

Robotics and Intelligent Systems

RIS Lab I - Fall 2022

Tea Stefanovska
Razvan Andrei Perial
Cristian-Mihai Stratulat

November 2022

Contents

1	Nodes, Topics, Services	2
2	Controlling the robot	6
2.1	LaunchFile	6
2.2	Rosbag	7
3	Creating Nodes	11
3.1	Creating a Node	11
3.2	Creating a launch file	12
4	Creating our own robot	12
4.1	Creating the urdf	12
4.2	Placement of the thrusters	12
4.3	Writing the node for generating thrust commands for our ROV	13
4.4	Creating and controlling our own robot	14
5	Additional information	16
5.1	Contributors	16
5.2	Github Repository	16

1 Nodes, Topics, Services

The task requires launching `rexrov_default.launch` file, found in `uuv_simulator/uuv_gazebo/launch/rexrov_demo/`, and inspecting the nodes, topics, services and messages that are being started. We can do this by executing the command `roslaunch rexrov_default.launch`. Launching the file will bring up a RVIZ simulation of a robot. The robot contains some components such as thrusters, sonar devices and a camera. The first step that needs to be done in order to have a clear understanding about what is happening when executing the launch file is to view a list of all the active nodes. A *node* is a single-purposed executable program that performs different computations, managed in packages. These programs can either publish or subscribe to a *topic*. We can view all active nodes using the command `rosnode list`. The command will produce the following output:

```
cristiutz@cristiutz-QEMU-Virtual-Machine:~/catkin_ws/src/uuv_simulator/uuv_gazebo/launch/rexrov_demos$ rosnode list
/rexrov/acceleration_control
/rexrov/ground_truth_to_tf_rexrov
/rexrov/joy_uuv_velocity_teleop
/rexrov/robot_state_publisher
/rexrov/thruster_allocator
/rexrov/urdf_spawner
/rexrov/velocity_control
/rosout
/rviz
```

Figure 1: List of nodes launched by the `rexrov_default.launch` file

From the figure above we see that there are 9 nodes that are being launched. However, this is all the information that we have so far. As mentioned about nodes can publish or subscribe to different topics. *Topics* are channels for communication between nodes. Publishing nodes are posting messages over a topic, while subscriber nodes are receiving the messages published to a topic. By using the command `rostopic list` we will be able to view all active topics. Executing the instruction will result in the following output:

```
cristiutz@cristiutz-QEMU-Virtual-Machine:~/catkin_ws/src/uuv_simulator/uuv_gazebo/launch/rexrov_demos$ rostopic list
/clicked_point
/initialpose
/move_base_simple/goal
/rexrov/cmd_accel
/rexrov/cmd_force
/rexrov/cmd_vel
/rexrov/current_velocity_marker
/rexrov/current_velocity_marker_array
/rexrov/dvl_sonar0
/rexrov/dvl_sonar1
/rexrov/dvl_sonar2
/rexrov/dvl_sonar3
/rexrov/ground_truth_to_tf_rexrov/euler
/rexrov/ground_truth_to_tf_rexrov/pose
/rexrov/home_pressed
/rexrov/joint_states
/rexrov/joy
/rexrov/pose_gt
/rexrov/rexrov/camera/camera_image
/rexrov/thruster_manager/input
/rexrov/thruster_manager/input_stamped
/rexrov/thrusters/0/input
/rexrov/thrusters/1/input
/rexrov/thrusters/2/input
/rexrov/thrusters/3/input
/rexrov/thrusters/4/input
/rexrov/thrusters/5/input
/rexrov/thrusters/6/input
/rexrov/thrusters/7/input
/rexrov/velocity_control/parameter_descriptions
/rexrov/velocity_control/parameter_updates
/rosout
/rosout_agg
/tf
/tf_static
```

Figure 2: List topics launched by the `rexrov_default.launch` file

Before mapping the relationship between nodes and topics, there are 2 more ROS concepts that we need to consider: *messages* and *services*, such that the final analysis to be complete.

Messages are the structures of data that is passed between nodes. There are several standard message types (ex: std_msgs, geometry_msgs, nav_msgs and sensor_msgs), but custom messages can be defined as well. Custom messages can be defined in .msg files stored in *jpkg_j/msg* directory. When executing the command *rosmsg list* we can see all the data that is being passed between nodes. In the output bellow, we can see that there are standard types passed, but as well there are custom messages passed.

```

actionlib/TestAction
actionlib/TestActionFeedback
actionlib/TestActionGoal
actionlib/TestActionResult
actionlib/TestFeedback
actionlib/TestGoal
actionlib/TestRequestAction
actionlib/TestRequestActionFeedback
actionlib/TestRequestActionGoal
actionlib/TestRequestActionResult
actionlib/TestRequestFeedback
actionlib/TestRequestGoal
actionlib/TestRequestResult
actionlib/TestResult
actionlib/TwoIntsAction
actionlib/TwoIntsActionFeedback
actionlib/TwoIntsActionGoal
actionlib/TwoIntsActionResult
actionlib/TwoIntsFeedback
actionlib/TwoIntsGoal
actionlib/TwoIntsResult
actionlib_msds/GoalID
actionlib_msds/GoalStatus
actionlib_msds/GoalStatusArray
actionlib_tutorials/AveragingAction
actionlib_tutorials/AveragingActionFeedback
actionlib_tutorials/AveragingActionGoal
actionlib_tutorials/AveragingActionResult
actionlib_tutorials/AveragingFeedback
actionlib_tutorials/AveragingGoal
actionlib_tutorials/AveragingResult
actionlib_tutorials/FibonacciAction
actionlib_tutorials/FibonacciActionFeedback
actionlib_tutorials/FibonacciActionGoal
actionlib_tutorials/FibonacciActionResult
actionlib_tutorials/FibonacciFeedback
actionlib_tutorials/FibonacciGoal
actionlib_tutorials/FibonacciResult
bond/Constants
bond/Status
control_msds/FollowJointTrajectoryAction
control_msds/FollowJointTrajectoryActionFeedback
control_msds/FollowJointTrajectoryActionGoal
control_msds/FollowJointTrajectoryActionResult
control_msds/FollowJointTrajectoryFeedback
control_msds/FollowJointTrajectoryGoal
control_msds/FollowJointTrajectoryResult
control_msds/GripperCommand
control_msds/GripperCommandAction
control_msds/GripperCommandActionFeedback
control_msds/GripperCommandActionGoal
control_msds/GripperCommandActionResult
control_msds/GripperCommandFeedback
control_msds/GripperCommandGoal
control_msds/GripperCommandResult
control_msds/JointControllerState
control_msds/JointJog
control_msds/JointTolerance
control_msds/JointTrajectoryAction
control_msds/JointTrajectoryActionFeedback
control_msds/JointTrajectoryActionGoal
control_msds/JointTrajectoryActionResult
control_msds/JointTrajectoryControllerState
control_msds/JointTrajectoryFeedback
control_msds/JointTrajectoryGoal
control_msds/JointTrajectoryResult
control_msds/PidState
control_msds/PointHeadAction
control_msds/PointHeadActionFeedback
control_msds/PointHeadActionGoal
control_msds/PointHeadActionResult
control_msds/PointHeadFeedback
control_msds/PointHeadGoal
control_msds/PointHeadResult
control_msds/SingleJointPositionAction
control_msds/SingleJointPositionActionFeedback
control_msds/SingleJointPositionActionGoal
control_msds/SingleJointPositionActionResult
control_msds/SingleJointPositionFeedback
control_msds/SingleJointPositionGoal
control_msds/SingleJointPositionResult
controller_manager_msds/Controllerstate
controller_manager_msds/ControllerStatistics
controller_manager_msds/ControllersStatistics
controller_manager_msds/HardwareInterfaceResources
diagnostic_msds/DiagnosticArray
diagnostic_msds/DiagnosticStatus
diagnostic_msds/KeyValue
dynamic_reconfigure/BoolParameter
dynamic_reconfigure/Config
dynamic_reconfigure/ConfigDescription
dynamic_reconfigure/DoubleParameter
dynamic_reconfigure/Group
dynamic_reconfigure/GroupState
dynamic_reconfigure/IntParameter
dynamic_reconfigure/ParamDescription
dynamic_reconfigure/SensorLevels
dynamic_reconfigure/StringParameter
gazebo_msds/ContactState
gazebo_msds/ContactsState
gazebo_msds/LinkState
gazebo_msds/LinkStates
gazebo_msds/ModelState
gazebo_msds/ModelStates
gazebo_msds/ODEJointProperties
gazebo_msds/ODEPhysics
gazebo_msds/PerformanceMetrics
gazebo_msds/SensorPerformanceMetric
gazebo_msds/Worldstate
geometry_msds/Accel
geometry_msds/AccelStamped
geometry_msds/AccelWithCovariance
geometry_msds/AccelWithCovarianceStamped
geometry_msds/Inertia
geometry_msds/InertiaStamped
geometry_msds/Point
geometry_msds/Point32
geometry_msds/PointStamped
geometry_msds/Polygon
geometry_msds/PolygonStamped
geometry_msds/Pose
geometry_msds/Pose2D
geometry_msds/PoseArray
geometry_msds/PoseStamped
geometry_msds/PoseWithCovariance
geometry_msds/PoseWithCovarianceStamped
geometry_msds/Quaternion
geometry_msds/QuaternionStamped
geometry_msds/Transform
geometry_msds/TransformStamped
geometry_msds/Twist
geometry_msds/TwistStamped
geometry_msds/TwistWithCovariance
geometry_msds/TwistWithCovarianceStamped
geometry_msds/Vector3
geometry_msds/Vector3Stamped
geometry_msds/Wrench
geometry_msds/WrenchStamped
map_msds/OccupancyGridUpdate
map_msds/PointCloud2Update
map_msds/ProjectedMap
map_msds/ProjectedMapInfo
nav_msds/GetMapAction
nav_msds/GetMapActionFeedback
nav_msds/GetMapActionGoal
nav_msds/GetMapActionResult
nav_msds/GetMapFeedback
nav_msds/GetMapGoal
nav_msds/GetMapResult
nav_msds/GridCells
nav_msds/MapMetaData
nav_msds/OccupancyGrid
nav_msds/Odometry
nav_msds/Path
pcl_msds/ModelCoefficients
pcl_msds/PointIndices
pcl_msds/PolygonMesh

```

Figure 3: First part of the list of messages

```

pcl_msgs/Vertices
roscpp/Logger
rosgraph_msgs/Clock
rosgraph_msgs/Log
rosgraph_msgs/TopicStatistics
rospy_tutorials/Floats
rospy_tutorials/HeaderString
sensor_msgs/BatteryState
sensor_msgs/CameraInfo
sensor_msgs/ChannelFloat32
sensor_msgs/CompressedImage
sensor_msgs/FluidPressure
sensor_msgs/Illuminance
sensor_msgs/Image
sensor_msgs/Imu
sensor_msgs/JointState
sensor_msgs/Joy
sensor_msgs/JoyFeedback
sensor_msgs/JoyFeedbackArray
sensor_msgs/LaserEcho
sensor_msgs/LaserScan
sensor_msgs/MagneticField
sensor_msgs/MultiDOFJointState
sensor_msgs/MultiEchoLaserScan
sensor_msgs/NavSatFix
sensor_msgs/NavSatStatus
sensor_msgs/PointCloud
sensor_msgs/PointCloud2
sensor_msgs/PointField
sensor_msgs/Range
sensor_msgs/RegionOfInterest
sensor_msgs/RelativeHumidity
sensor_msgs/Temperature
sensor_msgs/TimeReference
shape_msgs/Mesh
shape_msgs/MeshTriangle
shape_msgs/Plane
shape_msgs/SolidPrimitive
smach_msgs/SmachContainerInitialStatusCmd
smach_msgs/SmachContainerStatus
smach_msgs/SmachContainerStructure
std_msgs/Bool
std_msgs/Byte
std_msgs/ByteMultiArray
std_msgs/Char
std_msgs/ColorRGBA
std_msgs/Duration
std_msgs/Empty
std_msgs/Float32
std_msgs/Float32MultiArray
std_msgs/Float64
std_msgs/Float64MultiArray
std_msgs/Header
std_msgs/Int16
std_msgs/Int16MultiArray
std_msgs/Int32
std_msgs/Int32MultiArray
std_msgs/Int64
std_msgs/Int64MultiArray
std_msgs/Int8
std_msgs/Int8MultiArray
std_msgs/MultiArrayDimension
std_msgs/MultiArrayLayout
std_msgs/String
std_msgs/Time
std_msgs/UInt16
std_msgs/UInt16MultiArray
std_msgs/UInt32
std_msgs/UInt32MultiArray
std_msgs/UInt64
std_msgs/UInt64MultiArray
std_msgs/UInt8
std_msgs/UInt8MultiArray
stereo_msgs/DisparityImage
tf/tfMessage
tf2_msgs/LookupTransformAction
tf2_msgs/LookupTransformActionFeedback
tf2_msgs/LookupTransformActionGoal
tf2_msgs/LookupTransformActionResult
tf2_msgs/LookupTransformFeedback
tf2_msgs/LookupTransformGoal
tf2_msgs/LookupTransformResult
tf2_msgs/TF2Error
tf2_msgs/TFMessage
theora_image_transport/Packet
trajectory_msgs/JointTrajectory
trajectory_msgs/JointTrajectoryPoint
trajectory_msgs/MultiDOFJointTrajectory
trajectory_msgs/MultiDOFJointTrajectoryPoint
turtle_actionlib/ShapeAction
turtle_actionlib/ShapeActionFeedback
turtle_actionlib/ShapeActionGoal
turtle_actionlib/ShapeActionResult
turtle_actionlib/ShapeFeedback
turtle_actionlib/ShapeGoal
turtle_actionlib/ShapeResult
turtle_actionlib/Velocity
turtlesim/Color
turtlesim/Pose
visualization_msgs/ImageMarker
visualization_msgs/InteractiveMarker
visualization_msgs/InteractiveMarkerControl
visualization_msgs/InteractiveMarkerFeedback
visualization_msgs/InteractiveMarkerInit
visualization_msgs/InteractiveMarkerPose
visualization_msgs/InteractiveMarkerUpdate
visualization_msgs/Marker
visualization_msgs/MarkerArray
visualization_msgs/MenuEntry

```

Figure 4: Second part of the list of messages

Services implement a request/response mechanism for inter-node communication. Topics are a one-way communication channel, where a publisher is putting data on the topic, and the subscriber is reading the data, but it doesn't go the other way around. Services provides this 2 way inter-node communication, and are generally used for infrequent signals at the node/ asking for some calculation. Service types are generally defined in .srv files. Executing the command `rossrv list` we get all services used in our simulation.

```

control_msgs/QueryCalibrationState
control_msgs/QueryTrajectoryState
control_toolbox/SetPldGains
controller_manager_msgs/ListControllerTypes
controller_manager_msgs/ListControllers
controller_manager_msgs/LoadController
controller_manager_msgs/ReloadControllerLibraries
controller_manager_msgs/SwitchController
controller_manager_msgs/UnloadController
diagnostic_msgs/AddDiagnostics
diagnostic_msgs/SelfTest
dynamic_reconfigure/Reconfigure
gazebo_msgs/ApplyBodyWrench
gazebo_msgs/ApplyJointEffort
gazebo_msgs/BodyRequest
gazebo_msgs/DeleteLight
gazebo_msgs/DeleteModel
gazebo_msgs/GetJointProperties
gazebo_msgs/GetLightProperties
gazebo_msgs/GetLinkProperties
gazebo_msgs/GetModelState
gazebo_msgs/GetModelProperties
gazebo_msgs/GetPhysicsProperties
gazebo_msgs/GetWorldProperties
gazebo_msgs/JointRequest
gazebo_msgs/SetJointProperties
gazebo_msgs/SetJointTrajectory
gazebo_msgs/SetLightProperties
gazebo_msgs/SetLinkProperties
gazebo_msgs/SetLinkState
gazebo_msgs/SetModelConfiguration
gazebo_msgs/SetModelState
gazebo_msgs/SetPhysicsProperties
gazebo_msgs/SpawnModel
laser_assembler/AssembleScans
laser_assembler/AssembleScans2
map_msgs/GetMapROI
map_msgs/GetPointMap
map_msgs/GetPointMapROI
map_msgs/ProjectedMapsInfo
map_msgs/SaveMap
map_msgs/SetMapProjections
nav_msgs/GetMap
nav_msgs/GetPlan
nav_msgs/LoadMap
nav_msgs/SetMap
nodelet/NodeletList
nodelet/NodeletLoad
nodelet/NodeletUnload
pcl_msgs/UpdateFilename

polled_camera/GetPolledImage
roscpp/Empty
roscpp/GetLoggers
roscpp/SetLoggerLevel
roscpp_tutorials/TwoInts
rospy_tutorials/AddTwoInts
rospy_tutorials/BadTwoInts
rviz/SendFilePath
sensor_msgs/SetCameraInfo
std_srvs/Empty
std_srvs/SetBool
std_srvs/Trigger
tf/FrameGraph
tf2_msgs/FrameGraph
topic_tools/DemuxAdd
topic_tools/DemuxDelete
topic_tools/DemuxList
topic_tools/DemuxSelect
topic_tools/MuxAdd
topic_tools/MuxDelete
topic_tools/MuxList
topic_tools/MuxSelect
turtlesim/Kill
turtlesim/SetPen
turtlesim/Spawn
turtlesim/TeleportAbsolute
turtlesim/TeleportRelative

```

Figure 5: List of services

After taking into account all the information above, we can start mapping out the relation between nodes. We can queue each node for information using `rosnode info name_of_the_node`. For example if we want to get all the information regarding the `acceleration_control` node we execute the command above and get the following output:

```

Node [/rexrov/acceleration_control]
Publications:
  * /rexrov/thruster_manager/input [geometry_msgs/Wrench]
  * /rosout [rosgraph_msgs/Log]

Subscriptions:
  * /rexrov/cmd_accel [geometry_msgs/Accel]
  * /rexrov/cmd_force [unknown type]

Services:
  * /rexrov/acceleration_control/get_loggers
  * /rexrov/acceleration_control/set_logger_level

```

Figure 6: Information about node acceleration_control

Some nodes that are remarkable would be the `velocity_control` node and the `acceleration_control` node. Together with the `uuv_velocity_teleop` and `thruster_allocator` they form the longest inter-node communication from this simulation.

`Velocity_control` node is subscribed to `cmd_vel` topic, receiving messages of type `geometry_msgs`,

and is publishing to cmd_accel topic a message of type geometry_msgs. As already noticed above, acceleration_control node is subscribed to the cmd_accel topic and is further publishing data to thruster_manager/input topic. As well, this node is having 2 services, get_loggers and set_loggers. From the way the mentioned nodes are defined and their relationships we can deduce that they are responsible for controlling the movement of the robot (velocity/acceleration).

We can repeat the step above for all node, and that would lead us to the complete analysis. Nonetheless ROS provides us with the option of graphing everything together. By running the command `rosrun rqt_graph rqt_graph`, we will create a visual diagram that would combine all the information above.

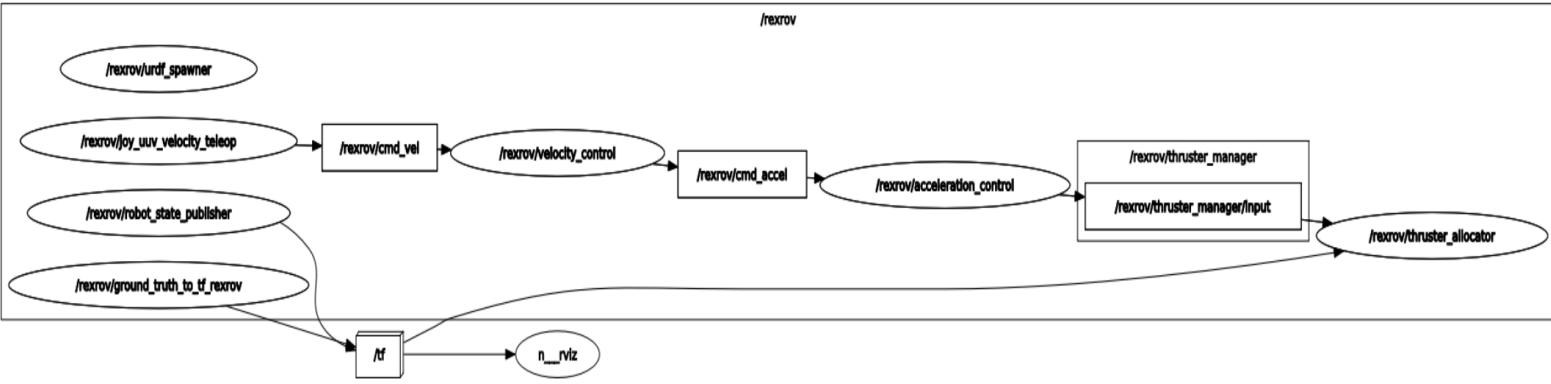


Figure 7: Full analysis generated by rqt_graph

2 Controlling the robot

Since we have already gained a deeper understanding of the nodes and topics running when the rexrov is launched using the `rexrov_default.launch` file, it is time for us to do something more interesting and more dynamic. Therefore, we will now explore how to launch the rexrov in an underwater world, and control its movements in the water.

In this section we will cover two ways of reaching our goal (moving the rexrov) - by writing a launch file that simultaneously spawns the underwater world, the rexrov and our keyboard control node, and by feeding/reproducing the rexrov movements previously recorded using rosbag.

2.1 LaunchFile

As mentioned, we will be writing a launch file, as its nature allows us to simultaneously start groups of nodes, launch other launch files and pass parameters to them, while also starting a master in the background. They are a powerful tool which replaces the labour of manually running nodes for non-trivial codes.

We wrote a launch file that launches three launch files simultaneously in order to achieve our goal: moving the robot under water. By using the ROS tool `find` we are able to locate the packages which contain our targeted launch files. We firstly include the world spawned by `uuv_gazebo_worlds/empty_underwater_world.launch` in order to launch the underwater environment, so we can later on spawn the rexrov inside it. Then, we include the `uuv_gazebo/rexrov_default.launch` file to spawn the rexrov, and last but not least (since `teleop_twist_keyboard` is not present in the

uuv_simulator directory) we include the *uuv_teleop/uuv_keyboard_teleop.launch* file in order to move the robot.

Of course, we also explored the possibility of installing the *teleop_twist_keyboard* package that contains the corresponding node, which we can invoke using the *node* tag and which output needs to be remapped to the topic */rexrov/cmd_vel*. For simplicity we decided to use the preexisting *uuv_keyboard_teleop.launch* file as the mapping is already taken care of.

The launch file is the following (*launcher.launch*):

```
<?xml version="1.0"?>

<launch>

  <include file ="$( find uuv_gazebo_worlds)/launch/empty_underwater_world.launch">
    <arg name="paused" value="false"/>
  </include>

  <include file ="$(find uuv_gazebo)/launch/rexrov_demos/rexrov_default.launch">
    <arg name = "namespace" value = "rexrov"/>
  </include>

  <include file ="$(find uuv_teleop)/launch/uuv_keyboard_teleop.launch">
    <arg name = "uuv_name" value = "rexrov"/>
  </include>

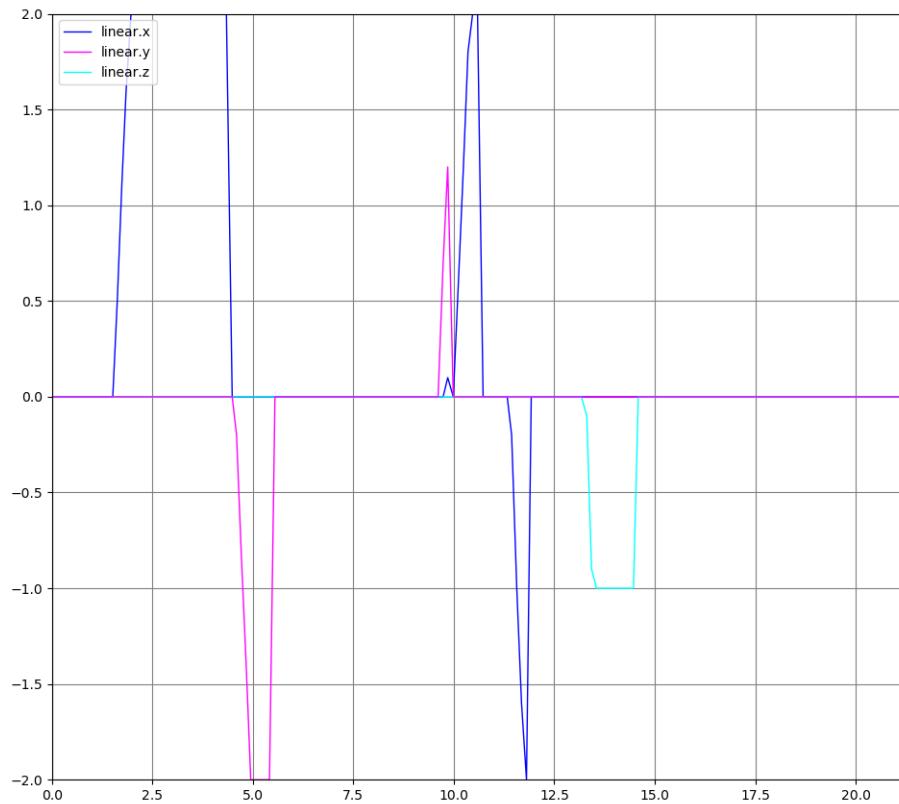
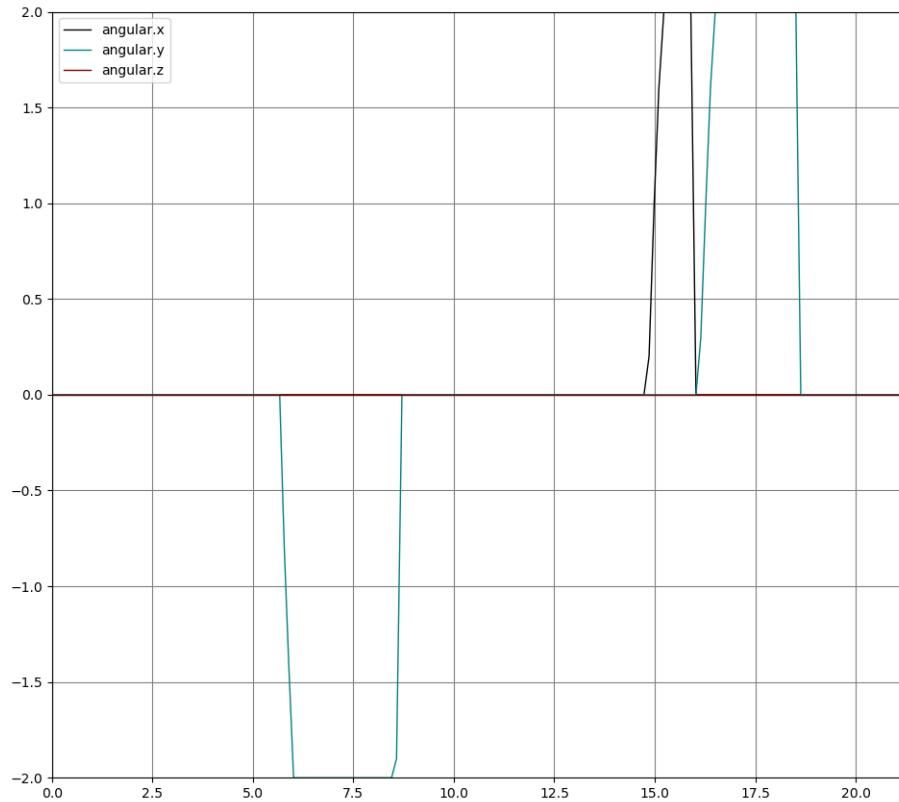
</launch>
```

Once the launch file is launched using the *roslaunch* command in the terminal, we can see the gazebo environment with the rexrov inside, while rviz also boots up. To control the robot, the terminal window running the launch file should be in focus and we can move the robot by pressing keys on our keyboard (w,s,a,d,z,x for position and i,k,j,l,q,l for orientation).

2.2 Rosbag

Moving forward, we moved our robot around with the corresponding keyboard keys. At the same time, we logged the topic *rexrov/cmd_vel* using *rosbag* to record. After that we stopped the keyboard control to let it replay the path it recorded.

We created an *rqt_plot* of the corresponding topic that we recorded with *rosbag* (*cmd_vel*), using *rqt_bag*, and plotted the position of the rexrov from the recording. To track the trajectory of the robot, we opened the rexrov's pose topic in gazebo and plotted its position and orientation axes, while the rosbag was playing in the background. The resulting plots can be seen below.

Figure 8: Linear velocities of the robot, plotted using `rqt_bag` in plotting modeFigure 9: Angular velocities of the robot, plotted using `rqt_bag` in plotting mode

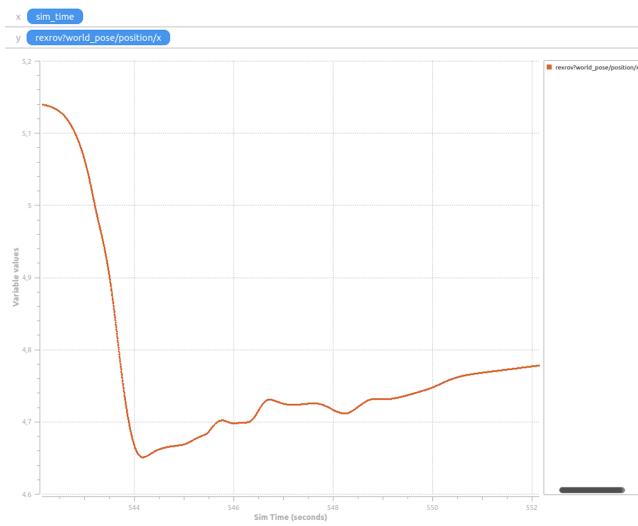


Figure 10: Position of rexrov - x axis

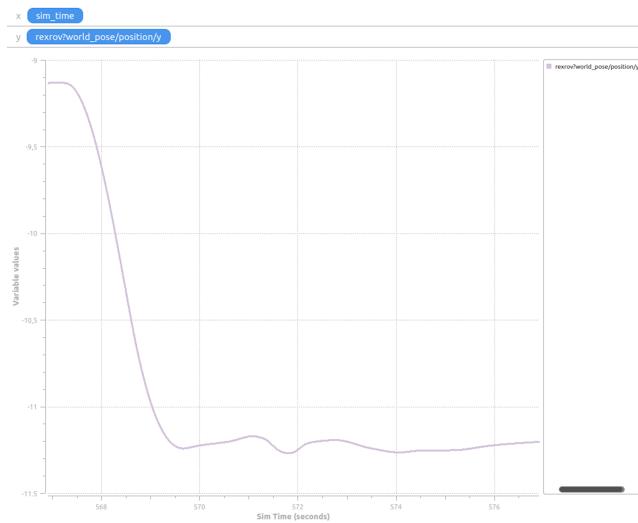


Figure 11: Position of rexrov - y axis

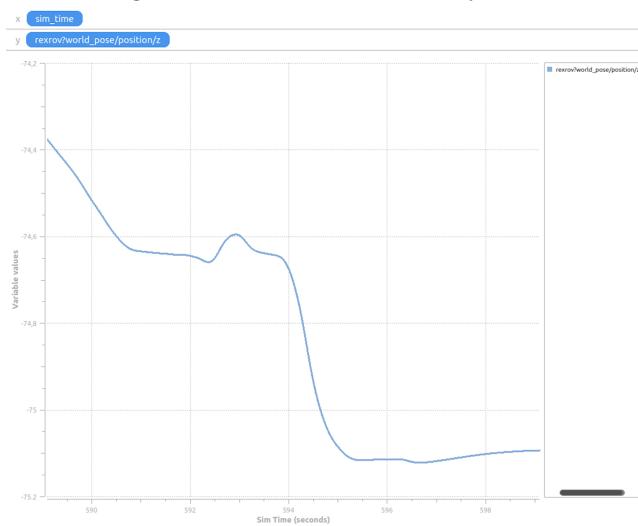


Figure 12: Position of rexrov - z axis

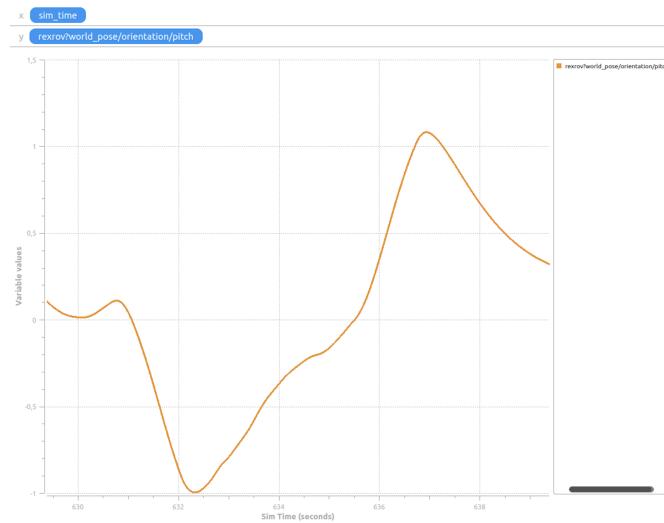


Figure 13: Orientation of rexrov - pitch

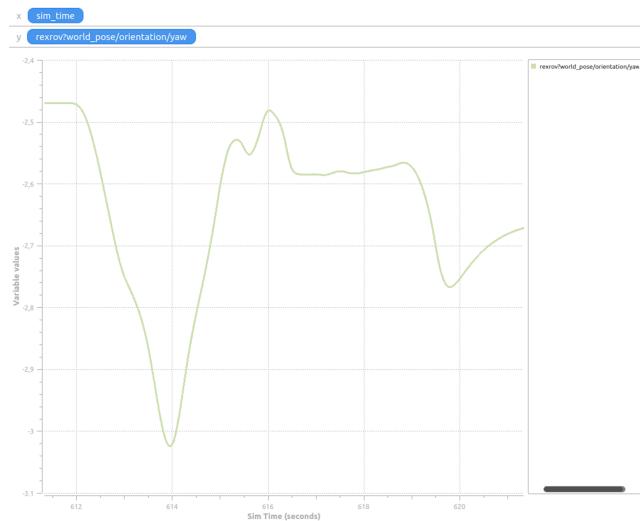


Figure 14: Orientation of rexrov - yaw

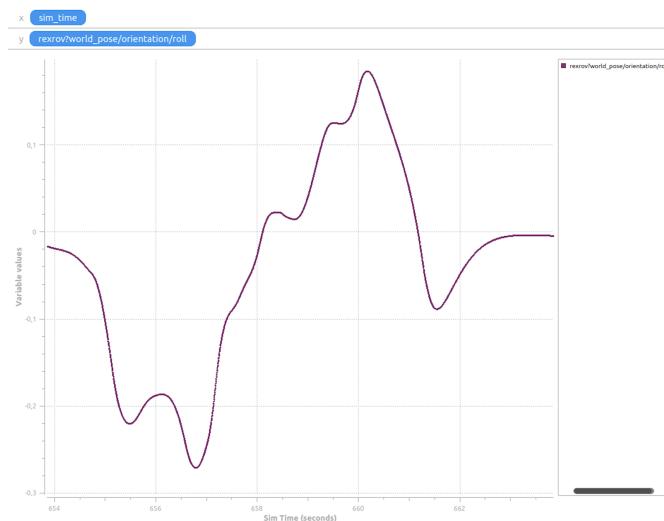


Figure 15: Orientation of rexrov - roll

Additionally we created an *rqt_graph* of the nodes and topics while running the keyboard control, and one after turning off the keyboard and playing the bag. This can be seen in the following graphs.

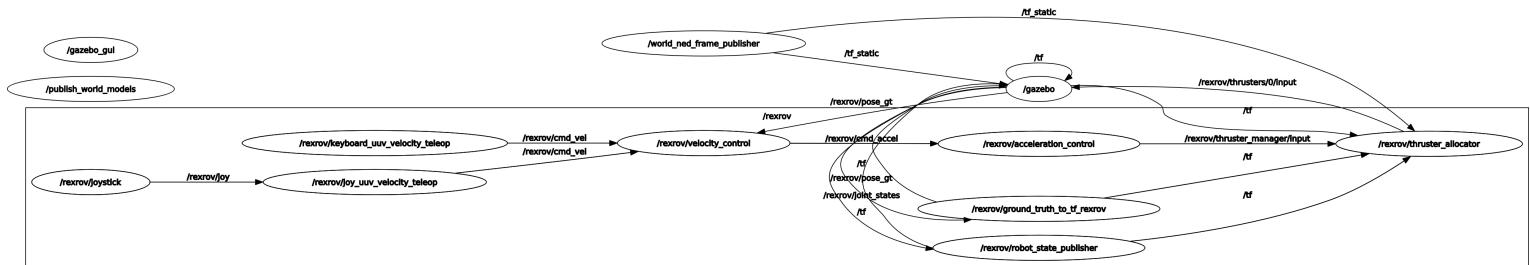


Figure 16: *rqt_graph* while keyboard control is running

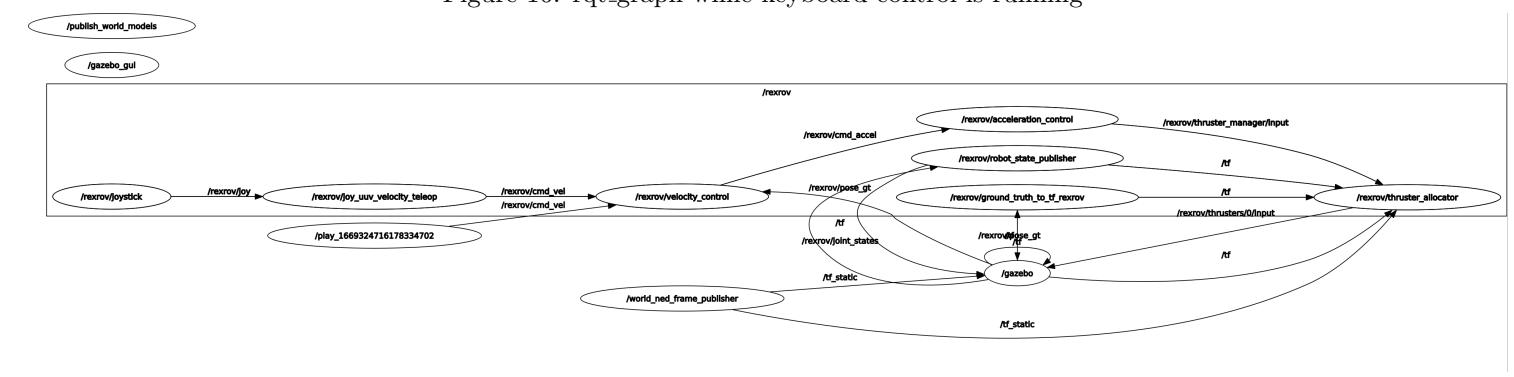


Figure 17: *rqt_graph* while rosbag play is running

3 Creating Nodes

This task guides us through the process of creating a node that outputs forces and torques as an input to control the AUV. The launch commands were executed as instructed and the AUV was successfully launched into the world.

3.1 Creating a Node

The vehicle_keyboard_teleop.py was used as a sample solution to writing a node that publishes the forces and torques to control the AUV. Using the *rosrun* command we ran the node and therefore the node successfully publishes to the topic: */rexrov/thruster_manager/input*.

Keyboard presses were being read off of the active terminal, the key strokes were turned into force-torque (wrench) values, which were published to the */rexrov/thruster_manager/input* topic. User can apply wrench in all directions, as well as change the magnitude of applied forces and torques using corresponding keyboard commands.

The corresponding node file can be found under the name *thruster_manager.py*.

3.2 Creating a launch file

A launch file similar to the one in exercise 2 was created and launched successfully (*thruster_manager_test.launch*). Empty underwater world with AUV was launched, followed by launch of the *thruster_manager.py* script, which spun up the respective node. Refer to the GitHub repository for launch file detailed documentation.

4 Creating our own robot

Finally, it was time for us to create our own robot, and launch it in the gazebo underwater world. In doing so, we used nodes that we have written in the previous tasks and one more node for taking in forces and torques as input and publishing thrust commands for our ROV.

4.1 Creating the urdf

For our URDF design we chose to go with a circular base using simple geometric shapes. We have a total of 12 links (including our base link) and 11 joints, tying everything to our base link. Each link contains a visual, collision and inertial tag, the last two being necessary for loading the robot into Gazebo.

- The *visual* tag takes care of how the specific link renders. Inside it, we can observe the use of tags such as *origin* (describes the position and orientation of the link's center of mass frame C relative to the link-frame L), *geometry* (describing the specific geometrical object used to define the link) and *material* (giving the link basic styling by using RBGA values).
- The *collision* tag describes the collision space for our objects. Like in the majority of the use-cases of the collision tag, we will want the geometry element in it to coincide with the one in the visual tag.
- Last but not least, the *inertial* tag, which contains mass and inertia tags. The inertial tag represents the 3x3 rotational inertia matrix which is specified with the *inertia* element. Since this is symmetrical, it can be represented by only 6 elements, as seen in our URDF model.

Once the individual links were created, we needed to join all of them with our *base link* using the *joint* tags.

As we can observe in the figures below, our robot has 7 thrusters (the motivation behind the number and placement will be discussed in the next section) and one fin, placed for stability purpose.

4.2 Placement of the thrusters

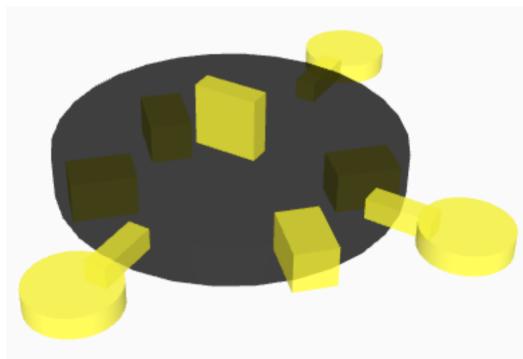


Figure 18: Visual representation of "Nessie", our ROV

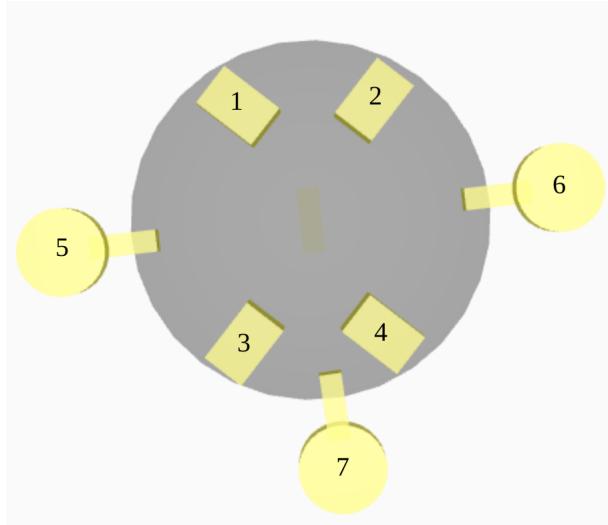


Figure 19: "Nessie" visual representation with numbered thrusters, for explanation reasons

When it came to deciding the most optimal placement of our thrusters, we decided to opt for a design that would provide 6 Degrees of Freedom for our ROV. As it can be observed in the two figures from above, we have placed 7 thrusters in total. The degrees of freedom are generated in the following manner:

- Thrusters numbered 1, 2, 3 and 4 give us the degrees of freedom on the X, Y and Z axis.
- Thrusters 4, 5 and 6 give us the degrees of freedom for the *Yaw*, *Pitch* and *Roll* movements.

4.3 Writing the node for generating thrust commands for our ROV

For this task, we have made our own node using `rospy`, named `thrusters.py`. In this node, as mentioned in the assignment PDF as well, we are subscribing to the topic on which the node written in the third task publishes, namely `/rexrov/thruster_manager/input`.

Furthermore, taking in values for the forces and torques from the topic mentioned earlier, we are using a service named `/gazebo/apply_body_wrench` in order to make our conversions to thrust commands. Finding this specific service was of great help for our project, since it made the conversions much easier, having to take care only to pass the correct parameters to the `/gazebo/apply_body_wrench` call.

Below, you can observe how our node functions when called, and what it prints in two different states: the initial one and after we have pressed keyboard commands as input for the node written in part 3.

```

z: 0.0
[ INFO] [1669405813.432471, 121.254000]: /
thrustersTorques: x: 0.0
y: 0.0
z: 0.0
[ INFO] [1669405813.435866663, 121.258000000]: ApplyBody
Wrench: reference_frame is empty/world/map, using inertial frame, transferring from body rel
ative to inertial frame
[ INFO] [1669405813.531757, 121.354000]: /thrustersForces: x: 0.0
y: 0.
z: 0
[ INFO] [1669405813.533192, 121.356000]: /thrustersTorques: x: 0.0
y: 0.0
z: 0.0
[ INFO] [1669405813.536759065, 121.360000000]: ApplyBodyWrench: reference_frame is empty/world/map,
using inertial frame, transferring from body relative to inertial frame
[ INFO] [1669405813.63]
2910, 121.454000]: /thrustersForces: x: 0.0
y: 0.0
z: 0.0
[ INFO] [1669405813.637602, 121.458000]:
]: /thrustersTorques: x: 0.0
y: 0.0
z: 0.0
[ INFO] [1669405813.652531446, 121.474000000]: Apply
BodyWrench: reference_frame is empty/world/map, using inertial frame, transferring from body
relative to inertial frame
[ INFO] [1669405813.732770, 121.554000]: /thrustersForces: x: 0.0
y: 0.0
z: 0.0

```

Figure 20: Initial state of what our node "hears"

```

[ INFO] [1669405813.126897, 120.950000]: /thru
storsForces: x: 0.0
y: 2.0
z: 0.0
[ INFO] [1669405813.128085, 120.950000]: /thrustersTorques: x:
0.0
y: 0.0
z: 0.0
[ INFO] [1669405813.131531078, 120.954000000]: ApplyBodyWrench: reference_fr
ame is empty/world/map, using inertial frame, transferring from body relative to inertial fr
ame
[ INFO] [1669405813.157415, 120.980000]: /thrustersForces: x: 0.0
y: 2.0
z: 0.0
[ INFO] [1669405813.158710, 120.982000]: /thrustersTorques: x: 0.0
y: 0.0
z: 0.0
[ INFO] [1669405813.16204893
3, 120.984000000]: ApplyBodyWrench: reference_frame is empty/world/map, using inertial frame
, transferring from body relative to inertial frame
[ INFO] [1669405813.187623, 121.010000]: /
thrustersForces: x: 0.0
y: 2.0
z: 0.0
[ INFO] [1669405813.189277, 121.012000]: /thrustersTorques
: x: 0.0
y: 0.0
z: 0.0
[ INFO] [1669405813.192901339, 121.016000000]: ApplyBodyWrench: referenc
e_frame is empty/world/map, using inertial frame, transferring from body relative to inertia
l frame

```

Figure 21: Print statement after applying forces to the robot (see that force y value has changed)

4.4 Creating and controlling our own robot

Finally, it was time to create write a launch file for our own robot. Our launch file is named *nessie_robot.launch*.

We started off by adding arguments with default values. This made it possible for us to add specific values to the command line instructions when launching our file.

Next up, we have included the *empty_underwater_world* launch file which was found in the *uuv_simulator* Gazebo launch files. This created our empty underwater world in which our robot

will spawn. After this, we have defined our *robot_description* as a parameter, in order to make the conversion of our URDF model possible into the Gazebo world.

Last but not least, we have included the launch files mentioned in part 4.3 into the final launch file. This made it possible for us to move our robot, by applying forces and torques and converting them in order to be able to use the thrusters from the keyboard.

Below, we can see images of our robot spawned into the Gazebo underwater world.

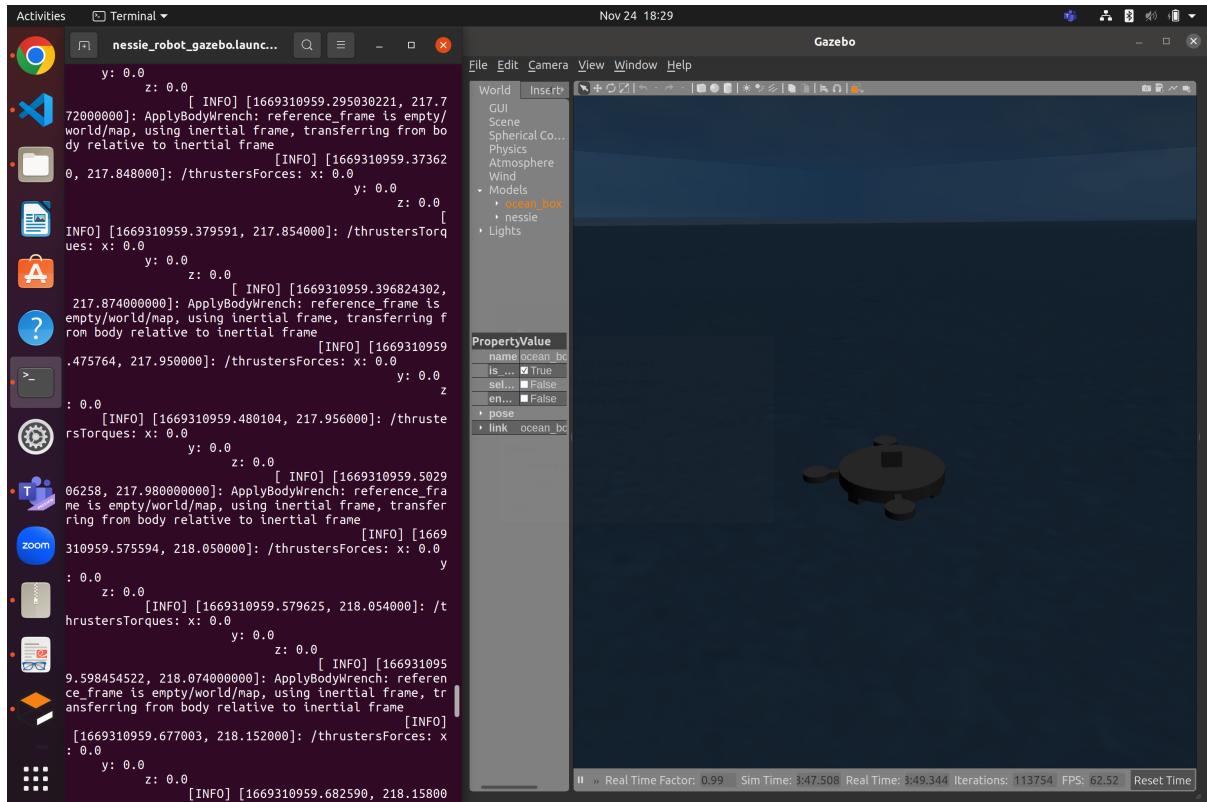


Figure 22: Nessie in underwater world, in state 0 (forces $x = y = z = 0$)

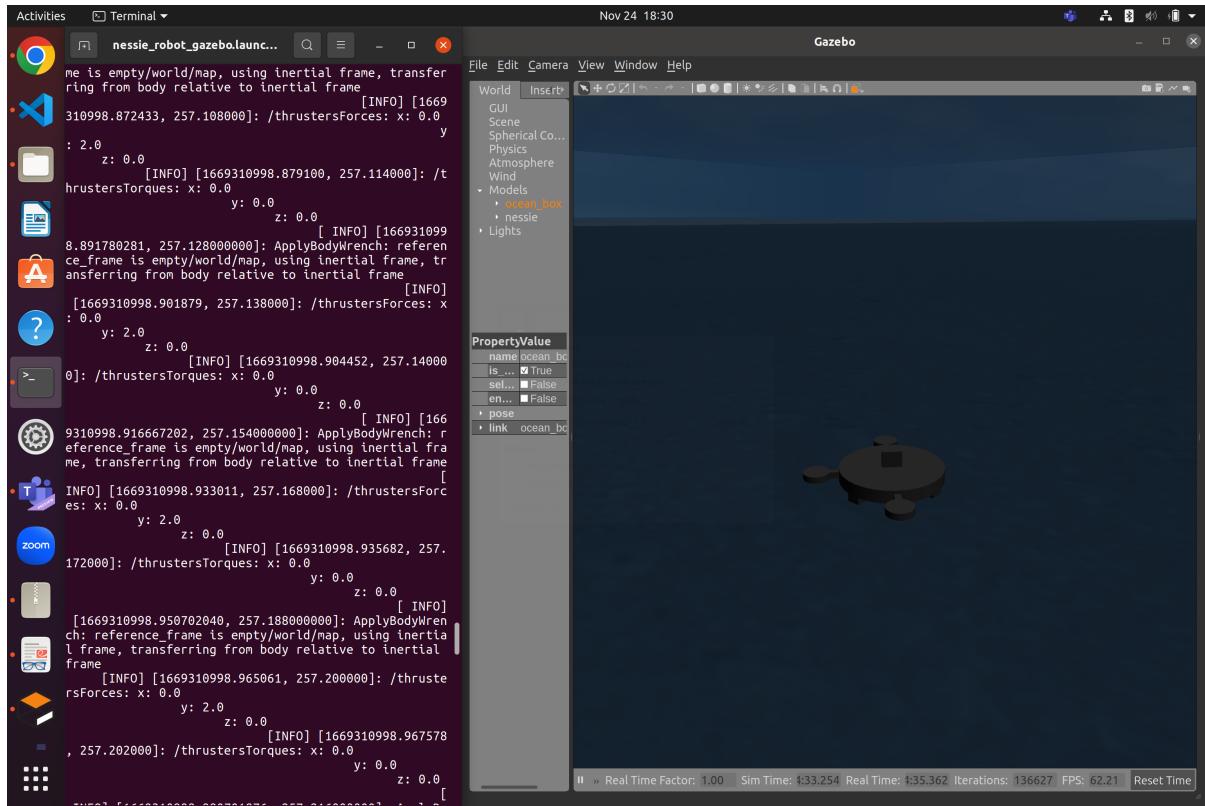


Figure 23: Nessie in underwater world, in state 0 (force y changes to value 2)

5 Additional information

5.1 Contributors

Part 1: a) and b) by Cristian-Mihai Stratulat - 4 hours

Part 2: a) and b) by Tea Stefanovska - 9 hours

Part 3: a) by Tea Stefanovska - 1 hour, and b) by Cristian-Mihai Stratulat - 1 hour

Part 4: a), b), c) and d) by Razvan Andrei Perial - 10 hours

Report writing by Cristian-Mihai Stratulat - 3 hours

5.2 GitHub Repository

RIS Lab 1 github repository