and for rotations separately, for each of 9 joints (6 arm + 3 torso), 7 joints (6 arm + 1 torso) and 6 joints (arm joints only). All of these variations are availble as member variables to all member

functions.

The member function " initMoveIncrement()" performs initializations for a specified move, based on the goal pose, move code, move distance and move duration. Given the desired (6-D) move distance, desired move duration and known update rate of the main loop, this member method sets how much incremental motion (in Cartesian space) should be performed each iteration.

The method "moveIncrement()" is responsible for performing a single, incremental update of the member vectors qcmd28 and qdot_cmd28. If a 6-DOF move is being performed, only the 6 affected joint angles (and joint velocities) are updated within the vector of 28 joint commands. If 7 joint angles are affected (including the torso joint-0 rotation), then only these 7 commands are updated in the 28-D vector. The "moveIncrement" method consults the values of lastCommandedJointAngles, which include feedback from the LLJC. There are safeguards to avoid discontinuous jump commands relative to the most recent LLJC approved command. MoveIncrement() anticipates extensions required for the jog servers to run compatibly with additional motion servers in parallel (e.g., to include independent neck and torso motion servers, and for operating two hands and two feet independently and concurrently, as anticipated for driving).

The specific means by which joint commands are incremented uses the lowest-level methods defined within the module "jog_increment_methods.cpp." Based on the move mode within the received goal message, low-level methods are invoked within a switch/case structure. There are, however, over 100 cases defined. The proliferation of cases motivates significant refactoring. Further, the case numbers currently are not mnemonic; "magic numbers" should be replaced with meaningful names. Nonetheless, the current program categorizes a variety of useful move types, including Cartesian motion along a single axis (in the pelvis frame or in the hand's grasp frame) for specified duration and distance, as well as a variety of experimental options to "unfreeze" and "reorient" to address problems of drift and joint limits.

It should be noted that, in the current construction, the original behavior of moving to the specified goal no longer appears as an option. The intent of the current construction was to move incrementally based on button clicks within a teach-pendant style GUI. However, the existing code is compatible with reinstating the functionality of moving to the specified goal, and this could be easily implemented as its own case. (TODO).

The case statements within the initMoveIncrement and moveIncrement methods merely provide connections between a move mode—specified within a goal message—and the corresponding kinematic technique to invoke. No kinematic computations are implemented within the jog_behavior_server_methods.cpp module. Rather, various approaches relating joint increments to desired Cartesian increments are detailed within the module "jog_increment_methods.cpp".

TouchIt Code, Jog Increment Methods: *cartesianJogMove* This module contains implementations of kinematics for initializing jog moves and performing incremental updates. The overall intent is to update the relevant joints to cause the selected hand to move closer to a goal pose. Additional methods assist this intent by attempting to correct for cumulative drift or to perform null-space moves to help avoid joint limits. The methods in this module rely heavily on Jacobians computed within the higher-level methods implemented in jog_behavior_server_methods.cpp.

The methods within this module should be refactored to eliminate dead code, to provide for more code re-use, and to improve the kinematic performance. More sophisticated techniques may be devised and developed and presumably implemented as methods within this module, potentially with little or no modification to the other modules of touchIt.

A particularly relevant method within this module is: cartesianJogMove(hand). This method performs the operations of computing a move increment using 7 joints, inserting the influence of these 7 perturbations into updated 28-D position and velocity vectors, and testing for termination conditions. The move increment is peformed by the method moveIncrementXdeltaAndSpinAxis(), and the termination conditions are tested using the method testDoneIncrementCartesianMove().

The method *moveIncrementXdeltaAndSpinAxis*() depends on a prior call to initMoveIncrement, within which the value of **xdelta_** is computed, consisting of the vector from initial hand grasp frame origin to the goal frame origin, divided by the number of increments required to reach the goal over the specified duration with increments occuring at the specified loop rate of the main program. The number of iterations to satisfy these specifications is computed as n_iters_max, which is set at the time a new move goal is received.

Given **xdelta**, a 3x1 vector of incremental translations, *moveIncrementXdeltaAndSpinAxis* uses the following technique to compute incremental commands to the 7 involved joints. The "Eigen" package method "*jacobiSvd*()" is used to compute a solution to the underdetermined equation: **xdelta** = **J_trans_3x7***qdelta7_nom**, where **xdelta** is the desired 3-D hand incremental translation, **J_trans_3x7** is the 3x7 translational Jacobian, and **qdelta7_nom** is a set of 7 angle perturbations to achieve the desired incremental Cartesian motion (assuming 6 arm joints plus torso joint-0). The algebraic equation is to be solved to find **qdelta7_nom.** Since there are 7 qdelta values to choose to produce 3 xdelta values, there is a (typically 4-D) space of valid solutions for **qdelta7_nom**. The chosen solution method uses singular-value decomposition. This method is one that returns a good solution and which is robust with respect to singularities of the Jacobian. Given only a 3-D incremental motion goal and 7 joints to utilize, a solution (incremental motion of the 7 involved joints) typically exists. In the present approach, this is the first priority to achieve, and the result is a nominal qdelta.

However, incremental Cartesian translation is not the only objective to achieve. It is also desired that motion satisfy some orientation constraints and, if possible, avoid joint limits. As a second priority, hand orientation considerations are considered in the context of the null space of the preceding solution. Given 3 translation objectives and 7 joints to use, there is typically a 4-dimensional null space of options. Motions in this null space will affect hand orientation as well as proximity to joint limits, but such null-space motions will not affect the first-priority hand displacement goal. The desired null space is computed using another "Eigen" method: *fullPivLu().kernel*(). This is one of the available methods for computing a set of basis vectors for the null space of the 3x7 translational Jacobian. The result is a "kernel", consisting of (typically) 4 vectors, each 7-dimensional (involving the 7 affected joints). These are conveniently contained within a matrix, **N**, consisting of 4 columns of 7-D joint-space vectors. Any weighted sum of these 4 vectors may be added to the nominal solution without affecting the first priority (translational displacement).

Assuming we desire to achieve the target hand displacement while keeping the hand orientation fixed, we equivalently desire that the rotation vector of the hand is a 3-D vector of zeros. However, we can compute the hand rotation that would result from invoking the

nominal solution: **omega_nom** = **J_ang_3x7*qdelta7_nom**. That is, for the first-priority joint-space solution of **qdelta7_nom**, we can compute how this will cause the hand to rotate incrementally, using the 3x7 angular Jacobian. This rotation is an error, and we wish to suppress it by invoking additional joint motions that lie in the null space of the translational Jacobian. With null-space vectors contained within the matrix **N**, the range of possibilities can be expressed as: **qdelta_null** = **N*n_null**, where **n_null** is a column vector of scale factors used to created a weighted sum of null-space **qdelta** vectors, and **qdelta_null** is the resulting weighted sum of **qdelta** vectors. We desire to find a set of weights **n_null** that corrects for hand orientation drift using only null-space motions. Equivalently, we desire to produce a corrective twist, **t_angular** = [0,0,0,-**omega_nom**^{T}]^T in terms of null-space weighting vectors, **n_null**, that satisfy:
t_angular = **J_ang_3x7***(**N*n_null) =** (**J_ang_3x7*N)*n_null**
This equation can be solved for **n_null,** e.g. using the singular-value decomposition method implemented in Eigen. The result for **n_null** allows computation of **qdelta_null** = **N*n_null,** which should be added to **qdelta7_nom** to correct for angular displacement of the hand's grasp frame.

In a slightly more general variation, we might not be concerned about some hand rotations. For example, the specific task of approaching a cylinder for grasp from above (i.e., with intent to grasp the end-cap) would not care about rotation about the hand normal. It would be desired to maintain the direction of the hand normal, but spin about this axis would be irrelevant. The same condition is true for manipulation with palm surfaces (not involving fingers)—e.g. on a valve wheel or steering wheel, where it is desired to orient the palm normal, but spin about the palm normal is irrelevant. A don't-care spin axis can also apply to the grasp-frame y-axis, if the object to be grasped is, e.g., a thin cylinder (like a pencil). In such cases, one would want the major axis of the object to align parallel to the grasp-frame y-axis, but rotation about the y-axis would be irrelevant.

For such situations, one may define a “spin axis” as an additional degree of freedom in the grasp-motion null space. Various methods within jog_increment_methods.cpp consider spin axes, although the kinematic approaches should be improved.

In addition to achieving hand position and orientation goals, it is also desired to avoid joint limits. If the task requirements are lower dimensional, and/or if more joints are recruited to perform a desired motion, there may be sufficient mobility within the resulting null space to perform the task goal while avoiding joint limits.

To take into consideration the joint limits, the method *compute_bumper_penalties*() is defined within the higher-level module “jog_behavior_server_methods.cpp.” This function computes a penalty value for each joint relative to the respective joint's mechanical limits. The penalty function is defined to be zero over most of a joint's range of motion, but rises sharply via an exponential function for encroachment beyond some defined soft limit. (Both the soft-limit location and the exponential factor are tubable parameters). In addition to computing this penalty function, the derivative, **bumper_penalties_prime**= d(penalty)/d(joint_angle) is computed as a vector in joint space.

One choice for how to include joint-limit violations is to add joint velocities proportional to the gradient d(penalty)/d(joint_angle). However, these limit-avoidance corrections should be performed only in null space. Assuming **qdelta_bumpers_nom** proportional to (negative) **bumper_penalties_prime**, then a projection of these joint velocities into the null space would yield: **n_bumpers** = -K*$\mathbf{N}^T$* **bumper_penalties_prime,** where **n_bumpers** is a vector of weights to apply to a weighted sum of null-space **dq** vectors to

produce joint-limit repulsion in null space, and K is a tunable factor for the speed of repulsion. The corresponding joint-space bumper motions would be: **qdelta_repulsion** = **N***n_bumpers = -K***N****N**^{T} * **bumper_penalties_prime.** Adding **qdelta_repulsion** to the incremental joint motion vector would produce the effect of attempting to repel joints from nearby mechanical limits while preserving the desired translational hand motion. This repulsion action can, however, interfere with trying to preserve hand orientation. Proposed improved kinematic approaches are suggested later in this document.

TouchIt Code, Jog Increment Methods: *testDoneIncrementCartesianMove*
In performing incremental motion updates in joint space, a responsibility of a moveIncrement method is to test for termination. Three conditions are considered for termination, tested within the method *testDoneIncrementCartesianMove: genericHaltTest(), testContactForce()* and *testTooFast().*

GenericHaltTest() checks if the proposed command would send the joints out of range, and if so, terminates the move and sets and error code (which is returned to the action client). In performing joint limit testing, there is a subtle consideration. If one were to merely test for joints out of bounds, then if an out-of-bounds error ever occurred, all future commands would be aborted. Equivalently, the system woud get "stuck" at a joint limit. To avoid this problem, the out-of-bounds checking considers both the joint-angle commands and the joint-velocity commands. If the joint velocities would result in the robot retreating from joint limits, then the action is allowed to proceed. This variant allows the robot to recover from (soft) joint-limit violations.

Assuming the joint commands are allowed, *GenericHaltTest()* next checks if the intended number of iterations has completed. Approximately, this would imply that the intended motion was successful in reaching its goal—assuming the cumulative Jacobian-based incremental motions did not result in excessive drift. If the number of planned iterations has been performed, then the "move_done" flag is set and a corresponding return code is assigned.

A second test for move completion is performed by *testContactForce().* In this method, hand forces are evaluated to check if contact has occurred. The hand-force signals used have been pre-processed by a separate node, within which corrections are made to subtract off the weight of the hands (since the wrist force/torque sensors feel the weight of the hands), subtract off bias offsets, and perform low-pass filtering. These signal-processing steps are intended to make the resulting signals more reliable and more sensitive to detecting contact. Within *testContactForce()* the (pre-processed) hand force vector is dotted with the hand motion direction vector to get the projection of hand force in the direction of motion. This step is intended to make contact detection more reliable, since contact force that is produced by commanded hand motion should be a reaction in the direction opposite to the hand motion. If the resulting force projection exceeds a tunable threshold, then the *testContactForce()* method sets the "move_done" flag to "true" and sets a corresponding return code.

The third generic halt test detects if joint motion commands are too fast. With scaling performed in the moveIncrement function, excessive velocity commands should not occur. Nonetheless, this function can trap conditions of improper velocity commands.

TouchIt Code, Jog Increment Methods, reorient methods: A variety of methods are implemented with attempts to enable the robot to recover from constraining poses. Incremental moves can result in drift in null space that puts the robot into clumsy

configurations, where subsequent incremental moves are constrained by joint limits or self collisions. To recover from undesirable poses, a “reorient” command can be used. These methods are experimental and lightly tested, and they should be improved.

An example attempt is: *moveIncrementReorientPrepCutWall*(). For this method, a nominal joint-space pose is pre-defined (as initialized with hard-coded values in “set_magic_numbers.cpp”), relevant to one of the tasks (and corresponding to one of the dedicated task GUIs).

In this method, the current robot pose (for the involved joints) is compared to the nominal pose, and a joint-space move increment is computed that would drive all joints to the respective nominal pose values over 2 seconds. These values are not used directly, however. Instead, they are projected onto the null space of the translational Jacobian. The result is a joint-space increment, **qdelta_unwind**, that attempts to drive the joints towards the nominal pose, but permits this only within a null space that preserves the current hand position. As a result, the joints should get closer to a pre-defined acceptable pose and the hand should remain at its current position, although the hand orientation would be affected. Desired hand orientation may be recovered using another “reorient” method.

The method *moveIncrementReorientGeneric2*() attempts to recover (partial) desired hand orientation by re-aligning a chosen axis of the hand grasp frame to a target direction. For example, if one is has defined the palm normal as the “spin axis” (about which orientation rotations are permitted), and if one has defined the desired spin axis direction (palm-normal approach direction), then pose recovery corresponds to tilting the palm-normal axis into alignment with the desired direction. This desired rotation can be computed as the cross product of the current spin axis (e.g., palm normal) direction with the desired spin axis direction. This cross product evaluates to a rotation direction and (approximate) magnitude to perform the desired orientation correction. The *moveIncrementReorientGeneric2()* method attempts to find joint-space incremental motions within the null space of the translational Jacobian to perform this orientation correction. These joint-space commands are returned, attempting to reorient the hand to the desired alignment while preserving the hand origin coordinates.

Conclusion: The touchIt package is a work in progress. Additional work to be done includes: eliminating obsolete code; refactoring for better code re-use; creating mnemonic labels for case numbers and return codes; re-integrating “grabIt” and GUI jog command interfaces within a single package; improving joint-repulsion methods and parameters to eliminate chatter near joint limits; developing better kinematic methods to exploit null-space options; and tuning and testing wrist force/torque sensor halt-on-touch behaviors.