Differential Kinematics Survival Guide, part II

Wyatt Newman
September, 2014

Introduction: In part-I of these notes, the translational "Jacobian" was introduced. These notes extend introduction of the Jacobian to include rotations. The "Kinematics and Dynamics Library" (KDL) is introduced for computing forward kinematics and Jacobians in ROS.

Translational Jacobian: In general, a Jacobian is a set of linear influence coefficients that relates how a perturbation within an input space, $\delta \boldsymbol{q}$, leads to a perturbation of an output space, $\delta \boldsymbol{p}$, expressed as: $\boldsymbol{J}_{trans}\delta \boldsymbol{q} = \delta \boldsymbol{p}$. For this expression to be defined, the function $\boldsymbol{p} = \boldsymbol{f}_p(\boldsymbol{q})$ must be differentiable. (Note: the dimensions of the input and output spaces may be different).

For robots, we define a "manipulator Jacobian" (although this is typically referred to simply as "the Jacobian" in context). In part I, a translational manipulator Jacobian was defined as $\boldsymbol{J}_{trans} = \delta \boldsymbol{p}/\delta \boldsymbol{q}$, where **p** is a 3x1 vector of coordinates of the origin of a frame of interest ("target" frame) with respect to a reference frame, and **q** is a vector containing all of the joint angles.

For (open kinematic chain) robots, we interpret the function $\boldsymbol{p} = \boldsymbol{f}_p(\boldsymbol{q})$ to be the mapping from joint angles to Cartesian coordinates (of the origin of the target frame). This mapping corresponds to "forward kinematics." For typical (open kinematic chain) robots, this mapping is always simple to compute and is always smooth and differentiable. (Closed kinematic chains, such as Stewart platforms require a separate discussion, not covered here).

Although the Jacobian is defined conceptually, one must specify both the target and reference frames before any numerical values can be assigned to elements of a Jacobian matrix. We may, for example, choose the target frame to be the "right_grasp_frame", which has been defined within the TeamHKU code to be a frame on the palm of the right hand (with the z-axis of this frame corresponding to the palm normal). It is often convenient to choose the pelvis frame to be the reference frame.

In addition to interpreting the translational Jacobian as a vector derivative, or as a relationship between input-space and output-space perturbations, one can also interpret the Jacobian in terms of relating input-space velocities to output-space velocities: $\dot{\boldsymbol{p}} = J_{trans}\dot{\boldsymbol{q}}$. The vector $\dot{\boldsymbol{p}}$ is a translational velocity of the origin of the target frame, $\dot{\boldsymbol{p}}^T = [v_x, v_y, v_z]$, conventionally in units of m/sec. The vector $\dot{\boldsymbol{q}}$ is comprised of the joint angular velocities (how fast each rotational joint is spinning), typically in units of rad/sec (for rotational joints). Considering the joints from Atlas' pelvis to the right-hand grasp frame, 9 joints affect the palm origin's position (3 torso joints and 6 arm joints). One may thus define a Jacobian that has 3 rows and 9 columns to map (relevant) joint velocities onto the hand velocity.

Understanding how joint perturbations (or joint velocities) affect hand positions (or hand velocities) is important, but insufficient. It is also necessary to understand target-frame orientation, and how joint motions affect target-frame orientation.

Rotational (Angular) Jacobian: Many alternative orientation representations exist, including roll-pitch-yaw angles, Euler angles, angle-axis representation, quaternions and 3x3 rotation matrices. We will refer to a "pose" as a complete definition of a frame, including both location of the frame's origin

and some specified representation of frame orientation. (In ROS, the default is to use a unit quaternion to define orientation). As noted, to specify values of a "pose" one must define both the reference and target frames. In ROS, a "pose" message contains pose data, and it also carries with it a label for the reference frame consistent with the pose data.

Unfortunately, representation of orientation is more difficult to express and to work with than positions. One cannot simply stack up roll, pitch and yaw angles into a 3x1 structure and call it an orientation "vector". Unlike linear spaces, the order of operations matters for rotation. For example, rotating an object by 90 degrees about its x axis, then 90 degrees about its z axis results in an orientation that is dramatically different from doing this in the opposite order (rotating about the z axis then about the x axis). Specification of order of rotations is not contained within a mere listing of 3 angles, and thus such a representation is insufficient. (Equivalently, addition is not commutative for such 3x1 expressions, and thus such representations violate requirements for a linear system).

On the other hand, if only differential rotations are involved, order of operations is not important. A small perturbation of rotation about the x axis, $\delta\varphi_x$, followed by a small perturbation of rotation about the z-axis, $\delta\varphi_z$, yields (to first order) the same resulting pose as these operations applied in reverse order. Thus, we *may* refer to $\delta\boldsymbol{\varphi}^T=[\delta\varphi_x,\delta\varphi_y,\delta\varphi_z]$ (a perturbation of orientation) as a vector. We may also refer to the time rate of change of orientation as $(\delta\boldsymbol{\varphi}/\delta t)^T=[\delta\varphi_x/\delta t,\delta\varphi_y/\delta t,\delta\varphi_z/\delta t]$, equivalently: $\boldsymbol{\omega}^T=[\omega_x,\omega_y,\omega_z]$.

The 3x1 rotation vector $\boldsymbol{\omega}$ is easy to visualize. It has a direction in space, corresponding to an axis of rotation, and the magnitude of this vector corresponds to the speed of rotation (rad/sec). The instantaneous rotation vector (angular velocity) of a target frame, $\boldsymbol{\omega}$, depends on how fast all of the relevant joints are moving at that instant.

We may define the angular Jacobian, $\boldsymbol{J}_A$, as: $\boldsymbol{J}_A=\delta\boldsymbol{\varphi}/\delta\boldsymbol{q}$. In terms of velocities, this is equivalent to: $\boldsymbol{\omega}=\boldsymbol{J}_A\dot{\boldsymbol{q}}$, for which the angular Jacobian relates joint velocities to an instantaneous rotation velocity vector (3x1) of the target frame.

Full manipulator Jacobian: Since both the translational and angular Jacobians pre-multiply a vector of joint velocities, we can combine these to write:

$$\boldsymbol{t}=\begin{bmatrix}\boldsymbol{v}\\ \boldsymbol{\omega}\end{bmatrix}=\begin{bmatrix}\boldsymbol{J}_{trans}\\ \boldsymbol{J}_{ang}\end{bmatrix}\dot{\boldsymbol{q}}=\boldsymbol{J}\,\dot{\boldsymbol{q}}$$

The above formula defines the "twist" vector, **t**, which is a 6x1 vector of three translational components and three rotational components of the target frame in Cartesian space. This generalized velocity vector captures all 6 degrees of freedom of how a frame is moving (i.e. how a pose is evolving). "Twist" is defined consistently in ROS as a message type in the "geometry_msgs" package.

The above formula also defines the full manipulator Jacobian, which incorporates both translational and rotational effects that scale with joint velocities, $\dot{\boldsymbol{q}}$. The full manipulator Jacobian always has 6 rows: three for translation and three for rotation, expressed in Cartesian coordinates. (One may choose to consider lower-dimensional output spaces, resulting in fewer Jacobian rows, but such special cases will not concern us here). The number of columns corresponds to the number of joints.

For Atlas, with 28 body joints, we could define the Jacobian to be 6x28. For the example of a target frame corresponding to Atlas's right-hand grasp frame and a reference frame defined as the pelvis

frame, this 6x28 Jacobian would be comprised mostly of zeros (with a column of zeros for each joint that has no influence on the target frame). Only 9 of the 28 joints affect the right-hand frame relative to the pelvis, and thus this 6x28 Jacobian would have 17 columns of all zeros. Alternatively, if Atlas were standing on his left foot, and if we chose our reference frame to be on the sole of his left foot, there would be 15 joints that influence the right-hand frame (6 joints of the left leg, 3 torso joints, and 6 joints of the right arm). We thus see that "the" Jacobian is ambiguous. We must be careful to define both the reference and target frames.

For the case of right-hand frame with respect to the pelvis, there are 9 influential joints. To simplify the Jacobian, we may define a 6x9 Jacobian that eliminates all 17 of the zero columns. However, we will need to be careful with the corresponding calculations, to define the consistent joint-velocity vector to be comprised of only the selected joints and in the correct stacking order.

Forward Kinematics and Jacobians in ROS: KDL and Eigen

In part-I of these notes, the "transform_listener" was introduced. The transform listener is very convenient, particularly for transforming sensor data into consistent frames for visualization. However, the transform listener does have its limitations. For one, the transform listener provides results only for the current pose of the robot. Thus, it is not helpful in performing "what if" planning analyses. Further, transform-listener data is not suitable for real-time feedback, since latency in tf data results in instability for feedback loops with closed-loop bandwidth faster than roughly 1Hz. Further, the transform-listener does not provide computations of Jacobians. An alternative is "KDL."

The package "KDL" (Kinematics and Dynamics Library, http://www.orocos.org/wiki/orocos/kdl-wiki) provides a useful library for performing common kinematic computations. One convenience of KDL in ROS is that the KDL library can parse your current robot model, so that changes to your model automatically get propagated to your KDL applications. The user does not have to enter hard-coded details of robot link lengths, offsets, home angles, etc. However, KDL's documentation can be confusing, and setting up consistent KDL applications can be tedious. For this reason, the HKU team code includes a "wrapper" for ease of use of KDL.

The HKU package "example_kdl" shows how you can use KDL to compute forward kinematics and Jacobians. One consequence of using different open-source efforts within ROS is that such independent efforts have different histories with different class and datatype definitions. For example, a 3-vector for use with a transform listener is instantiated with: tf::Vector3. A 3-vector in KDL (which is actually an object with member functions) is instantiated with: KDL::Vector. In yet a third (independent) effort, the package "Eigen" is very useful for linear-algebra operations. Eigen defines a class for 3-vectors (also with member methods) as: Eigen::Vector3d. One cannot simply copy a vector of one type to another type (because are incompatible objects). One must translate from one type to another. For the HKU wrapper of KDL, "Eigen" data types are used (and translated internally as part of the wrapper). Eigen types are preferred, since these offer considerable advantages for linear-algebra operations.

In the example code "example_kdl.cpp", headers for the Eigen library are included. Most of these headers are not needed for this simple example, but they are useful for some of the more advanced Eigen methods (e.g., for analyzing Jacobian null-space options). The Eigen package is referenced in the "package.xml" file as well. "KDL" is also referenced as a dependency in the "package.xml" file. However, no KDL headers are listed explicitly, as these are brought in by the wrapper code. The wrapper code "task_variables" is also listed as a dependency in the "package.xml" file, and a corresponding header file is included in the example source code. Note: this wrapper was written by

Vladimir Ivan of U. Edinburgh for TeamHKU. In ongoing work at U. Edinburgh, the “task_variables” package is being replaced by an improved package, “Kinematica”, which is also built on KDL (not yet released at the time of this writing).

In the example source code (example_kdl.cpp), one instantiates an object of type TaskVariableXYZRPY (called “fwd_kin_solver” in the example code). The following lines of code are invoked to initialize the solver, including specifying the target and reference frames, and acquiring kinematic model details based on the current robot model in use. (The robot parameters are made available through ROS's “parameter server”).

```
TaskVariableSolver TemporaryHelper;
TemporaryHelper.Init();
fwd_kin_solver_ = new TaskVariableXYZRPY(TemporaryHelper.my_tree);
fwd_kin_solver_->Init(std::tuple<std::string, std::string>("pelvis","right_grasp_frame"));
```

In performing initialization, the solver discovers that there are only 9 joints involved in determining the pose of the target frame (right hand grasp) relative to the reference frame (pelvis). Consequently, Jacobians that it computes will have only 9 columns. The column numbering corresponds to joints ordered sequentially from the reference frame to the target frame. For this example, the first three torso joints are numbered the same as Atlas's convention (0,1,2). However, the remaining joints that affect the target frame are joints 22 through 27 (per Atlas definition), but will be found as joints 3 through 8 (in the same order) with respect to the 6x9 Jacobian computed by KDL.

Use of the kinematic solver is illustrated by the following lines:

```
Eigen::MatrixXd g_J;  //Jacobian from (28) joint angles onto chosen target frame, w/rt chosen
//reference frame (nominally, pelvis)
Eigen::VectorXd g_x; // forward kinematics answer: x,y,z,r,p,y
// vector to store joint angles for which Jacobian and fwd kin solutions are desired
Eigen::VectorXd g_q;
 // vector to hold joint angles initialized to all zeros
  g_q = Eigen::MatrixXd::Zero(28, 1);

  // default test values of joint angles are all zero.  Try some alternative values:
 g_q(23)=3.14159/2;  // lower the right arm from the shoulder, jnt 23;

//  invoke solver; need to use .data() to convert from Eigen type to compatible kdl argument
// results of the solver are filled into g_J and g_x
fwd_kin_solver_->Solve(g_q.data(), g_J, g_x);
```

The q, x and J objects are all “Eigen” types. In invoking the member method “Solve” of the fwd_kin_solver object, only the data (28 values) from the Eigen object are used. Although only 9 joints affect the solution, all 28 joint values must be provided (in original Atlas numbering convention). The solver will then populate the (Eigen-type) objects g_J and g_x with a 6xN Jacobian (where “N” is the number of involved joints—9, in this example) and a 6x1 forward kinematic solution.

Conveniently, the “cout” operator nicely formats “Eigen” vectors and matrices for debugging display. The line:

`std::cout<<g_x.transpose()<<std::endl;` // output is x,y,z,R,P,Y,

uses the member function “transpose()”, which turns the column vector g_x into a row vector (in this case, for more convenient display).

The Eigen matrix “g_J” is the 6x9 matrix that relates perturbations (or velocities) of the 9 influential joints to perturbations (or velocities) of the target-frame position and orientation (e.g., twist vector).

The Eigen object “g_x” is not actually a vector. It is used as a convenient container to hold the forward-kinematics solution. The first three components correspond to the coordinates of the origin of the target frame. The last three components are roll, pitch and yaw angles (which is one means of defining target-frame orientation).

Running this example *does* require that the ROS parameter server is informed of the robot model parameters, and also that a “roscore” is running. This can be satisfied by starting up Gazebo/drcsim, e.g. using: `roslaunch hku_worlds drc_practice_task_4.launch` (which brings up Atlas in one of the example world scenarios). Once Gazebo is running, the example code can be run with the command:
`rosrun example_kdl example_kdl`

This node will run one iteration of kinematic computations (ignoring Atlas' actual pose, using only the hard-coded joint angles), display the results of the forward kinematics and the Jacobian, then return. Different joint angle values may be specified in the source file, and the resulting solutions may be interpreted with respect to Atlas.

A second example, “example2_kdl.cpp”, instead gets its joint values from drcsim's (alternatively, the physical robot's) publication of atlas/atlas_state. These values are used to compute forward kinematics and a right-hand Jacobian, which are updated and displayed at 2Hz (set by the chosen timer frequency). The robot can be repositioned (e.g., using play_file trajectories), and the forward-kinematic results will be updated every 0.5 seconds. The KDL code is fast enough that computations can keep up with the full bandwidth (1kHz) of Atlas state publications. The 2Hz rate is only for illustration and human readability.

In example2, the top 3 rows of the solution (position of the target-frame origin) are stripped off and installed in a proper 3-vector (p_target). The last 3 values of the solution (R,P,Y angles) are used within an “Eigen” method to convert this representation into a corresponding 3x3 rotation matrix. The results are displayed to the user in a terminal via “cout” output.

A Numerical Inverse-Kinematics Approach for Positioning: A sub-problem of inverse kinematics is to solve for reaching a goal position (3-D Cartesian coordinates of the origin of a frame of interest). The solution will be a set of joint angles that results in placing the origin of the frame of interest coincident with a specified goal position. Pursuing our present example, we consider the origin of the right-hand grasp frame (expressed with respect to the pelvis frame), as influenced by the 6 joints of the right arm (i.e., with torso joints frozen). Since there are 6 joints available to satisfy 3 constraints (the x, y and z coordinates of the right-hand grasp frame origin), the problem is underdetermined. Nonetheless, it is often/easily the case that there are no solutions (e.g., merely pick the desired hand origin to be beyond reach of the robot, and there will be no IK solutions). Assuming it is possible to reach the goal, typically there will be an infinity of solutions. Nonetheless, it can be difficult to find *any* of these solutions.

A numerical approach to finding a solution uses the pseudo-inverse of the Jacobian. This is partially illustrated with example3_kdl.cpp. In this program, the 3x6 translational Jacobian is extracted from the full 6x9 Jacobian corresponding to right-hand pose as a function of torso and right-arm joints. This Jacobian, called Jtrans_3x6, has the relationship:
dp = Jtrans_3x6 * dq6
where "dp" is a perturbation of the hand origin (3-D Cartesian, expressed in pelvis coordinates) as a function of a 6-vector of right-arm joint perturbations, "dq6". Given a vector of joint angles, q28, we can compute forward kinematics to obtain the corresponding Cartesian coordinates of the right hand, p_hand. We desire to find a solution q28_soln, that results in p_hand = p_desired, where p_desired is a 3-vector of specified Cartesian coordinates of a desirable position of the right-hand grasp-frame origin, expressed with respect to the pelvis frame.

For some guess of joint angles, q6 (a subset of q28), there is an error in p_hand. The vector p_err = p_desired – p_hand is a 3-vector that points from the current hand pose to the desired hand pose. Since Jacobians are defined with respect to perturbations, we could choose to scale down this vector (if it is not small) to obtain dp: a three-vector of a desirable hand-origin perturbation. We can then ask the question, can we find a 6-vector, dq6, that achieves the desired hand perturbation, dp. Equivalently, can we solve:
dp = Jtrans_3x6 dq6
for dq6, given dp and given Jtrans_3x6.

If Jtrans_3x6 were square, one could solve for dq by inverting the Jacobian, and premultiplying dp by this inverted Jacobian. However, since Jtrans_3x6 is not square, there is no unique inverse. One candidate inverse is the "pseudo-inverse", which yields a solution with the minimum norm of dq6.

Using available member functions in Eigen, a solution can be found with the operation:

dq6_soln = Jtrans_3x6.jacobiSvd(Eigen::ComputeThinU | Eigen::ComputeThinV).solve(dp_hand);

(see example3_kdl.cpp). The resulting 6-vector dq6 may be added to the current guess for the joint-space solution to obtain an improved solution. However, if the value of dp_hand is too large, then the assumption of linearity (constant Jacobian) will be violated, and the adjustment to the solution may result in an even worse hand error. Thus, it is important to take small steps if using this algorithm.

After updated q6 with q6+= dq6, the process may be repeated. The new joint angles result in a new hand pose. This new hand pose results in a new hand error, dp_hand. Also, the Jacobian should be recomputed at the new joint angles. The new Jacobian and new dp_hand lead to a new dq6, and thus a new q6. This process can be repeated iteratively until dp_hand is small enough, at which point the algorithm terminates with the joint-space solution q6.

The Jacobian Null Space: Another useful method in Eigen is:

```
Eigen::MatrixXd ker = Jtrans_3x6.fullPivLu().kernel();
```

(also invoked in example3_kdl.cpp). This operation yields a "ernel" comprised of columns of null-space vectors in joint space. That is, any one of these columns, when premultiplied by the translational Jacobian, will result in zero perturbation of the hand origin. This also means that the particular solution obtained previously is augmented by a family of solutions described by:
dq6_soln += Ker*w_vec
where "Ker" is the kernel (typically 6x3, except at singularities) and "w_vec" is a vector of weights (3x1, for a 6x3 kernel). The weight vector may be filled with any (typically, 3) values and the result

will yield another valid solution for q6 (although this only applies to perturbations).

This motion can be visualized anthropomorphically. Try putting your hand out in front of you (with elbow bent). Imagine a dot on your palm. Now, move your right arm while trying to keep the dot on your palm stationary in space. Most noticeably, your elbow can swing up and down while keeping the palm-dot stationary (and there are many more alternative motions that keep the palm-dot stationary). All of these arm motions are "null-space" motions.

Having a null space (in this case, a 3-D null space, corresponding to the three kernel columns and the three weights in w_vec) is an opportunity to satisfy additional tasks beyond the stated task (i.e. placing the palm origin at specify 3-D coordinates). One useful additional consideration is to attempt to avoid joint limits. The algorithm described above provides a means to discover some q6 that achieves p_hand = p_desired. However, the solution obtained might not be feasible—notably, because the angles obtained might be outside the physically-possible range of motion.

To attempt to address this concern, the algorithm may include a dq6 contribution generated from null-space vectors that act to avoid joint limits. A simple implementation is to try to attract the joints towards their respective center or range of motion. A vector, q_ctrs, is defined in example3_kdl.cpp, which is hard-coded with the joint angles corresponding to center of range of motion for all 6 right-arm joints. A particular joint-space solution, q6_soln, may be far from these joint centers. To try to find an alternative solution further from joint limits, it is desirable to move in the direction:
dq_to_ctr = q_ctrs – q6_soln;
However, attraction towards the centers of range of motion should not sacrifice the condition that p_hand = p_desired. Thus, the centering motion, dq_to_ctr , should be projected onto the null space, resulting in a dq perturbation that tends to attract towards joint-range centers while not affecting the hand position. Note that this correction also must be scaled such that it is a perturbation.

Numerical Inverse Kinematics Including Orientation: So far, we have only considered use of the translational Jacobian and finding solutions that satisfy achieving goal hand origin coordinates. More commonly, one would want to achieve a fully specified hand pose—both position and orientation. To extend the analysis, we must make use of the angular Jacobian and define an orientation error.

For a full specification of desired pose, one would provide both **p**_desired (a 3x1 vector of desired origin of the frame of interest) and **R**_desired (a 3x3 orientation matrix describing the desired orientation of the frame of interest).

Assuming an iterative approach, with **q**_est being the current estimate for a joint-space solution, forward kinematics yields the corresponding frame-origin location, **p**_hand, and frame orientation, **R**_hand. The position error is: d**p** = **p**_desired – **p**_hand (identical to that defined earlier).

For orientation error, we may consider performing a rotation, expressed as a 3x3 matrix R_delta, such that: **R**_desired = **R**_delta***R**_hand. Since both **R**_desired and **R**_hand are known, we can solve for **R**_delta as: $\boldsymbol{R}_{delta} = \boldsymbol{R}_{desired}\,\boldsymbol{R}_{hand}^{-1}$.

The desired rotation can be re-described in terms of a angle-axis notation, comprised of a rotation angle, rot_err_ang, about a rotation axis, a_axis (a 3x1 vector, expressed in the reference frame). Conversion from an R matrix to angle-axis notation is described, e.g. at:
http://en.wikipedia.org/wiki/Axis%E2%80%93angle_representation

The conversion may be implemented as an axis of rotation computed as:

$\boldsymbol{a}$ = [R_delta(3,2)-R_delta(2,3);
R_delta(1,3)-R_delta(3,1);
R_delta(2,1)-R_delta(1,2)];
(with Matlab-style indexing of matrices, starting from 1)

and the unit-length axis vector is normalized as:
$\hat{\boldsymbol{a}} = \boldsymbol{a}/\|\boldsymbol{a}\|$

The corresponding angle of rotation to impose about the above axis of rotation is given by:

$\delta\theta_{err}$ = acos(0.5*(R_delta(1,1)+R_delta(2,2)+R_delta(3,3)-1));

Both the displacement error, d**p**, and the rotation error, $\delta\varphi = \hat{\boldsymbol{a}}\,\delta\theta_{err}$, must be scaled down (if necessary) to qualify as "small" perturbations. If small, these can be stacked into a single 6-vector to construct a corresponding "twist" vector, **t,** where $\boldsymbol{t}^T = \left[\delta\boldsymbol{p}^T, \delta\varphi^T\right]/dt$

with the relation:
$\boldsymbol{t}\,dt = \boldsymbol{J}\,\delta\dot{\boldsymbol{q}}\,dt = \boldsymbol{J}\,\delta\boldsymbol{q}$

In the above, the factor "dt" is a time increment, introduced to reconcile units when referring to a "twist" vector comprised of velocities. This factor scales the twist vector arbitrarily, so it can be used to scale down **t** so that $\delta\boldsymbol{q}$ is a perturbation.

If only the 6 arm joints are involved, then the Jacobian is 6x6, $\delta\boldsymbol{q}$ is 6x1 and **t** is 6x1. When solving for $\delta\boldsymbol{q}$, the solution will be unique, provided the Jacobian is non-singular. Jacobians that are nearly singular lead to numerical instability, so this condition should be considered. Additionally, the rotation angle solution should be coerced into the closest solution option of arc-cosine.

In this generalization, the computed value of $\delta\boldsymbol{q}$ should be added to the estimated solution **q**_soln. The entire process is then repeated, iteratively, computed forward kinematics, updated **p**_hand, **R**_hand and Jacobian, leading to updated twist, **t**, and a new computed $\delta\boldsymbol{q}$. The loop is repeated until convergence to the goal meets some tolerance—or until some other condition dictates termination (e.g., out-of-reach detected or ill-conditioned Jacobian). If a solution is achieved, then the corresponding joint angles must be tested against the range-of-motion limits to see if the solution is viable. For a 6-DOF pose specification and 6 joint angles to adjust, there is no null space to exploit to attempt to avoid joint limits.

For Atlas' arms, full 6-DOF inverse-kinematic solutions are rare. Desired poses are almost always unobtainable. As a consequence, it is useful to first explore forward kinematics to understand what poses in space are reachable. Within such regions, numerical inverse-kinematic solutions may be pursued productively.