Introduction to Point Clouds with Human-Machine Interfacing

Wyatt Newman

March, 2013

These notes introduce example code for specifying and publishing 3-D points of interest using mouse input interacting with 3-D point-cloud data. Two packages (corresponding to two nodes) are introduced: point_cloud_HMI and hand_follow_hmi. The first enables one to pick points of interest from a 3-D scene, and the other package shows how a robot controller can respond to the points of interest (in this case, by attempting to reach the selected point with the robot's right hand).

To run the code, you should start up the drc simulator and preset the robot in an interesting pose. This should include an environment with sufficient contrast that stereo vision yields recognizable points, and at least some of these points should be reachable by the robot's right hand. For example, the robot can be preset in the DRC vehicle with its head tilted down to look at the vehicle controls, which are within reach of the right hand.

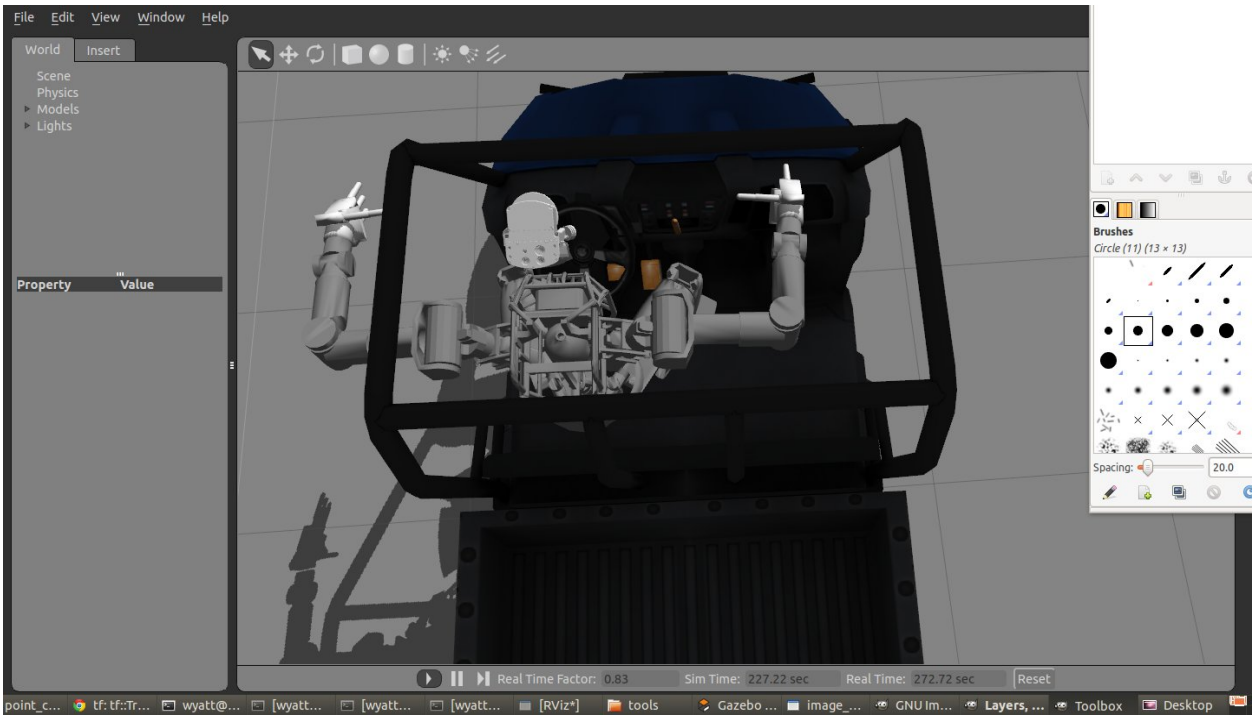
Point Cloud Interaction: The accompanying package, "point_cloud_HMI", contains ROS code that illustrates combining operator (mouse) interaction with OpenCV, affiliating 2-D image points with 3-D cloud points, and publishing selected 3-D points as ROS "stamped" points (carrying timing and frame information with the coordinate data).

The example program subscribes to the topic: /multisense_sl/camera/points2, which is publishes a "pointClould" synthetically derived from stereo vision from the robot's sensor head. A point cloud is a collection of 3-D points. The points may carry additional information, such as RGB color data. The point coordinates are referenced to some coordinate frame. There is a ROS datatype for pointcloud messages that includes a header carrying the named coordinate frame in which the points are referenced, as well as a time stamp. Points in a point cloud may be organized or unorganized. For 3-D points that originate from stereo vision, LIDAR scanners or a Kinect-like device, the data is typically organized in a 2-D matrix. Each cell of the 2-D matrix may contain a point with 3-D coordinates and (optionally) color data. Organized point-cloud data thus has a natural mapping to 2-D images. If the color data for each cloud point is assigned to a 2-D matrix of the same dimensions (preserving the i,j ordering), the resulting matrix may be interpreted as a 2-D image. Conceptually, this makes sense. When we view a scene with our eyes, the light impinging on the retina originates from points in a 3-D environment, but each of these light sources projects onto a 2-D receptive surface. Data from the retina can be expressed as an equivalent 2-D array of intensities of colors. Each cell of this array is affiliated with some 3-D point in the environment.

For the simulated stereo vision from Atlas's sensor head, 3-D points are inferred by triangulation of corresponding pixels in the right and left cameras. A challenge in stereo vision is to identify pairs of pixels from right and left cameras that correspond to the same point in the environment. If this correspondence can be found, then 3-D coordinates of the point in the environment can be computed from optics and geometry. Finding correspondence of pixels in left and right cameras is the biggest challenge. This can be accomplished in part using image processing in small regions of interest with specific filters. It is expected that the light source from an i,j pixel in the left eye will also impinge on a pixel in the right eye that is close to the same i,j coordinates. If a convolution filter is applied to a small region centered about an (i,j) pixel of interest in the left eye, the same filter can be applied to the right eye, tested in multiple evaluations close to being centered about (i,j). When testing the filter centered at various trial locations in the right-eye image, the result that most closely matches that of the left-eye filter is assumed to indicate a match for corresponding pixels. Variations on this conceptual algorithm are optimized for speed and implemented in FPGAs to yield fast stereo vision.

There are obvious problems with this approach. For example, some scenes are featureless (e.g. a flat, gray floor). In this case, filters centered at all trial locations yield the same quality of fit, and thus no corresponding points can be found. Featureless regions may include the sky, walls, floors, roads, etc. Surfaces with interesting texture or key features in the environment, such as corners, are more readily recognized in stereo vision. However, surfaces with high-frequency texture (e.g. grass) can present difficulty for stereo vision.

The image below is a screenshot of the Gazebo display, looking over Atlas's shoulder with Atlas seated in the DRC vehicle with his neck tilted down.



The figure below is a screen-capture of an “rviz” display of Atlas’s left camera feed corresponding to the situation shown above.

In this view, all pixels of the 2-D matrix are filled in with color values.

The image below is the corresponding stereo image converted to 2-D. That is, stereo processing was applied to the above scene, using both left and right cameras. A new 2-D image was created, where pixels in the synthesized 2-D image correspond to pixels in the left eye. In this new image, each left-eye pixel for which left/right correspondence could be found is populated with the color of the stereo point. However, as shown, most of the pixels in this scene failed to find a left/right correspondence. For these invalid pixels, the color is set to pure red.

As is apparent from the above image, a minority of points have satisfied the right/left correspondence test. The steering wheel, part of a pedal, part of the gear shift handle, Atlas's right and left knees and parts of the dash have been recognized in 3-D.

Although the left camera image is clearer and is complete, it is less useful for human interaction. Most of the pixels in the camera view do not have a valid correspondence to a 3-D point in the stereo point cloud. Thus, selecting one of these points typically will not allow deducing the corresponding 3-D coordinates. In the image synthesized from the point cloud, however, it is clear which points are able to provide 3-D coordinate information.

The program "point_cloud_HMI.cpp" provides an example of how a user can interact with 2-D images to imply 3-D points. This program has two callback functions. One of these, "cloud_callback()", responds to new pointCloud messages from Atlas's sensor head. The other, "onMouse()", responds to user mouse clicks.

The callback function "cloud_callback()" receives ROS messages of ROS type PointCloud2. Similar to ROS's integration of OpenCV, these messages may be converted into a format consistent with an existing library of code, the "Point Cloud Library" (PCL). This conversion is done with the line:

```
// convert Input cloud from ROS format to PCL format
pcl::fromROSMsg(*ros_input_cloud, *pcl_input_cloud);
```

Once the received 3-D points are converted to PCL format, one can use the PCL library. Use of the PCL library is described in http://www.pointclouds.org/documentation/, and specifics of integrating PCL with ROS can be found in: http://www.ros.org/wiki/pcl/Tutorials. The point-cloud library includes many useful functions for analyzing 3-D sets of points, including matching points to models, fitting points to surfaces, computing surface normal, etc.

In the present example, received pointclouds are converted from ROS message types into PCL format. From here, the point cloud is converted to a 2-D image in ROS message format using:

```
pcl::toROSMsg (*ros_input_cloud, ROS_image_of_input_cloud);
```

The cloud_callback() function performs this conversion, making 2-D images synthesized from 3-D point clouds available (via global memory, in this implementation).

Within the "onMouse()" callback function, the synthesized 2-D image is converted to openCV format using:

```
cv_ptr = cv_bridge::toCvCopy(ROS_image_of_input_cloud, enc::BGR8); //convert ROS message to openCV
```

The 2-D image is displayed to the user with:

```
img = cv_ptr->image; // synonym...slightly simpler
imshow("image_window", img); //update the image display
```

The displayed image is available for user interaction. By mousing over the display window and clicking on a pixel, the corresponding (i,j) coordinates of the selected pixel in the image are updated in global memory variables, mouse_click_x and mouse_click_y.

In the cloud_callback() function, the values of mouse_click_x and mouse_click_y are used to index into the matrix of organized pointCloud points. From this matrix cell, the corresponding 3-D point information can be obtained. In the present example, the cloud_callback() function checks if the selected pixel has valid 3-D data, and if so, it publishes the result to the topic "userPickedPoint." The message type published is a geometry_msgs::PointStamped. This message type inherits the time stamp and the coordinate-frame name from the point cloud from which the point is extracted. Publishing this point thus allows for transformations using "tf" by whatever node subscribes to the new "userPickedPoint" topic.

An inconvenient feature of this implementation is that published points can come from relatively old pointCloud messages. New pointcloud messages are processed only after each mouse click. Receipt of a new pointCloud produces a new 2-D image. This new 2-D image is kept static until the user clicks on a point. This point then refers to the pointCloud from which the 2-D image was generated. If the user waits more than about 10 seconds between mouse clicks, then the resulting published stamped point will have a time stamp that is fairly old (correctly indicating that the data obtained was from an older pointCloud). When a subscriber tries to transform this point to a new frame, e.g. using tf_listener.transformPoint(), the transform listener likely will not

have retained the correct transforms corresponding to the time in the past at which the pointCloud was acquired. If the pointCloud originates from an immobile source, this would not be an issue (and the time stamp could be faked to present it as fresh data). However, the robot's head will be moving, and thus the 3-D data must be referred to the pose that corresponded to the time of data acquisition. A failure of tf.transformPoint() is thus appropriate, and a "try/catch" structure should be used to allow for such failures.

A work-around with the present code is to click twice on points of interest. The first click may be aged data, but the second click will be from an updated point cloud. In fact, the first click may deliberately be in an invalid area (colored red). This will invoke receipt of a new point cloud and display of a fresh 2-D image—and the invalid selected pixel will be recognized as invalid and will not get published. Subsequently, clicking a viable point on the fresh image will result in good data being published.

The commented code for this HMI is appended, and the ROS package is in the HKU team directory under "tools". The code can be compiled with: rosmake point_cloud_HMI, and it may be run with:
rosrun point_cloud_HMI point_cloud_HMI.

The DRC simulator should be running and publishing stereo point clouds for this code to execute. Confirmation of operation can be obtained by doing: rostopic echo userPickedPoint, which will show that mouse clicks on the synthesized 2-D display result in publishing 3-D data to this topic.

Robot Control Based on HMI Input: A second package, "hand_follow_hmi", demonstrates how one can use input from the point_cloud_HMI node to control the robot. In this example, a frame on the robot's right hand (/right_f1_base) is to be driven to the coordinates of a point published on userPickedPoint. In this example, the program does not attempt to orient the hand appropriately—it only attempts to get the hand-frame origin to coincide with the 3-D user specified point.

The main program is "atlas_hand_follow_hmi_main.cpp", and this code relies on modules to compute Jacobians and perform smooth moves in joint space. A callback function, HMI_callback(), responds to incoming messages on the userPickedPoint topic. Received points are copied to the global variable g_PickedPoint for use by the main program.

In the main loop, the program constantly checks for the coordinates of the chosen hand frame, expressed in the /pelvis frame using:

```
tf_listener.lookupTransform("/pelvis", "/right_f1_base", ros::Time(0), transform);
rHandOriginActual= transform.getOrigin();
```

Additionally, the requested hand-frame location is obtained from the published HMI point. However, this point must be transformed into a consistent frame (the /pelvis frame). This is accomplished using the "tf" function "transformPoint()" as follows:

```
tf_listener.transformPoint("/pelvis", g_PickedPoint, tfPickedPoint);
```

The result of this call is that the desired point (from the HMI node) is transformed into /pelvis-frame coordinates, contained in the variable tfPickedPoint. As noted earlier, if the received point is too old (as per its timestamp), tf will not be able to transform it. The transformPoint() call is thus contained in a try/catch block to tolerate failures.

Valid, transformed 3-D location requests are copied into the variable rHandOriginDesired, and this vector becomes the destination for the chosen right-hand frame origin. To move towards this goal, a hand error is computed as:

```
rHandErrorVec= rHandOriginDesired-rHandOriginActual;
```

Incremental joint commands are computed using the Jacobian-transpose method:

```
//compute dqVec for dp step towards goal as dqVec= J'dp
jacobianComputer.computeRightHandJTransposeDp(rHandErrorVec,dqVec);
```

This computation results in a recommended delta-q (increment of joint angles) to move the hand closer to its destination. The delta-q vector is tested to see if it is overly large, and if so, the entire vector is scaled down. The resulting vector is then added to the vector of absolute joint-angle commands—but only for right-arm joints. (Waist-joint motions are thus suppressed).

```
for (unsigned int i=22;i<26;i++) //only increment right-arm angles (not back joints)
jointSpaceCommand[i]+=dqVec[i];  //increment the previous joint-space command by vector dq
```

The resulting joint-space vector then gets set as the new joint-space goal:

```
atlasJointControl.setNewJointSpaceGoal(jointSpaceCommand, moveDuration);
```

The speed of moves can be adjusted by changing the moveDuration value and/or the maximum permitted delta-q evaluate for scaling the dqVec.

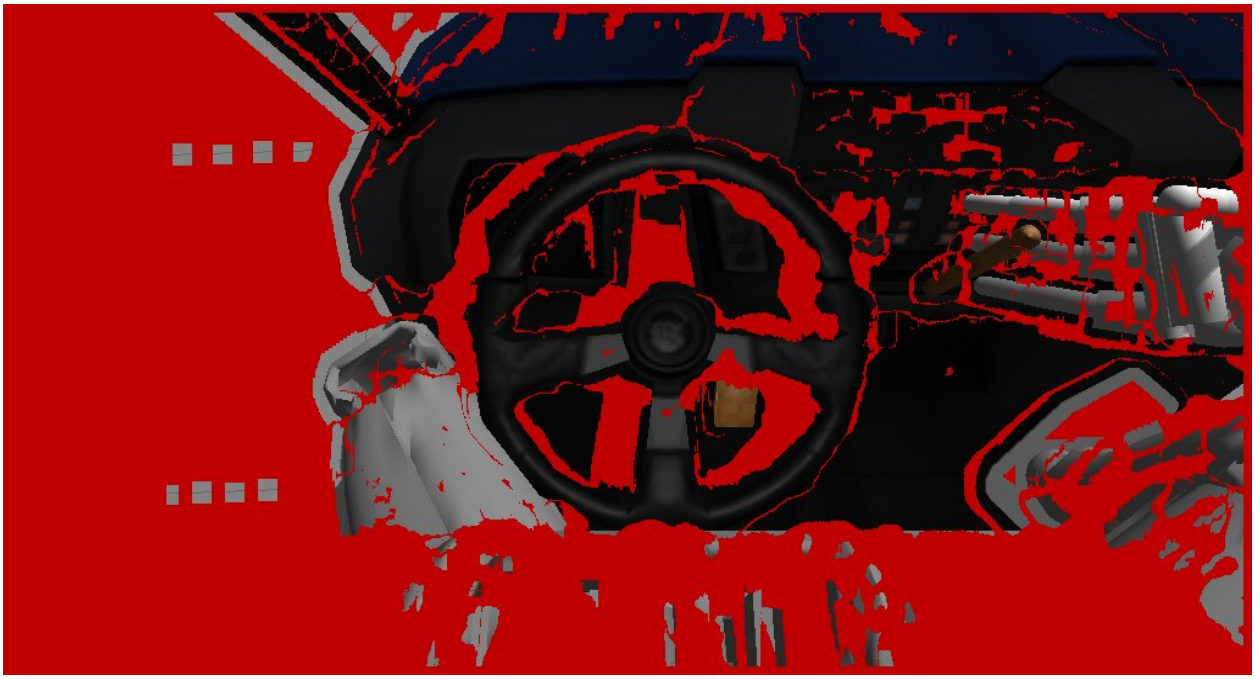
In the example implementation, joint-angle commands are incremented each iteration using the Jacobian-transpose method. Corrections are constantly updated based on hand error relative to the desired location. If the target location is reachable without interference, this algorithm should converge to zero error (equivalent to integral-error control). However, like integral-error control, one must be cautious of integrator wind-up. If motion towards the target location results in contact interference, the joint commands will continue to integrate as the robot struggles to reach the target. This can result in excessive force. If the robot succeeds in getting through the barrier, its hand motion might overshoot dramatically.

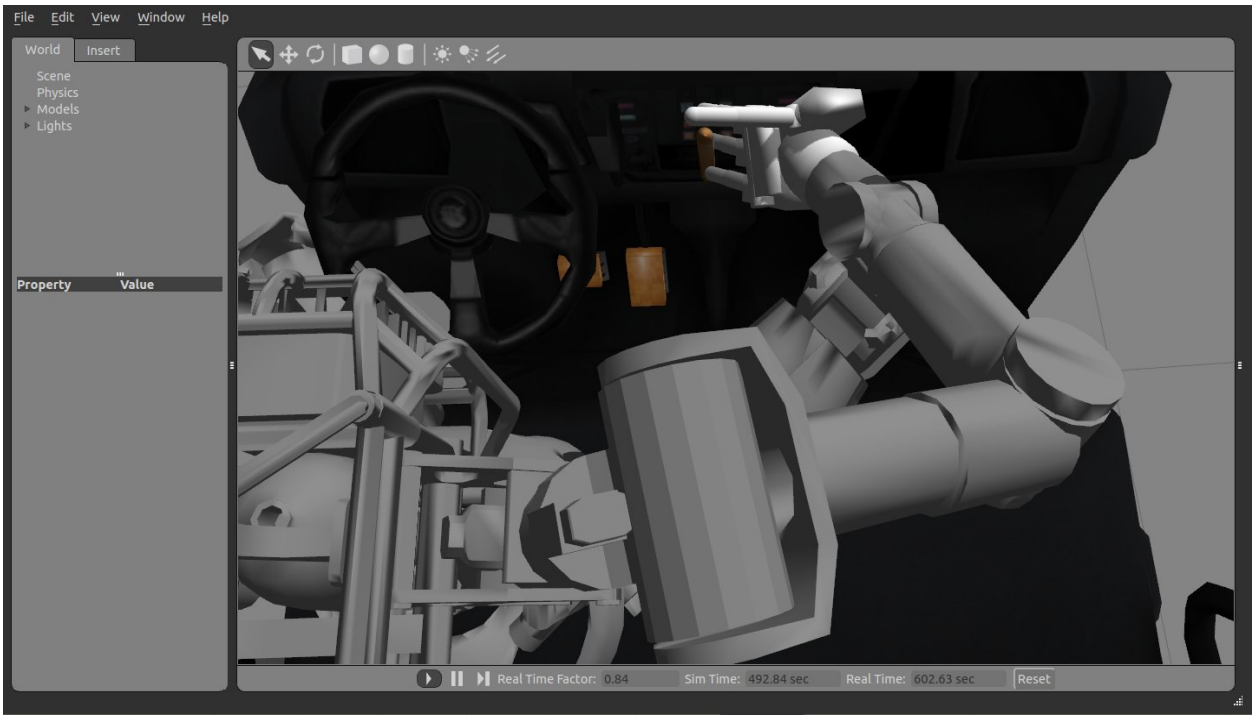
An alternative approach is to increment the actual (rather than commanded) joint angles. This would result in gentler interactions, coming to a halt at interference. However, it would also result in less accurate convergence, e.g. due to gravity droop.

Running the Example Code: To try out the HMI-driven arm control with the illustrated scenario, first start up the DRC simulator (with DRC vehicle), preposition Atlas in the DRC vehicle, and run the "one_shot_move" utility to bend the neck down. This will allow a view of the vehicle controls. The one_shot_move program should be run before starting the hand_follow_hmi program, or the latter program will undo the neck bend.

Once the simulator is pre-staged, start up the two new demo nodes: hand_follow_hmi and point_cloud_HMI. Make sure that the Atlas simulator is in "run", not "pause" mode, so new images are being published. The point_cloud_HMI program will present a 2-D image, like that shown herein. Mouse over the image and click on a non-red point (a valid 3-D point). It may be necessary to click twice, if the image is "old."

The hand_follow_hmi node will print out ROS_INFO() messages to its terminal, confirming receipt of a new goal. The Atlas robot will move its right arm, attempting to place its hand frame at the selected coordinates. The following images show the result of clicking on the brown shifter handle. Atlas has attempted to move its hand to the selected coordinates, as shown in the Gazebo screenshot and the synthesized 2-D image view.





Conclusion: The example code presented shows how one can use a mouse for human interaction with 3-D sensor data from the robot. This information can be used to control the robot.

Many improvements can and should be made. For example, clicking on a single point is error prone, since the chosen point might be far off due to noise. To reduce this source of error, point-cloud processing could be used to find a coherent constellation of points near the selected point, and possibly near in color-space to the selected point. Outliers from this constellation could be rejected and the centroid of the result could be used to specify the desired point. A K-means clustering algorithm or a RANSAC algorithm might be used for this purpose. More generally, selection of a point might be done in context—e.g., with desire to grasp the steering wheel. Clicking on one or more points on the steering wheel may be used as seed values to help a point-cloud processing algorithm identify the steering wheel, e.g. as a fit to a cylinder of known dimensions. A grasp point, including approach orientation, may be deduced from the model fit to the data. This would generate a more intelligent and less error-prone goal for the robot.

For motion control with respect to HMI input, it is clear that collision-free paths must be planned to achieve desired grasp poses. At present, the hand motion can result in collisions with the environment and/or self collisions. Further, for successful grasp, it will be important to control hand orientation as well as hand position.

```
// point_cloud_HMI; Wyatt Newman, March 6, 2013
// demo program to show how to interact with point-cloud data
// Theory of operation:
//  *a callback function listens for pointclouds on topic: /multisense_sl/camera/points2, which is an
ordered pointcloud
//    (i.e. can be accessed from a matrix of points) from emulated stereo vision
//   This pointcloud is (realistically) imperfect, with noise and lots of NaN entries (failure to find
corresponding features
//   in left/right cameras)
//  *the pointcloud callback, "cloud_callback", checks if there has been a new mouse-click, and if so,
it
//    uses the mouse-click pixel coordinates to index into the point cloud, extracting 3-D coordinates
//    These 3-D coordinates are used to populate a geometry_msgs::PointStamped object, which gets a
copy of the pointCloud header
//    This stamped point is suitable for use with tf to cast results into alternative frames;
//    Each stamped point (if valid data) is published on topic "userPickedPoint"
//  *2-D images created from 3-D pointclouds are displayed and processed within a mouse callback
function, "onMouse()"
//    This function gets the display-window's pixel coordinates of the mouse-click location, and saves
these as global vars
//    for use by the cloud_callback function.  The 2-D image coordinates allow indexing into the 3-D
point cloud to infer 3-D
//    coordinates from a 2-D mouse click
//
//   Note: When the user enters a mouse click, the corresponding 3-D datapoint is published,
inheriting the header of the saved
//    (corresponding) point cloud.  However, if the user waits more than about 10sec between mouse
clicks, the resulting published
//    point will be fairly old, and a corresponding transform from this time is not likely to be in
cache.  Thus, the "consumer"
//    of this data may get a fault from tf_listener.transformPoint() in attempting to transform this
data to another frame.
//    The consumer should try/catch to allow for this possibility.  The user can click a point twice
in succession to make sure
//    the second transmission corresponds to a "fresh" timeStamp.

// To run the program, first create a live publisher for the topic: /multisense_sl/camera/points2
(i.e. start up drcsim, or playback
//    some recorded bagfile data).  Then start up this node.  Click on the display window and observe
the ROS_INFO stating
//    the image x,y, the corresponding 3-D point, x,y,z, and the RGB colors of both the 2-D image and
the corresponding 3-D point
// Also, in another terminal, run: rostopic echo /userPickedPoint.  At each mouse click, the
corresponding 3-D data (and header)
//    will be echoed.  This much demonstrates that the HMI code is functioning.  Note that the
reference frame for the data
//    is "frame_id: /left_camera_optical_frame", and the consumer of this data should transform it
into an appropriate frame

// Standard ROS Includes
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <sensor_msgs/Image.h>

// PCL includes
#include <pcl/ros/conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>

// OpenCV includes
#include <image_transport/image_transport.h> // don't need the transport headers for this routine;
#include <image_transport/subscriber_filter.h>

#include <cv_bridge/cv_bridge.h>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

//geometry messages
#include <geometry_msgs/PointStamped.h>

#include <ros/ros.h>
#include <sensor_msgs/image_encodings.h>
```

```
using namespace cv;
namespace enc = sensor_msgs::image_encodings;

// Function prototypes here
sensor_msgs::PointCloud2 filter_rgb(const sensor_msgs::PointCloud2ConstPtr& input_cloud,
                                                                                int l_r,
                                                                                int u_r,
                                                                                int l_g,
                                                                                int u_g,
                                                                                int l_b,
                                                                                int u_b);

// Global variables here: use these to communicate among main and callbacks
ros::Publisher cloud_pub; // will publish the cloud
ros::Publisher image_pub; // publish cloud converted to image
ros::Publisher pickedPointPub; //published picked point

sensor_msgs::Image ROS_image_of_input_cloud;

int mouse_click_x=0;
int mouse_click_y=0;

bool getPtFlag=false;
bool firstCloud=true;
bool pickedTarget=false;
pcl::PointCloud<pcl::PointXYZRGB> pcl_remembered;  //persistent pointCloud


//utility to copy transient point cloud into persistent memory
void copyCldPtrToCloud(pcl::PointCloud<pcl::PointXYZRGB>::Ptr
pcl_input_cloud,pcl::PointCloud<pcl::PointXYZRGB> &pcl_remembered) {
       for(unsigned int i = 0; i < pcl_input_cloud->size(); ++i)
        {

                // copy each point from input cloud pointer to output cloud (which is global)
                pcl_remembered.points[i].x = pcl_input_cloud->points[i].x;
                pcl_remembered.points[i].y = pcl_input_cloud->points[i].y;
                pcl_remembered.points[i].z = pcl_input_cloud->points[i].z;
                pcl_remembered.points[i].r = pcl_input_cloud->points[i].r;
                pcl_remembered.points[i].g = pcl_input_cloud->points[i].g;
                pcl_remembered.points[i].b = pcl_input_cloud->points[i].b;
        }
        pcl_remembered.header = pcl_input_cloud->header;
}



void cloud_callback(const sensor_msgs::PointCloud2ConstPtr& ros_input_cloud)
{
        int pcl_index;
        sensor_msgs::PointCloud2 ros_output_cloud;
        geometry_msgs::PointStamped userPickedPoint; // fill this from pointCloud data and publish
        float cloudPtX;
        float cloudPtY;
        float cloudPtZ;
        int  cloudPtR;
        int  cloudPtG;
        int  cloudPtB;

        if (firstCloud) { // do initializations here...
          // convert ROS input cloud to PCL format
          pcl::PointCloud<pcl::PointXYZRGB>::Ptr pcl_input_cloud(new
pcl::PointCloud<pcl::PointXYZRGB>);
          // convert Input cloud from ROS format to PCL format
          pcl::fromROSMsg(*ros_input_cloud, *pcl_input_cloud);
          ROS_INFO("received first pointcloud");
          ROS_INFO("width of cloud: %d ",pcl_input_cloud->width);
          ROS_INFO("height of cloud: %d",pcl_input_cloud->height);

          // convert input cloud to ROS image; this image will be suitable for 2-D openCV processing,
```

```
          // and its i,j matrix coordinates will correspond to i,j matrix coordinates of the point
cloud
          // thus, if one selects a pixel in 2-D, can look up corresponding 3-D coords...assuming the
pixel is valid
          pcl::toROSMsg (*ros_input_cloud, ROS_image_of_input_cloud); //convert the cloud
          // publish the ROS image message: ... not really necessary for this demo
          image_pub.publish(ROS_image_of_input_cloud);

          // first call: set up width, height and number of points for the global point-cloud object
"pcl_remembered"
          // this is needed since 2-D mouse selections relate to a prior pointCloud.  Thus, need to
save the prior
          // point Cloud in order to associate the 2-D mouse clicks with the corresponding 3-D points
          pcl_remembered.width = pcl_input_cloud->width;
          pcl_remembered.height = pcl_input_cloud->height;
          pcl_remembered.points.resize( pcl_remembered.width *  pcl_remembered.height);

          copyCldPtrToCloud(pcl_input_cloud,pcl_remembered); //stores current pointCloud for later
recall
          firstCloud=false;  //flag (global) to recognize that this initialization is done;
                             // would be cleaner with classes and a constructor
        }

        if(getPtFlag) {
          getPtFlag=false; // here if flag says a new mouse click selected a point; reset the trigger

          //if here, then two things happened: got a new pointcloud message and a new mouse click
occurred
          // mouse click refers to the OLD point cloud, so access data from pcl_remembered

          //pcl::PointXYZRGB pcl_pt; // could use this to help comb through pointCloud...

          // look up 3-D point info corresponding to mouse-click coords--compute 1-D index:
          pcl_index =  mouse_click_y * (pcl_remembered.width) + mouse_click_x; //computed 1-D index

          //alt: could use pcl_pt = pcl_input_cloud->at(row,col);  then access as pcl_pt.x, etc
          // get the data;

          cloudPtX = pcl_remembered.points[pcl_index].x;
          cloudPtY = pcl_remembered.points[pcl_index].y;
          cloudPtZ = pcl_remembered.points[pcl_index].z;
          ROS_INFO("selected point x,y,z = %f %f %f",cloudPtX,cloudPtY,cloudPtZ);
          cloudPtR = pcl_remembered.points[pcl_index].r;
          cloudPtG = pcl_remembered.points[pcl_index].g;
          cloudPtB = pcl_remembered.points[pcl_index].b;
          ROS_INFO("color of pt, RGB = %d %d %d",cloudPtR,cloudPtG,cloudPtB);

          // should publish target..will need to convert from left-camera optical frame

          // recognize new target only if data is valid:
          // the following tests if any of the values are NaN
          if (cloudPtX!=cloudPtX || cloudPtY!=cloudPtY || cloudPtZ!=cloudPtZ)  {
              pickedTarget=false;
              ROS_INFO("bad point: NaN data");
          }
          else  {
             pickedTarget=true; //this global flag indicates that a valid new point has been selected
             // publish this identified point as a PointStamped, suitable for use with frame transforms
             userPickedPoint.point.x=cloudPtX;
             userPickedPoint.point.y=cloudPtY;
             userPickedPoint.point.z=cloudPtZ;
             userPickedPoint.header = pcl_remembered.header; // this will keep the same time stamp and
reference frame
                                                             // as the original point cloud

             // and publish the result
             pickedPointPub.publish(userPickedPoint);
          }


          // make a new 2-D image from point cloud available for more mouse clicks
          // convert input cloud to ROS image and publish it:
```

```
          // Input cloud in PCL format
          pcl::PointCloud<pcl::PointXYZRGB>::Ptr pcl_input_cloud(new
pcl::PointCloud<pcl::PointXYZRGB>);

          // convert Input cloud from ROS format to PCL format
          pcl::fromROSMsg(*ros_input_cloud, *pcl_input_cloud);


          // here is an example of color filtering of a point cloud:
          // its use is suppressed here, but it works;
          // currently only allowing near black colors through--hard-coded filter bounds
          /*
          ros_output_cloud = filter_rgb(ros_input_cloud,
                                                                0, 47,  // red
                                                                0, 42,  // green
                                                                0, 52); // blue

          cloud_pub.publish(ros_output_cloud);
          */

          pcl::toROSMsg (*ros_input_cloud, ROS_image_of_input_cloud); //convert the cloud
          image_pub.publish(ROS_image_of_input_cloud); //publish this 2-D image--not necessary for
this demo, since
                                                       // ROS_image_of_input_cloud is a global var

          copyCldPtrToCloud(pcl_input_cloud,pcl_remembered); //save the new cloud to refer to after
            //receiving the next mouse click
        }

}



// callback function for mouse events
// save the coords of a mouse click on scene of interest--save these as global vars and use in
pointCloud callback
// also, make note that a new click occurred by setting the global flag "getPtFlag"
void onMouse(int event, int x, int y, int flags, void* param)
{
    Mat img,img3;
    cv_bridge::CvImagePtr cv_ptr;  //use cv_ptr->image in place of "image" in OpenCV manual example

    cv_ptr = cv_bridge::toCvCopy(ROS_image_of_input_cloud, enc::BGR8); //convert ROS message to openCV
format

    img = cv_ptr->image; // synonym...slightly simpler

    //img2 = img.clone();  //example of how to copy an image

    // respond only to left button clicks, in this example code:
    if (event == CV_EVENT_LBUTTONDOWN)
    {
        mouse_click_x = x;  //pass mouse coords as globals
        mouse_click_y = y;

        // some debug and info display:
        Vec3b p = img.at<Vec3b>(y,x);
        ROS_INFO("image RGB = %d %d %d",p[2],p[1],p[0]);
        ROS_INFO("image x,y = %d %d",mouse_click_x,mouse_click_y);
        getPtFlag=true; // flag to indicate new mouse click needs processing
    }

    imshow("image_window", img); //update the image display
}



// example of color filtering directly in pointCloud space
sensor_msgs::PointCloud2 filter_rgb(const sensor_msgs::PointCloud2ConstPtr& ros_input_cloud,
                                                                        int l_r,
                                                                        int u_r,
                                                                        int l_g,
```

```
                                                                int u_g,
                                                                int l_b,
                                                                int u_b)
{
        // Input cloud in PCL format
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr pcl_input_cloud(new pcl::PointCloud<pcl::PointXYZRGB>);
        // Filtered cloud in PCL format
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr pcl_filtered_cloud(new
pcl::PointCloud<pcl::PointXYZRGB>);

        sensor_msgs::PointCloud2 ros_filtered_cloud; // Filtered Cloud in ROS Format

        // convert Input cloud from ROS format to PCL format
        pcl::fromROSMsg(*ros_input_cloud, *pcl_input_cloud);

        pcl_filtered_cloud->width = pcl_input_cloud->width;
        pcl_filtered_cloud->height = pcl_input_cloud->height;
        pcl_filtered_cloud->points.resize(pcl_filtered_cloud->width * pcl_filtered_cloud->height);

        for(unsigned int i = 0; i < pcl_input_cloud->size(); ++i)
        {
                if(pcl_input_cloud->points[i].r < l_r || pcl_input_cloud->points[i].r > u_r)
                        continue;
                if(pcl_input_cloud->points[i].g < l_g || pcl_input_cloud->points[i].g > u_g)
                        continue;
                if(pcl_input_cloud->points[i].b < l_b || pcl_input_cloud->points[i].b > u_b)
                        continue;

                // if made it to here then the point is within all 3 color ranges
                // so copy the point to the filtered cloud
                pcl_filtered_cloud->points[i].x = pcl_input_cloud->points[i].x;
                pcl_filtered_cloud->points[i].y = pcl_input_cloud->points[i].y;
                pcl_filtered_cloud->points[i].z = pcl_input_cloud->points[i].z;
                pcl_filtered_cloud->points[i].r = pcl_input_cloud->points[i].r;
                pcl_filtered_cloud->points[i].g = pcl_input_cloud->points[i].g;
                pcl_filtered_cloud->points[i].b = pcl_input_cloud->points[i].b;
        }

        // convert Filtered cloud from PCL format to ROS format
        pcl::toROSMsg(*pcl_filtered_cloud, ros_filtered_cloud);

        ros_filtered_cloud.header = ros_input_cloud->header;

        return ros_filtered_cloud;
}



int main(int argc, char** argv)
{
        ros::init(argc, argv, "point_cloud_HMI");
        ros::NodeHandle nh_processed,nh_orig_image,nh_handGoal;
        //image_transport::ImageTransport it(nh_orig_image); // fancy class used for dealing with
image topics

        ros::Rate looprate(10); //10Hz max loop rate
        ROS_INFO("subscribing to multisense points2...");

        // subscribe to this pointcloud2 topic
        ros::Subscriber sub = nh_orig_image.subscribe("/multisense_sl/camera/points2", 1,
cloud_callback);

        // convert the incoming, original pointcloud into an image and publish it to this topic:
        image_pub = nh_orig_image.advertise<sensor_msgs::Image>(ros::this_node::getName() + "/
cloud2image", 1);
        // the above is not really needed, since the mouse-interaction is integrated in this node

        // publish the processed point-cloud topic to this topic...
        // uncomment to restore
        //cloud_pub = nh_processed.advertise<sensor_msgs::PointCloud2>(ros::this_node::getName() + "/
cloud", 1);
```

```
        // publisher for selected points:
        pickedPointPub = nh_handGoal.advertise<geometry_msgs::PointStamped>("userPickedPoint",1);


        ROS_INFO("waiting for pointcloud from Atlas..."); // need to receive an image from callback
before can display it;
        // first iteration of callback fnc sets "firstCloud" flag to "false"

        while(firstCloud) {
           ros::spinOnce(); // let the callback get an image;
           ros::Duration(0.1).sleep();
        }

        ROS_INFO("setting up named window");
        namedWindow("image_window"); //window for openCV displays

        // assign "onMouse()" as the callback fnc for mouse events
        setMouseCallback("image_window", onMouse, 0);

        // done with setups; enter main loop;  callbacks do all the work now

        while(ros::ok()) {
          ros::spinOnce(); // allow getting fresh pointclouds received;
          waitKey(100); // images update only after mouse events;
          looprate.sleep();
        }
}
```

```
// main file for atlas control moving right hand to HMI selected 3-D point: wsn  March 6, 2013
// theory of operation:
// callback fnc HMI_callback() subscribes to topic "/userPickedPoint"
// HMI_callback() copies the incoming PointStamped to a global variable, and sets the flag
"HmiPointIsGood"
// This point is interpreted as the desired origin of frame "/right_f1_base" on the palm of the right
hand
// The right-arm Jacobian is computed and dq is computed based on J'*dp, where dp is the hand error
(in pelvis frame)
// Jacobian and dq are recomputed iteratively.  Should result in hand converging on goal

#include "AtlasJointControl.h"
#include "JacobianComputer.h"
#include <geometry_msgs/PointStamped.h> //datatype of published desired hand position from HMI


geometry_msgs::PointStamped g_PickedPoint; //global var to get values from HMI callback
osrf_msgs::JointCommands g_jntCmd;
double jntCmdsFromAtlas[NJoints];

bool HmiPointIsGood = false;

//callback to get hand position commands from HMI
void HMI_callback(const geometry_msgs::PointStamped& pickedPoint) {
  ROS_INFO("got point: x,y,z= %f %f %f",pickedPoint.point.x,pickedPoint.point.y,pickedPoint.point.z);
  g_PickedPoint=pickedPoint;
  HmiPointIsGood=true; //notify "main" that it is OK to use g_PickedPoint
}

//callback to see what was commanded to the robot; odd behavior--does not seem to recognize
// commands since start-up, e.g. neck bend
void atlas_cmds_callback(const osrf_msgs::JointCommands& jntCmds) {
  g_jntCmd= jntCmds; // make info globally available
  for (unsigned int i=0;i<NJoints;i++)
     jntCmdsFromAtlas[i]=jntCmds.position[i];
}

//helper fnc to find max abs val of entries in a vector
double findMaxAbs(double xvec[],int Npts) {
 double maxval;
 maxval=fabs(xvec[0]);
 for (int i=1;i<Npts;i++)
  if (fabs(xvec[i])>maxval)
    maxval=fabs(xvec[i]);
 return (maxval);
}

int main(int argc, char** argv)
{
  double looprateHz = 10.0;
  tf::Vector3 rHandOriginDesired;
  tf::Vector3 rHandOriginActual;
  tf::Vector3 rHandPrev;
  tf::Vector3 drHand;
  tf::Vector3 JdqRHand;
  tf::Vector3 rHandErrorVec;
  double jointSpaceCommand[NJoints];
  double dqVec[NJoints];
  tf::StampedTransform transform;
  double prevAngles[NJoints];
  double currentAngles[NJoints];
  double dq[NJoints];
  geometry_msgs::PointStamped tfPickedPoint;  // HMI point transformed to pelvis frame

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

  // 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;                         //
  ros::NodeHandle nh_subHMI;
```

```
  ros::NodeHandle nh_subCmds;

  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, can 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);

  // set up subscriber to HMI input:
  ros::Subscriber sub_HMI = nh_subHMI.subscribe("/userPickedPoint", 1,HMI_callback);

  //subscriber to atlas commands:
  ros::Subscriber sub_cmds = nh_subCmds.subscribe("/atlas/joint_commands", 1,atlas_cmds_callback);


  // wait for valid data from joint commands callback--see what is currently commanded to the robot
joints
  jntCmdsFromAtlas[0]=-1000.0; //init w/ impossible value; cycle until value is replaced
  while (jntCmdsFromAtlas[0]<-500.0) {
    ros::spinOnce();
  }

  ROS_INFO("current joint angle commands: ");
  atlasJointControl.displayAngles(jntCmdsFromAtlas);

  //look at the actual joint angles at this moment:
  //test for bad initial value in sensed angles--put there via constructor of atlasJointControl; cycle
until value is replaced
  atlasJointControl.getCurrentAngles(currentAngles);
  while (currentAngles[0]<-500.0) {
    ros::spinOnce();
    atlasJointControl.getCurrentAngles(currentAngles);
  }
  ROS_INFO("current angles:");  //debug: display the received joint angles; should all be ~0.0
                                // for Atlas in start-up pose
  atlasJointControl.displayCurrentAngles();

  JacobianComputer jacobianComputer; // create a Jacobian computer object; might bury this inside
another object...

  //where is the hand now?  (w/rt pelvis frame)
  bool tf_not_ready=true;
  int ntries=0;
  while(tf_not_ready) {
    try {
          tf_not_ready=false;
          tf_listener.lookupTransform("/pelvis", "/right_f1_base", ros::Time(0), transform);
    }
    catch (tf::TransformException ex) { //do nothing
      tf_not_ready=true;
      ntries++;
      ROS_INFO("waiting for right_f1_base frame; ntries = %d",ntries);
      ros::Duration(0.3).sleep();
    }
  }

  rHandOriginActual= transform.getOrigin(); // extract the origin of r_hand frame, relative to pelvis
  ROS_INFO("initial hand pos = %f %f %f",rHandOriginActual[0],rHandOriginActual[1],rHandOriginActual
[2]);

  // for smooth start-up, set hand goal coincident with current, actual hand position
  tfPickedPoint.point.x= rHandOriginActual[0];  //init right-hand goal w/rt pelvis
```

```
  tfPickedPoint.point.y= rHandOriginActual[1];
  tfPickedPoint.point.z= rHandOriginActual[2];

  double goalAngles[NJoints];
  double moveDuration; //can specify speed of move via this

  //to start, command robot to go to its current pose...should stand still
  atlasJointControl.getCurrentAngles(currentAngles);

  //may initialize move command to current angles:
  //better is to use current commands...having trouble with neck command snapping back to zero
  //atlasJointControl.setNewJointSpaceGoal(jntCmdsFromAtlas,jntCmdsFromAtlas, 1.0);// move time=1Sec
  atlasJointControl.setNewJointSpaceGoal(currentAngles,currentAngles, 1.0);// move time=1Sec

  // define a right-hand goal position: initially, same as current hand position
  rHandOriginDesired[0]= tfPickedPoint.point.x;
  rHandOriginDesired[1]= tfPickedPoint.point.y;
  rHandOriginDesired[2]= tfPickedPoint.point.z;


  //ROS_ERROR("false alarm");

  ROS_INFO("starting main loop...");
  while(ros::ok())  // main loop
  {
   double qmax = 0.001;
   if(atlasJointControl.isMoveDone()) { // test if time to set a new move
       // get the current hand position:
       tf_listener.lookupTransform("/pelvis", "/right_f1_base", ros::Time(0), transform);
       rHandOriginActual= transform.getOrigin(); // extract the origin of r_hand frame, relative to
pelvis

       ROS_INFO("hand pos = %f %f %f",rHandOriginActual[0],rHandOriginActual[1],rHandOriginActual[2]);

       // get desired hand position from HMI...transform this point to the pelvis frame:
       // input PickedPoint (updated by callback) and transform to tfPickedPoint
       if (HmiPointIsGood) {
        try{
         tf_listener.transformPoint("/pelvis", g_PickedPoint, tfPickedPoint); //tfPickedPoint is goal
in pelvis frame
         //ROS_INFO("main: HMI orig = %f %f %
f",g_PickedPoint.point.x,g_PickedPoint.point.y,g_PickedPoint.point.z);
         ROS_INFO(" ");
         ROS_INFO("desired hand pose...");
         ROS_INFO("main: HMI xfmd = %f %f %
f",tfPickedPoint.point.x,tfPickedPoint.point.y,tfPickedPoint.point.z);
         rHandOriginDesired[0]= tfPickedPoint.point.x;
         rHandOriginDesired[1]= tfPickedPoint.point.y;
         rHandOriginDesired[2]= tfPickedPoint.point.z;
          }
        catch(tf::TransformException ex)
         {
           // transform will fail if point's timestamp is too old
           ROS_ERROR("HMI catch: failed to do point transform");
         }
         HmiPointIsGood=false; //only transform new points
       }
       else
         ROS_INFO("no new HMI point");

       rHandErrorVec= rHandOriginDesired-rHandOriginActual;
       ROS_INFO("hand err = %f %f %f",rHandErrorVec[0],rHandErrorVec[1],rHandErrorVec[2]);


       jacobianComputer.computeRightHandJacobians(tf_listener); //  compute a right-hand Jacobian

       //compute dqVec for dp step towards goal as dqVec= J'dp
       jacobianComputer.computeRightHandJTransposeDp(rHandErrorVec,dqVec);

       // make sure dqVec is not too large: find max value
       qmax = findMaxAbs(dqVec,NJoints);
       if (qmax>0.05)
```

```
        for (unsigned int i=0;i<NJoints;i++)
          dqVec[i]*=(0.05/qmax); // scale entire vec so max increment does not exceed 0.05 rad in move
duration


       ROS_INFO("J'dp[22-25]= %f %f %f %f",dqVec[22],dqVec[23],dqVec[24],dqVec[25]);


       atlasJointControl.getCurrentGoalJointAngles(jointSpaceCommand); //look up previous joint-space
cmd vec

       for (unsigned int i=22;i<26;i++) //only increment right-arm angles (not back joints)
         jointSpaceCommand[i]+=dqVec[i];  //increment the previous joint-space command by vector dq

       ROS_INFO("cmd[22-25]= %f %f %f %f",jointSpaceCommand[22],jointSpaceCommand[23],
                                          jointSpaceCommand[24],jointSpaceCommand[25]);

       moveDuration=0.3; // this is slow; could speed up w/ shorter moveDuration and/or larger max dq
per iteration
       atlasJointControl.setNewJointSpaceGoal(jointSpaceCommand, moveDuration);
     }


   // next line commands a joint-space incremental move--invoked at frequency "looprate"
   atlasJointControl.update();  //compute/publish incremental motion commands towards current goal

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