Interfacing to the DARPA Robotics Challenge simulator 2.0:

Wyatt Newman

February, 2013

These notes are an update to the November, 2012 document "Interfacing to the DARPA Robotics Challenge simulator." The present document describes changes for interfacing to the 2.0 release of the drcsim simulator and adds some additional descriptions of the Atlas robot model.

Installation of the drcsim-2.0 simulator should be found in the directory: xxx /usr/share/drcsim-2.0.

Robot URDF Model File:

The robot model (including inertial properties, visual properties, collision properties, and kinematic properties) is contained in the file: /usr/share/ros/models…atlas_sandia_hands.urdf. (This model includes the Sandia hands—alternative models are also present). The URDF (Universal Robot Descriptor Format) file describes 28 movable joints of the robot, plus 12 joints each in the right and left hands. Sensor frames are also defined. (See the accompanying graphical display of frames and joints). A description of the URDF format can be found at: http://www.ros.org/wiki/urdf/Tutorials.

An abbreviated example of link and joint specifications, extracted from the atlas_sandia_hands.urdf file, appears below:

```
</link>
  <link name="utorso">
    <inertial>
      <mass value="18.484"/>
      <origin rpy="0 -0 0" xyz="0.02 -0.001 0.211"/>
      <inertia ixx="0.395" ixy="0" ixz="0.083" iyy="1.089" iyz="-
0.003" izz="0.327"/>
    </inertial>
  </link>



<link name="r_clav">
    <inertial>
      <mass value="2.369"/>
      <origin rpy="0 -0 0" xyz="0.014 -0.058 0.029"/>
      <inertia ixx="0.004" ixy="-0.001" ixz="0" iyy="0.006" iyz="0"
izz="0.007"/>
    </inertial>
  </link>
```

These extracted lines define two links: "utorso" and "r_clav". Each link has an associated coordinate frame. The link's mass and its center of mass location is specified relative to the link's coordinate frame. E.g., for r_clav, the mass is 2.369kg, and the center of mass is located at (x,y,z) = (`0.014, -0.058, 0.029)`, as measured in the link's reference frame.

The rotational inertia for this link is (with units kg-m^2)

$$\mathbf{I} = \begin{bmatrix} 0.004 & -0.001 & 0 \\ -0.001 & 0.006 & 0 \\ 0 & 0 & 0.007 \end{bmatrix}$$

The off-diagonal terms indicate that the inertia is not specified with respect to principal axes. Rather, the moments of inertia are expressed with respect to a frame with origin at the center of mass and axes oriented by a roll-pitch-yaw description relative to the link's reference frame. For the r_clav example, the inertial-frame orientation is `rpy="0 -0 0"`, i.e. oriented coincident with the link's reference frame. (If the rotational inertia matrix had been diagonal, this would have indicated that the principal axes are coincidentally aligned with the link-frame axes).
Link specifications have additional properties (not shown here), including visualization properties (the display appearance) and collision properties (typically a solid model, such as one or more cylinders, that is simpler to check for collisions than the more detailed visual model).

A kinematic chain of links is defined by specifying *joints* that connect links. For the above links, there is a revolute (hinge-like) joint called `r_arm_usy`, defined as follows:

```
<joint name="r_arm_usy" type="revolute">
  <origin rpy="0 -0 0" xyz="0.024 -0.221 0.289"/>
  <axis xyz="0 0.5 -0.866025"/>
  <parent link="utorso"/>
  <child link="r_clav"/>
</joint>
```

A joint relates a "parent" link (utorso, in this example) to a "child" link (r_clav, in this example). The child link's frame is defined such that its origin lies on the parent/child joint axis. E.g., the origin of the r_clav link lies on the rotational axis of the r_arm_usy joint. It is also necessary to define the orientation of the child frame relative to the parent's frame. This specification seems ambiguous in the URDF description, but the apparent intent is that this orientation (given by: `rpy="0 -0 0"` for the current example) is the orientation of the child frame relative to the parent frame *when the value of joint-angle rotation of `r_arm_usy` is zero*. (Clearly, the child-link frame changes relative to the parent frame as a function of joint angle for the connecting joint, so there are no static values of rpy that specify this relationship in general).

Although the child-link's frame has an origin that lies on the parent/child connecting joint axis, the parent/child joint axis is not necessarily conveniently aligned with any of the frame axes. For the case of the `r_arm_usy` joint, the joint is oriented as:

```
<axis xyz="0 0.5 -0.866025"/>
```

which defines a unit direction vector, specified in the child-link's frame. Note that although the child link's frame rotates relative to the parent link, the joint-axis expressed in the child-link frame remains valid. (The direction of the parent/child joint axis could also have been expressed in the parent frame, and this specification would also be constant as the robot moves. The choice to express it in the child frame appears arbitrary, and unfortunately inconsistent with specification of the "origin" values, which are expressed in the parent frame).

Most of the joints in the Atlas model are aligned with one of the child-link frame axes. The clavicle is a notable exception, as this joint is tilted 30-degrees relative to the "spine" (z-axis of the utorso frame).

The complete specification of the `r_arm_usy` joint follows:

```
<joint name="r_arm_usy" type="revolute">
    <origin rpy="0 -0 0" xyz="0.024 -0.221 0.289"/>
    <axis xyz="0 0.5 -0.866025"/>
    <parent link="utorso"/>
    <child link="r_clav"/>
    <dynamics damping="0.1" friction="0"/>
    <limit effort="212" lower="-1.9635" upper="1.9635" velocity="12"/>
    <safety_controller k_position="100" k_velocity="100"
soft_lower_limit="-11.9635" soft_upper_limit="11.9635"/>
  </joint>
```

The joint specification includes properties of the joint motion, including range-of-motion limits (`lower="-1.9635" upper="1.9635"`) specifies a minimum and maximum angle of the range of motion (in radians) . The corresponding actuator has a maximum torque of `effort="212"` N-m. The joint also has viscous friction, defined as `damping="0.1"` N-m, but Coulomb friction has been set to zero. The actuator also has a velocity limit of `velocity="12"` rad/sec. A safety controller is defined to help prevent hitting angle hard stops at high speed.

Sending joint commands to the Atlas simulator:

This section is based on the tutorial example provided at:
http://gazebosim.org/wiki/Tutorials/drcsim/2.0/sending_joint_controller_commands_over_ros

In release 2.0, the drcsim simulator has joint commands bundled in a single structure

```
osrf_msgs::JointCommands jointcommands;
```

This message type contains vectors for: name, position, velocity, effort, kp_position, ki_position, kd_position, kp_velocity, i_effort_min, i_effort_max. The vectors are of length 24 (the number of movable body joints). Joint positions are commanded by populating jointcommands.position[i] for all 24 joints, then publishing the jointcommands message on the topic "/atlas/joint_commands". Control-gain values for proportional-error feedback may be set in the jointcommands.kp_position[i] entries, and these values will be used when the jointcommands message is published. Integral-error (ki), derivative (kd) and velocity gains (kp) may be specified. If non-zero integral-error feedback is used, then i_effort_min and i_effort_max values should be set as anti-windup protection. In addition to actuator torques produced by PID feedback of a position controller, actuator torques may be specified explicitly via the "effort" terms. Velocity commands may be used in addition to position commands.

In the tutorial example, there are 24 lines of the form:

```
  jointcommands.name.push_back("atlas::back_lbz");
  jointcommands.name.push_back("atlas::back_mby");
…
```

These commands establish the relationship between joint names and index numbers for commanding the Atlas joints. (e.g., index 0 is the back_lbz axis, index 1 is the back_mby joint, etc.).

An extension of the Gazebo drcsim-2.0 tutorial for joint commands follows. The example code sequences through a list of pre-defined joint-space poses (specification of 24 joint angles). Each of these goal states is reached through smooth, coordinated motion of all 24 joints, arriving at the respective desired joint angles simultaneously over a move duration specified for each goal. The code runs with the atlas simulator launched as:

```
roslaunch atlas_utils atlas_sandia_hands.launch
```

The accompanying code should be compiled with:

```
rosmake atlas_jnt_ctl
```

and should be run with:

```
rosrun atlas_jnt_ctl atlas_jnt_ctl
```

The code consists of four C++ files and four header (*.h) files. The specific example does the following: starting from the "home" position (standing with legs together and arms outstretched in an "iron cross" position), the robot tips backwards, falling to the ground on its back. The robot then brings its arms forward, leans forward from the waist, and lifts and opens its legs. As a result, the robot sits upright on the ground. This is accomplished in 3 steps, each assigned 3.0 seconds to complete. A discussion of illustrative sections from each file follows. The full code is attached.

The file “atlas_jnt_ctl_main.cpp” has as its main loop the following code:

```
while(ros::ok()) // main loop
{
 //moveSequencer.update();   //advances through  moves; keeps track  of moveNum, q_des, hasChanged
//if (moveSequencer.hasChanged()) {
//    moveSequencer.getQDes(qDes);
//    atlasJointControl.setNewJointSpaceGoal(qDes);
if(atlasJointControl.isMoveDone()) { // test if time to sequence to next move
if (!moveSequencer.isDoneWithAllMoves())
     ROS_INFO("prev  move done; sequence to next: ");
  { //can attempt to advance to next move--get message if unable
    moveSequencer.update(goalAngles);
    atlasJointControl.setNewJointSpaceGoal(goalAngles, moveSequencer.getMoveDuration());
    ROS_INFO("move ID = %d",moveSequencer.getCurrentMoveSequenceID());
    ROS_INFO("move duration:  %f",moveSequencer.getMoveDuration());

   }
  }
 atlasJointControl.update();  //compute/publish incremental motion commands towards current  goal
//atlasJointControl.test_tf(tf_listener); // this shows how to pass the tf_listener  to a method
                  // for use by atlasJointControl.   Will need this for Jacobians
 looprate.sleep();
 ros::spinOnce();
}
```

This code checks if a gross move is complete. If the most recent goal is still unattained, the robot is commanded to move incrementally towards that goal, with the intent of arriving at the specified “moveDuration” time. This is accomplished with the member method “update()” of the object “atlasJointControl”.

If the most recent goal state has been achieved, then a new goal state (if available) is established. This is done with the member function “update()” of the object “moveSequencer,” which copies goal angles from a pre-populated list. (In the future, this list may be populated dynamically by another node, but in this illustration, the goals are pre-established). Values of the next goal (28 angles) are populated in the reference argument, “goalAngles.” These goals angles and the corresponding move duration are imposed through the action of: atlasJointControl.setNewJointSpaceGoal().

The class designs of MoveSequencer and AtlasJointControl are appear in *.cpp and *.h files of the same names.

The implementation of the “update()” method of MoveSequencer is:

```
// update() populates goalAngles with next macro joint-space goal
// and sets corresponding moveDuration_ value
void MoveSequencer::update(double goalAngles[NJoints]){
  if (currentMoveSequenceID_< lastValidMoveSequenceID_) {
    currentMoveSequenceID_++;
    for (unsigned int i=0;i<NJoints;i++)
      goalAngles[i]=moveSequenceGoalQueue_[currentMoveSequenceID_][i];
    moveDuration_ = moveSequenceDurationQueue_[currentMoveSequenceID_];
  }
  //if at last valid move, do nothing to update index, time or goal angle values
  else {
    doneWithMoves_ = true;
  }
}
```

In this member function, if there is another valid goal state in the moveSequenceGoalQueue, then this next move is copied to goalAngles, and the associated move duration is updated as well. The move number is incremented to prepare for the next copy—or to detect if the array of goals is exhausted.

The atlasJointControl.update() function uses an object of class SmoothJointSpaceMove to compute an incremental update towards a macro goal. The update() method populates the vector qOut with updated angle commands and dDotOut with consistent joint velocities. These values are used by atlasJointControl by copying them to the respective elements of the jointCommands message, which is then published on the "/atlas/joint_commands" topic. This causes the Atlas simulator to control its joints to these positions and velocities. The AtlasJointControl::update() function follows:

```
void AtlasJointControl::update(){
  double qOut[NJoints];
  double qDotOut[NJoints];
  // main fnc of this class: call this once per control cycle
  // hand off all the work to the smoothJointSpaceMove object, which
  // will fill the vectors aOut and qDotOut with updated commands
  smoothJointSpaceMove.update(qOut,qDotOut);
  moveDone_ = smoothJointSpaceMove.getMoveDone(); //make moves status known to AtlasJointControl
  // copy these new joint angle and joint velocity commands to jointCommands_ for publication
  for (unsigned int i=0;i<NJoints;i++) {
  jointCommands_.position[i]=qOut[i];
  jointCommands_.velocity[i]=qDotOut[i];
  }
  pub_joint_commands_.publish(jointCommands_); // send out joint commands to Atlas
  // displayCurrentAngles(); //DEBUG
}
```

Initialization of a "MoveSequencer" object includes invoking the member function:

```
void MoveSequencer::initMoveSequences()
```

This function includes blocks of code like the following:

```
// step 1: from home position, move arms backwards and toe off to fall bkwds;
ang_goals[r_arm_usy]=0.1;
ang_goals[l_arm_usy]=0.1;
ang_goals[r_leg_uay]= 1.0; //try to point toes
ang_goals[l_leg_uay]= 1.0;
moveTime=2.0; // take 3 seconds to do this
addSequence(ang_goals, moveTime);//push this on the queue:
```

To add more hard-coded goals to the sequence of moves, one may add more lines of this type, setting new values for desired joint angles. These joint angles will be achieved through smooth, coordinated incremental updates. Specification of goal angles must be in the order expected in the jointCommands message. While it is convenient to send over 28 commands in a single message (vs. publishing to 28 separate, named topics—one for each joint command), use of vectors loses some of the mnemonic labeling. To address this, names are associated with joint indices in the header file "joint_names.h". Lines from this file look like:

```
 //for the back:
const unsigned int back_lbz=0;
const unsigned int back_mby=1;
const unsigned int back_ubx=2;
//neck:
const unsigned int neck_ay=3;
…
```

Conclusion: This update shows how to command joint angles to the Atlas robot. The code attempts to encapsulate some of the ROS details, allowing the "main" program to be smaller. It also extends the tutorial example to produce smooth, coordinated moves in joint space.

Another extension that will be valuable is producing smooth moves in task space (Cartesian space), allowing body motions (e.g., movement of a hand) with respect to sensor values. Additionally, joint-space commands of the hands' fingers should be organized similarly.

```
// main file for atlas_jnt_ctl package: wsn  Feb 15, 2013
#include "AtlasJointControl.h"
#include "MoveSequencer.h"


int main(int argc, char** argv)
{
 double looprateHz = 10.0;

 // ROS set-ups:
 ros::init(argc, argv, "atlas_jnt_ctl"); // call this node "atlas_jnt_ctl"

 // pointer to node handle, for use by drcsim demo code, now moved to AtlasJointControl constructor
 ros::NodeHandle* rosnode = new ros::NodeHandle(); //note--need to request a nodehandle before
                                    // ros::Time works (not sure why)
 ros::NodeHandle nh_subCB;               //
 tf::TransformListener tf_listener;  // create a transform listener

 while (!ros::Time::isValid()) {} // make sure we are receiving valid clock values

 ros::Rate looprate(looprateHz); //will perform sleeps to enforce loop rate of "looprateHz" Hz
 ros::Time startTime= ros::Time::now(); // get the current time, which defines our start
                               // not really needed, but may come in handy

 //create an instance of an AtlasJointControl...  See AtlasJointControl.cpp
 AtlasJointControl atlasJointControl(rosnode,looprateHz);

 // if want to use method member of class as callback func, must do it this way:
 ROS_INFO("setting up subscriber to joint states");
 ros::Subscriber sub = nh_subCB.subscribe("/atlas/joint_states", 1,
         &AtlasJointControl::getJointStatesCB, &atlasJointControl);


 ROS_INFO("receiving joint state data...");
 for (int i=0;i<5;i++) { //wait for callbacks to get data
  // really, should test the data to make sure it is valid...not just wait and hope
  ros::Duration(0.5).sleep();
  ros::spinOnce();
 }

 ROS_INFO("current angles:");  //debug: display the received joint angles; should all be ~0.0
                         // for Atlas in start-up pose
 atlasJointControl.displayCurrentAngles();

 MoveSequencer moveSequencer;  // create a sequencer object

 ROS_INFO("starting main loop...");

 double currentAngles[NJoints];
 double goalAngles[NJoints];

//to start, command robot to go to its current pose...should stand still
 atlasJointControl.getCurrentAngles(currentAngles);
 //initialize move command to current angles:
 atlasJointControl.setNewJointSpaceGoal(currentAngles,currentAngles, 1.0);// move time=1Sec

 while(ros::ok())  // main loop
 {
```

```
  //moveSequencer.update();  //advances through moves; keeps track of moveNum, q_des, hasChanged
  //if (moveSequencer.hasChanged()) {
  //    moveSequencer.getQDes(qDes);
  //    atlasJointControl.setNewJointSpaceGoal(qDes);
  if(atlasJointControl.isMoveDone()) { // test if time to sequence to next move
    if (!moveSequencer.isDoneWithAllMoves())
        ROS_INFO("prev move done; sequence to next: ");
    { //can attempt to advance to next move--get message if unable
      moveSequencer.update(goalAngles);
      atlasJointControl.setNewJointSpaceGoal(goalAngles, moveSequencer.getMoveDuration());
      ROS_INFO("move ID = %d",moveSequencer.getCurrentMoveSequenceID());
      ROS_INFO("move duration: %f",moveSequencer.getMoveDuration());
    }
  }

  atlasJointControl.update();  //compute/publish incremental motion commands towards current goal
  //atlasJointControl.test_tf(tf_listener); // this shows how to pass the tf_listener to a method
                    // for use by atlasJointControl.  Will need this for Jacobians

  looprate.sleep();
  ros::spinOnce();
 }
 return 0;
}
```

```
#ifndef ATLAS_JOINT_CONTROL_H
#define ATLAS_JOINT_CONTROL_H

#include <math.h>
// class definition for AtlasJointControl

#include <ros/ros.h>
#include <ros/subscribe_options.h>
#include <boost/thread.hpp>
#include <boost/algorithm/string.hpp>
#include <sensor_msgs/JointState.h>
#include <osrf_msgs/JointCommands.h>
#include <tf/transform_listener.h>
#include "std_msgs/String.h"
#include "joint_names.h"
#include "SmoothJointSpaceMove.h"  //class definition for smooth move

class AtlasJointControl
{
  private:  //private member variables; use "_" suffix to indicate private member vars
  //member variables
  osrf_msgs::JointCommands jointCommands_; //holds all commands to send to Atlas

  //ros::NodeHandle xxx,xxx;
  ros::NodeHandle nh_jntPub_;  // node handle for joint command publisher
  ros::Publisher pub_joint_commands_; //here is the publisher--gets initialized in constructor

  ros::NodeHandle* rosnode_;

  //variables to store values from state callback func
  double jointAngles_[NJoints]; //these should always be the "freshest" angles available from Atlas
  double jointVels_[NJoints];
  double jointEfforts_[NJoints];

  double prevJointAngles_[NJoints]; //these get updated by jointState callbacks
  double deltaJointAngles_[NJoints];

  double loopFreq_;

  double goalJointAngles_[NJoints];

  //Jacobian matrices to be populated by callback fnc
  double J_lin_r_arm[3][9]; // translational Jacobian from pelvis to right hand
  double J_rot_r_arm[3][9]; // rotational Jacobian from pelvis to right hand

  double J_lin_r_knee[3][3]; // translational Jacobian from pelvis to right knee
  double J_lin_l_knee[3][3]; // translational Jacobian from pelvis to left knee

  bool moveDone_; // flag to tell if current move is complete--as pass from smoothJointSpaceMove

  //instantiate a SmoothJointSpaceMove object:
  SmoothJointSpaceMove smoothJointSpaceMove;


  public:    // member functions

  AtlasJointControl(ros::NodeHandle* rosnode, double loopFreq);   //constructor
```

```
 ~AtlasJointControl()
 {  //destructor...do nothing
 }

 void update();  // main fnc of this class: call this once per control cycle
 //could not make this work--member method as callback function
 // instead, made it a stand-alone function in the implementation file

 void getJointStatesCB(const sensor_msgs::JointState::ConstPtr &_js);
 void displayCurrentAngles();
 void getCurrentAngles(double qVals[NJoints])
   { for (unsigned int i=0;i<NJoints;i++) qVals[i]=jointAngles_[i]; }

 void doPushBacks();
 void initJointCommands();
 void test_tf(tf::TransformListener &tf_listener);
 //void setNewJointSpaceGoal(double newGoalJointAngles[NJoints]);
  //the following version advances goal from previous goal
 void setNewJointSpaceGoal(double newGoalJointAngles[NJoints],double moveDuration);
 //the following version explicitly specifies the start angles--not previous goal
 void setNewJointSpaceGoal(double startAngles[NJoints],
                      double newGoalJointAngles[NJoints],double moveDuration);

 bool isMoveDone() { return(moveDone_); }

 // instantiate some helpful objects here:

 // SmoothJointSpaceMove smoothJointSpaceMove; //no constructor arguments?
 // JacobianComputer jacobianComputer; // again, no constructor args?


};


#endif
```

```
// this is the implementation file (member methods) for the AtlasJointControl class

#include "AtlasJointControl.h"  //AtlasJointControl class definition, plus more #includes
#include "SmoothJointSpaceMove.h"

//constructor: initialize entities, create xform listener; instantiate useful objects
AtlasJointControl::AtlasJointControl(ros::NodeHandle* rosnode_arg, double loopFreq) {
  ROS_INFO("AtlasJointControl constructor: loopFreq arg= %f",loopFreq);
  rosnode_ = rosnode_arg; // make a persistent copy of rosnode argument
  loopFreq_=loopFreq; //and for loop frequency

  // this assignment for commanding joint states worked:
  pub_joint_commands_ = nh_jntPub_.advertise<osrf_msgs::JointCommands>(
                      "/atlas/joint_commands", 1, true);
  doPushBacks();  //sizes jointCommands_ and creates numerical indices

  initJointCommands(); // put in alternative gains and starting angles here, if desired

  //clear out   double prevJointAngles_[Njoints], deltaJointAngles_[Njoints];
  for (unsigned int i=0;i<NJoints;i++) {
    jointAngles_[i]=0.0;  //these are nonsense until the callback fnc gets real values
    prevJointAngles_[i]=0.0; //callback fnc will update these
    deltaJointAngles_[i]=0.0;
  }

  moveDone_=true;
  //smoothJointSpaceMove.initParams(q_desired,q_start,moveTime,moveDone);
  // inform smoothJointSpaceMove of the loopFreq_
  smoothJointSpaceMove.setTimeStep(loopFreq_);

  ROS_INFO("Done with initializations");
  // NOTE: must set up subscriptions from parent level; use:
  //  ros::Subscriber sub = nh_subCB.subscribe("/atlas/joint_states", 1,
  //            &AtlasJointControl::getJointStatesCB, &atlasJointControl);

}

void AtlasJointControl::update(){
  double qOut[NJoints];
  double qDotOut[NJoints];
  // main fnc of this class: call this once per control cycle
  // hand off all the work to the smoothJointSpaceMove object, which
  // will fill the vectors aOut and qDotOut with updated commands
  smoothJointSpaceMove.update(qOut,qDotOut);
  moveDone_ = smoothJointSpaceMove.getMoveDone(); //make moves status known to AtlasJointControl

  // copy these new joint angle and joint velocity commands to jointCommands_for publication
  for (unsigned int i=0;i<NJoints;i++) {
    jointCommands_.position[i]=qOut[i];
    jointCommands_.velocity[i]=qDotOut[i];
  }

  pub_joint_commands_.publish(jointCommands_); // send out joint commands to Atlas
  // displayCurrentAngles();  //DEBUG

}

//  void newMoveGoal(double q_start[NJoints],double q_goal[NJoints],double moveDuration);
```

```
void AtlasJointControl::setNewJointSpaceGoal(double newGoalJointAngles[NJoints],double moveDuration)
{
 //use the smoothJointSpaceMove object to initialize data for a new move
 // tell it to start from the most recently commanded joint angles
 smoothJointSpaceMove.newMoveGoal(goalJointAngles_,newGoalJointAngles,moveDuration);

 moveDone_=false; //reset flag: new move, so not done

 //the Goal angles may have been modified by newMoveGoal, if necessary for safety
 for (unsigned int i=0;i<NJoints;i++)
  goalJointAngles_[i]=newGoalJointAngles[i]; //accept new values into private data
}

void AtlasJointControl::setNewJointSpaceGoal(double startAngles[NJoints],
                    double newGoalJointAngles[NJoints],double moveDuration) {
 //use the smoothJointSpaceMove object to initialize data for a new move
 // tell it to start from the specified startAngles
 smoothJointSpaceMove.newMoveGoal(startAngles,newGoalJointAngles,moveDuration);

 moveDone_=false; //reset flag: new move, so not done

 //the Goal angles may have been modified by newMoveGoal, if necessary for safety
 for (unsigned int i=0;i<NJoints;i++)
  goalJointAngles_[i]=newGoalJointAngles[i]; //accept new values into private data

}

void AtlasJointControl::test_tf(tf::TransformListener &tf_listener)
{
 tf::Vector3 r_hand_origin;
 tf::StampedTransform transform;

 tf_listener.lookupTransform("/pelvis", "/r_hand", ros::Time(0), transform);
 r_hand_origin= transform.getOrigin(); // extract the origin of r_hand frame, relative to pelvis
 ROS_INFO("update: r_hand origin x,y,z = %f %f %f",r_hand_origin[0],
            r_hand_origin[1],r_hand_origin[2]);
}


//member method as callback func--but cannot do setup in constructor; just do from parent
void AtlasJointControl::getJointStatesCB(const sensor_msgs::JointState::ConstPtr &_js)
{
 for (unsigned int i=0;i<NJoints;i++)
  {
    prevJointAngles_[i]=jointAngles_[i];

  jointAngles_[i]=_js->position[i];  //fill these AtlasJointControl-private variables
  jointVels_[i]=_js->velocity[i];
  jointEfforts_[i]=_js->effort[i];

  deltaJointAngles_[i]=jointAngles_[i]-prevJointAngles_[i];
  }
}

void AtlasJointControl::displayCurrentAngles(){
 // simply displays joint angles using ROS_INFO()

 ROS_INFO(" ");
```

```
 ROS_INFO("back and neck:  %f %f %f %f",jointAngles_[0],jointAngles_[1],jointAngles_[2],jointAngles_[3]);
 ROS_INFO("l_leg  uhz to lhy: %f %f %f", jointAngles_[4],jointAngles_[5],jointAngles_[6]);
 ROS_INFO("l_leg kny to lax:  %f %f %f", jointAngles_[7],jointAngles_[8],jointAngles_[9]);
 ROS_INFO("r_leg  uhz to lhy: %f %f %f", jointAngles_[10],jointAngles_[11],jointAngles_[12]);
 ROS_INFO("r_leg kny to lax:  %f %f %f", jointAngles_[13],jointAngles_[14],jointAngles_[15]);

 ROS_INFO("l_arm  usy to ely: %f %f %f", jointAngles_[16],jointAngles_[17],jointAngles_[18]);
 ROS_INFO("l_arm elx to mwx:  %f %f %f", jointAngles_[19],jointAngles_[20],jointAngles_[21]);

 ROS_INFO("r_arm  usy to ely: %f %f %f", jointAngles_[22],jointAngles_[23],jointAngles_[24]);
 ROS_INFO("r_arm elx to mwx:  %f %f %f", jointAngles_[25],jointAngles_[26],jointAngles_[27]);

}

void AtlasJointControl::doPushBacks() { //tediousness from drcsim tutorial
 //joint commands is now an N-element object; joints are referred to by index
  // must match those inside AtlasPlugin
  jointCommands_.name.push_back("atlas::back_lbz");
  jointCommands_.name.push_back("atlas::back_mby");
  jointCommands_.name.push_back("atlas::back_ubx");
  jointCommands_.name.push_back("atlas::neck_ay");
  jointCommands_.name.push_back("atlas::l_leg_uhz");
  jointCommands_.name.push_back("atlas::l_leg_mhx");
  jointCommands_.name.push_back("atlas::l_leg_lhy");
  jointCommands_.name.push_back("atlas::l_leg_kny");
  jointCommands_.name.push_back("atlas::l_leg_uay");
  jointCommands_.name.push_back("atlas::l_leg_lax");
  jointCommands_.name.push_back("atlas::r_leg_uhz");
  jointCommands_.name.push_back("atlas::r_leg_mhx");
  jointCommands_.name.push_back("atlas::r_leg_lhy");
  jointCommands_.name.push_back("atlas::r_leg_kny");
  jointCommands_.name.push_back("atlas::r_leg_uay");
  jointCommands_.name.push_back("atlas::r_leg_lax");
  jointCommands_.name.push_back("atlas::l_arm_usy");
  jointCommands_.name.push_back("atlas::l_arm_shx");
  jointCommands_.name.push_back("atlas::l_arm_ely");
  jointCommands_.name.push_back("atlas::l_arm_elx");
  jointCommands_.name.push_back("atlas::l_arm_uwy");
  jointCommands_.name.push_back("atlas::l_arm_mwx");
  jointCommands_.name.push_back("atlas::r_arm_usy");
  jointCommands_.name.push_back("atlas::r_arm_shx");
  jointCommands_.name.push_back("atlas::r_arm_ely");
  jointCommands_.name.push_back("atlas::r_arm_elx");
  jointCommands_.name.push_back("atlas::r_arm_uwy");
  jointCommands_.name.push_back("atlas::r_arm_mwx");


  unsigned int n = jointCommands_.name.size();
  if (NJoints!=n)
      ROS_WARN("UH-OH!  NJoints = %d and n = %d",NJoints,n);
  else ROS_INFO("NJoints = n...whew!");

  jointCommands_.position.resize(n);
  jointCommands_.velocity.resize(n);
  jointCommands_.effort.resize(n);
  jointCommands_.kp_position.resize(n);
  jointCommands_.ki_position.resize(n);
  jointCommands_.kd_position.resize(n);
```

```
  jointCommands_.kp_velocity.resize(n);
  jointCommands_.i_effort_min.resize(n);
  jointCommands_.i_effort_max.resize(n);
  ROS_INFO("done resizing");

  for (unsigned int i = 0; i < n; i++)
  {
    std::vector<std::string> pieces;
    boost::split(pieces, jointCommands_.name[i], boost::is_any_of(":"));

    rosnode_->getParam("atlas_controller/gains/" + pieces[2] + "/p",
      jointCommands_.kp_position[i]);

    rosnode_->getParam("atlas_controller/gains/" + pieces[2] + "/i",
      jointCommands_.ki_position[i]);

    rosnode_->getParam("atlas_controller/gains/" + pieces[2] + "/d",
      jointCommands_.kd_position[i]);

    rosnode_->getParam("atlas_controller/gains/" + pieces[2] + "/i_clamp",
      jointCommands_.i_effort_min[i]);
    jointCommands_.i_effort_min[i] = -jointCommands_.i_effort_min[i];

    rosnode_->getParam("atlas_controller/gains/" + pieces[2] + "/i_clamp",
      jointCommands_.i_effort_max[i]);

    jointCommands_.velocity[i]     = 0;
    jointCommands_.effort[i]       = 0;
    jointCommands_.kp_velocity[i]  = 0;
  }
  //ROS_INFO("done w/ pushback method");
}


/*
rosmsg show osrf_msgs/JointCommands:
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort
float64[] kp_position
float64[] ki_position
float64[] kd_position
float64[] kp_velocity
float64[] i_effort_min
float64[] i_effort_max
*/

void AtlasJointControl::initJointCommands() {
  // wsn: choose gains and initial values here... display the gains

    //provide damping on back, neck and legs
    for (unsigned int i = 0; i <= r_leg_lax; i++)
        {
```

```
          //jointCommands_.kp_position[i] =0.0; //turn off proportional gain
          jointCommands_.ki_position[i] = 0.0; // turn off integral gain
          jointCommands_.kp_velocity[i]  = 1.0; // provide damping
        }

    // increase proportional gain on back_lbz joint
    jointCommands_.kp_position[back_lbz]= 2000.0;
    jointCommands_.kp_velocity[back_lbz]  = 2.0;

    // increase proportional gain on uhz leg joints
    jointCommands_.kp_position[r_leg_uhz]= 100.0;
    jointCommands_.kp_position[l_leg_uhz]= 100.0;
    jointCommands_.kp_position[r_leg_mhx]= 100.0;
    jointCommands_.kp_position[l_leg_mhx]= 100.0;

    //hips need more damping:
    jointCommands_.kp_velocity[r_leg_uhz]  = 20.0;
    jointCommands_.kp_velocity[r_leg_uhz]  = 20.0;
    jointCommands_.kp_velocity[r_leg_mhx]  = 20.0;
    jointCommands_.kp_velocity[l_leg_mhx]  = 20.0;

   ROS_INFO("Init jointCommands_");
   for (unsigned int i=l_arm_usy;i<=r_arm_mwx;i++)
     jointCommands_.kp_velocity[i]=2.0; // add some damping to all arm DOF's

   // display right-arm gains:
   ROS_INFO("right-arm gains: Kp, Kv: ");
   for (unsigned int i=r_arm_usy; i<=r_arm_mwx;i++)
    ROS_INFO("%f  %f",jointCommands_.kp_position[i], jointCommands_.kp_velocity[i]);

   ROS_INFO("back-joint gains: Kp, Kv: ");
   for (unsigned int i=back_lbz;i<=back_ubx;i++)
    ROS_INFO("%f  %f",jointCommands_.kp_position[i], jointCommands_.kp_velocity[i]);

   ROS_INFO("right-leg gains: Kp, Kv: ");
   for (unsigned int i=r_leg_uhz; i<=r_leg_lax; i++)
    ROS_INFO("%f  %f",jointCommands_.kp_position[i], jointCommands_.kp_velocity[i]);


}
```

```
//MoveSequencer.h
#ifndef MOVE_SEQUENCER_H
#define MOVE_SEQUENCER_H

#include <math.h>
// class definition for AtlasJointControl

#include <ros/ros.h>
#include "joint_names.h"

const int goalQueueSize=256;

class MoveSequencer
{
  private:  //private member variables; use "_" suffix to indicate private member vars
  //variables to store values from state callback func
  double goalJointAngles_[NJoints];
  double moveDuration_;
  int currentMoveSequenceID_;
  int lastValidMoveSequenceID_;
  bool doneWithMoves_;
  bool queueFull_;

  double moveSequenceGoalQueue_[goalQueueSize][NJoints];  // ugly; need to define a circular buffer
                                        // of "goal" objects
  double moveSequenceDurationQueue_[goalQueueSize];       //duration should be part of the "goal" object

  void initMoveSequences(); // initialize an array of pose goals

  public:    // member functions

  MoveSequencer();   //constructor

  ~MoveSequencer()
  {  //destructor...do nothing
  }

  double getMoveDuration() { return(moveDuration_);}
  int getCurrentMoveSequenceID() { return(currentMoveSequenceID_); }
  int getLastValidMoveSequenceID() { return(lastValidMoveSequenceID_); }
  bool isDoneWithAllMoves() { return(doneWithMoves_); }
  bool isQueueFull() { return(queueFull_); }

  void update(double goalAngles[NJoints]);  // populates goalAngles with next macro joint-space goal
                                      // and sets corresponding moveDuration_ value

  void addSequence(double qGoal[NJoints], double moveDuration);

  void displayAngles(double q[NJoints]);

};


#endif
```

```
// MoveSequencer implementation file:
#include "MoveSequencer.h"

//constructor
MoveSequencer::MoveSequencer(){
 lastValidMoveSequenceID_=-1;
 currentMoveSequenceID_= -1; //index into goal queue; first update will advance this to legal value of 0
 moveDuration_=0.0;
 doneWithMoves_= true;
 queueFull_=false;
 initMoveSequences(); // this is where we'll define a sequence of desired joint states
                      // hard coded; need to make more flexible--e.g. as a subscriber
                      // also make a member function to receive a new goal state/moveDuration
}


// update() populates goalAngles with next macro joint-space goal
// and sets corresponding moveDuration_ value
void MoveSequencer::update(double goalAngles[NJoints]){
 //need to convert this to circular queue
 ROS_INFO("moveSequencer.update: lastValidMoveSequenceID_= %d",lastValidMoveSequenceID_);
 if (currentMoveSequenceID_< lastValidMoveSequenceID_) {
  currentMoveSequenceID_++;
  ROS_INFO("moveSequencer.update: currentMoveSequenceID_= %d",currentMoveSequenceID_);
  //currentMoveSequenceID_%=goalQueueSize; // modulo index for circular queue
  for (unsigned int i=0;i<NJoints;i++)
   goalAngles[i]=moveSequenceGoalQueue_[currentMoveSequenceID_][i];
  moveDuration_ = moveSequenceDurationQueue_[currentMoveSequenceID_];
 }
 //if at last valid move, do nothing to update index, time or goal angle values
 else {
  moveDuration_=0.5; //stay at same goals for 0.5 sec...parent must detect that we are out of moves
  doneWithMoves_ = true;
  ROS_INFO("exhausted move queue at currentMoveSequenceID_= %d",currentMoveSequenceID_);
 }
}

void MoveSequencer::addSequence(double qGoal[NJoints], double moveDuration) {
 if (lastValidMoveSequenceID_ < goalQueueSize-1) { // need to make this circular buffer
  doneWithMoves_= false;
  lastValidMoveSequenceID_++;
  lastValidMoveSequenceID_%=goalQueueSize;
  for (unsigned int i=0;i<NJoints;i++) {
    moveSequenceGoalQueue_[lastValidMoveSequenceID_][i]=qGoal[i];  //add goal to the queue
    moveSequenceDurationQueue_[lastValidMoveSequenceID_]=moveDuration;
  }
  ROS_INFO("addSequence angles for sequenceID %d, duration %f ",
       lastValidMoveSequenceID_,moveDuration);
  displayAngles(qGoal);
 }
 //if no room, say queue is full
 else
  queueFull_=true;

}


//hard-code some goals and push them on the queue:
```

```
//OK for start-up, but additional moves should come from addSequence()
void MoveSequencer::initMoveSequences(){
 // first move:
 double ang_goals[NJoints];
 double moveTime;

 for (unsigned int i=0;i<NJoints;i++) {
  ang_goals[i]=0.0;  //test: start w/ goal = home pose
  moveTime=1.0; //for 1 second
 }
 addSequence(ang_goals, moveTime);//push this on the queue:

 // step 1:  from home position, move arms backwards and toe off to fall bkwds");
 ang_goals[r_arm_usy]=0.1;
 ang_goals[l_arm_usy]=0.1;
 ang_goals[r_leg_uay]= 1.0; //try to point toes
 ang_goals[l_leg_uay]= 1.0;
 moveTime=2.0; // take 3 seconds to do this
 addSequence(ang_goals, moveTime);//push this on the queue:

 // step 2:
 //return arms to iron cross pose:
 ang_goals[r_arm_usy]= -1.0;
 ang_goals[l_arm_usy]= -1.0;
 ang_goals[l_arm_shx]= -1.1;
 ang_goals[r_arm_shx]= 1.1;
 ang_goals[l_arm_ely]= 1.0;
 ang_goals[r_arm_ely]= 1.0;

 ang_goals[l_leg_mhx]= 0.5;  // spread legs in a "V"
 ang_goals[r_leg_mhx]= -0.5;
 ang_goals[l_leg_lhy]= -0.25; //a little leg lift to help leg spread
 ang_goals[r_leg_lhy]= -0.25;
 ang_goals[r_leg_uay]= 1.0; //try to point toes
 ang_goals[l_leg_uay]= 1.0;
 ang_goals[l_leg_uhz]= 0.2; // not confident of these angles
 ang_goals[r_leg_uhz]= -0.2;
 moveTime=3.0; // take 3 seconds to do this
 addSequence(ang_goals, moveTime);//push this on the queue:

 //step 3:
 ang_goals[l_arm_elx]= 0.8; //bend elbows
 ang_goals[r_arm_elx]= -0.8;
 ang_goals[l_arm_uwy] = 1.0;
 ang_goals[r_arm_uwy] = 1.0;
 ang_goals[l_arm_mwx] = 0.1;
 ang_goals[r_arm_mwx] = -0.1;// need to negate for symmetry

 ang_goals[back_mby]= 0.8; // lean fwd from back joint

 // and lift legs as well:
 ang_goals[l_leg_lhy]= -0.8;  //more leg lifts (L-sit); -0.5, -0.5 works w/ sit-up
 ang_goals[r_leg_lhy]= -0.8;

 ang_goals[l_leg_uhz]= 0.5; // not confident of these angles
 ang_goals[r_leg_uhz]= -0.5;

 moveTime=3.0; // take 3 seconds to do this
```

```
  addSequence(ang_goals, moveTime);//push this on the queue:


}

void MoveSequencer::displayAngles(double q[NJoints]){
  // simply displays joint angles using ROS_INFO()

  ROS_INFO(" ");
  ROS_INFO("back and neck:  %f %f %f %f",q[0],q[1],q[2],q[3]);
  ROS_INFO("l_leg  uhz to lhy: %f %f %f", q[4],q[5],q[6]);
  ROS_INFO("l_leg kny to lax:  %f %f %f", q[7],q[8],q[9]);
  ROS_INFO("r_leg  uhz to lhy: %f %f %f", q[10],q[11],q[12]);
  ROS_INFO("r_leg kny to lax:  %f %f %f", q[13],q[14],q[15]);

  ROS_INFO("l_arm  usy to ely: %f %f %f", q[16],q[17],q[18]);
  ROS_INFO("l_arm elx to mwx:  %f %f %f", q[19],q[20],q[21]);

  ROS_INFO("r_arm  usy to ely: %f %f %f", q[22],q[23],q[24]);
  ROS_INFO("r_arm elx to mwx:  %f %f %f", q[25],q[26],q[27]);
  ROS_INFO(" ");

}
```

```
// class definition for SmoothJointSpaceMove
#ifndef SMOOTH_JOINT_SPACE_MOVE_H
#define SMOOTH_JOINT_SPACE_MOVE_H

// it is meant to increment joint-space commands to produce smooth motions
#include <ros/ros.h> // just for use of ROS_INFO, ROS_WARN
#include <math.h>
#include "joint_names.h"


class SmoothJointSpaceMove
{
 private:  //private member variables; use "_" suffix to indicate private member vars
 //member variables
 double timeStep_;  //this variable gets set upon instantiation and remains constant
 int moveNumber_;   // this variable increments with each new move command

 // these variables get filled in at the start of each new move command
 double jointStartAngles_[NJoints];
 double goalJointAngles_[NJoints];
 double angIncCmds_[NJoints];
 double nomVelCmds_[NJoints];
 double moveDuration_;
 double nMotionIncs_;
 bool validMoveRequest_;

 //these values get updated with each increment of motion
 double jointVelCmds_[NJoints];
 double jointAngleCmds_[NJoints];
 bool jointMovesDone_[NJoints];
 bool moveDone_;

 //void initNewMove(xxx); // sets up parameters for a new move;
 bool isValidCommand(double q_goal[NJoints]); // test validity of a move; substitute valid values
 // internal helper function to update a single joint angle

 double f_ang_update(double ang_cur,double ang_goal,double ang_inc, bool &motionDone);
 public:    // member functions

 //constructor: provide looptime; preferably, provide valid initial angles
 //SmoothJointSpaceMove(double q_init[NJoints],double looptime);
 SmoothJointSpaceMove(); //constructor

 ~SmoothJointSpaceMove()
 {  //destructor...do nothing
 }

 void setTimeStep(double loopFreq);

 // main fnc of this class: call this once per control cycle
 // updates joint command vector by one increment, and fills in velocity command vector
 // if reached goal, commands no increment, commands velocity=0 and sets moveDone to true
 void update(double q_in[],double q_out[]);

 // specifies joint-angle values and duration for a new move
 // returns true if specified move is valid (within joint limits), or false if unsafe
 //  (and then substitutes a reasonable alternative in q_goal)
 // specify moveDuration in seconds
```

```
  void newMoveGoal(double q_start[NJoints],double q_goal[NJoints],double moveDuration);

  //accessor methods:
  int getMoveNumber() { return(moveNumber_); }
  bool getMoveDone() { return(moveDone_); }
  void getJointMovesDone(double movesDone[NJoints])
    { for (unsigned int i=0;i<NJoints;i++) movesDone[i]=jointMovesDone_[i]; }


};


#endif
```

```
// this is the implementation file (member methods) for the SmootJointSpaceMove class

#include "SmoothJointSpaceMove.h"  // class definition

const double minIncMove=0.001;  //tolerance on min move size

//constructor: initialize entities, create xform listener; instantiate useful objects
SmoothJointSpaceMove::SmoothJointSpaceMove() {
  ROS_INFO("SmoothJointSpaceMove constructor");

  timeStep_=0.1; // prevent invalid timestep specification; default to 10Hz

  moveNumber_=0;  // move counter; increment this value for each new move command

  // make sure a call to "update()" has valid initial goals (should produce no motion)
  moveDone_=true; //specifies if all joints have reached their goals
  validMoveRequest_=true; // tells if most recent move request was valid
  nMotionIncs_=1.0; //prevent divide by zero if update without proper newMove() cmd
  moveDuration_=1.0; //legal default value, 1 Second

  for (unsigned int i=0;i<NJoints;i++) {
    angIncCmds_[i]=0.001; //legal, minimal increment
    jointMovesDone_[i]=true; // specifies joint-by-joint if command has reached the goal
    jointStartAngles_[i]=0.0;  //
    goalJointAngles_[i]=jointStartAngles_[i]; //want no motion: goal=start
    nomVelCmds_[i]=0.0; // zero velocities by default
  }

}


//initialize all necessary values in preparation for incremental moves to a new goal
// possible issue: do not allow specification of qStarts?  Always use
// private memory of last set of joint commands?
// Maybe make two functions:  one that allows specification of qStarts, and one that does not\

//xxx problem here...increments not being set correctly!!

void SmoothJointSpaceMove::newMoveGoal(double qStarts[],double qGoals[],double moveDuration) {
 moveDone_=false;
 moveNumber_++; // record that have received a new move command
 validMoveRequest_ = isValidCommand(qGoals); // test if command is valid; alter if necessary
 nMotionIncs_=moveDuration/timeStep_;
 moveDuration_=moveDuration; // keep a record of this value, set by arg

 for (unsigned int i=0;i<NJoints;i++) {
   jointStartAngles_[i]=qStarts[i];  //
   jointAngleCmds_[i]=jointStartAngles_[i];
   goalJointAngles_[i]=qGoals[i];
   angIncCmds_[i] = (qGoals[i]-qStarts[i])/nMotionIncs_;
   if (fabs(angIncCmds_[i])<minIncMove) {
     if (angIncCmds_[i]>0.0) angIncCmds_[i]=minIncMove;
     else angIncCmds_[i]= -minIncMove;
   }

   //ROS_INFO("newMoveGoal: angIncCmds[%d]=%f",i,angIncCmds_[i]); //debug
   nomVelCmds_[i] = angIncCmds_[i]/timeStep_; //nominal velocity for each joint
   jointMovesDone_[i]=false; // specifies joint-by-joint if command has reached the goal
```

```
  }

}

// dummy validity-checker function--need to write actual checker
bool SmoothJointSpaceMove::isValidCommand(double q_goal[NJoints]) {
  return(true);
}

void SmoothJointSpaceMove::setTimeStep(double loopFreq)
{

  if (loopFreq>0.01)
      timeStep_ = 1.0/loopFreq;  // time increment of control loop; presumed update rate of jnt cmds
  else
    timeStep_=100.0;  //100 seconds is too long
  if (timeStep_<0.001)  // make sure timeStep is valid
    {
     timeStep_=0.001; // prevent invalid timestep specification
     ROS_WARN("invalid timestep specification in SmoothJointSpaceMove constructor; substituting 1ms");
    }
  if (timeStep_>10.0)
    ROS_WARN("warning: timeStep in smoothJointSpaceMove is large: %f",timeStep_);
  ROS_INFO("smooth jointspace move timestep set to: %f",timeStep_);
}

//here's the main deal...
//update() will put values in qOut[] and qDotOut[], updated as incremental motions
// from previous joint commands towards previously specified goal
//If want a new goal, must invoke a new call to newMoveGoal()
void SmoothJointSpaceMove::update(double qOut[NJoints], double qDotOut[NJoints]) {
  // main fnc of this class: call this once per control cycle

  moveDone_=true; //assume move is done, unless prove otherwise
  // loop through all joints, using helper fnc to incrementally update commands

  double joint_sqr_err=0;
  for(unsigned int i=0;i<NJoints;i++) {
    bool motionDone;

    //compute an incremental angle update for the i'th joint; indicate if motion is complete for ith jnt
    // using reference variable bool &motionDone
    qOut[i]=f_ang_update(jointAngleCmds_[i],goalJointAngles_[i],angIncCmds_[i], motionDone);
    jointAngleCmds_[i]=qOut[i]; // save current commands in private array
    joint_sqr_err+= (qOut[i]-goalJointAngles_[i])*(qOut[i]-goalJointAngles_[i]);

    jointMovesDone_[i]=motionDone;
    if (!motionDone) {
      qDotOut[i]=nomVelCmds_[i];
      //ROS_INFO("jnt %d not done; qOut= %f, goal=%f, inc=%f",i,qOut[i],goalJointAngles_[i]
      //         ,angIncCmds_[i]);
      moveDone_=false; // if any joint is not done, then entire move is not yet complete
    }
    else   //here if this joint's motion is complete
      qDotOut[i]=0.0; // command zero velocity if move is done, joint by joint

    jointAngleCmds_[i]=qOut[i]; // remember most recent angle commands
    jointVelCmds_[i]=qDotOut[i];  // also most recent velocity commands
```

```
  }
  //ROS_INFO("smooth update: jnt cmd[r_arm_usy] = %f",jointAngleCmds_[r_arm_usy]); //debug
  ROS_INFO("smooth update: jnt_err = %f",sqrt(joint_sqr_err)); //debug
}


//helper function to update a single joint angle
double SmoothJointSpaceMove::f_ang_update(double ang_cur,double ang_goal,double ang_inc,
                                         bool &motionDone)
{
  double ang_new;
  if (ang_inc>0.0) // for positive motion
   {
    if ((ang_goal-ang_cur) > ang_inc)
    {
    ang_new = ang_cur+ang_inc; // move towards goal by one increment
    motionDone= false; //signal motion not complete
       }
    else
    {
    ang_new=ang_goal; // less than one increment to goal--or else overstepped goal, or exactly at goal already
    motionDone= true; //signal that motion is complete
       }
   }
  else // case for zero for negative increment
   {
    if ((ang_goal-ang_cur) < ang_inc)
       {
    ang_new = ang_cur+ang_inc; // move towards goal by one (negative) increment
    motionDone= false; //signal motion not complete
       }
    else
    {
    ang_new = ang_goal; //within one increment of goal--else overstepped (should not happen)
    motionDone= true; //signal that motion is complete
       }
   }
  return(ang_new);

}
```

```
#ifndef JOINT_NAMES_H
#define JOINT_NAMES_H
 // make names for indices...
  const unsigned int NJoints=28;  //number of body joints= 28
  //for the back:
  const unsigned int back_lbz=0;
  const unsigned int back_mby=1;
  const unsigned int back_ubx=2;
  //neck:
  const unsigned int neck_ay=3;

  // legs:
  const unsigned int l_leg_uhz=4;
  const unsigned int l_leg_mhx=5;
  const unsigned int l_leg_lhy=6;
  const unsigned int l_leg_kny=7;
  const unsigned int l_leg_uay=8;
  const unsigned int l_leg_lax=9;

  const unsigned int r_leg_uhz=10;
  const unsigned int r_leg_mhx=11;
  const unsigned int r_leg_lhy=12;
  const unsigned int r_leg_kny=13;
  const unsigned int r_leg_uay=14;
  const unsigned int r_leg_lax=15;

  // these for the arms:
  const unsigned int l_arm_usy=16;
  const unsigned int l_arm_shx=17;
  const unsigned int l_arm_ely=18;
  const unsigned int l_arm_elx=19;
  const unsigned int l_arm_uwy=20;
  const unsigned int l_arm_mwx=21;

  const unsigned int r_arm_usy=22;
  const unsigned int r_arm_shx=23;
  const unsigned int r_arm_ely=24;
  const unsigned int r_arm_elx=25;
  const unsigned int r_arm_uwy=26;
  const unsigned int r_arm_mwx=27;
#endif
```


pelvis
xyz: -0.0125 0 0
rpy: 0 -0 0
back_lbz
ltorso
xyz: 0 0 0.09
rpy: 0 -0 0
back_mby
mtorso
xyz: 0 0 0.05
rpy: 0 -0 0
back_ubx
utorso
xyz: 0.024 0.221 0.289
rpy: 0 -0 0
l_arm_usy
l_clav
xyz: 0 0.075 0.036
rpy: 0 -0 0
l_arm_shx
l_scap
xyz: 0 0.185 0
rpy: 0 -0 0
l_arm_ely
l_uarm
xyz: 0 0.121 0.013
rpy: 0 -0 0
l_arm_elx
l_larm
xyz: 0 0.188 -0.013
rpy: 0 -0 0
l_arm_uwy
l_farm
xyz: 0 0.058 0
rpy: 0 -0 0
l_arm_mwx
l_hand
xyz: 0 0.1 0
rpy: 1.57079 -1.20861e-16 1.57079
left_sandia_hand_joint
left_palm
xyz: 0.14 0 0
rpy: 0 0 -0.3
left_f0_base
xyz: -0.05 0 0
rpy: 0 -0 0
left_f0_fixed_accel
xyz: 0 0 0
rpy: 0 -0 0
left_f0_j0
left_f0_0
xyz: 0.028 0 0
rpy: 1.5708 -0 0
left_f0_j1
left_f0_1
xyz: 0.03 0 0
rpy: -1.5708 0 0
left_f0_1_accel
xyz: 0.05 0 0
rpy: 0 -0 0
left_f0_j2
left_f0_2
xyz: 0.04 0 0
rpy: -1.5708 0 0
left_f0_2_accel
xyz: 0.115 -0.03122 0
rpy: 0 0 -0.3
left_f1_base
xyz: -0.05 0 0
rpy: 0 -0 0
left_f1_fixed_accel
xyz: 0 0 0
rpy: 0 -0 0
left_f1_j0
left_f1_0
xyz: 0.028 0 0
rpy: 1.5708 -0 0
left_f1_j1
left_f1_1
xyz: 0.03 0 0
rpy: -1.5708 0 0
left_f1_1_accel
xyz: 0.05 0 0
rpy: 0 -0 0
left_f1_j2
left_f1_2
xyz: 0.04 0 0
rpy: -1.5708 0 0
left_f1_2_accel
xyz: 0.09 -0.06245 0
rpy: 0 0 -0.3
left_f2_base
xyz: -0.05 0 0
rpy: 0 -0 0
left_f2_fixed_accel
xyz: 0 0 0
rpy: 0 -0 0
left_f2_j0
left_f2_0
xyz: 0.028 0 0
rpy: 1.5708 -0 0
left_f2_j1
left_f2_1
xyz: 0.03 0 0
rpy: -1.5708 0 0
left_f2_1_accel
xyz: 0.05 0 0
rpy: 0 -0 0
left_f2_j2
left_f2_2
xyz: 0.04 0 0
rpy: -1.5708 0 0
left_f2_2_accel
xyz: 0.03995 0.04383 0.014
rpy: -4.21801e-06 -1.5708 2.3562
left_f3_base
xyz: -0.05 0 0
rpy: 0 -0 0
left_f3_fixed_accel
xyz: 0 0 0
rpy: 0 -0 0
left_f3_j0
left_f3_0
xyz: 0.028 0 0
rpy: 1.5708 -0 0
left_f3_j1
left_f3_1
xyz: 0.03 0 0
rpy: -1.5708 0 0
left_f3_1_accel
xyz: 0.05 0 0
rpy: 0 -0 0
left_f3_j2
left_f3_2
xyz: 0.04 0 0
rpy: -1.5708 0 0
left_f3_2_accel
xyz: 0.16555 0 0.50823
rpy: 0 -0 0
neck_ay
head
xyz: 0.01125 0 -0.0155
rpy: 0 -0 0
center_bottom_led_frame_joint
center_bottom_led_frame
xyz: 0.01125 0 0.0105
rpy: 0 -0 0
center_top_led_frame_joint
center_top_led_frame
xyz: -0.0446 0 0.088
rpy: 0 -0 0
hokuyo_joint
hokuyo_link
xyz: 0.03 0 0.015
rpy: 0 -0 0
head_hokuyo_joint
head_hokuyo_frame
xyz: 0 0.035 -0.002
rpy: 0 -0 0
left_camera_frame_joint
left_camera_frame
xyz: 0 0 0
rpy: -1.5708 -1.14194e-16 -1.5708
left_camera_optical_frame_joint
left_camera_optical_frame
xyz: -0.01443 0.07452 0.050346
rpy: 4.07081e-18 -0.15 0.53
left_led_frame_joint
left_led_frame
xyz: 0 -0.035 -0.002
rpy: 0 -0 0
right_camera_frame_joint
right_camera_frame
xyz: 0 0 0
rpy: -1.5708 -1.14194e-16 -1.5708
right_camera_optical_frame_joint
right_camera_optical_frame
xyz: -0.01443 -0.07452 0.050346
rpy: -4.07081e-18 -0.15 -0.53
right_led_frame_joint
right_led_frame
xyz: 0.024 -0.221 0.289
rpy: 0 -0 0
r_arm_usy
r_clav
xyz: 0 -0.075 0.036
rpy: 0 -0 0
r_arm_shx
r_scap
xyz: 0 -0.185 0
rpy: 0 -0 0
r_arm_ely
r_uarm
xyz: 0 -0.121 0.013
rpy: 0 -0 0
r_arm_elx
r_larm
xyz: 0 -0.188 -0.013
rpy: 0 -0 0
r_arm_uwy
r_farm
xyz: 0 -0.058 0
rpy: 0 -0 0
r_arm_mwx
r_hand
xyz: 0 -0.1 0
rpy: -1.57079 -1.20861e-16 -1.57079
right_sandia_hand_joint
right_palm
xyz: 0.14 0 0
rpy: 0 -0 0.3
right_f0_base
xyz: -0.05 0 0
rpy: 0 -0 0
right_f0_fixed_accel
xyz: 0 0 0
rpy: 0 -0 0
right_f0_j0
right_f0_0
xyz: 0.028 0 0
rpy: 1.5708 -0 0
right_f0_j1
right_f0_1
xyz: 0.03 0 0
rpy: -1.5708 0 0
right_f0_1_accel
xyz: 0.05 0 0
rpy: 0 -0 0
right_f0_j2
right_f0_2
xyz: 0.04 0 0
rpy: -1.5708 0 0
right_f0_2_accel
xyz: 0.115 0.03122 0
rpy: 0 -0 0.3
right_f1_base
xyz: -0.05 0 0
rpy: 0 -0 0
right_f1_fixed_accel
xyz: 0 0 0
rpy: 0 -0 0
right_f1_j0
right_f1_0
xyz: 0.028 0 0
rpy: 1.5708 -0 0
right_f1_j1
right_f1_1
xyz: 0.03 0 0
rpy: -1.5708 0 0
right_f1_1_accel
xyz: 0.05 0 0
rpy: 0 -0 0
right_f1_j2
right_f1_2
xyz: 0.04 0 0
rpy: -1.5708 0 0
right_f1_2_accel
xyz: 0.09 0.06245 0
rpy: 0 -0 0.3
right_f2_base
xyz: -0.05 0 0
rpy: 0 -0 0
right_f2_fixed_accel
xyz: 0 0 0
rpy: 0 -0 0
right_f2_j0
right_f2_0
xyz: 0.028 0 0
rpy: 1.5708 -0 0
right_f2_j1
right_f2_1
xyz: 0.03 0 0
rpy: -1.5708 0 0
right_f2_1_accel
xyz: 0.05 0 0
rpy: 0 -0 0
right_f2_j2
right_f2_2
xyz: 0.04 0 0
rpy: -1.5708 0 0
right_f2_2_accel
xyz: 0.03995 -0.04383 0.014
rpy: 4.21801e-06 -1.5708 -2.3562
right_f3_base
xyz: -0.05 0 0
rpy: 0 -0 0
right_f3_fixed_accel
xyz: 0 0 0
rpy: 0 -0 0
right_f3_j0
right_f3_0
xyz: 0.028 0 0
rpy: 1.5708 -0 0
right_f3_j1
right_f3_1
xyz: 0.03 0 0
rpy: -1.5708 0 0
right_f3_1_accel
xyz: 0.05 0 0
rpy: 0 -0 0
right_f3_j2
right_f3_2
xyz: 0.04 0 0
rpy: -1.5708 0 0
right_f3_2_accel
xyz: 0.0599153 0 -0.01415
rpy: 0 -0 0
imu_joint
imu_link
xyz: 0 0.089 0
rpy: 0 -0 0
l_leg_uhz
l_uglut
xyz: 0 0 0
rpy: 0 -0 0
l_leg_mhx
l_lglut
xyz: 0.05 0 -0.05
rpy: 0 -0 0
l_leg_lhy
l_uleg
xyz: -0.05 0 -0.374
rpy: 0 -0 0
l_leg_kny
l_lleg
xyz: 0 0 -0.422
rpy: 0 -0 0
l_leg_uay
l_talus
xyz: 0 0 0
rpy: 0 -0 0
l_leg_lax
l_foot
xyz: 0 -0.089 0
rpy: 0 -0 0
r_leg_uhz
r_uglut
xyz: 0 0 0
rpy: 0 -0 0
r_leg_mhx
r_lglut
xyz: 0.05 0 -0.05
rpy: 0 -0 0
r_leg_lhy
r_uleg
xyz: -0.05 0 -0.374
rpy: 0 -0 0
r_leg_kny
r_lleg
xyz: 0 0 -0.422
rpy: 0 -0 0
r_leg_uay
r_talus
xyz: 0 0 0
rpy: 0 -0 0
r_leg_lax
r_foot