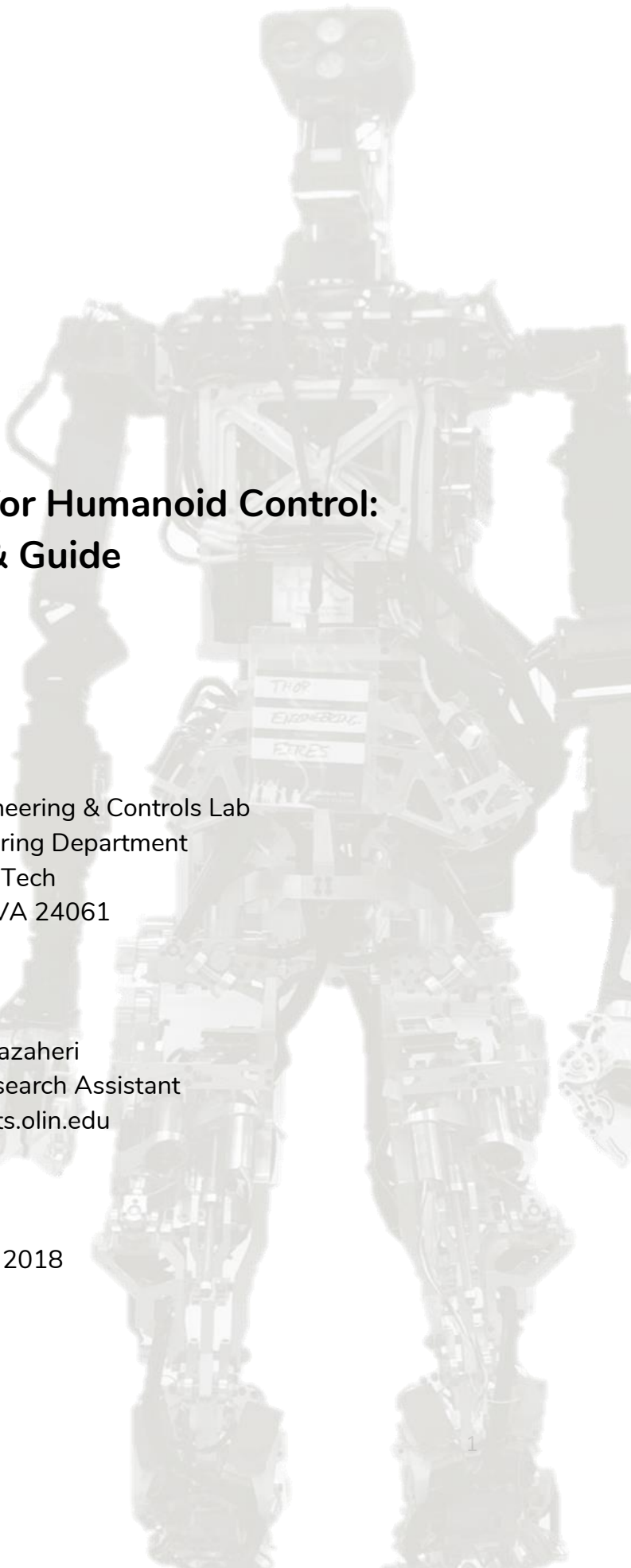


Software Architecture for Humanoid Control: Review & Guide

Terrestrial Robotics Engineering & Controls Lab
Mechanical Engineering Department
Virginia Tech
Blacksburg, VA 24061

Ava Lakmazaheri
Undergraduate Research Assistant
ava@students.olin.edu

Summer 2018

Table of Contents:

Preface

The purpose of this document is to serve as a guide for implementing software control for a humanoid robot. We will begin by grounding this investigation in its real-world implications. A majority of the report will focus on what have been identified as two primary solutions to our control problem. To augment poor documentation in this area, we will discuss both solutions on a conceptual level and provide step-by-step instructions to begin implementing and adapting them. This is not a standalone document, so we will refer to published resources when available. We will end by formulating conclusions on the viability of each software and describe steps to take moving forward.

Overview

In the broad mission to improve quality of life, humanoid robots have a unique niche. Their capability for dexterity and intelligence enables them to perform life-critical tasks such as disaster search-and-rescue and assisting people with severe motor impairment.

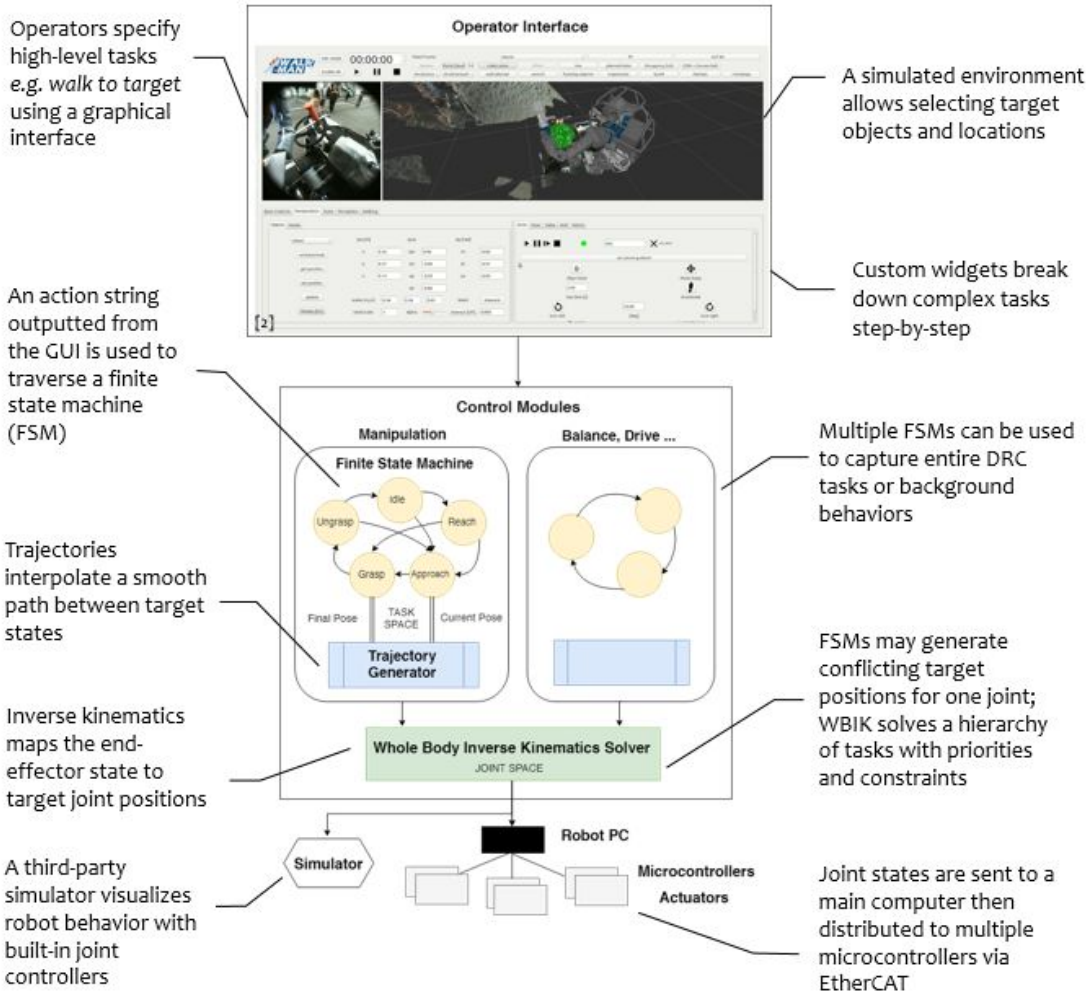
The DARPA Robotics Challenge (DRC) sought to promote innovation in this area by inviting teams to develop a humanoid robot that can perform disaster-response operations (drive a utility vehicle, traverse rubble, enter a building, close a valve to a leaking pipe, etc.) semi-autonomously and in real-time.

In this work, we aim to identify and adapt a software architecture that supports such advanced locomotion and manipulation tasks for the TREC Lab's own humanoid robot.

Given that this robot is still in development, and to make the future process robust to design change, we are interested in a universal system that decouples high-level logic from low-level hardware control. We believe that focusing on such cross-robot frameworks will broadly enable investigators to bypass infrastructure-level programming and invest more in research-level efforts.

Control Architecture

Our first goal was to conceptualize a robust operating logic for a generic humanoid robot. After studying various strategies from the DRC, we synthesized the following system flow that maps abstract tasks to low-level control.



Software Review

In order to realize our software architecture, we began by evaluating existing open-source control applications for functionality, modularity, and simplicity. We heeded special attention to systems with an abstracted hardware layer for easy adaptation to the TREC Lab's robots. We also made sure to prioritize real-time operation.

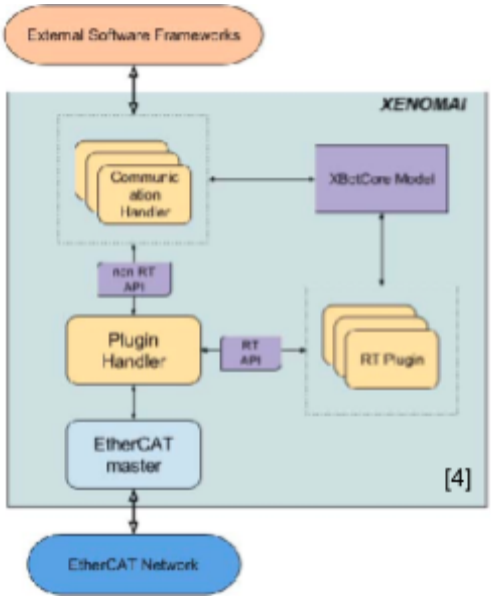
Our primary findings were: 1) XBotCore: a software platform that supports real-time humanoid control and 2) IHMC Open Robotics Software: a set of libraries for out-of-the-box planning and robot operation. Both allow modelling abstract behaviors using finite state machines, multi-tasking via a whole-body inverse kinematics engine, and simplifying complex structures into a universal robot description format.

XBotCore

Overview

XBotCore is a software platform. It does not contain the control algorithms necessary to operate a humanoid, but enables communication between user-developed modules, third-party software, and robot hardware by extending the Robot Operating System (ROS) middleware.

A primary feature of this software is its real-time control capabilities. XBotCore's plugin architecture was designed to ensure a 1 kHz control loop communicating to the robot even with intensive background computations. To learn more about the platform, read: On the Design and Evaluation of XBotCore [4].



Given its high transmission rate, XBotCore was originally designed for EtherCAT based robots. However, later versions of the software support any device driver via a hardware abstraction layer[1].

The development of XBotCore stemmed from the Italian Institute of Technology's DARPA Robotics Challenge entry. To read about this entry at length, including their software architecture and the XBotCore control loop process, see: WALK-MAN: A High-Performance Humanoid Platform for Realistic Environments [7].

We installed XBotCore and created non-real time communication plugins to test the software. These modules enable robot control via either direct joint trajectories or an inverse kinematics solver. A ROS node then visualizes whole-body robot behavior in the third-party simulator Gazebo.

[1] Though this is not our focus, we note the following resource to guide further reading: Towards a Robot Hardware Abstraction Layer (R-Hal) Leveraging the XBot Software Framework [6].

Installation

The source code for XBotCore is available at https://github.com/ADVRHumanoids. In order to install this software, it is **not** sufficient to clone the XBotCore repository. The ADVR Humanoid group has created a superbuild process that installs XBotCore and all of its dependencies needed to run.

Successfully installing and running XBotCore requires a Linux 16.04 OS. A clean OS is preferable, since you do will not face issues with broken packages. To check that the XBotCore build process is working on a clean OS, use Travis CI: a continuous integration tool for building and testing Github software projects: https://travis-ci.org/ADVRHumanoids/advr-superbuild.

To install XBotCore from terminal:

Create an advr-superbuild folder in the home directory.

```
0. $ cd ~
1. $ git clone --depth=50 --branch=master
https://github.com/ADVRHumanoids/advr-superbuild.git
```

Fix C++ compilation.

```
2. $ export CXX=g++
3. $ export CC=gcc
```

Install necessary packages.

```
4. $ sudo apt-get update
5. $ sudo apt-get install g++
6. $ sudo apt-get install libboost-all-dev
7. $ sudo python -m pip install pip==9.0.1
8. $ sudo apt-get install cmake
```

Configure git account.

```
9. $ git config --global user.name "your github name"
10. $ git config --global user.email "your github email"
```

Download ROS and other large packages via basic setup.

```
11. $ cd ~/advr-superbuild
12. $ ./scripts/basic_setup.sh
```

Source robotology setup file. This will allow you to run XBotCore executables later.

```
13. $ chmod 777 ./robotology-setup.bash
14. $ source ./robotology-setup.bash
15. $ echo "source ./robotology-setup.bash" >> ~/.bashrc
```

Check ROS distribution. In this example, we have used ROS Kinetic.

```
16. $ cd /opt/ros/
17. $ ls
```

Source ROS setup.

```
18. $ source /opt/ros/kinetic/setup.bash
```

Install a few more packages.

```
19. $ pip install catkin_pkg
20. $ pip install empy
```

Create directory for build files.

```
21. $ cd ~/advr-superbuild
22. $ mkdir -p build
23. $ cd build
```

Make superbuild.

```
24. $ cmake -DADVR-CORE=ON ..
25. $ cmake -DSUPERBUILD_XBotGUI=OFF ..
26. $ make
```

If the build was successful, you should see populated folders in advr-superbuild/external and advr-superbuild/build. If it was not, you will see empty folders and/or CMake files from the clone.

Running XBotCore

In one terminal, run `roscore`. In a second terminal, run `XBotCore -D`. If the command is not found, change directories to advr-superbuild, `source ./robotology-setup.bash` and try again.

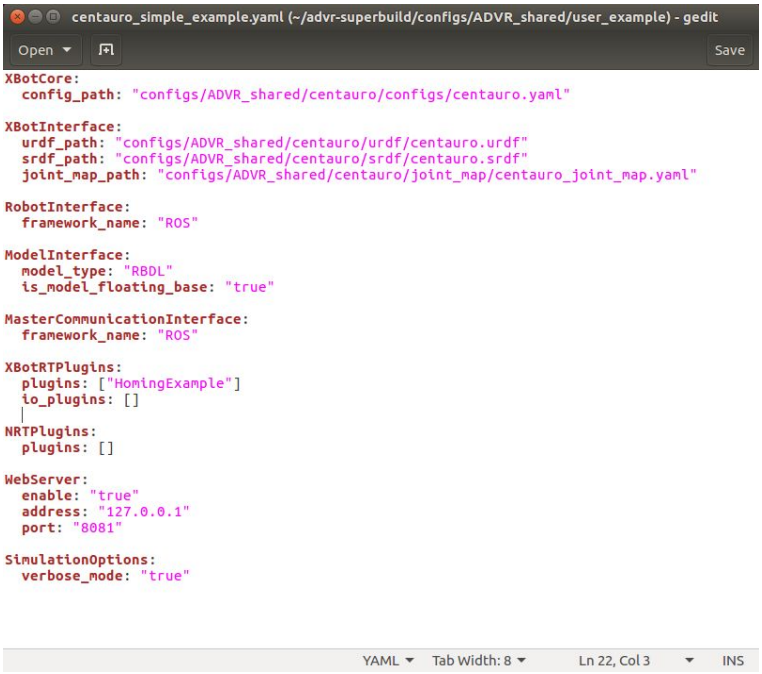
The `XBotCore -D`(ummy) command is running the shared object file XBotCore-1.0.0 found in advr-superbuild/build/install/bin. This is a compiled version of XBotMain.cpp. (https://github.com/ADVRHumanoids/XBotCore/blob/master/src/XBotMain.cpp)

Since XBotCore does not include its own control algorithms, this is all that will run:

```
xbotcore@xbotcore-Inspiron-7373:~$ XBotCore -D
[info] XBotCore using config file /home/xbotcore/advr-superbuild/configs/ADVR_shared/user_example/centauro_simple_example.yaml
[info] XBotCore using SCP log with the following parameters
[info] remote_username : centauro
[info] remote_ip_address : 10.24.4.77
[info] remote_log_folder_path : /home/centauro/CENTAURO_LOG
[success] DUMMY HAL interface found!
[warning] in loadTransmissionPlugins! Config file does NOT contain optional node TransmissionPlugins!
[success] HomingExample RT plugin found!
[success] XBotCommunicationPlugin RT plugin found!
[success] XBotLoggingPlugin RT plugin found!
[success] XBotNRTRef RT plugin found!
[success] Thread XBOT: start looping
[info] Initializing plugin HomingExample
[success] Plugin HomingExample initialized successfully!
[info] Initializing plugin XBotCommunicationPlugin
[warning] Filter ON by default, cutoff frequency is 1 Hz
[success] Plugin XBotCommunicationPlugin initialized successfully!
[info] Initializing plugin XBotLoggingPlugin
[success] Plugin XBotLoggingPlugin initialized successfully!
[info] Initializing plugin XBotNRTRef
[success] Plugin XBotNRTRef initialized successfully!
[success] Thread Loader: start looping
[success] WEB_SERVER INTERFACE found!
[info] XBotCore server running at http://127.0.0.1:8081
[success] Thread XBOT_COMMHANDLER: start looping
```

As previously mentioned, XBotCore uses plugins to read from and write to the robot. The XBotMain function creates a Plugin Handler, responsible for queuing and running RT Plugins, as well as a Communication Handler, which performs the same job for NRT Plugins.

Both Handlers look for plugins that are listed in the specified YAML configuration file. Before running `XBotCore -D`, you can check which configuration is being used with `get_xbot_config` or set it with `set_xbot_config PATH/TO/CONFIG.yaml`. The figure below shows the contents of the centauro_simple_example configuration file. The only plugin being loaded is HomingExample, a testing file compiled by the superbuild. This is why the `XBotCore -D` prints `Initializing plugin HomingExample`.



In order to use XBotCore for its intended control purposes, we need to create our own plugins.

Understanding Plugins

As described in [WALK-MAN: A High-Performance Humanoid Platform for Realistic Environments](#) [7], we can interact with plugins via three interfaces.

1) *Switch interface* ..... start/stop plugin
2) *Command interface* ..... send robot command to plugin
3) *Status interface* ..... receive robot status from plugin

In a complete project (as described in the Control Architecture section), the plugins (previously called control modules) are manipulated by the GUI. For example, the output of a button press on the GUI may call the *switch interface*, which in turn starts the corresponding plugin. Since we are testing plugins in isolation, we can mimic this behavior from the terminal.

XBotCore's plugins operate through ROS services. Thus, to call a plugin interface, we simply use rosservice calls like these:

```
rosservice call /PLUGIN_NAME_switch 1
rosservice call /PLUGIN_NAME_cmd "turn valve"
```

We can also print all available plugins with `rosservice list`.

Creating a Plugin

Install an IDE for CMake called KDevelop.

1. `$ wget -O KDevelop.AppImage https://download.kde.org/stable/kdevelop/5.2.3/bin/linux/KDevelop-5.2.3-x86_64.AppImage`
2. `$ chmod +x KDevelop.AppImage`

Generate an XBot Plugin.

3. `$ cd advr-superbuild/external`
4. `$ source ./robotology-setup.bash`
5. `$ generate_XBot_Plugin.sh PLUGIN_NAME`

Write plugin.

6. `$ ./KDevelop.AppImage`
7. `Open / Import Project` and navigate to `advr-superbuild/external/PLUGIN_NAME`
8. Edit plugin code

Build plugin

9. Change build directory to `PATH/TO/advr-superbuild/build/PLUGIN_NAME`
10. Right-click on project. `Build`.

Install plugin in KDevelop

11. Right-click on project. `Open Configuration…`
12. `CMake >` Set `CMAKE_INSTALL_PREFIX` to `PATH/TO/advr-superbuild/build/install`
13. Right-click on project. `Install`.

Add plugin to configuration file.

14. Open YAML configuration file
15. Add `PLUGIN_NAME` to `RT` or `NRT Plugins`

To walk-through the structure of a plugin, we will consider the contents of a dummy `myPlugin`.

`init_control_plugin`: function called when the Plugin Handler or Communication Handler initializes this plugin. Add a print statement here `std::cout << "myPlugin init done!" << std::endl;`

`on_start`: function called when the switch interface is set to "start" (`rosservice call /myPlugin_switch 1`). Here, print the initial position reference `std::cout << "Position reference : " << _q0 << std::endl;`

`on_stop`: function called when the switch interface is set to "stop" (`rosservice call /myPlugin_switch 0`).

`control_loop`: called on every control loop once the plugin has been started. Here we can parse command strings sent via the command interface. For example,

```
if(current_command.str() = "move"){
 std::cout << "Moving..." << std::endl;
}
if(current_command.str() = "turn"){
 std::cout << "Turning..." << std::endl;
}
```

If myPlugin has been set up with the steps described above and added to centauro_simple_example.yaml, then running `XBotCore -D` will call a Plugin Handler to initialize it. Here we see the print statement from **`init_control_plugin`**.

```
xbotcore@xbotcore-Inspiron-7373:~$ XBotCore -D
[info] XBotCore using config file /home/xbotcore/advr-superbuild/configs/ADVR_shared/user_example/centauro_simple_example.yaml
[info] XBotCore using SCP log with the following parameters
[info] remote_username : centauro
[info] remote_ip_address : 10.24.4.77
[info] remote_log_folder_path : /home/centauro/CENTAURO_LOG
[success] DUMMY HAL interface found!
[warning] in loadTransmissionPlugins! Config file does NOT contain optional node TransmissionPlugins!
[success] HomingExample RT plugin found!
[success] myPlugin RT plugin found!
[success] XBotCommunicationPlugin RT plugin found!
[success] XBotLoggingPlugin RT plugin found!
[success] XBotNRTRef RT plugin found!
[success] Thread XBOT: start looping
[info] Initializing plugin HomingExample
[success] Plugin HomingExample initialized successfully!
[info] Initializing plugin myPlugin
myPlugin init done!
[success] Plugin myPlugin initialized successfully!
[info] Initializing plugin XBotCommunicationPlugin
[warning] Filter ON by default, cutoff frequency is 1 Hz
[success] Plugin XBotCommunicationPlugin initialized successfully!
[info] Initializing plugin XBotLoggingPlugin
[success] Plugin XBotLoggingPlugin initialized successfully!
[info] Initializing plugin XBotNRTRef
[success] Plugin XBotNRTRef initialized successfully!
[success] Thread Loader: start looping
[success] WEB_SERVER INTERFACE found!
[info] XBotCore server running at http://127.0.0.1:8081
[success] Thread XBOT_COMMHANDLER: start looping
```

Now if we send commands via the switch and command interface, we can check that the plugin is running as expected from our print statements.

```
[success] WEB_SERVER INTERFACE found!
[info] XBotCore server running at http://127.0.0.1:8081
[success] Thread XBOT_COMMHANDLER: start looping
[info] Starting plugin : myPlugin
Position reference : 0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
moving command
turning command
```

```
xbotcore@xbotcore-Inspiron-7373:~$ source advr-superbuild/robotology-setu
xbotcore@xbotcore-Inspiron-7373:~$ rosservice list
/CommHandler/get_loggers
/CommHandler/set_logger_level
/HomingExample_cmd
/HomingExample_restart
/HomingExample_status
/HomingExample_switch
/MasterCommunicationInterface_switch
/XBotCommunicationPlugin_cmd
/XBotCommunicationPlugin_restart
/XBotCommunicationPlugin_status
/XBotCommunicationPlugin_switch
/XBotLoggingPlugin_restart
/XBotNRTRef_restart
/centauro/position_controller/get_joint_names
/myPlugin_cmd
/myPlugin_restart
/myPlugin_status
/myPlugin_switch
/rosout/get_loggers
/rosout/set_logger_level
/test_service
xbotcore@xbotcore-Inspiron-7373:~$ rosservice call /myPlugin_switch 1
success: True
message: ''
xbotcore@xbotcore-Inspiron-7373:~$ rosservice call /myPlugin_cmd move
success: True
xbotcore@xbotcore-Inspiron-7373:~$ rosservice call /myPlugin_cmd turn
success: True
xbotcore@xbotcore-Inspiron-7373:~$
```

Introduction to ROS

As mentioned earlier, XBotCore extends the Robot Operating System (ROS)[2]. This setup provides all ROS capabilities, but packages real-time control operations into a single node called XBotCore. In the following section, we will focus on non-real time operations (controlling a robot in simulation), and thus refer to ROS in lieu of XBotCore.

In case the reader is not familiar with ROS, here are some important notes. ROS has a graph-based structure, where operations are designated to **nodes**. All nodes are registered to a master `roscore` and thus can communicate with each other in a peer-to-peer network. Modes of this communication include 1) **topics**, to which nodes may publish or subscribe **messages** asynchronously and 2) **services**, in which nodes synchronously call functions of other nodes.

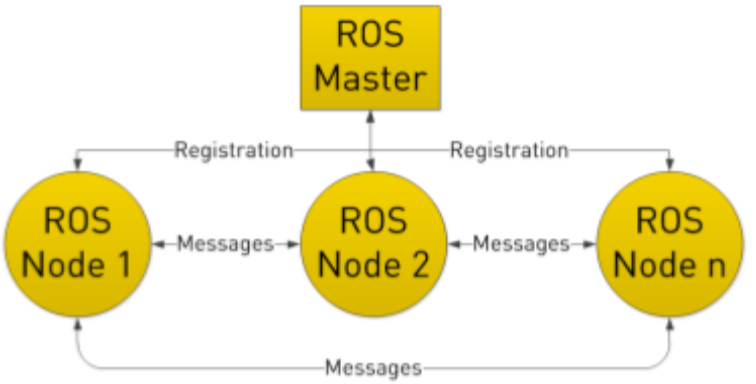


Image taken from http://www.clearpathrobotics.com

ROS is substantially well-documented. To learn and practice with more information, I recommend the following books.

1) Programming Robots with ROS by Brian Gerkey, William Smart, Morgan Quigley
2) A Systematic Approach to Learning Robot Programming with ROS by Wyatt S. Newman
3) ROS Basics in 5 Days by Ricardo Téllez, Alberto Ezquerro, and Miguel Ángel Rodríguez

[2]For a partially complete description of how XBotCore extends standard ROS messages, see https://github.com/ADVRHumanoids/XBotCore/wiki/XBotCore-ROS-integration

Creating a Controllable Robot in ROS

This tutorial will walk through step-by-step how to visualize and control a simulated humanoid. To access the working code or reference certain files, see https://github.com/alakmazaheri/sim-robot-control

PART 1: Setting Up a Workspace
To facilitate our work in ROS and Python, we will need to set up a catkin workspace.

Create a workspace [name: urdf_ws]

```
1. $ mkdir -p ~/urdf_ws/src
2. $ cd ~/urdf_ws/src
3. $ catkin_init_workspace
4. $ cd ..
5. $ catkin_make
6. $ source devel/setup.bash
7. $ echo "source ~/urdf_ws/devel/setup.bash" >> ~/.bashrc
```

Create a package [name: poppy]

```
8. $ cd ~/urdf_ws/src
9. $ catkin_create_pkg poppy rospy
```

PART 2: Create a Robot Model
ROS developers use the XML-based Unified Robot Description Format (URDF) for representing robot models. In this example, we will be modelling the open-source 3D-printable humanoid Poppy. Since she is open-source, she had an existing URDF file which we have modified for our purposes.

Create a URDF folder for Poppy to live in:

```
1. $ cd ~/urdf_ws/src/poppy
2. $ mkdir urdf
```

In this folder, place the modified URDF `poppy.urdf.xacro` as found on Github. URDF files are long and not pleasant to read or create in full, so here is a quick peek into what is going on inside.

LINKS AND JOINTS:
A robot can be defined as a set links chained together by joints. For each link, we define visual, collision, and inertial properties in the URDF.

For each joint, we define its parent and child link. In this example, the right upper arm link is defined and connected to the right forearm via the elbow.

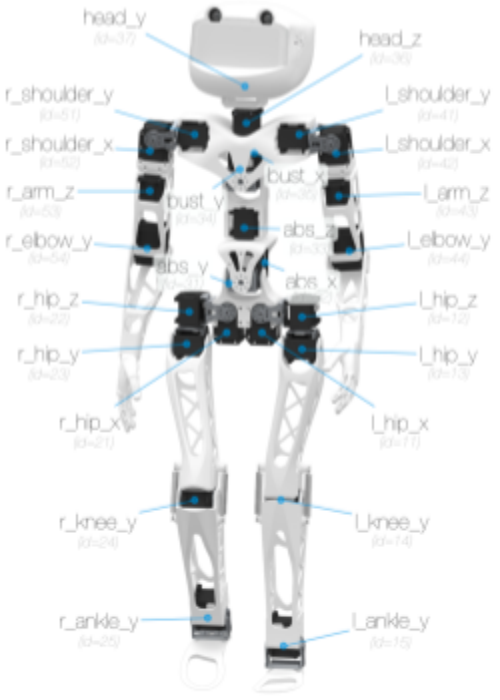


Image taken from www.poppy-project.org

```
<link name="r_upper_arm">
        <visual>
                <origin xyz="0 0 0" rpy="0 0 0"></origin>
                <geometry>
                        <mesh filename="package://urdf_tutorial/meshes/poppy_meshes/r_upper_arm_visual.STL"></mesh>
                </geometry>
                <material name="">
                        <color rgba="0.9 0.9 0.9 1.0"></color>
                </material>
        </visual>
        <collision>
                <origin xyz="0 0 0" rpy="0 0 0"></origin>
                <geometry>
                        <mesh filename="package://urdf_tutorial/meshes/poppy_meshes/r_upper_arm_respondable.STL"></mesh>
                </geometry>
        </collision>
        <inertial>
                <origin xyz="0.01 0.01 0.01" rpy="0 0 0"></origin>
                <mass value="0.008"></mass>
                <inertia ixx="0.3" ixy="0.004" ixz="0.002" iyy="0.3" iyz="0.001" izz="0.2"></inertia>
        </inertial>
        <mass>0.00838</mass>
</link>
<joint name="r_elbow_y" type="revolute">
        <origin xyz="0 0.11175 -0.01" rpy="0 0 0"></origin>
        <parent link="r_upper_arm"></parent>
        <child link="r_forearm"></child>
        <axis xyz="-1 0 0"></axis>
        <limit effort="3.1" lower="-0.0174532925199" upper="2.58308729295" velocity="7.0"></limit>
</joint>
```

Note that Poppy's visual and collision geometries come from CAD meshes (.STL files). For ease of use, we have reproduced these files on our Github: urdf_src/poppy/urdf/poppy_meshes. We have to place these files where the system can find packages, in opt/ros/kinetic/share.

We have encountered an issue where placing the meshes directly into this folder results in them not being loaded. For a quick fix, run `sudo apt-get install ros-kinetic-urdf-tutorial`. This will create a meshes package in opt/ros/kinetic/share/urdf_tutorial. Copy and paste poppy_meshes into this folder.

TRANSMISSION:
To make the robot controllable, we must also create a transmission component for each defined joint. Note that this section was not in the original Poppy URDF and was added to control the robot in simulation.

```
<transmission name="tran54">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="r_elbow_y">
                <hardwareInterface>PositionJointInterface</hardwareInterface>
        </joint>
        <actuator name="motor54">
                <hardwareInterface>PositionJointInterface</hardwareInterface>
                <mechanicalReduction>1</mechanicalReduction>
        </actuator>
</transmission>
```

We will be using the Gazebo simulator to visualize the robot's behavior. Thus, as final step in the URDF, we must load the ros_control plugin for Gazebo

```
<!-- Gazebo plugin for ROS Control -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so"/>
</gazebo>
```

PART 3: Adding Controllers

How do we make all these joints move? Each joint (or group of joints) will be manipulated via a ROS controller of our choice. Note that different controllers expect different types of commands.

Let's create a folder where we can keep our controller configurations (`mkdir ~/urdf_ws/src/poppy/config`)

When dealing with arms and legs, we generally envision long, fluid motions. For these joint groups, we will use trajectory controllers. For the torso and head, position controllers are fine.

A YAML configuration file (`poppy_controllers.yaml`) will specify these controllers and their associated joints.

```yaml
left_arm_controller:
 type: "position_controllers/JointTrajectoryController"
 joints:
   - l_shoulder_y
   - l_shoulder_x
   - l_arm_z
   - l_elbow_y

right_arm_controller:
 type: "position_controllers/JointTrajectoryController"
 joints:
   - r_shoulder_y
   - r_shoulder_x
   - r_arm_z
   - r_elbow_y

left_leg_controller:
 type: "position_controllers/JointTrajectoryController"
 joints:
   - l_hip_x
   - l_hip_z
   - l_hip_y
   - l_knee_y
   - l_ankle_y

right_leg_controller:
 type: "position_controllers/JointTrajectoryController"
 joints:
   - r_hip_x
   - r_hip_z
   - r_hip_y
   - r_knee_y
   - r_ankle_y
```

```yaml
head_controller:
 type: "position_controllers/JointGroupPositionController"
 joints:
   - head_z
   - head_y

torso_controller:
 type: "position_controllers/JointGroupPositionController"
 joints:
   - bust_y
   - bust_x
   - abs_y
   - abs_x
   - abs_z

joint_state_controller:
 type: "joint_state_controller/JointStateController"
 publish_rate: 50
```

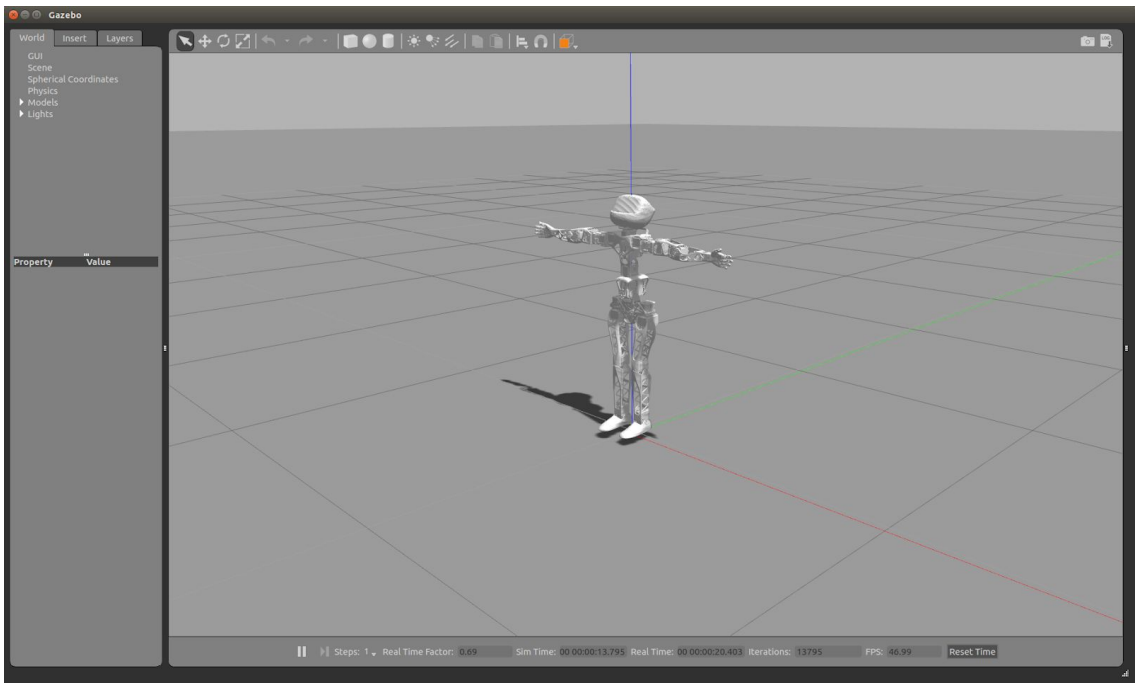
PART 4: Launching

Create a folder for your launch files: `mkdir ~/urdf_ws/src/poppy/launch.`

`main.launch` is what you will be calling from terminal. It begins by calling a secondary launch file `gazebo.launch`, which creates a ROS node that runs the Gazebo simulator and loads a robot model from your URDF. `<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"… >` Once this is done, `main.launch` spawns the controllers needed to manipulate that robot.

```
<rosparam command="load" file="$(find poppy)/config/poppy_controllers.yaml"/>

<node name="poppy_controller_spawner" pkg="controller_manager" type="spawner"
      args="left_arm_controller
      right_arm_controller
      head_controller
      torso_controller
      left_leg_controller
      right_leg_controller
      joint_state_controller
      --shutdown-timeout 3"/>
```

To see this in action, run **`roslaunch poppy main.launch`**[3]. An empty Gazebo world will open with Poppy in a default position. To facilitate testing, we have fixed Poppy's pelvis to the world frame in the URDF. To make her free to move (and fall over), simply comment out the world link and world joint in the URDF.



[3] If you get an error that `main.launch` is neither a launch file in package `poppy` nor is `poppy` a launch file name, try `source devel/setup.bash` again.

PART 5: Controlling

View the controllers you created from terminal with `rostopic list`.

```
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gazebo_gui/parameter_descriptions
/gazebo_gui/parameter_updates
/head_controller/command
/joint_states
/left_arm_controller/command
/left_arm_controller/follow_joint_trajectory/cancel
/left_arm_controller/follow_joint_trajectory/feedback
/left_arm_controller/follow_joint_trajectory/goal
/left_arm_controller/follow_joint_trajectory/result
/left_arm_controller/follow_joint_trajectory/status
/left_arm_controller/state
/left_leg_controller/command
/left_leg_controller/follow_joint_trajectory/cancel
/left_leg_controller/follow_joint_trajectory/feedback
/left_leg_controller/follow_joint_trajectory/goal
/left_leg_controller/follow_joint_trajectory/result
/left_leg_controller/follow_joint_trajectory/status
/left_leg_controller/state
/right_arm_controller/command
/right_arm_controller/follow_joint_trajectory/cancel
/right_arm_controller/follow_joint_trajectory/feedback
/right_arm_controller/follow_joint_trajectory/goal
/right_arm_controller/follow_joint_trajectory/result
/right_arm_controller/follow_joint_trajectory/status
/right_arm_controller/state
/right_leg_controller/command
/right_leg_controller/follow_joint_trajectory/cancel
/right_leg_controller/follow_joint_trajectory/feedback
/right_leg_controller/follow_joint_trajectory/goal
/right_leg_controller/follow_joint_trajectory/result
/right_leg_controller/follow_joint_trajectory/status
/right_leg_controller/state
/rosout
/rosout_agg
/tf
/tf_static
/torso_controller/command
```

CONTROLLING FROM TERMINAL

Publish to each controller topic according to its unique message type. You can check a message type with `rostopic type /TOPIC_NAME`

Here is an example trajectory message for the left arm.

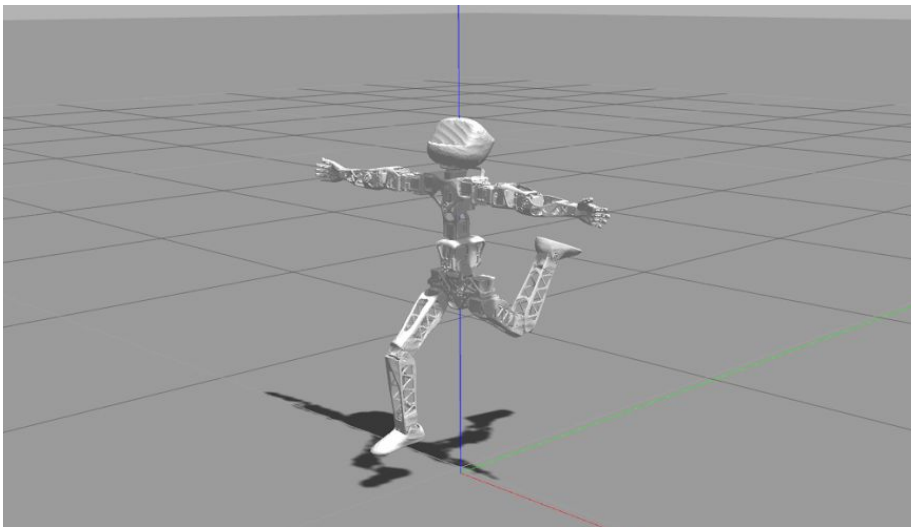
```
rostopic pub /left_arm_controller/command trajectory_msgs/JointTrajectory
'{joint_names: ["l_shoulder_y", "l_shoulder_x", "l_arm_z", "l_elbow_y"],
points: [{positions: [1.5, -1.5, 1.75, 1.75], time_from_start: [1.5, 0.0]},
{positions: [-1, 1.5, 1.2, -1.5], time_from_start: [2.0, 1.0]}]}' -1
```

CONTROLLING FROM SCRIPT

In an executable Python script, initialize a ROS node and publish to your desired topic.

```
rospy.init_node('[NODE NAME]')
pub = rospy.Publisher('[TOPIC NAME]', [MESSAGE TYPE, queue_size=1)
pub.publish(MESSAGE)
```

For an example of this, see run.py in the urdf_ws/src/poppy/scripts folder. Poppy will move her legs in a continuous trajectory to feign running.



In addition to this visual confirmation in Gazebo, we can check that Poppy's joint positions are changing by subscribing to the ROS topic joint_states. The command `rostopic echo /joint_states` outputs a continuous stream of joint positions, velocities, and efforts.

Since Poppy is a bipedal robot with many degrees of freedom, we don't have as many simple scripts to demonstrate multitasking capabilities. To see a mobile robot that can perform navigation in conjunction with manipulation, refer to our R2D2 package. It was created from the same general process as Poppy, adapted from the ROS tutorial Building a Visual Robot Model with URDF from Scratch.

```
roslaunch r2d2 main.launch
cd urdf_ws/src/r2d2/scripts
./drive_gripper or ./arm_traj.py
```

A Note on Trajectories:
We previously mentioned two methods of ROS communication: topics and services. Generally speaking, if you send a command via a topic or service, it must be completed before you can execute the next command. To work around this, ROS implements a third communication process called **actions**. These are time-extended and goal-oriented tasks; you can specify an action and it will run until it completes its goal in parallel with other processes.

Since trajectories can be time-extending, the trajectory controller implements both a topic interface and an action interface. ROS topics with the extension `/follow_joint_trajectory/[cancel] [feedback] [goal] [results] [status]` are manifestations of the action interface.

Knowing that this will prove useful for navigation and manipulation tasks, we attempted to implement a simple action for an arm trajectory. This still requires more work to make operational, but the shell for its client and server based interaction can also be found in the scripts folder.

Higher-Level Control: IK Engine

The previous section described low-level control in ROS. Building higher-level processes requires significantly more coding and development. In this final ROS section, we will show a simple type of higher-level control that is already provided by the ROS MoveIt! Software: inverse kinematics (IK).

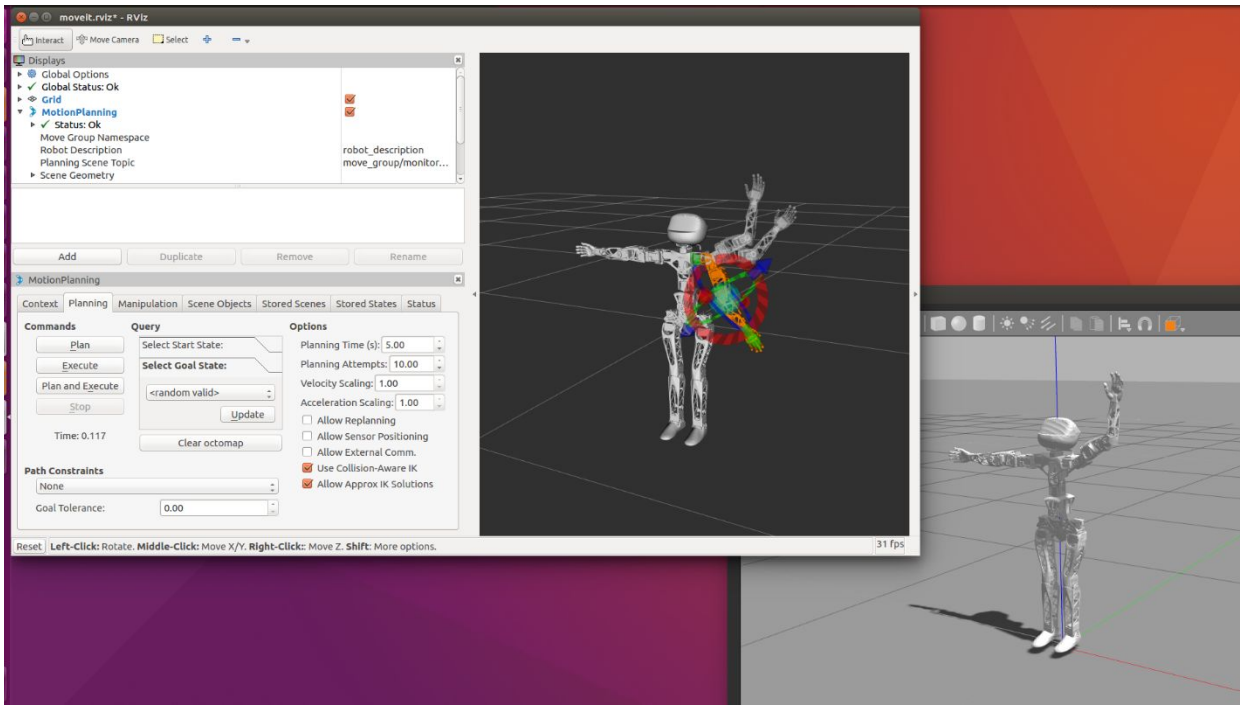
Install MoveIt! for ROS Kinetic

```
1. $ sudo apt-get install ros-kinetic-moveit
2. $ source /opt/ros/kinetic/setup.bash
```

Now, follow the Configure MoveIt steps found in the Programming Robots with ROS book (a full PDF is available online). Adapt the Cougarbot name and joint references for your Poppy counterparts.

To execute, run each command in a separate terminal:

```
roslaunch poppy main.launch
roslaunch poppy_moveit_config move_group.launch
roslaunch poppy_movet_config moveit_rviz.launch config:=True
```



Congratulations! You now have a robot model that is controllable through both direct joint trajectories and end effector target locations.[4]

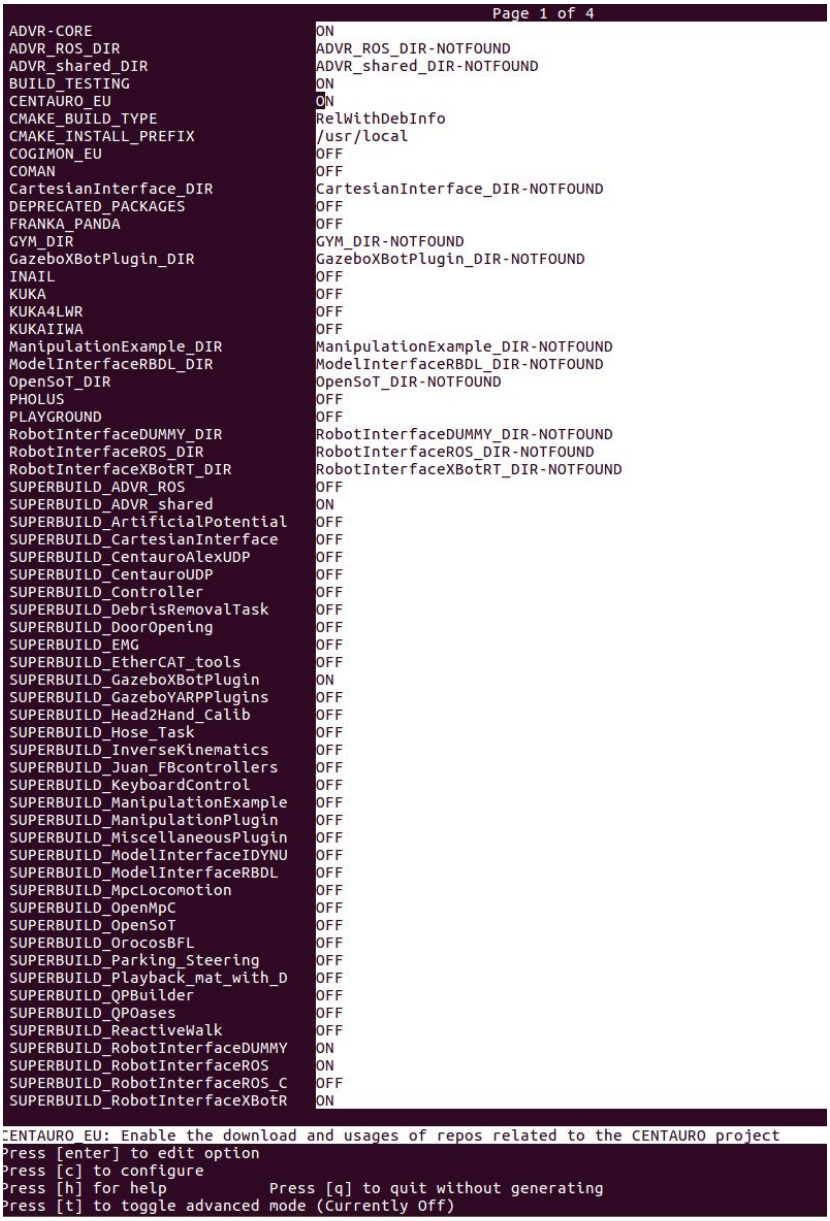
[4] It is possible to import the MoveIt IK operations into a Python script (see Chapter 11 in the Programming Robots with ROS book), but you run the frequent risk of generating impossible-to-reach trajectories without the GUI, in which case the script will simply not execute. We have yet to implement this successfully for the Poppy Humanoid. The NASA R2 robot that is used in the book has since been removed from its open source location and thus also leads to a host of challenges in following their listed instructions.

Projects Using XBotCore

We now have some experience with creating URDF robot descriptions and ROS controllers. Both will be extremely useful tools for building on top of XBotCore in the future.

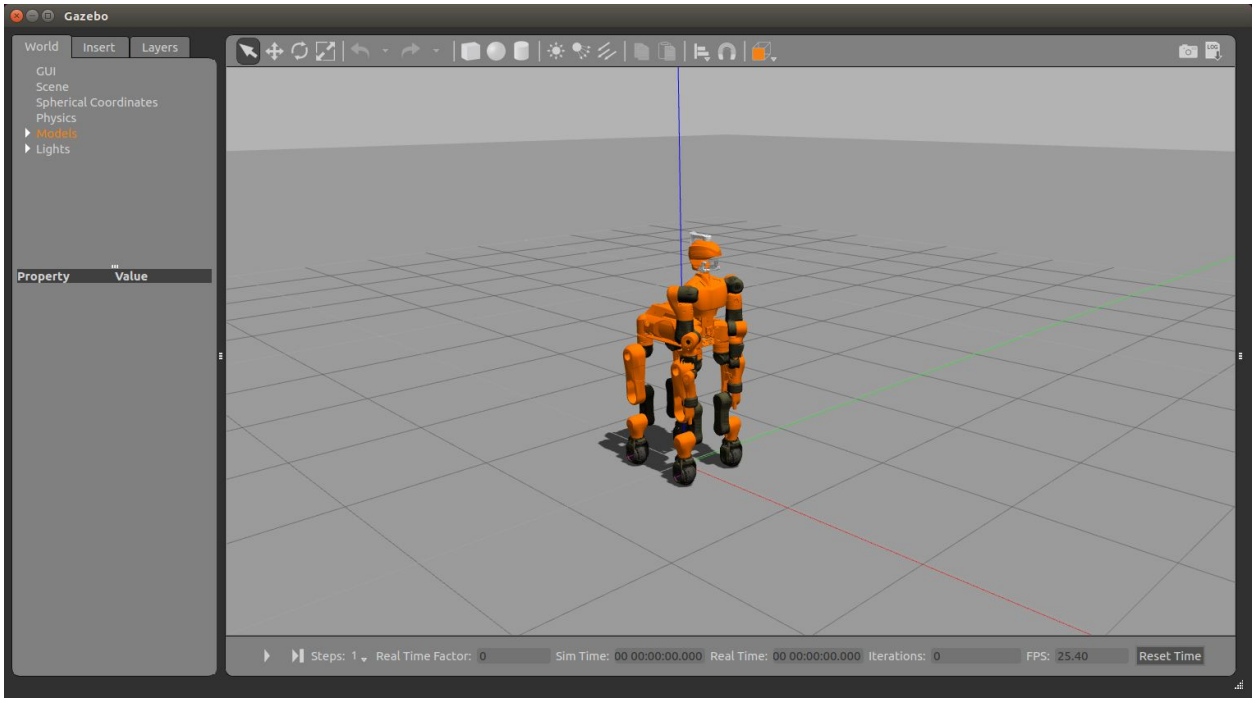
Are there any open-source projects that have built on top of XBotCore already? Perhaps. The XBot superbuild does not only contain the XBotCore software platform. In fact, it appears to have a huge database of control algorithms, robot models, user interfaces, and walking/manipulation examples out of other IIT projects.

When we installed XBotCore, we turned off the CMake GUI and only installed the bare minimum to run XBotCore. Now, navigate to the advr-superbuild/build folder and run `ccmake ..` to see the GUI in full.



You can scroll through all the available projects and toggle them ON/OFF with the enter key. Press [c] to configure sub-dependencies until the option to press [g] to generate appears. This will exit the GUI. Now run `make` from terminal to build all of the projects you have turned on.

We have attempted to compile the WALK-MAN project, its predecessor BIG-MAN, and the Centauro robot for testing. Unfortunately, every additional project that is turned on comes with a host of build errors to work through. At this time, the documentation does not exist to work through these errors. Thus, while XBot-related projects may prove useful, their pursuit will require a substantial time investment.



IHMC Open Robotics Software

Overview

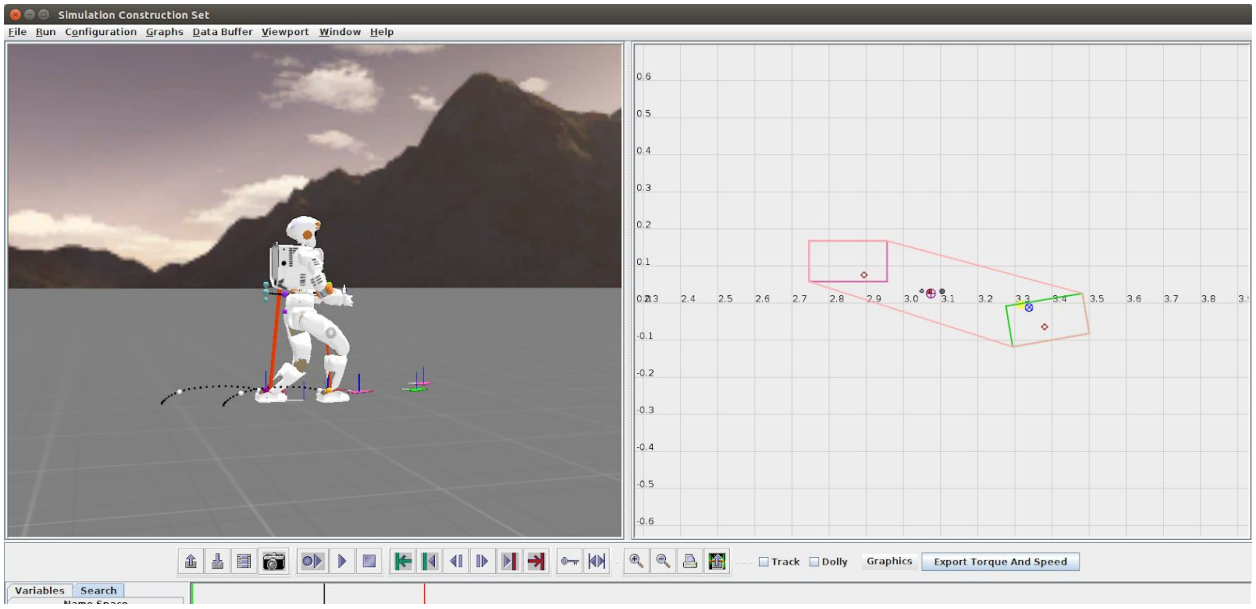
As an open-source initiative, the Institute for Human and Machine Cognition (IHMC) has published the near-complete control code[5] from their DRC entry. This robotics software contains legged locomotion algorithms for walking and an optimized momentum-based controller core for managing multiple tasks.

Our goal is to understand the workings of this software so that we can adopt and eventually adapt it. At the present time, its source code is disorganized and documentation lacking. Existing resources clue us in on how to run specific demonstrations, such as how to make NASA's Valkyrie robot walk. Beyond this, the software is highly impenetrable. In this section, we report on our efforts to:

1) Gain a basic understanding of IHMC's control architecture.
2) Familiarize ourselves with the organization of the code. We want to map our theoretical understanding into the exact lines of code that implement it, so that we can swap out algorithms.
3) Identify low-level communication to a simulator or physical robot.
4) Add and execute high-level behaviors (e.g. turn valve, open door, walk to location).

Quick Start Demonstration

For an initial demonstration of its abilities, IHMC provides thorough instructions for setting up a walking simulation of the Valkyrie robot. Follow the Quick Start steps from ihmcrobotics.github.io, but make sure to download the newest version of Eclipse for Java Developers (Photon R instead of Mars 2).



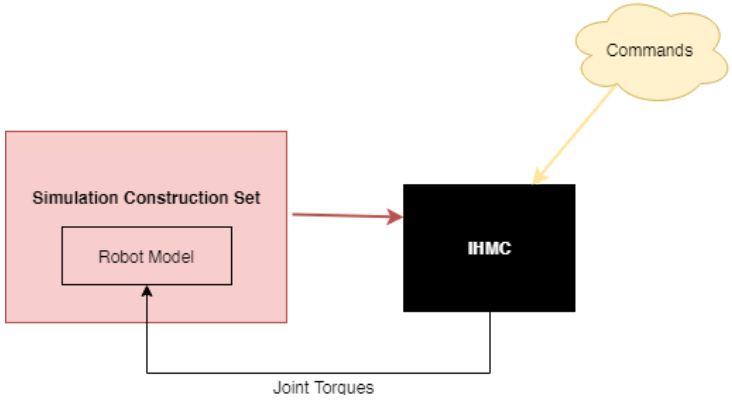
[5] It excludes the operator interface and low level hardware drivers to interface with Boston Dynamics' Atlas robot.

How it Works

PART 1: The Basics

We can represent any robot in IHMC using the Robot class. This class contains all of the joint and link information needed to fully model the robot in any configuration at any time. We will alter the state of the robot via **Robot Controllers**. What constitutes a Robot Controller is very loosely defined; we only require that it contains an `initialize` and a **real-time `doControl` function**. IHMC's in-house simulator, the Simulation Construction Set (SCS), is responsible for both visualizing the current robot state and calling any active Robot Controllers to execute their `doControl` function in real-time.

For now, think of IHMC's control schema as a black-boxed Robot Controller. When we run the SCS GUI and press simulate, it asks IHMC to perform its `doControl` function. In turn, IHMC looks for user-specified commands and maps them into desired joint movements. SCS takes these values and updates the robot model accordingly. The process continues to loop until the user stops the program.



PART 2: The Control

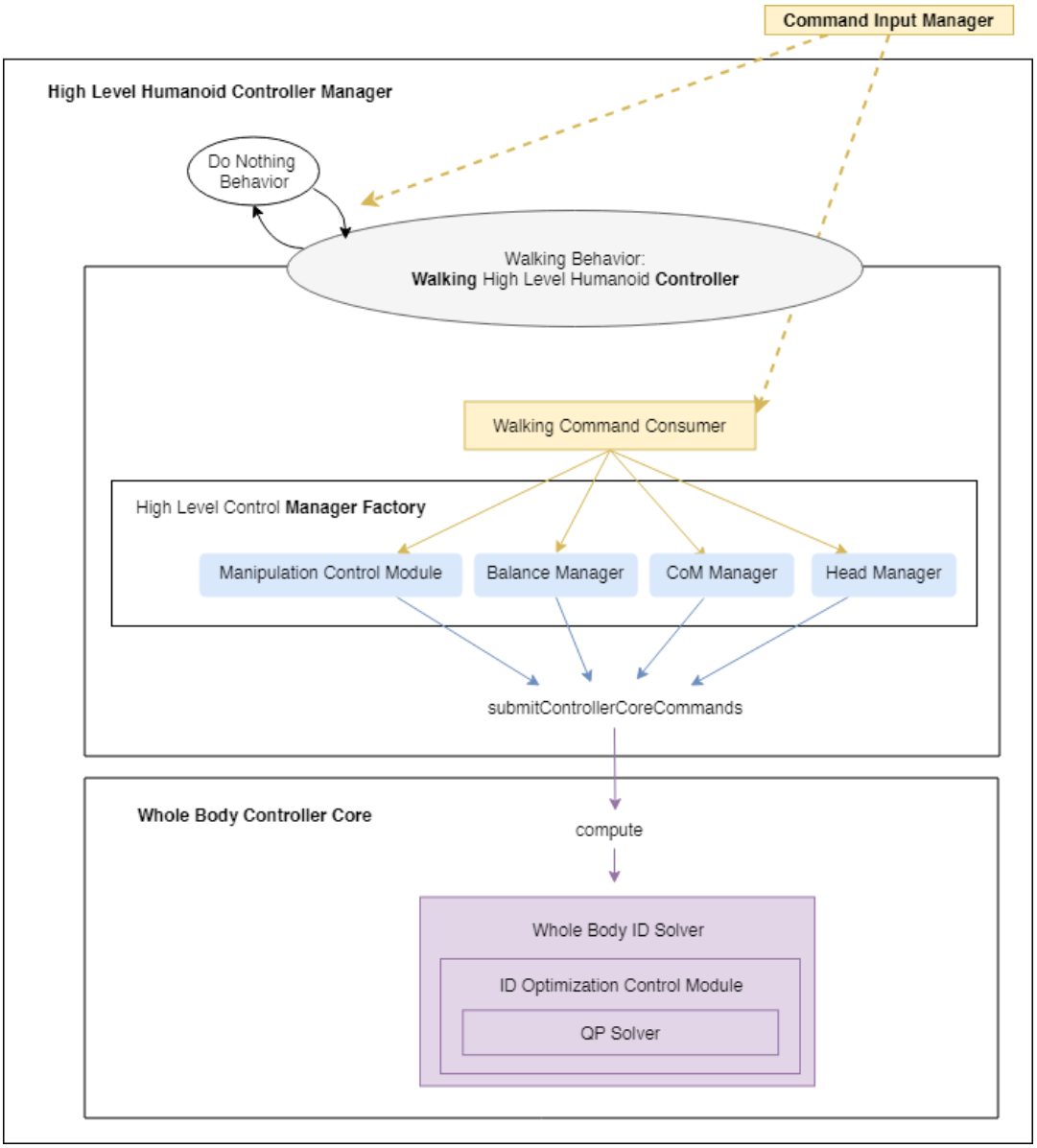
What is happening in the IHMC black box? How does it handle complex, whole-body tasks?

The Robot Controller we abstracted IHMC to is actually called a **High Level Humanoid Controller Manager**[6]. It contains a very high-level state machine with only two options: "Do Nothing" or "Walking". This terminology is misleading, as the walking state actually manages all aspects of the body. It can, for example, manipulate the robot's arms without moving its legs. We will almost always be in the "Walking" state.

The first task of the Controller Manager's **`doControl`** function is to confirm the current high-level state. It checks for a user command requesting a transition, then runs the `doAction` function of the current state. Given the predominance of the "Walking" state, we will only examine this behavior.

[6] You can find the source code for this class in your Gradle Project under
Project and External Dependencies > CommonWalkingControlModules-0.9.0.jar >
us.ihmc.commonWalkingControlModules.highLevelHumanoidControl.
From now own, we will refer to the location of important source code using the us.ihmc package notation.

The walking behavior is implemented by a Walking High Level Humanoid Controller[7], often referred to as a **Walking Controller**. In its **`doAction`** function, the Walking Controller reads in a variety of user-specified commands and distributes them to different **managers**. Managers exist to compartmentalize the control of various robot subsystems. For example, Head and Neck Trajectory Commands[8] are sent to the Head Manager, while Arm and Hand Trajectory Commands[8] are sent to the Manipulation Manager.



Once all managers have internalized their commands, `doAction` calls `updateManagers`. This function makes each manager internally map their given commands into computable motion objectives (desired accelerations and momentum values) that are compatible with the IHMC Controller Core.

[7] us.ihmc.commonWalkingControlModules.highLevelHumanoidControl.highLevelStates
[8] us.ihmc.humanoidRobotics.communication.controllerAPI.command (IHMCHumanoidRobotics-0.9.0.jar)

The Walking Controller's `doAction` function has now completed, so we return to the High Level Humanoid Controller Manager's `doControl`.

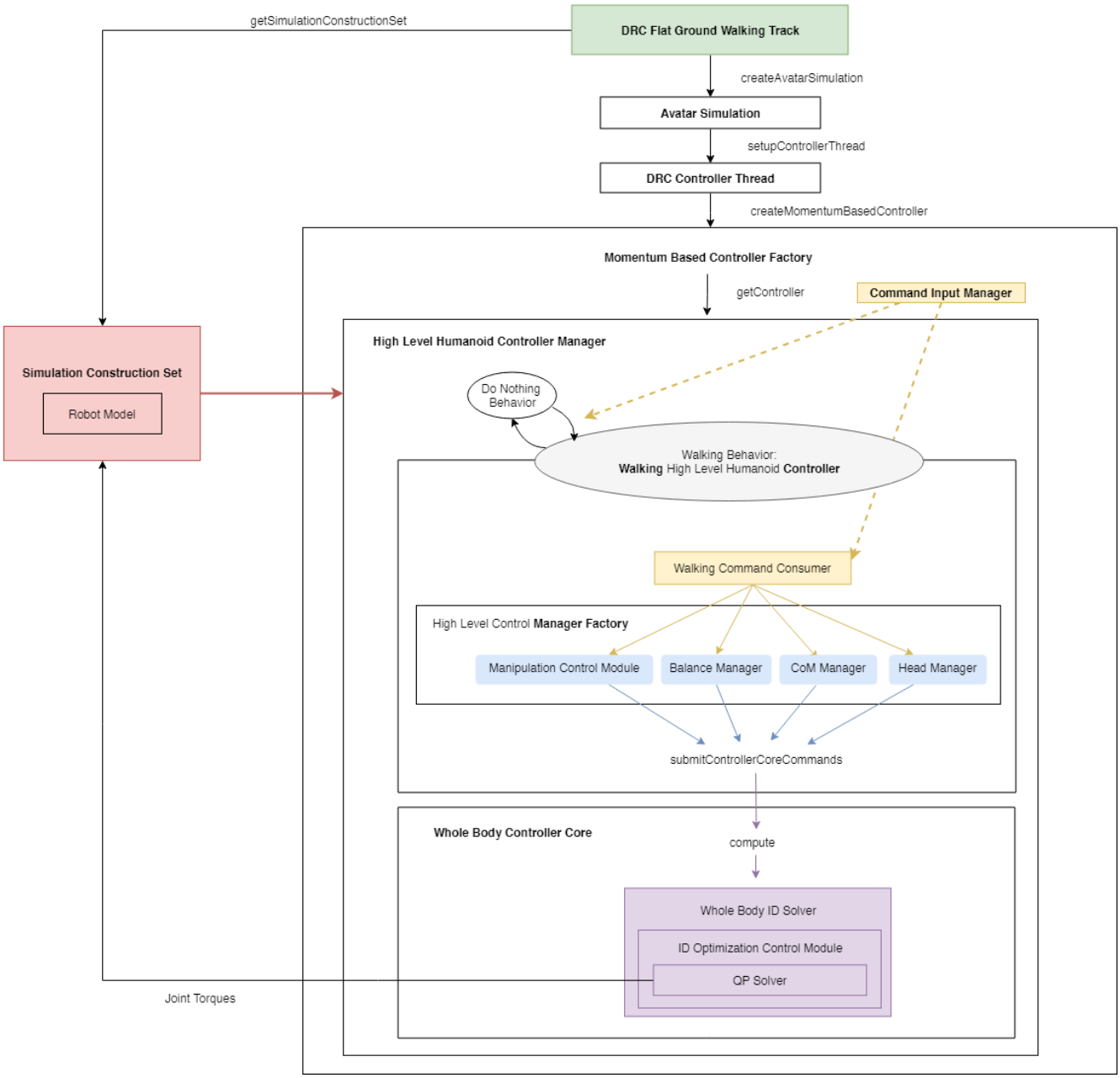
Here, we submit our generated list of motion objectives to the **Whole Body Controller Core**[9]. The controller core sends these commands to a Whole Body Inverse Dynamics Solver[9], resolves conflicting joint instructions with a QP solver[10], and outputs desired torques. For more details on the Controller Core, see Design of a Momentum-Based Control Framework and Application to the Humanoid Robot Atlas [1] or https://ihmcroboticsdocs.github.io/ihmc-open-robotics-software/docs/01-controllercore.html.

Now, the Simulation Construction Set can update the robot model by applying the calculated torques at each joint. We have successfully used the entire IHMC schema to send commands to the robot.

[9] us.ihmc.commonWalkingControlModules.controllerCore
[10] us.ihmc.commonWalkingControlModules.momentumBasedController.optimization

PART 3: The Demonstration

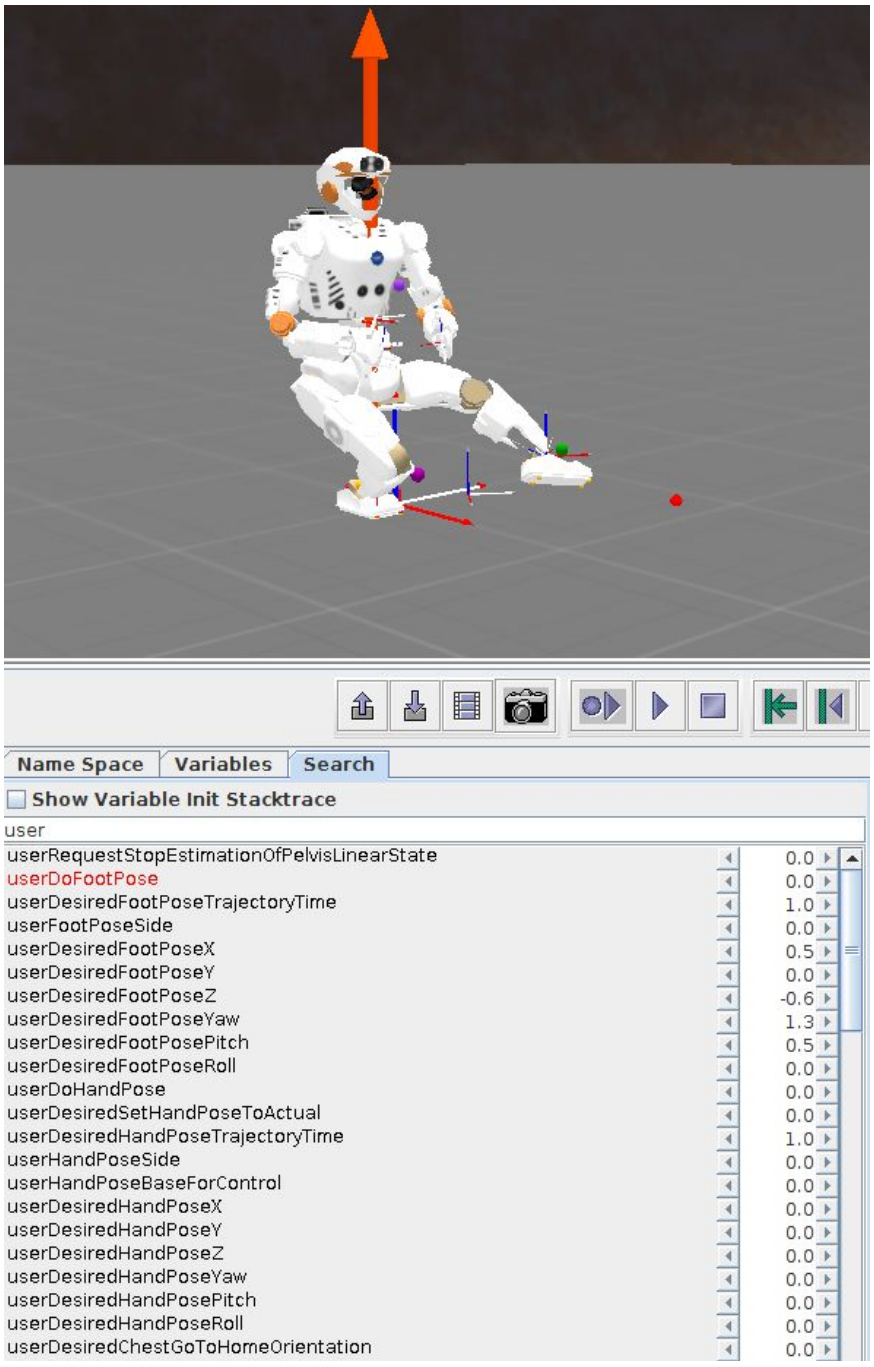
When you ran Valkyrie Demo from the Quick Start section, you successfully implemented all of this control architecture. How is this possible? The demo only had a few lines of code, and none of them mentioned a robot controller. The answer is that the Valkyrie Demo hinges upon the DRC Flat Ground Walking Track[11]. If you follow the declaration of this class, you will see the function calls: `createAvatarSimulation` > `setupControllerThread` > `createMomentumBasedController` > `getController`. This `getController` function returned a High Level Humanoid Controller Manager, initializing the entire IHMC flow.



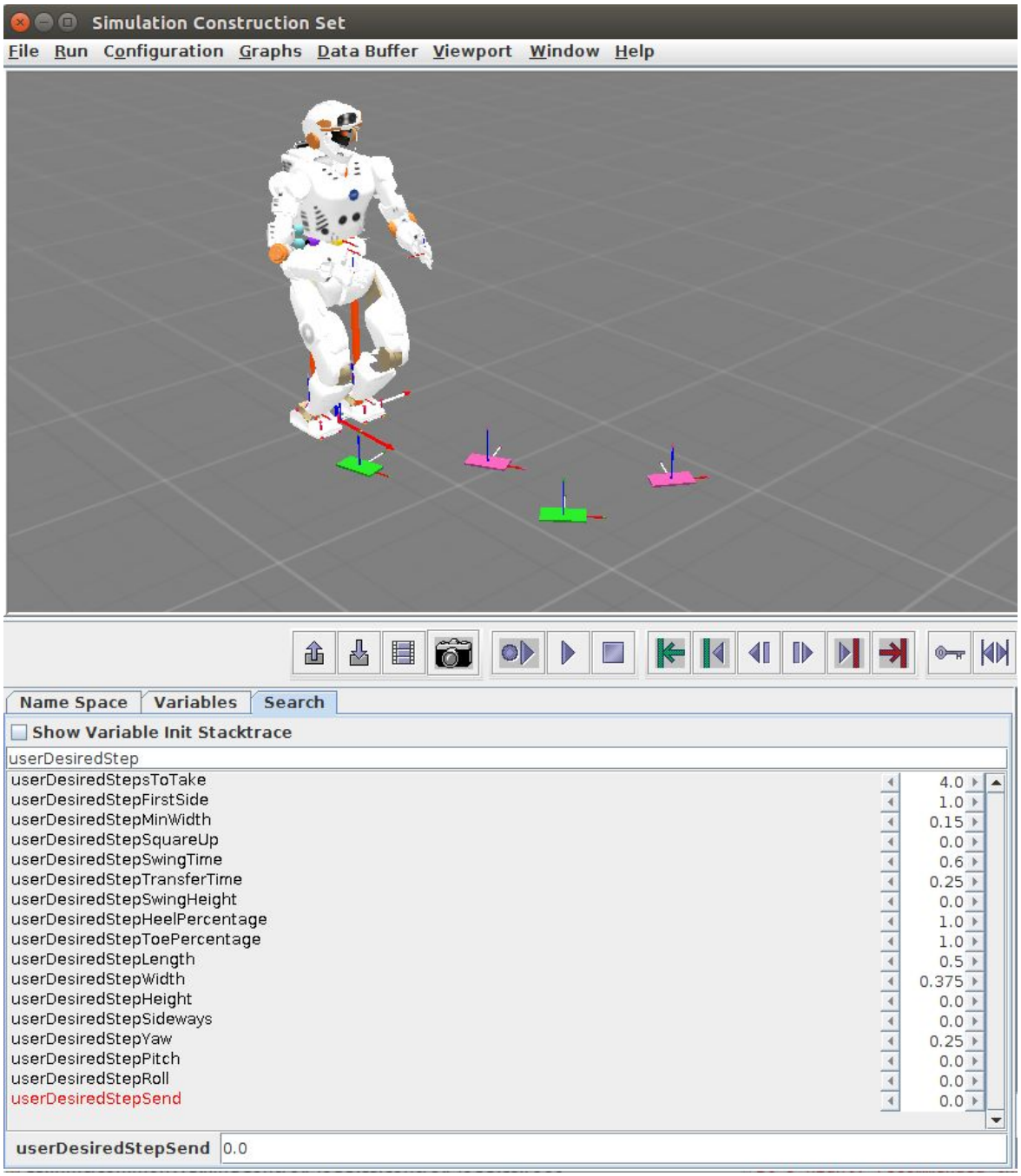
[11] us.ihmc.avatar (IHMCAvatarInterfaces-0.9.0.jar)

User Commands

A key activator in this process is the user providing commands to the Command Input Manager. How does this happen? Submitting a message to the Command Input Manager most often occurs through SCS YoVariables. Many of these variables begin with the word `user`. We can input desired poses (e.g., change the values of `userDesiredFootPose[X][Y][Z][Yaw][Pitch][Roll]`) and execute them (e.g., with the boolean `userDoFootPose`). Pose options which execute individual trajectories are available for the hand, foot, chest, and pelvis.



Additionally, we can specify footstep parameters for controlled walking. For example, if we change `userDesiredStepLength` to 0.5, `userDesiredStepYaw` to 0.25, `userDesiredStepsToTake` to 4.0, and `userDesiredStepSend` to 1.0 (true), the robot will take four steps on our specified curved path and then stop. We can further alter which foot the robot starts with, the timing of each step, and so on. This is a highly controllable alternative to the the continuous walking motion initiated by the `walk` variable from the Quick Start Demonstration.



Where are these user capabilities set up? When we configured the Momentum Based Controller[12], there was an option to enable user input, set to true by default. This created a User Desired Controller Command Generator[13] which in turn started the User Desired Command Generators[13] for the chest, foot, hand, and pelvis. Each initialized their own `user` labeled YoVariables. For example, the User Desired Foot Pose Controller Command Generator[13] is what allowed us to change the desired X, Y, Z, Yaw, Pitch, and Roll values of either foot. When we changed the boolean value of `userDoFootPose`, the `variableChanged` function submitted these values to the Command Input Manager.

For long-term use, the TREC Lab may seek to create a Operator Interface by which to more intuitively specify commands. We should know how to send messages to the Command Input Manager directly and how to build off of these command generators. For more examples of submitting messages to the Command Input Manager[14], open the call hierarchy of its `submitCommand` function.

USER SCRIPT

Thus far, we have only focused on controlling the robot directly from the Simulation Construction Set. However, the IHMC framework also allows XML files of desired command sequences to be run independently.

According to IHMC's report DARPA Robotics Challenge (DRC) Using Human-Machine Teamwork to Perform Disaster Response with a Humanoid Robot [5], a "record" feature can automatically serialize all communication received from the UI into an XML script. It then allows for adaptable playback of these complex movements at any time.

These features extend IHMC's Operator Interface, and thus are not included in the Open Robotics Software. However, the mechanism to parse XML files is captured by the **Script Based Controller Command Generator[15]**. This converts a script's object stream into IHMC controller core commands. A potential next step for the TREC Lab is to create and execute such XML scripts to further automate our control process.

[12] See the `getController` function in the Momentum Based Controller Factory: us.ihmc.commonWalkingControlModules.highLevelHumanoidControl.factories
[13] us.ihmc.commonWalkingControlModules.controllerAPI.input.userDesired
[14] us.ihmc.communication.controllerAPI (IHMCCommunication-0.9.0.jar)
[15] us.ihmc.humanoidBehaviors.behaviors.scripts.engine (IHMCHumanoidBehaviors-0.9.0.jar)

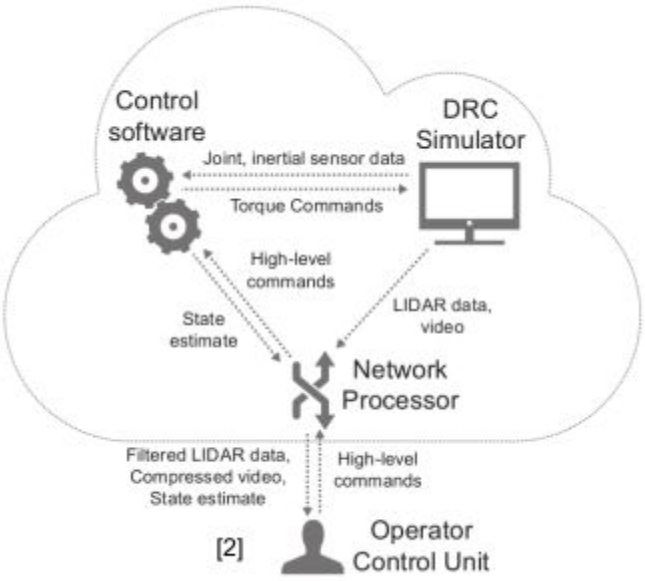
High-Level: Behaviors

Of course, we do not want to stop at specifying trajectory commands for each robot limb. Ideally, we instruct the robot to perform an abstract task (e.g. open door) and the system internally maps this into limb trajectories and then to low-level controller objectives.

A major benefit of using IHMC is that they have already developed some key behaviors, including DRC tasks like opening doors and turning valves as well as goal-based tasks like walking to a location. Our next major undertaking is to understand these high-level behaviors.

For this effort, we will turn our attention away from the Valkyrie Demo and focus on the **Open Humanoids Simulator**[16]. The Open Humanoids Simulator sets up and accesses the IHMC control schema slightly differently than the Valkyrie Demo. Instead of creating an Avatar Simulation through the DRC Flat Ground Walking Track, the Open Humanoids Simulator creates a DRC Simulation Starter. The DRC Simulation Starter's `createAvatarSimulation` function sets up the SCS, GUI, and Momentum Based Controller Factory internally.

The advantage of the DRC Simulation Starter is that it also creates a DRC Network Processor. This sets up a bunch of interesting modules (UI, Sensor, ROS, Text to Speech, LIDAR, etc.). Among them, we enable the **Behavior Module** to run high-level tasks. To learn more about the Network Processor and the speed of the illustrated communication, read [Summary of Team IHMC's virtual robotics challenge entry](#) [2].



In this section, we will be editing the Open Humanoids Simulator script and some other pertinent behavior classes. Since we can not edit these .class files, we need to create an identical Java Class in our Gradle project (.java) and make changes to the uncompiled version. If the filename and package exactly match the compiled version, it will always be called instead of its .class equivalent.

[16] us.ihmc.valkyrie (Valkyrie-0.9.0.jar)

There is an import issue in the Open Humanoids Script which we will ignore for now. The `import org.ros.internal.message.Message` returns an error, so for preliminary testing comment it out as well as all of the ROS lines that depend on it.

DIAGNOSTIC BEHAVIOR

We will first use the Open Humanoids Simulator in its Automatic Diagnostic mode. In the `main` function, change its first boolean argument (`runAutomaticDiagnosticRoutine`) to `true`.

Run the script. Recall from the Quick Start Demonstration that Java Applications must be run with a custom configuration. To create this for the Open Humanoids Simulator, select `Run Configurations` and create a `new Java Application`. In the application that is made, add "`-Xms4096m -Xmx4096m`" to VM arguments, so the simulator has enough memory to run.

Now you can execute the Open Humanoids Simulator. When you do so, you will be creating an **IHMC Humanoid Behavior Manager**[17]. The exact call hierarchy is `OpenHumanoidsSimulator > simulationStarter.startSimulation > createSimulation > startNetworkProcessor > DRCNetworkProcessor > setupBehaviorModule > IHMCHumanoidBehaviorManager.`

The IHMC Humanoid Behavior Manager creates a very important **Behavior Dispatcher**[18]. The Behavior Dispatcher is responsible for managing a Behavior State Machine (this is different from the high-level state machine we encountered earlier). Essentially, we can register every possible high-level behavior as a state. Then, when we request a behavior to run, the dispatcher is responsible for transitioning to and executing that state. By default, the state machine only has one state: a Do Nothing Behavior.

Once we initialize the state machine, we can populate it with our custom behaviors. The IHMC Humanoid Behavior Manager will do this either through the function `createAndRegisterAutomaticDiagnostic` or `createAndRegisterBehaviors.`

Since we are starting simple with the Automatic Diagnostic mode, we will run `createAndRegisterAutomaticDiagnostic`. It creates a single **Diagnostic Behavior**[19] and adds it to the state machine with `dispatcher.addBehavior(…)`. In order to transition from the Do Nothing Behavior to our Diagnostic Behavior, we request it from the dispatcher with **`dispatcher.requestBehavior(…)`**. This `requestBehavior` function is the key to starting behaviors. For a high-level GUI, we want a `RUN` button to execute `dispatcher.requestBehavior`.

[17] us.ihmc.humanoidBehaviors (IHMCHumanoidBehaviors-0.9.0.jar)
[18] us.ihmc.humanoidBehaviors.dispatcher
[19] us.ihmc.humanoidBehaviors.behaviors.diagnostic

What does the Diagnostic Behavior do? Actually, it can do a lot of things. There are a host of Diagnostic Tasks which use sequenced limb movements to emulate higher level operation.

```
public enum DiagnosticTask
{
   CHEST_ROTATIONS,
   PELVIS_ROTATIONS,
   BOOTY_SHAKE,
   SHIFT_WEIGHT,
   COMBINED_CHEST_PELVIS,
   ARM_MOTIONS,
   ARM_SHAKE,
   UPPER_BODY,
   FOOT_LIFT,
   FOOT_POSES_SHORT,
   FOOT_POSES_LONG,
   RUNNING_MAN,
   BOW,
   KARATE_KID,
   WHOLE_SCHEBANG,
   SQUATS,
```

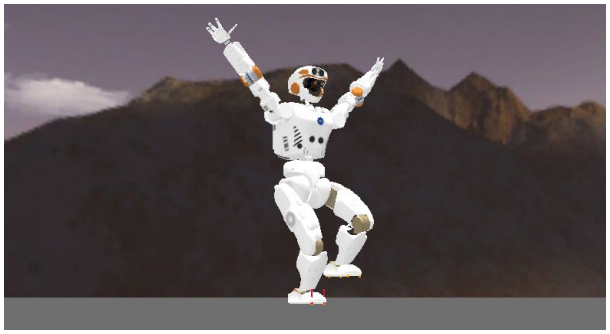
```
private void sequenceChestRotations(double percentOfJoin
private void sequencePelvisRotations(double percentOfJoi
private void sequenceShiftWeight()
private void sequenceMovingChestAndPelvisOnly()
private void sequenceArmPose(RobotSide robotSide)
private void sequenceFootPoseShort()
private void submitFootPosesShort(RobotSide robotSide)
private void sequenceFootPoseLong()
```

It will be helpful at this point to take a break and look through the contents of these functions. They quite literally are sequences of submitting hand, foot, chest, and pelvis poses, positions, and orientations. We can take these "submit" functions at their word, or we can work through their implementation. If you choose the latter, be warned that the notation gets a bit dense. The short version is that something like an Arm **Trajectory Message** will contain the key information about manipulating target limbs. An Arm **Trajectory Behavior** will know how to send packets to the Network Processor. A **Trajectory Task** takes as input a Trajectory Message and a Trajectory Behavior. Once a Trajectory Task is submitted to the Pipeline, it has been sent to the robot.

What actually runs in a Diagnostic Behavior? In the `doControl` function (run when we enter the Diagnostic Behavior state), we start with an Automatic Diagnostic Routine. We can change this routine by calling a sequence in the `automaticDiagnosticRoutine` function. The default `sequenceSimpleWarmup` takes some time to run, so you may want to change this to the shorter `sequenceSquats`.

Then, we handle the requested diagnostic behavior. We can provide an initial request when we declare the Diagnostic Behavior with `requestedDiagnostic.set(DiagnosticTask.KARATE_KID)`. This initial request will be run immediately after the Automatic Diagnostic Routine.

Now if we run Open Humanoids Simulator and press simulate, we will see the robot do a squat, then perform the karate kid behavior. After it has completed both, it will continue to loop through Diagnostic Behavior's `doControl` function. We will not see any more movement, because nothing is being requested.



To run another sequence, we must submit a new requested diagnostic using the `requestDiagnosticBehavior` function. Again, this would make for a great button on a high-level GUI.

DRC BEHAVIORS

Now we leave the world of Diagnostic Behaviors. Back in the Open Humanoids Simulator, change the boolean argument `runAutomaticDiagnosticRoutine` to `false`. In `public OpenHumanoidsSimulator`, add these two lines:

```
networkProcessorParameters.enableBehaviorModule(true);
networkProcessorParameters.enableBehaviorVisualizer(true);
```

These parameters were already set in the Diagnostic version. The first line allows us to create the IHMC Humanoid Behavior Manager which launched the Behavior Dispatcher and everything followed. The second line should enable the relevant behavior YoVariables to appear on the SCS GUI.

The IHMC Humanoid Behavior Manager will again initialize the Behavior Dispatcher and a Behavior State Machine with a single Do Nothing state. However, since we are not in the diagnostic mode, we will now run the IHMC Behavior Manager's `createAndRegisterBehaviors` function. This adds many behaviors to the state machine, including a Diagnostic Behavior, but also Pick Up Ball, Turn Valve, Walk Through Door and many more. Each behavior that is added (with `dispatcher.addBehavior`) is connected to every other state such that we can always transition to any behavior.

Take a few moments here to examine the contents of each Behavior class. You can open them directly from the IHMC Humanoid Behavior Manager using the Open Declaration (F3) tool, or find many of them in `us.ihmc.humanoidBehaviors.behaviors.complexBehaviors`.

Complex behaviors are built up from the primitive behaviors we have already encountered. For example, the `BasicPipeLineBehavior` uses `WalkToLocation` and `GoHome` Behaviors from the Diagnostic Tasks. The DRC Valve Task (`TurnValveBehaviorStateMachine`) concatenates other complex behaviors: `searchForValve`, `walkToInteractableObject`, `resetRobot`, `graspAndTurnValve`. These, in turn, are sets of primitive Trajectory Behaviors.

We will now be editing the IHMC Humanoid Behavior Manager's `createAndRegisterBehaviors` function. First, comment out the entire block pertaining to the Blob Filtered Sphere Detection Behavior. This requires OpenCV tools that we have not set up, and will return a fatal error if included[20].

After the behaviors have been added to the dispatcher, also insert the line:
`dispatcher.requestBehavior(HumanoidBehaviorType.TEST_PIPELINE);`
This will start our desired behavior upon running Open Humanoids Simulator.

[20] We attempted to add missing OpenCV .jar dependencies with the following instructions https://udallascs.wordpress.com/2014/03/30/adding-opencv-and-configuring-to-work-with-eclipse-and-java/, but it did not resolve the error:

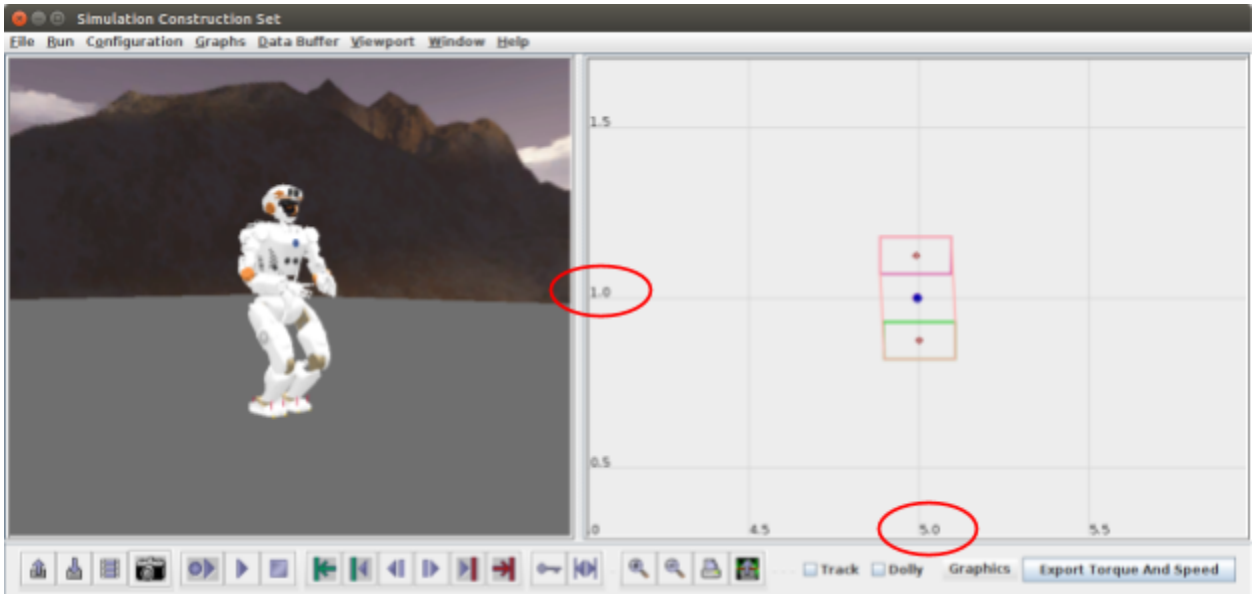
Java HotSpot(TM) 64-Bit Server VM warning: You have loaded library /home/.../libopencv_java310.so which might have disabled stack guard. The VM will try to fix the stack guard now.
It's highly recommended that you fix the library with 'execstack -c <libfile>', or link it with '-z noexecstack'.
/home/.../libopencv_java310.so: libIlmImf.so.6: cannot open shared object file: No such file or directory
Exception in thread "main" java.lang.UnsatisfiedLinkError: Cannot load org/opencv/opencv_java310

We can further edit our chosen behavior, in this case `BasicPipeLineBehavior`[21]. For now, let's change the target pose to walk to from (0,0) to (5,1).

Now run OpenHumanoidsSimulator. In the Console, print statements should show that the IHMC Humanoid Behavior Manager is initializing and behaviors are loading.

```
[INFO] (DRCNetworkProcessor.java:361): Connecting to controller using intra process
communication
[INFO] IHMCHumanoidBehaviorManager: Initializing
[INFO] (ObjectDetectorFromCameraImages.java:173): Attempting to connect to valve
detector...
PickUpBallBehaviorStateMachine queue size 1
```

After pressing simulate, we will see Valkyrie walk to position (5,1).



We can run behaviors like ***TEST_PIPELINE***, ***TEST_STATEMACHINE***, and ***EXAMPLE_BEHAVIOR*** out of the box. The other behaviors require building a DRC environment with interactable objects (a door, valve, ball, fiducial marker, etc.).

Future work in this area should include setting up such an environment and creating custom behaviors. Ultimately, we want to create our own Operator Interface that allows us to click on target objects and launch high-level operations.

[21] us.ihmc.humanoidBehaviors.behaviors.complexBehaviors

Low-Level: Using the Controller Core

For those not interested in the entire IHMC software stack, this section explains how to use its Whole Body Controller Core in isolation to perform low-level control.

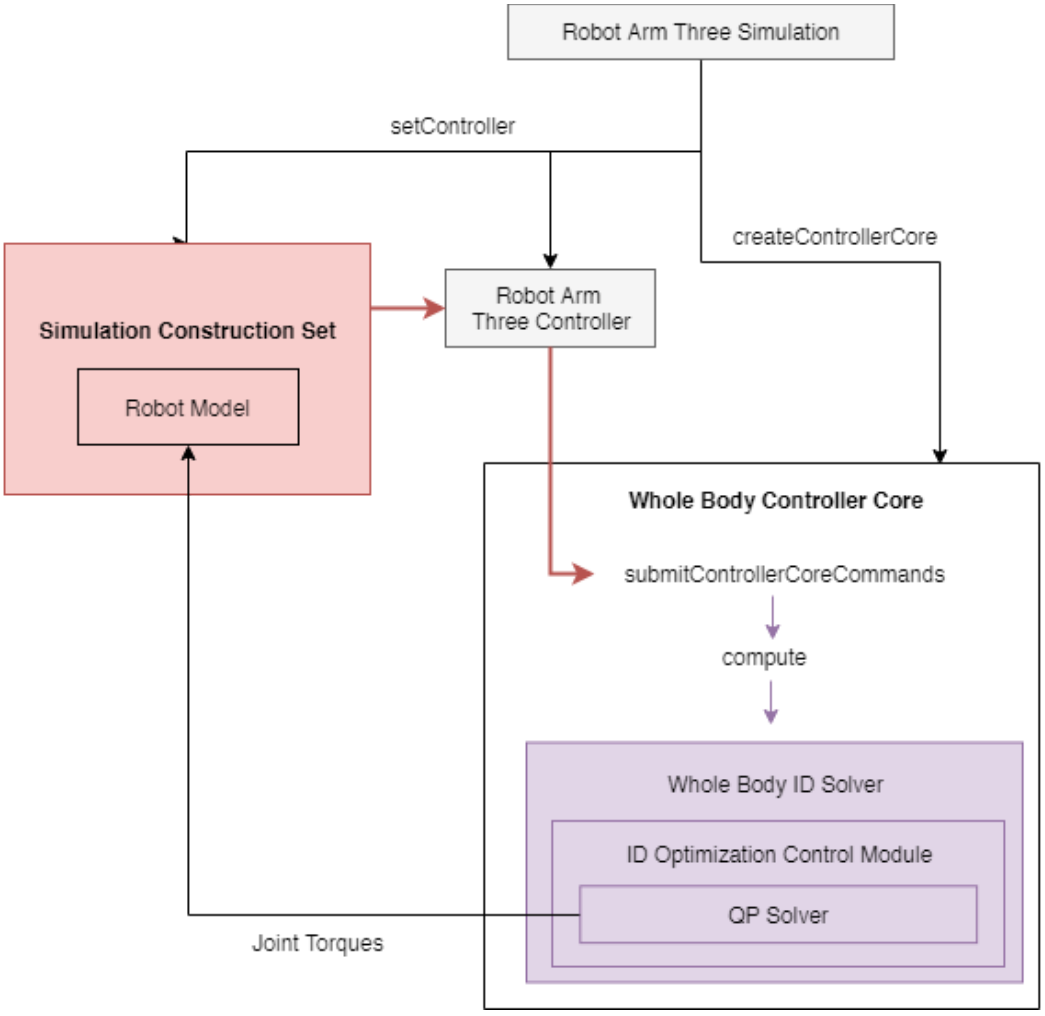
Recall that the Simulation Construction Set updates its environment by running active Robot Controllers. Instead of using IHMC's High Level Controller Manager, we will create a low-level Robot Arm Controller.

For joint-space operation, `doControl` creates a Jointspace Feedback Control Command with desired joint positions and velocities. For task-space operation, `doControl` creates a Spatial Feedback Control Command with desired position, orientation, linear and angular velocities of the end effector. In either case, the Feedback Control Command is then translated to a Controller Core Command, which in turn is submitted to the Whole Body Controller Core.

The Whole Body Controller Core is an extremely useful tool because it performs all of the computations to transform desired motion objectives to joint torque outputs. With an additional integrator tool, it also outputs the desired position and velocity of each joint. Thus, the user can update the robot model by setting the desired effort or joint configuration.

To dig into examples of creating a robot controller and accessing the Whole Body Controller Core, follow the steps of this tutorial: https://github.com/ihmcrobotics/dynamic-walking-tutorial-2018/wiki. Pay special attention to the **RobotArmThreeController**. I highly recommend tracing the bolded steps of its `doControl` function with the Open Declaration (F3) tool.

```
public void doControl()
{
    ...
    // Create a task space command
    SpatialFeedbackControlCommand command = new SpatialFeedbackControlCommand();
    command.set(desiredEndEffectorPosition, desiredEndEffectorLinearVelocity);
    command.set(desiredEndEffectorOrientation, desiredEndEffectorAngularVelocity);

    //  Create the controller core command equivalent
    ControllerCoreCommand controllerCoreCommand = new
    ControllerCoreCommand(controllerCoreMode);
    controllerCoreCommand.addFeedbackControlCommand(command);

    // Submit objectives to be achieved to the controller core. This internally
    submits a command to the ID Solver, which maps objectives into desired
    accelerations, in turn sent to the QP solver
    wholeBodyControllerCore.submitControllerCoreCommand(controllerCoreCommand);

    // Run WBCC computations. This asks the ID Solver to compute, which calls upon
    the QP solver.
    wholeBodyControllerCore.compute();

    // Return desired torque, position and effort for this tick of the control loop
    JointDesiredOutputListReadOnly outputForLowLevelController =
    wholeBodyControllerCore.getOutputForLowLevelController();

    // Write the controller output to the simulated joints (iterated by jointEnum)
    ...
    robotArm.setDesiredEffort(jointEnum,jointDesiredOutput.getDesiredTorque());
}
```


Robot Arm Three Simulation
setController
createControllerCore
Simulation Construction Set
Robot Model
Robot Arm
Three Controller
Whole Body Controller Core
submitControllerCoreCommands
compute
Whole Body ID Solver
ID Optimization Control Module
QP Solver
Joint Torques

Accessing the ORS

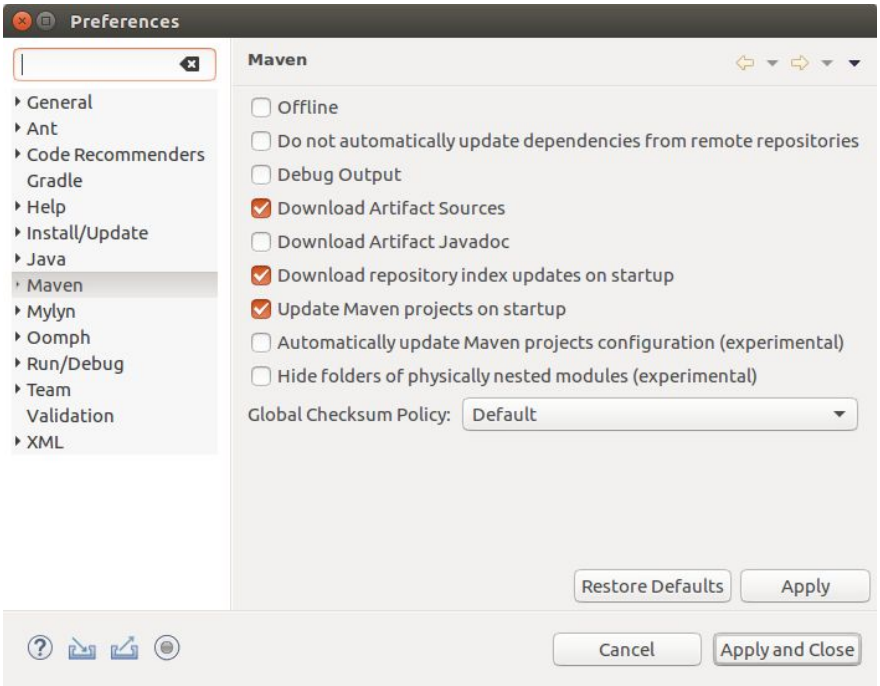
As shown by the Valkyrie Demo, you do **not** need to clone and build IHMC's code from their Github in order to access the Open Robotics Software (ORS). The critical step in lieu of this process was adding a compile dependency statement (`us.ihmc:Valkyrie:0.9.0`) in the build.gradle file. This downloads IHMC's .jar and .pom files which are the compiled versions of what you can see on Github.

Let's extend this process for another simulation - say, ValkyriePushRecoveryTrack[22]. Looking at the import statements, we see we are missing packages like Euclid, YoVariables, and SimulationToolkit.

We can find the exact names in http://dl.bintray.com/ihmcrobotics/maven-release/us/ihmc/ and add the statements as follows:

```
compile 'us.ihmc:euclid:0.8.2'
compile 'us.ihmc:ihmc-yovariables:0.3.4'
compile 'us.ihmc:SimulationConstructionSet:0.9.0'
```

To use Maven successfully, we make the following changes under Window > Preferences.



Finally, to update our dependencies, right-click on the Gradle Project ValkyriePushRecoveryTask resides in, and run Gradle > Refresh Gradle Project. The Progress and Console tabs will show the ORS downloads occuring.

We can view our downloads at any time by expanding the Project and External Dependencies folder on the Package Explorer.

[22] https://github.com/ihmcrobotics/ihmc-open-robotics-software/blob/develop/valkyrie/src/main/java/us/ihmc/valkyrie/simulation/ValkyriePushRecoveryTrack.java

BUGS:

It is worth noting that the IHMC code has some errors that appear due to updating inconsistency; we actually have to make a few changes to make ValkyriePushRecoveryTrack work.

1) DRC Robot Target

The avatar package for Valkyrie does not have a robot target. You can check this by expanding Project and External Dependencies > IHMCAvatarInterfaces-0.9.0.jar > us.ihmc.avatar.drcRobot.

To fix, first ensure you have the following import statement:
`import us.ihmc.avatar.drcRobot.DRCRobotModel`

Now remove the Robot Target import statement:
`import us.ihmc.avatar.drcRobot.RobotTarget`

When you import the robot model, change
`new ValkyrieRobotModel(RobotTarget.SCS, false)` to
`new ValkyrieRobotModel(DRCRobotModel.RobotTarget.SCS, false)`

2) Floating Root Joint Robot

Similar to the above issue, we need to import Floating Root Joint Robot from Simulation Construction Set instead of Simulation Construction Set Tools.

```
import ...simulationConstructionSetTools.util.HumanoidFloatingRootJointRobot →
import us.ihmc.simulationconstructionset.HumanoidFloatingRootJointRobot;
```

2) Vector 3D vs. Vector 3d

The Euclid 3D vector has different parameters than the Javax 3d version. Though the script imports Euclid, it uses the Javax version. So add the import statement `import javax.vecmath.Vector3d` and change all references of `Vector3D` to `Vector3d`.

3) Casting YoVariables

The simulation construction set getVariable returns YoVariable<?>. For some reason, we get an error trying to cast this into a Boolean YoVariable. If you go to the source for this, there is a comment about how casting makes an angel lose it wings. We don't want this for angels, so for now just comment out all references to the YoVariable `enable`. This isn't the best practice either, but the script can run without it.

Now ValkyriePushRecoveryTrack should run. Simulate walking as with ValkyrieDemo and press PushRobot when ready to apply the external force.

Valkyrie to THOR

Though not on the Github like Valkyrie or Atlas, Thor and Escher are also supported by IHMC. Their files can be found on the Maven link http://dl.bintray.com/ihmcrobotics/maven-release/us/ihmc/ and included accordingly.

Unfortunately running these simulation classes also returns errors due to outdated import statements. See the Marvin computer for a successful Thor simulation.

Projects Using IHMC

The Open Humanoids Project (https://github.com/openhumanoids) out of MIT and the University of Edinburgh appear to be adapting MIT's control work from DRC to use with IHMC's controller core. Their Github wiki serves as a guide to operating IHMC on the physical Valkyrie robot. https://github.com/openhumanoids/wiki/wiki

A key feature of this work is MIT's user interface: The Director. To read about this software, see Director: A User Interface Designed for Robot Operation with Shared Autonomy [3]. We hoped that by adopting the MIT/IHMC schema, we could easily instruct high-level IHMC capabilities and begin to work with the capabilities of an advanced UI.

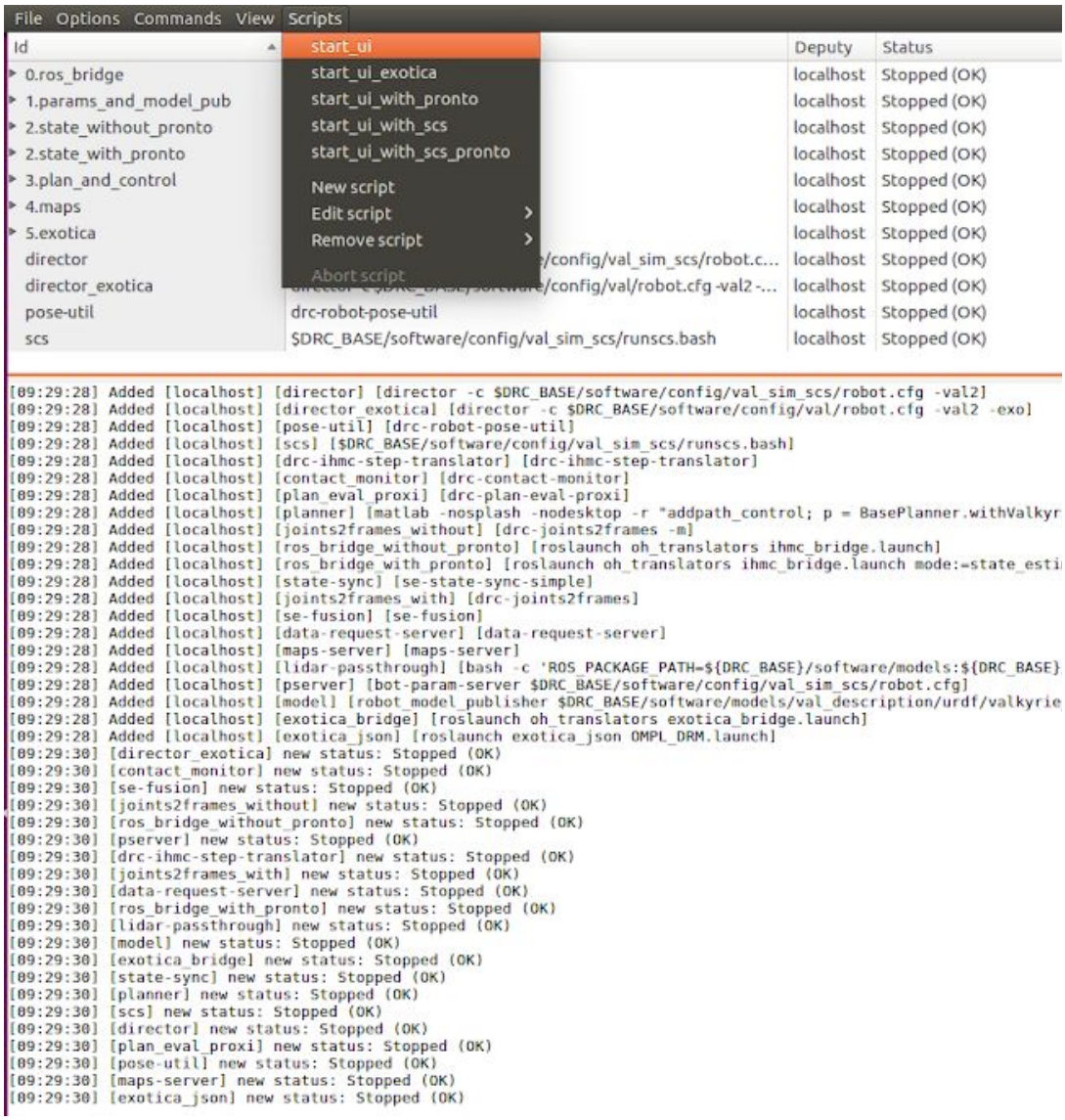
To install these software tools and link them together, we followed Open Humanoid's instructions "To run Valkyrie paired with Drake Designer UI" on their Github Wiki: https://github.com/openhumanoids/wiki/wiki/Installing-IHMC's-Atlas-and-Valkyrie-Codebase

We have already installed IHMC ORS, so we can run `NetworkParametersCreator`. The next step is to install the Open Humanoids Software. Unfortunately, at the time of writing this report, the provided "Install OpenHumanoids" link is broken. Instead, we followed the README instructions in the oh-distro repository https://github.com/openhumanoids/oh-distro, which contains the Director and Drake (MIT control software). The first line heeds us "don't expect this to work!" and indeed, we have not gotten it operational. We followed the steps as best we could (without having licenses for MATLAB, MOSEK, or Gurobi).

To establish a bridge, the Wiki instructs us to run `roscore` and the procman process manager. Perhaps due to our incomplete installation, the `bot-procman-sheriff` command is not recognized by default. To use procman, follow these additional steps:

1) `$ git clone git@github.com:RobotLocomotion/libbot.git`
2) `$ cd libbot`
3) `$ sudo make BUILD_PREFIX=/usr/local -j`
4) `Add the following to .bashrc`
    a) `PATH=$PATH:~/libbot/build/bin`
    b) `export PATH`

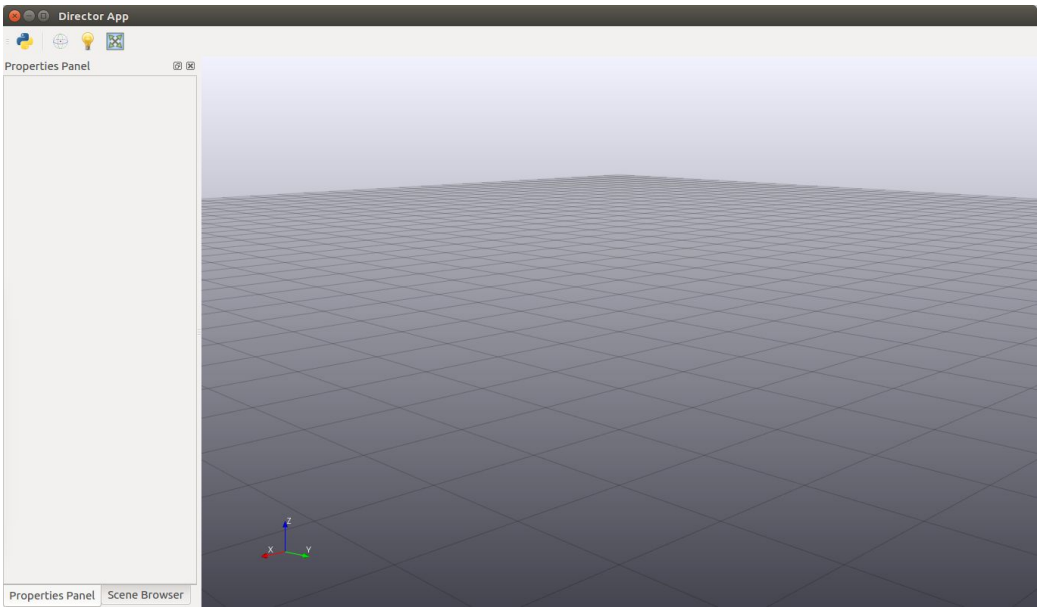
Now `cd ~/oh-distro/software/config/val_sim_scs` and run `bot-procman-sheriff -l robot.pmd`. In the window that pops up, go to scripts in top toolbar and run "ui_script".



This should launch the Director, planners, and IHMC bridge, but instead returns errors for “No such file or directory” for each.

```
[model] starting
[pserver] starting
[model] ERROR executing [robot_model_publisher $DRC_BASE/software/models/val_description/urdf/valkyrie_sim.urdf]
[model] execv: No such file or directory
[model] exited with status 255
[ros_bridge_without_pronto] exited with status 1
[joints2frames_without] starting
[joints2frames_without] ERROR executing [drc-joints2frames -m]
[joints2frames_without] execv: No such file or directory
[joints2frames_without] exited with status 255
[planner] starting
[planner] ERROR executing [matlab -nosplash -nodesktop -r "addpath_control; p = BasePlanner.withValkyrie(2); p.run();"]
[planner] execv: No such file or directory
[drc-ihmc-step-translator] starting
[planner] exited with status 255
[drc-ihmc-step-translator] ERROR executing [drc-ihmc-step-translator]
[drc-ihmc-step-translator] execv: No such file or directory
[contact_monitor] starting
[drc-ihmc-step-translator] exited with status 255
[contact_monitor] ERROR executing [drc-contact-monitor]
[contact_monitor] execv: No such file or directory
[plan_eval_proxi] starting
[contact_monitor] exited with status 255
[plan_eval_proxi] ERROR executing [drc-plan-eval-proxi]
[plan_eval_proxi] execv: No such file or directory
[plan_eval_proxi] exited with status 255
[maps-server] starting
[lidar-passthrough] starting
[maps-server] ERROR executing [maps-server]
[maps-server] execv: No such file or directory
[maps-server] exited with status 255
[data-request-server] starting
[data-request-server] ERROR executing [data-request-server]
[data-request-server] execv: No such file or directory
[data-request-server] exited with status 255
[director] starting
[director] ERROR executing [director -c $DRC_BASE/software/config/val_sim_scs/robot.cfg -val2]
[director] execv: No such file or directory
[lidar-passthrough] exited with status 127
[director] exited with status 255
```

This is almost certainly due to the broken installation process. If one follows the installation for Director a standalone software (https://github.com/RobotLocomotion/director), then Director runs with commands such as `build/install/bin/directorPython src/python/tests/testMainWindowApp.py`[23]



[23] https://github.com/RobotLocomotion/director/issues/609

Conclusions

We conclude this phase with two software platforms ready and operational. For use moving forward, we recommend the IHMC Open Robotics Software. Having identified the key components in its control architecture, we can now adapt them for the TREC Lab's use.

Certainly, more developmental work is required to generate our own high-level behaviors and to pair the IHMC software with custom hardwire drivers. However, the XBotCore alternative requires significantly more algorithm development and coding to achieve even our baseline control purposes.

We must also consider that without an operational robot to test the solutions, it is difficult to draw solid conclusions about the real-time efficacy of either software.

In this vein, our future plans include developing a full simulation of THOR in the DRC environment, testing and further developing real-time commands on a microcontroller, and ultimately controlling a full physical robot to perform high-level tasks.

References

[1] Koolen, T. et al. "Design of a momentum-based control framework and application to the humanoid robot Atlas" *International Journal of Humanoid Robotics*

[2] Koolen, T. et *al*., "Summary of Team IHMC's virtual robotics challenge entry," *2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, Atlanta, GA, 2013, pp. 307-314. doi:10.1109/HUMANOIDS.2013.7029992

[3] Marion, P. , Fallon, M. , Deits, R. , Valenzuela, A. , Pérez D'Arpino, C. , Izatt, G. , Manuelli, L. , Antone, M. , Dai, H. , Koolen, T. , Carter, J. , Kuindersma, S. and Tedrake, R. (2017), Director: A User Interface Designed for Robot Operation with Shared Autonomy. J. Field Robotics, 34: 262-280. doi:10.1002/rob.21681

[4] Muratore, Luca & Laurenzi, Arturo & Mingo Hoffman, Enrico & Rocchi, Alessio & G Caldwell, Darwin & Tsagarakis, Nikos. (2017). On the Design and Evaluation of XBotCore, a Cross-Robot Real-Time Software Framework. Journal of Software Engineering for Robotics. 8. 164-170. 10.6092/JOSER_2017_08_01_p164.

[5] Pratt, J. “DARPA Robotics Challenge (DRC) Using Human-Machine Teamwork to Perform Disaster Response with a Humanoid Robot”. Florida Institute for Human and Machine Cognition Inc. Pensacola United States, 2017.

[6] Rigano, Giuseppe & Muratore, Luca & Laurenzi, Arturo & Mingo Hoffman, Enrico & Tsagarakis, Nikos. (2018). Towards a Robot Hardware Abstraction Layer (R-Hal) Leveraging the XBot Software Framework. 10.1109/IRC.2018.00036.

[7] Tsagarakis, N. G., Caldwell, D. G., Negrello, F. , Choi, W. , Baccelliere, L. , Loc, V. , Noorden, J. , Muratore, L. , Margan, A. , Cardellino, A. , Natale, L. , Mingo Hoffman, E. , Dallali, H. , Kashiri, N. , Malzahn, J. , Lee, J. , Kryczka, P. , Kanoulas, D. , Garabini, M. , Catalano, M. , Ferrati, M. , Varricchio, V. , Pallottino, L. , Pavan, C. , Bicchi, A. , Settimi, A. , Rocchi, A. and Ajoudani, A. (2017), WALK-MAN: A High-Performance Humanoid Platform for Realistic Environments. J. Field Robotics, 34: 1225-1259. doi:10.1002/rob.21702