TOGAC29_AMBF_IN_ConnectMatlab2AMBF.docx

The following code and example output images were produced on a virtual machine Linux platform running Ubuntu 20.04 with ROS Noetic installed and the user is name 'robotics'. AMBF has been installed and the Galen Robot ADF files have been built, made, and located on the home drive in a folder called Galen_AMBF_Simulation outside of the /ambf folder. MATLAB 2022b was used to control the simulation.

The MATLAB communication here was done both inside the virtual machine with a MATLAB inside the virtual machine, and also performed via MATLAB outside of the virtual machine communicating with ROS from the windows 10 desktop through the virtual machine.

1. Contents

2. Connecting MATLAB to ROS 2
3. Connecting MATLAB to the AMBF Simulation - The MATLAB Python Environment 3
4. Publishing and Subscribing to topics 5
5. Sending and Receiving Messages 6

2. Connecting MATLAB to ROS

To use the ROS commands in MATLAB you must have the ROS Toolbox. With this addon you can use ROS subscribers and publishers to communicate with ROS over ROS messages.
Ensure that ROS has been started by typing:

```
roscore
```

In a separate terminal. Notice the ROS_MASTER_URI.

```
robotics@ubuntu:~$ roscore
... logging to /home/robotics/.ros/log/c4595706-c458-11ed-8fa6-3dd695f6de02/roslaunch-ubuntu-71659.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://ubuntu:46823/
ros_comm version 1.15.15


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.15

NODES

auto-starting new master
process[master]: started with pid [71674]
ROS_MASTER_URI=http://ubuntu:11311/

setting /run_id to c4595706-c458-11ed-8fa6-3dd695f6de02
process[rosout-1]: started with pid [71691]
started core service [/rosout]
```

The entry after ROS_MASTER_URI, in this case 'http://ubuntu:11311/', will be the address you will use to start the MATLAB communication with ROS. In most cases, if ROS is being run on the same machine, MATLAB will default to the master address, but if running on a network or across a virtual machine, you will need this address.
In the MATLAB terminal type:

```
rossinit
```

And it should connect you to the ROS master node. If you're connecting across a network or virtual machine use:

```
rossinit('http://ubuntu:11311/')
```

And then you should connect to ROS.
In the case that errors regarding the python distribution or other compatibility issues see the next section.
End the ROS to MATLAB communication by running:

```
rosshutdown
```

It is best not to have ROS open when working through the next section.

3. Connecting MATLAB to the AMBF Simulation - The MATLAB Python Environment

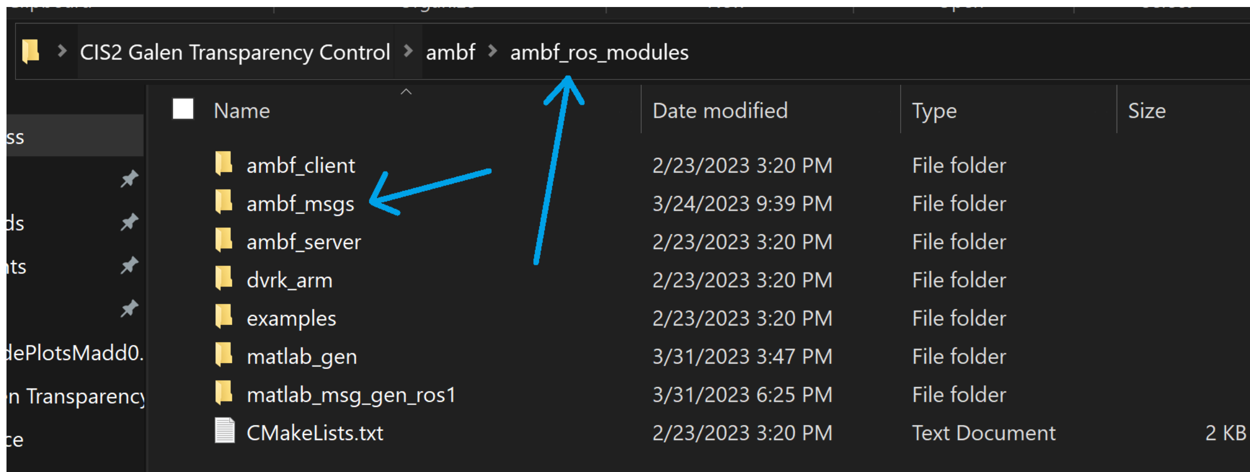
AMBF uses custom ROS messages to communicate over ROS. We need to create these messages and connect them to MATLAB so they appear when the:

```
rosmsg list
```

command is run. Some ROS packages will also have issues running without a proper Python environment setup for MATLAB. Here we will discuss how to create these. Install or find a distribution of python 3.9 to use with MATLAB. Ensure you can find a python 3 exe within an accessible directory for MATLAB.
Note that if you are using MATLAB 2022b then you will need a version of python 3.9x. For some distributions it specifically requires python 3.9.13. It may not specify that python 3.9.13 is required, but it may give continual errors if 3.9.13 is not the python version used.
To build the environment enter the following command:

```
ros.internal.createOrGetLocalPython(true)
```

It may fail but will give you the option to open a box under one of the underlined error messages that will lead to a window where you can locate the python exe. After selecting the python exe from the directory, press okay in the window to have MATLAB build the python environment. MATLAB will create its own python environment in
/home/username/.matlab/R2022b/ros1/glnxa64/venv
After the environment has been made, it is time to create the custom messages for ROS and add them to the rosmg list. For the custom ambf messages, these messages are found in the ambf_msgs folder under ambf_ros_modules.



Ensure MATLAB can access a directory where the ambf_ros_modules folder is available. It may be easiest to copy the ambf folder into the directory you plan to work in with MATLAB, since you will be adding this directory to the workspace path in the next following steps. Run the command with your own directory in place to the ambf_ros_modules folder:

```
rosgenmsg('/home/robotics/ambf/ambf_ros_modules')
```

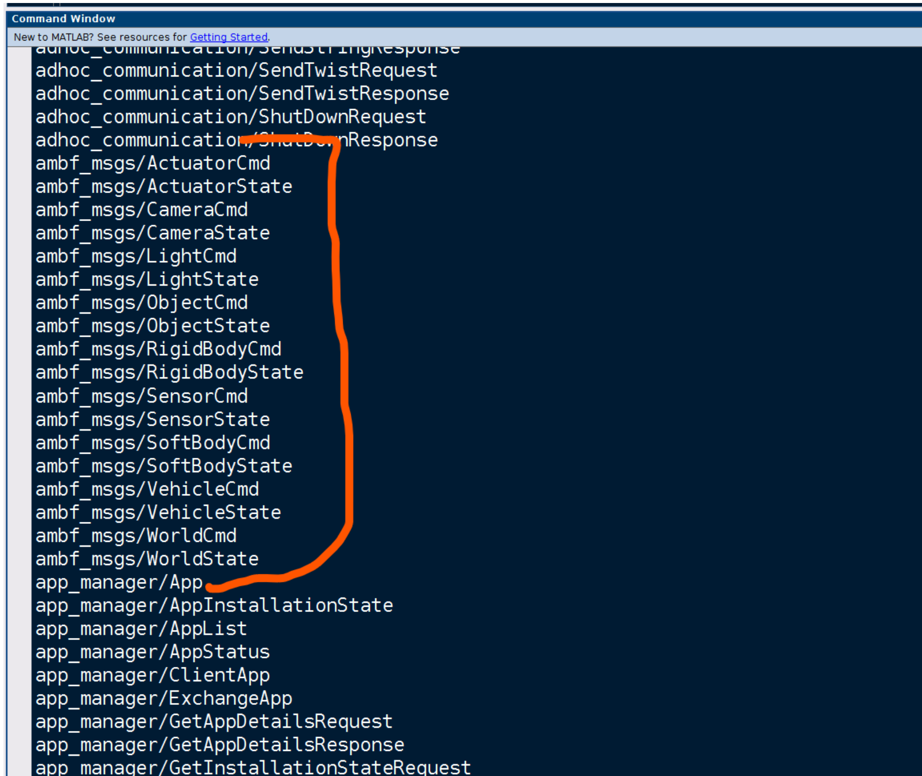
To generate the custom ambf_commands. Then follow the output steps 1 through 3 in the MATLAB terminal to complete the message setup process.

```
>> rosgenmsg('/home/robotics/ambf/ambf_ros_modules')
Identifying message files in folder '/home/robotics/ambf/ambf_ros_modules'..Done.
Validating message files in folder '/home/robotics/ambf/ambf_ros_modules'..Done.
[1/1] Generating MATLAB interfaces for custom message packages... Done.
Running catkin build in folder '/home/robotics/ambf/ambf_ros_modules/matlab_msg_gen_ros1/glnxa64'.
Build in progress. This may take several minutes...
Build succeeded.build log

To use the custom messages, follow these steps:

1. Add the custom message folder to the MATLAB path by executing:

addpath('/home/robotics/ambf/ambf_ros_modules/matlab_msg_gen_ros1/glnxa64/install/m')
savepath

2. Refresh all message class definitions, which requires clearing the workspace, by executing:

clear classes
rehash toolboxcache

3. Verify that you can use the custom messages.
   Enter "rosmsg list" and ensure that the output contains the generated
   custom message types.

>>
```

If you have not already, run:

```
rosmsg list
```

and verify the following messages are available. We will be using the RigidBodyCmd and RigidBodyState messages.



4. Publishing and Subscribing to topics

After custom messages have been created, it is possible to publish and subscribe to the AMBF nodes to communicate with the robot. Ensure ROS is running by running roscore in a separate terminal. Then initiate the robot you want to control in AMBF. Here we will run the Galen robot at 200Hz with fixed time steps. This will allow slower or less powerful machines to control the robot with less latency. If your machine can handle the processing while running AMBF and MATLAB this may not be a concern for you.

```
./ambf_simulator --launch_file ~/Galen_AMBF_Simulation/ADF/launch.yaml -l 0 -p 200 -t 1
```

After starting ROS and AMBF, run rosinit in MATLAB to bridge the communication.
Now we can publish and subscribe to the ambf topics. The main topics we will work with are Command and State found at /ambf/env/Base/Command and /ambf/env/Base/State respectively. We can create a subscriber to control the joint variables and joint commands and other items with the following command:

```
galen_cmd = rospublisher('/ambf/env/Base/Comamnd','ambf_msgs/RigidBodyCmd')
```

The following should appear and now galen_cmd has been assigned your publisher.

```
>> galen_cmd = rospublisher('/ambf/env/Base/Command','ambf_msgs/RigidBodyCmd')

galen_cmd =

  Publisher with properties:

         TopicName: '/ambf/env/Base/Command'
    NumSubscribers: 0
        IsLatching: 1
       MessageType: 'ambf_msgs/RigidBodyCmd'
        DataFormat: 'object'
```

Creating a subscriber is much the same. Type the following command.

galen_sub = rossubscriber('/ambf/env/Base/State','ambf_msgs/RigidBodyCmd')

The following should appear and now galen_sub has been assigned your subscriber.

```
>> galen_state_sub = rossubscriber('/ambf/env/Base/State','ambf_msgs/RigidBodyState')

galen_state_sub =

  Subscriber with properties:

        TopicName: '/ambf/env/Base/State'
    LatestMessage: [0×1 RigidBodyState]
      MessageType: 'ambf_msgs/RigidBodyState'
       BufferSize: 1
    NewMessageFcn: []
       DataFormat: 'object'
```

We can now send messages to the publisher and receive messages from the subscriber. With the latest MATLAB distributions we can also include 2 other inputs to the creation of publishers and subscribers. If we know the data type to expect from the publisher and subscriber nodes then we can initiate these within their creation to help speed up the message communication time as follows. Create your publisher and subscribers as follows.

galen_cmd=rospublisher('/ambf/env/Base/Command','ambf_msgs/RigidBodyCmd',"DataFormat","struct");

joint_sub = rossubscriber('/ambf/env/Base/State','ambf_msgs/RigidBodyState',"DataFormat","struct");

When the data type is used as shown above, "struct", anytime messages are sent, you must ensure that the data type you send or receive using messages is of the correct type. Most MATLAB variables come in the form of doubles, but the AMBF joint_cmds expects singles, and AMBF joint_cmds_types expects int8's.

5. Sending and Receiving Messages

To send messages or receive messages from the publishers and subscribers we first need to create a rosmessage. Use the following command to create the message:

galen_msg = rosmessage('ambf_msgs/RigidBodyCmd')

This will show the different message items, and their expected types, which we can use to control various aspects of the robot. We will mainly use JointCmdsTypes and JointCmds

```
>> galen_msg = rosmessage('ambf_msgs/RigidBodyCmd')

galen_msg =

  ROS RigidBodyCmd message with properties:

              MessageType: 'ambf_msgs/RigidBodyCmd'
                TYPEFORCE: 0
             TYPEPOSITION: 1
             TYPEVELOCITY: 2
                   Header: [1×1 Header]
                     Pose: [1×1 Pose]
                   Wrench: [1×1 Wrench]
                    Twist: [1×1 Twist]
         CartesianCmdType: 0
           JointCmdsTypes: [0×1 int8]
                JointCmds: [0×1 single]
     PublishChildrenNames: 0
        PublishJointNames: 0
    PublishJointPositions: 0

  Use showdetails to show the contents of the message
```

Notice the TYPEFORCE, TYPEPOSITION, and TYPEVELOCITY values. These indicate the values we can input into an array of length equal to the number of joints of our robot (in the galen case this is 6) and tell the robot whether to control the joints with either force, position, or velocity. We will use velocity here so we will create an 6x1 array of 2's. [2,2,2,2,2,2].
We will assign this value as an int8 to the galen_msg.JointCmdsTypes property with this command:

```
galen_msg.JointCmdsTypes = int8([2,2,2,2,2,2])
```

Now we can send a message using our publisher. With the publisher called galen_cmd, we can send any information we've added to galen_msg to the robot with the following command.

```
send(galen_cmd,galen_msg)
```

Now the robot knows any input commands to JointCmds will use the values as reference velocity commands.
Input a velocity command on the robot by entering the following:

```
galen_msg.JointCmds = single([1,1,1,0.5,0,0])
```

Now send this change to the robot:

```
send(galen_cmd,galen_msg)
```

Notice the robot move some joints and others stay. The robot joints move according to the command given in the array.
We can receive the current robot state information similarly by receiving a message from the subscriber we made earlier, galen_sub. Type:

```
galen_true_state = receive(galen_sub)
```

And we can see the following output.

```
>> galen_true_state = receive(galen_state_sub)

galen_true_state =

  ROS RigidBodyState message with properties:

        MessageType: 'ambf_msgs/RigidBodyState'
             Header: [1×1 Header]
         Identifier: [1×1 String]
               Name: [1×1 String]
           PInertia: [1×1 Point]
               Pose: [1×1 Pose]
              Twist: [1×1 Twist]
             Wrench: [1×1 Wrench]
            SimStep: 1234115
           WallTime: 1.5652e+03
            SimTime: 1.5652e+03
               Mass: 0
      ChildrenNames: {6×1 cell}
         JointNames: {6×1 cell}
     JointPositions: [6×1 single]
    JointVelocities: [6×1 single]
       JointEfforts: [6×1 single]

  Use showdetails to show the contents of the message
```

Here we can further access and observe the force, velocity, or positions with JointEfforts, JointPositions, or JointVelocities. We will look at the current velocity by typing the following after receiving the subscriber.

```
Current_velocity = galen_true_state.JointVelocities
```

And the velocity will be assigned to Current_velocity.

We will have to continually send messages to move the robot and continually receive messages to update the current velocity in a loop to have accurate information, but now we have a way of sending commands to the robot and receiving the robot state.

Finally, similar to how we can make publishers and subscribers more efficient by identifying the data type on initiation, we can do the same to the message initiation with the following:

```
galen_command_message = rosmessage('ambf_msgs/RigidBodyCmd',"DataFormat","struct");
```

And we can send and receive commands just the same as we did before.