Interfacing to the DARPA Robotics Challenge simulator:

Wyatt Newman

Updated: September 2013

These notes provide a brief introduction to interfacing to the DARPA Robotics Challenge (DRC) "Gazebo" simulator via ROS (Robot Operating System) code. Except as noted, the code described herein communicates with a simulation of the Atlas robot (see http://en.wikipedia.org/wiki/Atlas_%28robot%29) or with Hong Kong University's Atlas robot without change. Code developed and tested on the simulator may be executed subsequently on the physical robot. Differences in simulated vs. actual behavior are expected, as a result of imperfect modeling and imperfect dynamic simulation. However, the "drcsim" model of Atlas is continuing to be refined by the Open Source Robotics Foundation (OSRF—see http://osrfoundation.org), providing increasingly accurate simulation capability.

The drcsim simulator was originally released by OSRF in October, 2012 with version 1.0. These notes refer specifically to version 2.7 of "drcsim." See http://gazebosim.org/wiki/DRC for more drcsim details and installation instructions. Getting drcsim installed to run correctly requires attention to considerable detail, including OS version (Ubuntu 12.04 "precise", 64-bit in the present case), version of ROS ("fuerte" in the present case), version of Gazebo (version 1.9 at present), and a compatible graphics GPU (nVidia graphics hardware preferred). Version 3.0 of drcsim is anticipated in the near future, and this may require installing more recent versions of Ubuntu and ROS. However, it is expected that the C++ code described here will continue to work with drcsim updates.

Within drcsim, ROS publisher and subscriber nodes have already been written. By launching the simulator in "gazebo" (which automatically starts up roscore as well), one can examine all of the topics that are available. Sensor nodes publish sensor information. Corresponding joint-control topics are subscribed to by the simulator. It is up to the user to interpret sensor values and to issue actuator commands to move the simulated robot in its virtual world.

The Boston Dynamics "Atlas" robot has a custom, proprietary interface that transmits sensor information and receives joint commands. Joint-sensor outputs and joint-command inputs are updated at 333Hz. Team HKU has written a ROS translator that acquires Atlas data and publishes it in the same format as the drcsim simulator. Similarly, the HKU interface subscribes to a ROS joint-command topic and communicates these commands (also at 333Hz) to the Atlas robot. With this "bridge" software, sending/receiving data to/from Atlas is identical to communicating with the drcsim simulator.

1) Running a Gazebo Simulation of Atlas: After complete installation, the DRC (DARPA Robotics Challenge) simulator may be launched with a desired "world" model. For example, the command:

roslaunch atlas_utils vrc_task_3.launch

will start up a simulation of Atlas initially standing within a "starting pen" with a hallway leading to an annex that contains some interesting models (including a table, a firehose, a spigot and a valve). The above instruction uses the standard "roslaunch" command, which refers to a launch script that starts multiple processes running. The argument "atlas_utils" refers to a ROS package. You can find this package directory by entering: *roscd atlas_utils*, which will take you to this directory. (*roscd package_name* works in general to navigate to a package named "package_name"). Within this directory, you will see a subdirectory called "launch." Generally, roslaunch will look for a directory named "launch" within any specified package. Within the atlas_utils launch directory, there are dozens of options of launch files, of which "vrc_task_3.launch" is just one.

The launch file starts up two "gazebo" processes: a gazebo server and a gazebo client. The server performs a dynamic simulation of models in the virtual world. Some models are static (such as a floor), but still require specification of contact properties (e.g. stiffness and friction). Other models are dynamic, which requires specification of more detail (mass, center of mass, inertia, geometry, etc). By far, the most complex model included by vrc_task_3.launch is the complete specification of the Atlas model. Details of this robot model are specified in a "universal robot description file" (URDF), located in the directory: /usr/share/drcsim-2.7/gazebo_models/atlas_description/atlas/atlas.urdf. (This directory will change with updates to drcsim, but the launch file will automatically refer to the appropriate directory).

By default, the gazebo server runs an Open Dynamics Engine (ODE) physics simulator. (See http://ode-wiki.org/wiki). ODE simulates complex dynamics of interacting bodies, including multi-jointed robots touching (or colliding with) modeled environments. This includes modeling foot forces/torques, hands contacting (movable or immovable) objects, and any body part contacting any model in the environment (e.g. stubbing a toe, banging a shin, bumping the torso or pelvis against furniture, …). Forces and torques from contacts are computed, and these stimuli are included in the computation of translational and rotational accelerations of all objects (including coupled robot links). Influences of actuator torques, joint kinematic constraints, joint friction, contact forces/torques, gravity, and Coriolis and centrifugal effects are all considered in computing body accelerations. ODE does an impressive job of simulating highly complex dynamics (although its treatment of contact friction needs improvement).

One of the most demanding requirements of ODE simulation is fast computation of contact forces between bodies. This computation requires reference to detailed geometric descriptions of the surfaces of modeled objects (the "collision model"). Each time step of simulation, Gazebo/ODE must identify which objects are in contact, and specifically where the contact sites are and what contact forces occur at these sites.

Gazebo also computes virtual sensor data, which is computationally demanding. Sensor values for Atlas include joint positions, joint velocities and joint actuator torques for all 28 joints. Wrist and ankle force/torque sensors are also simulated, as well as Inertial Measurement Unit (IMU) values for an IMU in the robot's pelvis (yielding 3 accelerations and 3 angular velocities).

More demanding still is simulation of the multisense_sl robot head, which includes stereo color cameras and 3-D LIDAR sensing. Virtual camera data is synthesized by computing how light reflects off of a modeled environment and impinges on pixel cells within a simulated camera. This computation includes camera properties (e.g. focal length and CCD imaging sensor resolution) as well as coordinate-frame transforms (appropriate viewpoint of the camera within the virtual world). Impressively, high-resolution, color images are synthesized and published at realistic speeds (e.g. 30Hz).

The gazebo server clearly has a very computationally demanding task. Nonetheless, on a (current) reasonably powerful PC, Atlas simulations can be performed at nearly real time (depending on the complexity of the world model).

The counterpart to the gazebo server is a gazebo client, which is the user interface. The client is responsible for graphical display as well as user controls. Via the graphical interface, one can run/pause the simulation, as well as pan/tilt/zoom viewpoints of the robot in the virtual world. The simulation properties can also be altered interactively. For example, the user can select/drag the robot (or other objects) to new locations, then resume simulation from the new state. Models can be added or deleted interactively as well. View-menu options allow alternatives to display contact sites, joint coordinate frames, centers of mass, or alternative renderings. Through the Gazebo client display, one can observe robot behaviors under program control while the robot interacts with a simulated environment.

2) Gazebo, ROS and Atlas: Gazebo, ROS and ODE were originally separate projects. However, they have been integrated with "bridge" software. As Gazebo invokes ODE and sensor emulation software to simulate Atlas, Gazebo transmits its simulated sensor data and receives actuator effort commands via ROS topics.

After starting up drcsim (e.g., using: *roslaunch atlas_utils vrc_task_3.launch*), 12 ROS nodes are launched, which use nearly 400 ROS topics. (type: *rosnode list* to see the list of nodes and *rostopic list* to see the list of topics). Most of the topics are sensor topics associated with the sensor head and with sensors integrated within the Sandia hands. Aside from the sensor head and the Sandia hands, there are 11 topics associated with the Atlas robot body.

Of the Atlas topics, the most valuable are: /atlas/atlas_command and /atlas/atlas_state. The former accepts user inputs (including joint torque commands) to cause Atlas to move. The simulator subscribes to this topic, and the user must publish to this topic to cause the actuators to move. The latter topic (/atlas/atlas_state) is published by the simulator. The user may write ROS nodes that subscribe to this topic to receive Atlas sensor data. This topic is where one can find the joint positions, joint velocities and joint actuator torques for all 28 joints, as well as the wrist and ankle force/torque sensors values and the pelvis IMU values.

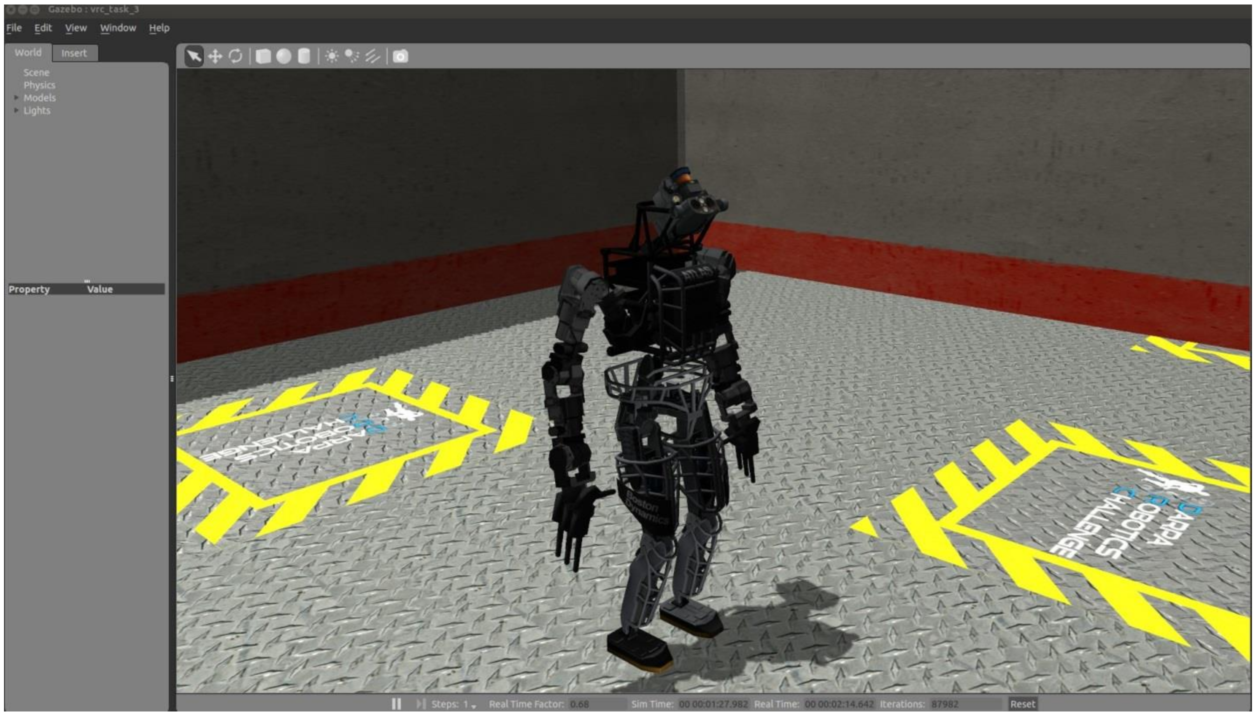
In a command window, entering: *rostopic hz /atlas/atlas_state* shows that the Atlas sensor values are being republished at an update rate of 1kHz. This corresponds to the simulator time step of 1msec. Note, though, that this reported frequency is based on simulated (not clock) time. Depending on the complexity of the simulation and the speed of the computer, the simulation might occur in slow motion relative to real time. The gazebo client window shows a "real time factor" value indicating the simulator speed relative to real time. A second caveat of the update rate of atlas_state is that the actual Atlas robot only updates its sensor data at 333Hz. In development, the user thus should not write code that depends on 1kHz sensor update rates. Entering*: rostopic echo /atlas/atlas_state* shows a display of what data is being published. (The output will scroll by too fast to read, but you can pause with ctl-S or halt it with ctl-C to scroll up the display to observe the data).

Entering: *rostopic info /atlas/atlas_state* reveals that this topic involves messages of type atlas_msgs/AtlasState. Entering: *rosmsg show atlas_msgs/AtlasState* reveals the details of this message type. Fields include vectors of joint positions, joint velocities and joint efforts, as well as IMU data of angular velocities and linear accelerations and 6-dimensional force/torque signals for both wrists and both ankles.

One of the sensor topics published by Gazebo is "/multisense_sl/camera/right/image_raw", which is synthetic camera output from the robot's right eye. Messages on this topic are of type "sensor_msgs/Image." Entering: *rostopic hz /multisense_sl/camera/right/image_raw*
shows that this is updated at 30 Hz and that it uses 38MB/s of bandwidth. Options exist for setting various parameters, including update rate, field of view and compressing settings. One may construct a node to perform image processing on such images, or run "rosbag record" to save the synthetic images for offline development of image-processing code.

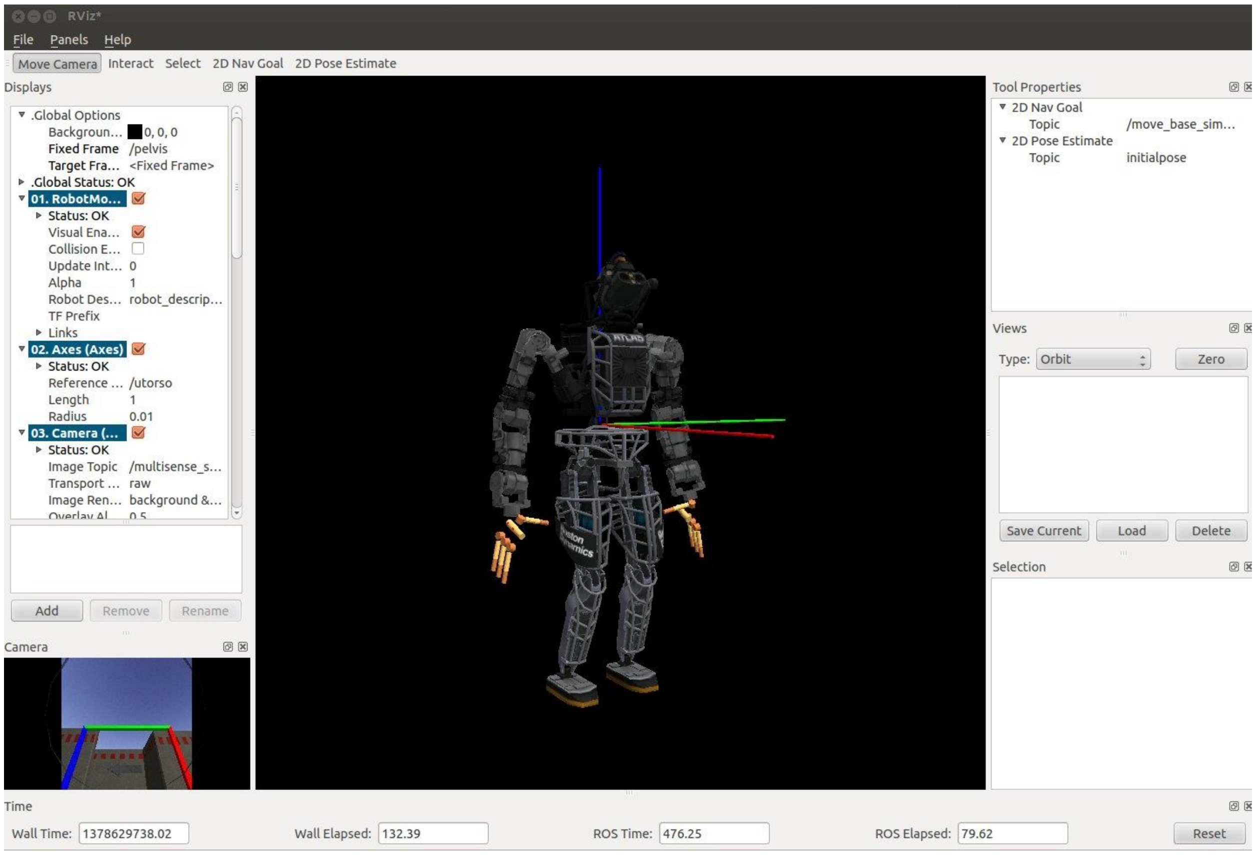
Another available topic is "/multisense_sl/laser/scan", to which gazebo publishes messages of type "sensor_msgs/LaserScan". This topic contains synthetic data from a simulated Hokuyo LIDAR in the robot's sensor head. Modeled objects within the robot's virtual world are sensed by both the cameras and the laser scanner. Data from the physical robot sensor head is published on topics of the same names and same message formats as the synthetic data published by the gazebo DRC simulator. User nodes that process the sensor data should be verbatim compatible with the real sensor data, requiring no code changes to switch between the simulator and the physical robot. The only difference would be that the sensor topics are published by Atlas instead of by Gazebo.

After entering *roslaunch atlas_utils vrc_task_3.launch*, the gazebo client will display an output that looks like the following:



View of Gazebo display running vrc_task_3.launch(zoomed on Atlas)

3) Visualizing data with rviz: Another useful ROS tool is "rviz", which offers a variety of visualization options. In an available terminal, type: **rosrun rviz rviz** This executes a node called "rviz" from a package of the same name. Rviz has options to be set by the user (what data to display), and these options can be saved in a configuration file and re-used the next time rviz is launched. The configuration settings for the screenshot below can be seen in the left panel labeled "displays." Importantly, the "fixed frame" value is set to "/pelvis." The options being displayed are "RobotModel", coordinate axes of one of the link frames (/utorso in this example), and synthesized images from the left color camera of the multisense_sl robot head.

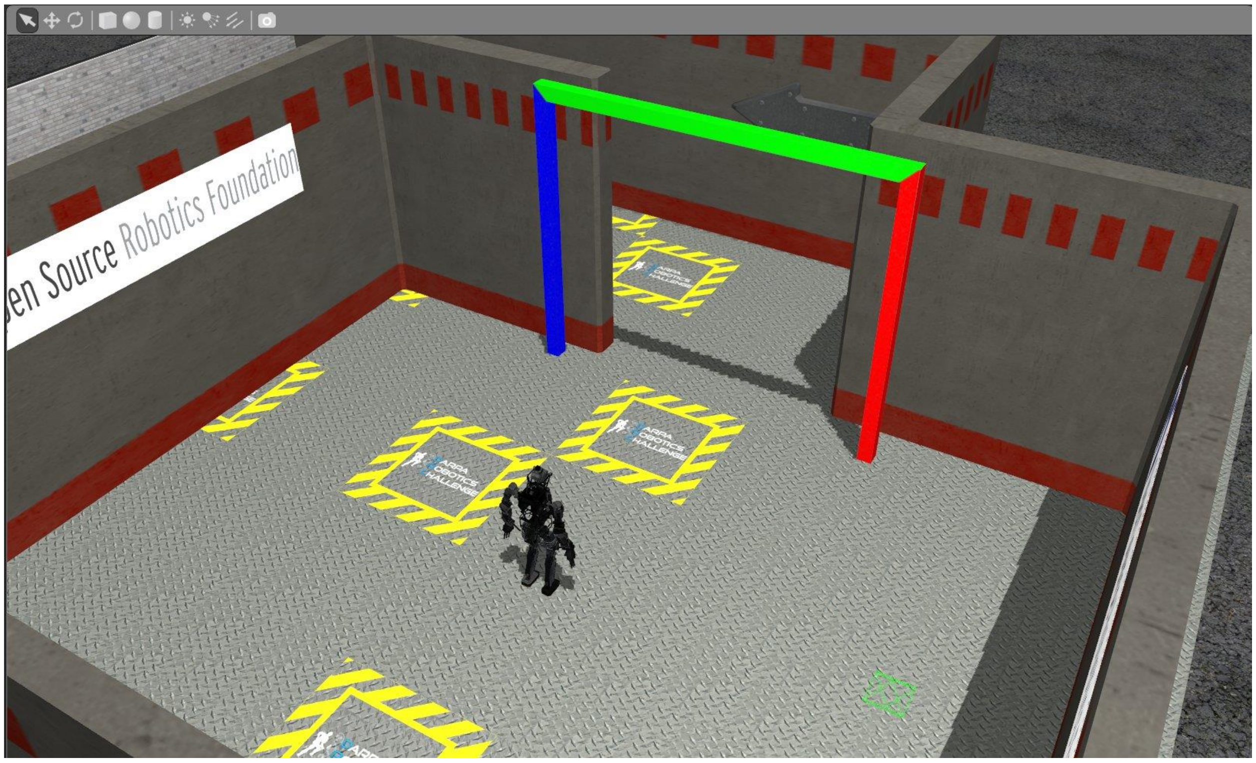


```
▼ .Global Options
    Backgroun...   0, 0, 0
    Fixed Frame    /pelvis
    Target Fra...  <Fixed Frame>
▸ .Global Status: OK
▼ 01. RobotMo...   ✓
  ▸ Status: OK
    Visual Ena...  ✓
    Collision E... ☐
    Update Int...  0
    Alpha          1
    Robot Des...   robot_descrip...
    TF Prefix
  ▸ Links
▼ 02. Axes (Axes)  ✓
  ▸ Status: OK
    Reference ...  /utorso
    Length         1
    Radius         0.01
▼ 03. Camera (...  ✓
  ▸ Status: OK
    Image Topic    /multisense_s...
    Transport ...  raw
    Image Ren...   background &...
    Overlay Al...  0.5

Tool Properties
▼ 2D Nav Goal
    Topic          /move_base_sim...
▼ 2D Pose Estimate
    Topic          initialpose

Views
Type: Orbit

Wall Time: 1378629738.02   Wall Elapsed: 132.39   ROS Time: 476.25   ROS Elapsed: 79.62
```

rviz screenshot of DRC simulator

The rviz view shows the same model of the DRC robot as displayed in the “gazebo” window, but it can be viewed from alternative viewpoints more easily. Further, rviz can display selected sensor data, which is updated dynamically as the robot moves.

The lower-left panel of the rviz display shows a virtual image from the robot’s left eye. This display shows a red, green and blue gateway through a door, as well as some blue sky. This view is constructed from the robot’s pose, which at the time had the head tilted upward. The screencapture below shows the Gazebo display of Atlas, with a wider angle and viewed from behind Atlas.



Gazebo screen capture: zoomed out, and looking from behind Atlas

From the above scene, one can interpret the synthesized camera image from Atlas' point of view. The RGB-colored gateway and doorway is in front of Atlas. With his head tilted up, he would see the green crossbar and sky above.

4) Moving joints under program control:

The following code constitutes a node capable of both subscribing to a gazebo joint state topic and publishing joint-angle commands to gazebo command topics.

```
/**  simple_joint_interface implementation file*/
#include <ros/ros.h>
#include <atlas_msgs/AtlasCommand.h> // message to send commands to Atlas
#include <atlas_msgs/AtlasState.h> // to subscribe to Atlas sensor messages
#include <iostream> //for interactive user I/O
using namespace std;

atlas_msgs::AtlasCommand atlasCommand; // here is an instance of the message
//type used to command Atlas; make it global so init_cmds() can access it
const int NJoints=28;  // mnemonic for number of Atlas joints
double g_jointAngles[NJoints]; //global variable to hold results from sensor-
//message callback

//callback to subscribe to joint state messages; copy results to global array
//for access by main()
void getJointStatesCB(const atlas_msgs::AtlasState& js) {
    for (unsigned int i = 0; i < NJoints; i++) {
        g_jointAngles[i] = js.position[i]; //copy to global var
    }
}
```

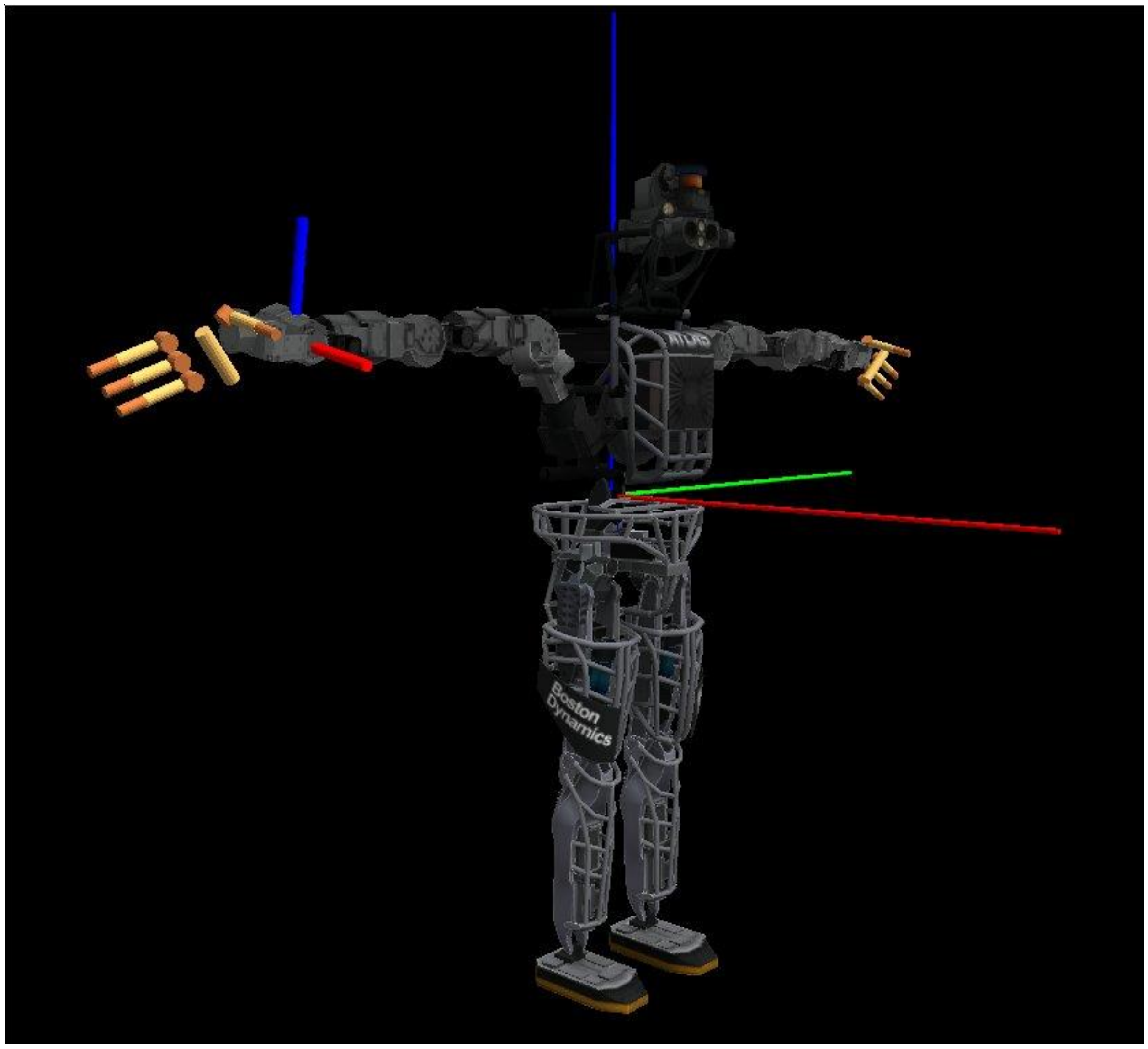
```
//function to initialize gains and setpoints:
void init_cmds() {
    // resize the vectors within this object to hold values for 28 joints
  atlasCommand.position.resize(NJoints);
  atlasCommand.velocity.resize(NJoints);
  atlasCommand.effort.resize(NJoints);
  atlasCommand.kp_position.resize(NJoints);
  atlasCommand.ki_position.resize(NJoints);
  atlasCommand.kd_position.resize(NJoints);
  atlasCommand.kp_velocity.resize(NJoints);
  atlasCommand.k_effort.resize(NJoints);
  atlasCommand.i_effort_min.resize(NJoints);
  atlasCommand.i_effort_max.resize(NJoints);

  // initial commands:
  for (int i=0;i<NJoints;i++) {
        atlasCommand.position[i] = 0.0;
        atlasCommand.velocity[i] = 0.0;
        atlasCommand.effort[i] = 0.0;
        atlasCommand.k_effort[i]    = 255; // for user control of all joints
  }


   // get jnt angles from sensor message, and use these for initial commands:
  for (int i=0;i<NJoints;i++) {
      atlasCommand.position[i]= g_jointAngles[i];
  }

  //set all the controller gains:
  for (unsigned int i = 0; i <= NJoints; i++)
  {
    atlasCommand.ki_position[i] = 0.0; // turn off integral gain
    atlasCommand.kp_velocity[i]  = 1.0; // provide inherent damping
  }

  atlasCommand.kp_position[0]= 5000.0; //1st torso joint:  N-m/rad
  atlasCommand.kd_position[0]  = 40.0; // velocity fdbk: N-m/(rad/sec)
  // …
  // set proportional gains on leg joints
  atlasCommand.kp_position[4]= 5000.0; // left leg, 1st hip jnt Kp
  atlasCommand.kp_position[10]= 5000.0; //right leg, same jnt
  atlasCommand.kp_position[5]= 1200.0;  //left leg, 2nd hip jnt
  atlasCommand.kp_position[11]= 1200.0;  //right leg, 2nd hip jnt
  // …
  //set gains for arms:
  atlasCommand.kp_position[16]= 3000.0; //left arm, 1st shoulder jnt
  atlasCommand.kd_position[16]= 30.0; //left arm velocity fdbk
  atlasCommand.kp_position[22]= 3000.0;   // right arm...
  atlasCommand.kd_position[22]= 30.0; //
  //… skip rest of code setting Kp and Kd values…
  // … (rest of these values are in the code on-line)
  // …
  atlasCommand.kp_position[27]= 500.0; //right arm wrist
  atlasCommand.kd_position[27]= 10.0;

}
```

```
int main(int argc, char** argv) {

    // ROS set-ups:
    ros::init(argc, argv, "simple_joint_interface");
    ros::NodeHandle nh;
    //here is the publisher to send cmds to Atlas
    ros::Publisher pub_joint_commands=
     nh.advertise<atlas_msgs::AtlasCommand>("/atlas/atlas_command", 1, true);
    ros::Rate timer(10); //timer to sleep for 0.1 sec

    ROS_INFO("setting up subscriber to atlas state");
    ros::Subscriber sub =
       nh.subscribe("/atlas/atlas_state", 1, getJointStatesCB);

    //let the subscriber get joint-state data to use for command
    //initialization; let it run for a while
    ROS_INFO("getting sensor messages: ");
    for (int i=0;i<30;i++) {
      timer.sleep(); //sleep for 0.1 sec
      ros::spinOnce();
    }

    ROS_INFO("initializing controller");
    init_cmds(); // initialize gains and joint commands to good values

  ROS_INFO("sending initial command to Atlas...might droop slightly");
  //send this command 10 times...communications initialization can require
  //some warm-up time
  for (int i=0;i<10;i++) {
      atlasCommand.header.stamp = ros::Time::now(); // puts time stamp in
       //header; not really needed in this simple example
      pub_joint_commands.publish(atlasCommand); // send jnt commands to Atlas
      ROS_INFO("sleep.");
      timer.sleep(); //sleep for 0.1 sec
      ros::spinOnce();
  }
  ROS_INFO("starting interactive loop...");
  int jnt_num;
  double angle_cmd;
    // main loop
    while (ros::ok()) {
        cout<<"enter a joint number, 0 through 27 (out of range to quit): ";
        cin>>jnt_num;
        if (jnt_num<0 || jnt_num>NJoints)
            break;
        cout<<"current joint angle is "<<g_jointAngles[jnt_num]<<endl;
        cout<<"enter an angle command: ";
        cin>>angle_cmd;
        atlasCommand.position[jnt_num]=angle_cmd;
        atlasCommand.header.stamp = ros::Time::now(); //enter time stamp
        pub_joint_commands.publish(atlasCommand); // send jnt commands
        ros::spinOnce(); // allow joint state subscriber to get a message
    }
    return 0;
}
```

In the above example interface node, the code subscribes to the topic atlas/atlas_state and publishes to the topic atlas/atlas_command using message types defined in atlas_msgs. Every command message sent to Atlas includes position commands, velocity commands, direct torque commands, and specification of control gains. An initialization function fills in valid values for control gains (though this function is not shown in full, for brevity, nor are the specified gains optimized). It is important that every command sent in an atlas_command message has valid values in every field. However, the gains set in the initialization function do not have to be altered subsequently, as these initial values are persistent.

As part of the initialization, it is also important that the first set of joint commands be close to the robot’s actual pose. Otherwise, the robot will jolt harshly when the first command is sent. To achieve a smooth start-up, the interface program subscribes to the atlas/atlas_state topic, from which it can observe the robot’s actual pose. The initialization function uses these values to initialize the joint commands to their current values. Essentially, the first command is: “go to your current pose.”

After initialization, the node enters a loop in which it prompts the user for a joint number (0 through 27), displays the current angle of that joint, then prompts for a new angle value from the user. The screencapture below shows the result of commanding all 28 joints to an angle of 0. This pose is the reference position. Coordinate axes corresponding to the pelvis frame are shown, where the colored axes R,G,B are the x, y and z axes, respectively.



Rviz view of Atlas in the reference pose (all joint angles command to 0 radians)

Note that use of the simple interface program here can result in very jerky motions. For example, commanding joint 17 (left-arm elevation) to move from the start-up pose (with arms down at sides) to 0 angle (left arm raised horizontal, out to the left side) results in a violent motion that likely will cause Atlas to lose his balance and fall down. You can get the arm to zero angle more carefully by commanding a sequence of joint angles, incrementally closer to the desired goal. In fact, this is a necessary next layer.

Since Atlas is not running any balance feedback control, he will fall down if you try to move the leg joints—or even if you create sufficient imbalance by moving the arms forward. For testing and development purposes, it can be convenient to initialize the simulator to "pin" the pelvis, so Atlas cannot fall over. This can be done as follows. First, launch the simulator with the following variation:

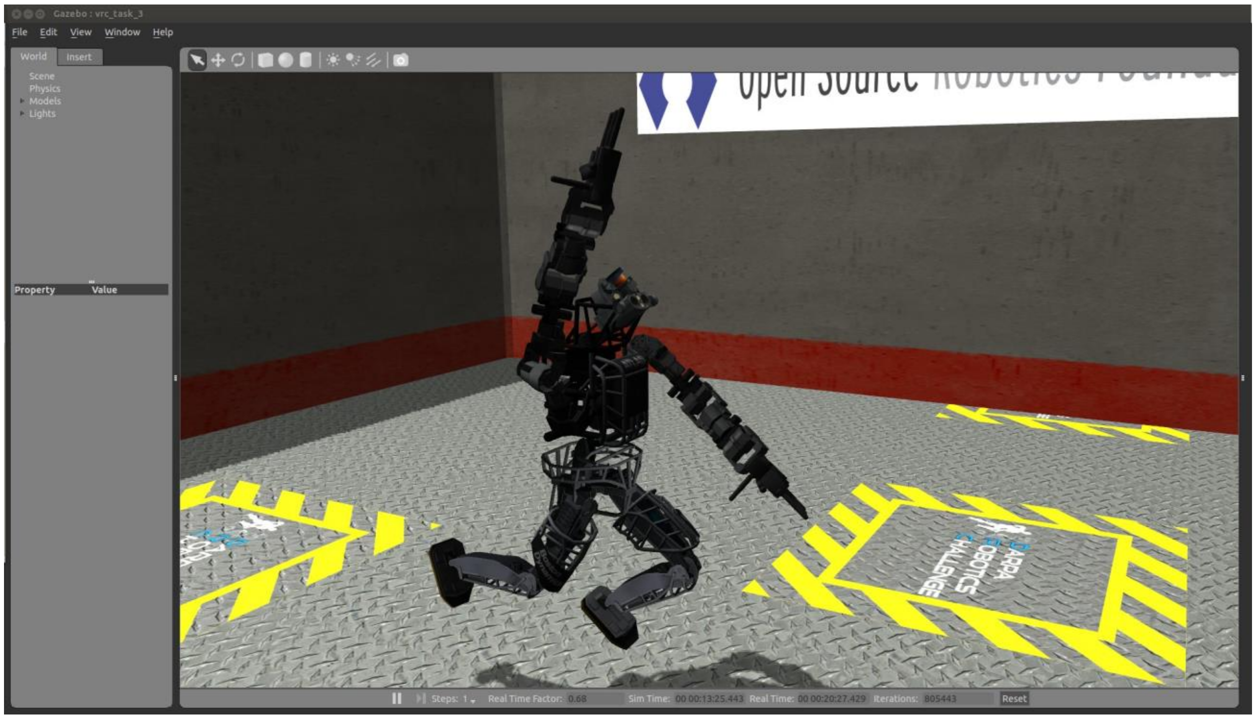
```
 VRC_CHEATS_ENABLED=1 roslaunch atlas_utils vrc_task_3.launch
```

Then, in another terminal, enter the following:

```
rostopic pub - - once  /atlas/mode  std_msgs/String '{data: "pinned_with_gravity"}'
```

This will make Atlas behave as though his pelvis frame were permanently fixed in space. You can then move the legs around (e.g. lift them). You will find, though, leg motions are still constrained by contact conditions (e.g. contact between the feet and the floor prevents additional knee extension).

The figure below shows a screencapture of Gazebo with Atlas pinned and legs commanded to angles such that the feet are both off the floor. The pelvis pinning keeps Atlas suspended in space, although gravity still tugs downwards affecting all joints.



Atlas in Gazebo with pelvis pinned

5) Atlas Joint Descriptions:
Atlas has 28 body joints, and referencing them is not intuitive. There are mnemonic names for the joints that map onto 28 integers (starting from zero). With respect to the reference pose shown, the joint motions may be described as follows. (Note that in more complex poses, the joint motions cannot be described as simply. This description is simply for introductory, descriptive purposes).

The first four joints are equivalent to a "spine." Joint 0 is the first back joint, which causes the robot to rotate its upper body relative to the pelvis about the z axis. Joint 1 is the second back (torso) joint, which causes the torso to bend forward (as though making a bow). Joint 2 is the third torso joint, which is a rotation about the x axis (leaning to the right, for positive rotation). Joint 3 is the neck joint, causing the head to nod up and down. (The neck cannot rotate left/right).

The next 6 joints (4 through 9) correspond to motions of the left leg, starting from the hip down to the ankle. Joint 4 is a hip rotation about the z axis (e.g. causing the left foot to point its toe to the left), joint 5 is about the x axis (causing the leg to swing out to the robot's left), and joint 6 is about the y axis (causing the left leg to swing forward/backward). Joint 6 is particularly important for walking and squatting. Joint 7 is the left knee joint. Joint 8 is the first ankle joint, causing the foot to rotate about the y axis (e.g. to stand on the robot’s toes like a ballet dancer). Joint 9 rotates the foot about the x axis (a common motion for twisted-ankle injuries in sports).

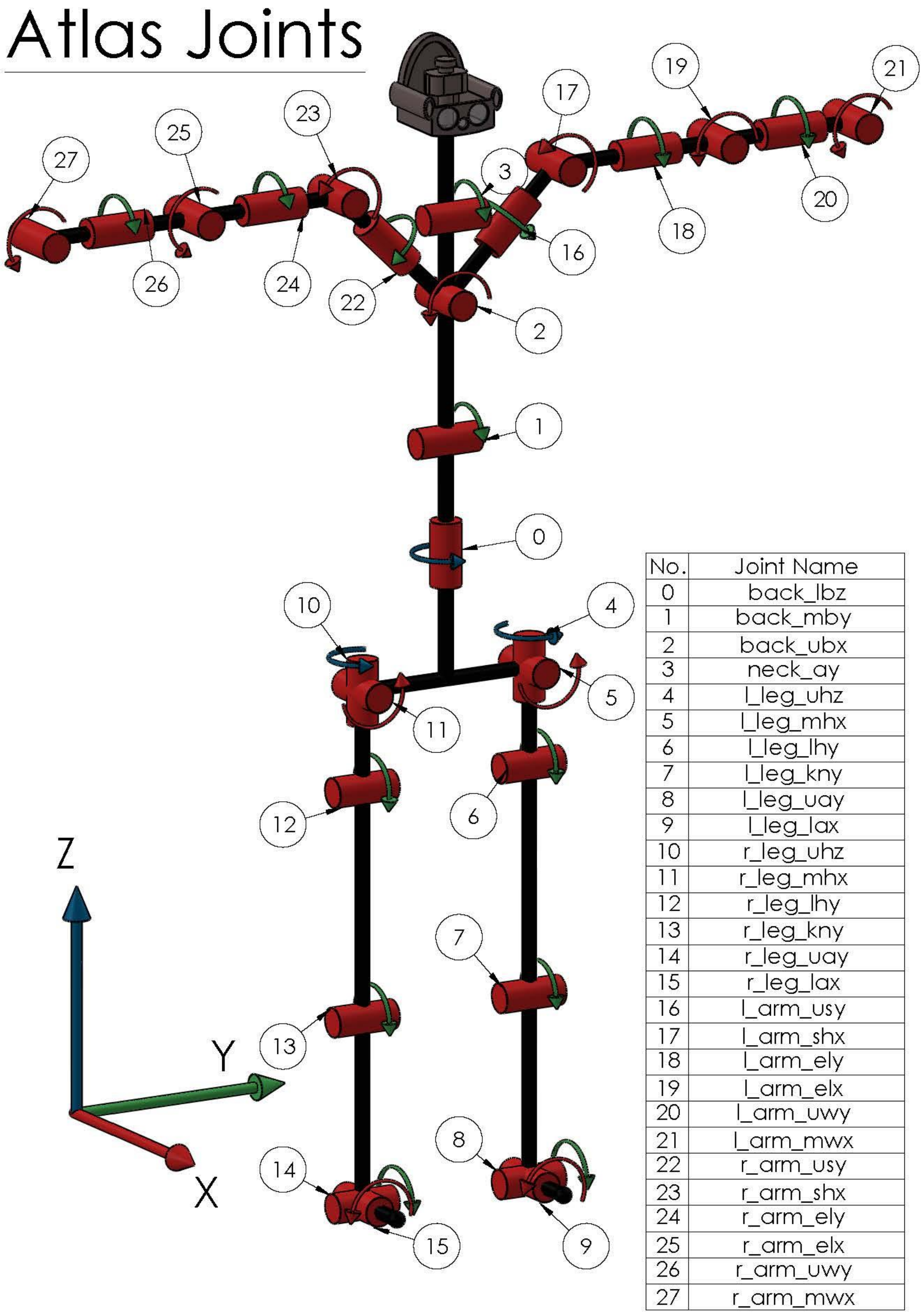
Joints 10 through 15 correspond to joints of the right leg, in exactly the same order and description. Note important sign differences, though. Joints 6, 7 and 8 (and joints 12, 13 and 14 for the right leg) are all rotations about the y axis, and these would be coordinated for a squatting motion, where the joint values 6 and 12 would be identical, 7 and 13 would be identical and 8 and 14 would be identical. However, the remaining joint correspondences (4/10, 5/11, 9/15) should be commanded to equal and opposite angles to achieve mirrored symmetry of motion (e.g., spreading the knees apart, or spreading the feet apart, symmetrically).

Joints 16 through 21 correspond to joints of the left arm, and joints 22 through 27 are the corresponding joints of the right arm. Joints 16/22 are the first shoulder joints, left/right. This axis does not cause a rotation about one of the pelvis-frame axes (from home position). Rather, this joint is tilted at 30 degrees from vertical in the y-z plane. Motion of this joint causes the arm to swing forward/back, but swept at a 30-degree angle. These joint commands must be negated for mirrored symmetry.

Joints 17/23 cause rotation about the x axis (when in the reference pose), i.e. lifting/lowering the arms. The zero angle corresponds to arms straight out at the sides (in an “iron cross” pose). These joint commands must be negated to for mirrored symmetric motions.

Joints 18/24 are rotations of the upper arm about the y axis. Joints 19/25 are elbow rotations. Joints 20/26 are forearm rotations, and joints 21/27 are wrist bend rotations.

The figure below illustrates the Atlas joint definitions.



No.	Joint Name
0	back_lbz
1	back_mby
2	back_ubx
3	neck_ay
4	l_leg_uhz
5	l_leg_mhx
6	l_leg_lhy
7	l_leg_kny
8	l_leg_uay
9	l_leg_lax
10	r_leg_uhz
11	r_leg_mhx
12	r_leg_lhy
13	r_leg_kny
14	r_leg_uay
15	r_leg_lax
16	l_arm_usy
17	l_arm_shx
18	l_arm_ely
19	l_arm_elx
20	l_arm_uwy
21	l_arm_mwx
22	r_arm_usy
23	r_arm_shx
24	r_arm_ely
25	r_arm_elx
26	r_arm_uwy
27	r_arm_mwx

Atlas joint definitions and directions of defined positive rotations

Next steps:
As can be observed from running this simple joint interface, one should never command large changes in angles to the joint interface node—and this should never be run on the physical robot. Instead, to get to a target set of joint angles, a sequence of incremental joint commands (e.g., updated about 10 increments per second) should be sent to Atlas, producing a smooth stream of commands that interpolate from the initial pose to the final pose over some number of seconds. The simple joint interface program presented here needs to be modified to accept streams of commands via subscription to a new ROS topic with an appropriate message type. The simple joint interface should be enhanced to check for joint-range restrictions and other safety considerations appropriate to a high-speed, low-level, joint-space context. Publication of smooth, incrementally updated joint-space trajectories should be performed by a higher-level another node. Such a joint-space interpolator could more appropriately accept coarse user commands, interpolate smooth trajectories to these goals, and publish these incremental goals for execution by the low-level joint interface.