Curio Rover Behavior Trees Tutorial

Jadrina De Andrade E Silva Malchik
École des Mines de Saint-Étienne

Donal O'Donovan
Munster Technological University

August 2023

Contents

1 Introduction

This paper seeks to outline the process of setting up the Curio rover for both navigation and control via a behavior tree. We shall go through all the necessary steps required to set up the different components of the system including the network, the files, the behavior trees and the navigation stack. The tutorial like format shall allow the reader to follow along and hopefully end up with a fully functioning version that they can then expand upon to get closer to goal of turning the Curio rover into an autonomous tour guide. We shall also focus more on the code that was specifically created for this project since the navigation stack is well documented in the ROS wiki. Keep in mind that we shall be referring to other reports and papers throughout this tutorial in order to minimize the length of it. Make sure to read those when indicated.

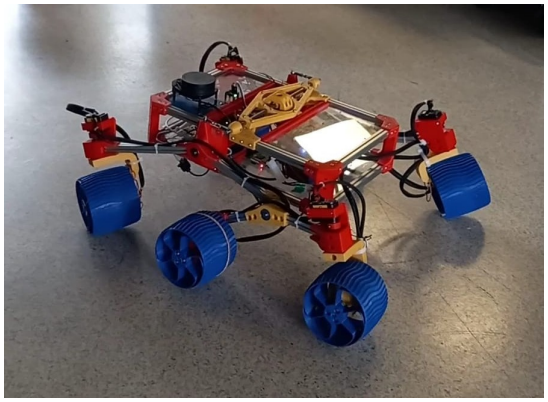

Figure 1: Ramona the Rover

2 Set-Up

2.1 Software

For the general set up we need a PC running Ubuntu 18.04, a Raspberry Pi 4 Model B and an Arduino Mega. If using an IMU then there needs to be an additional Raspberry Pi 3 Model B. We shall use ROS in order to connect all these things together. It is also necessary to have a router or mobile hotspot to create a local network. This network does not need to have internet access while you run the rover. You will need internet to download the packages.

First we will set up the PC. This will act as the master ROS node which will then orchestrate all the communication between the other nodes. Follow the instructions on the ROS wiki to install ROS in Ubuntu. Make sure to install the **melodic** distribution of ROS. Next follow the instructions on the Installation section in `https://github.com/srmainwaring/curio`. At this point you

should be able to run the curio simulation. Then install the necessary packages for behavior trees with the following commands:

```
    sudo apt install ros-melodic-py-trees
    sudo apt install ros-melodic-py-trees-ros
    sudo apt install ros-melodic-py-trees-msgs
    sudo apt install ros-melodic-rqt-py-trees
```

Usually to install a ROS package you can do so by following this template (any underscores in the name turn to dashes):

```
sudo apt install ros-{dist name}-{package name}
```

On a separate note the versions of py-trees and py-trees-ros that are compatible with ROS melodic are not the latest, they are version 0.6.x and 0.5.x respectively in case you want to download the package from GitHub. Therefore, when looking for information or tutorials use the following links: `http://docs.ros.org/en/melodic/api/py_trees/html/` `http://docs.ros.org/en/melodic/api/py_trees_ros/html/` We also downloaded a local version of py_trees_ros in order to be able to modify things. Go to curio_ws/src on the terminal and type in:

```
    git clone https://github.com/splintered-reality/py_trees_ros.
    git
    git checkout release/0.5.x
```

You also need the obstacle_detector package. Git clone it into your curio workspace from this GitHub repository: `https://github.com/tysik/obstacle_detector`. Make sure you repeat the catkin_make command whenever you install a new package through GitHub. After you do so go to the file called curio_navigation.launch and paste the following block of code after the block labeled move_base.

```
    <!-- obstacle detector -->
  <node name="nodelet_manager" pkg="nodelet" type="nodelet" args="
    manager" output="screen">
    <param name="num_worker_threads" value="20"/>
  </node>

  <node name="obstacle_extractor" pkg="nodelet" type="nodelet" args
    ="load obstacle_detector/ObstacleExtractor nodelet_manager">
    <remap from="scan" to="sensors/laser" />
    <param name="active"               value="true"/>
    <param name="use_scan"             value="true"/>
    <param name="use_pcl"              value="false"/>

    <param name="use_split_and_merge"    value="true"/>
    <param name="circles_from_visibles"  value="true"/>
    <param name="discard_converted_segments" value="true"/>
    <param name="transform_coordinates"  value="true"/>

    <param name="min_group_points"     value="5"/>

    <param name="max_group_distance"   value="0.8"/>
    <param name="distance_proportion"  value="0.00628"/>
```

```
    <param name="max_split_distance"   value="0.2"/>
    <param name="max_merge_separation" value="0.5"/>
    <param name="max_merge_spread"     value="0.2"/>
    <param name="max_circle_radius"    value="1"/>
    <param name="radius_enlargement"   value="0.3"/>

    <param name="frame_id"             value="map"/>
  </node>
```

To properly use the ROS installation we need to add two lines to our bashrc file. Edit it using the following command:

```
gedit ~/.bashrc
```

Or if you dont have gedit.

```
nano ~/.bashrc
sudo apt install gedit
```

Then on the PC add the two following lines:

```
source /opt/ros/melodic/setup.bash
source ~/curio_ws/devel/setup.bash
```

On the Pi 4 we used the same workspace but a different distribution of ROS.

```
source /opt/ros/kinetic/setup.bash
source ~/curio_ws/devel/setup.bash
```

Then on the Pi 3 we have a different name for our catkin workspace and a different ROS version.

```
source /opt/ros/noetic/setup.bash
source ~/catkin_ws/devel/setup.bash
```

Then go to the following GitHub repository: `https://github.com/JadrinaDA/CurioBTs` and clone the repository. There you have two folders, one contains the folders needed for the simulation and one contains the ones needed for the physical system. First just use the folder called Simulation, put the curio folder in the curio_ws/src/curio and put the other two in curio_ws/src or catkin_ws/src if using a separate catkin folder for the behavior trees. Then make all these files executable using the chmod +x command. Go to the folder on the terminal and type in:

```
chmod +x filename
```

Now you have everything installed to use the simulation, skip to the next section of this tutorial if you want to start with that. If you'd like to go ahead and set up the physical system continue reading.

2.2 Setting up the Pis

For the physical system not only do you need to have the master PC you will also need to Raspberry Pis. One Raspberry Pi 4 Model B to run the controller and one Raspberry Pi 3 Model B to run the IMU. For the Pi 4 you can install the normal Debian OS. Then install ROS kinetic. After that you can install the

curio workspace again following the instructions on the Installation section in `https://github.com/srmainwaring/curio`. Now you need to make some edits to these files. First go to curio_base/config/base_controller.yaml and change the wheel radius multiplier from 1 to 1.04.

For the Pi 3 things are simpler, take the latest image from `https://learn.ubiquityrobotics.com/noetic_pi_image_downloads` (thank you Ubiquity Robotics) and install it in the Pi. This image comes with ROS noetic already installed. If you choose to install ROS yourself be careful, there tends to be a lot of issues, and make sure you install noetic. Follow section 4.5 when you are ready to finish installing the software for the IMU.

2.3 Networking and Synchronization

We recommend keeping one user account in the master PC for using the simulation and one user for using the physical system as this allowed for greater flexibility and robustness when it came to testing and debugging. To do so, follow these steps:

First create a new user in Ubuntu, in our case we named it Curio and the password is cur1o#23. Then make the same changes to the .bashrc that you made in the previous section adding the sources. To enable this user to access the ROS files you will need to add them to sudoers. Go back to your initial user and type in this command in the terminal (username is the user's name, in our case curio).

```
usermod -aG sudo username
```

Then type in this command to check it worked:

```
sudo whoami
```

If the output is root it worked! Now to save yourself the hassle of typing in the password again and again modify the sudoers file.

```
visudo
```

And then add to the bottom (username is again the user's name)

```
username  ALL=(ALL) NOPASSWD:ALL
```

Now you should be able to run ROS from this user. Try running a turtlesim node to check.

You also have to edit the bashrc file in order to reflect this new network. Make sure all devices are connected to the same network before getting the IP addresses. If using the asus network router, supplied with the NASA ORD project, make sure the PC is connected to the asus network and not asus_5g when getting its IP address. Follow the procedure in section 5.4.4 in Ciaran Fahy's tutorial document.

Repeat the steps in the section 2.1 to get all necessary files and packages. This time when you get to copying the files from the final GitHub repository use the folder titled Physical.

The next step is to set up TeamViewer so you can easily access the RPi 4. Install the free version of TeamViewer on both the master PC and on the RPi 4. Set up the Pi to have a constant password and set it to launch the app on startup. To set up a constant password you have to sign in with your TeamViewer account. The password I set was: curio123!A#. Make sure LAN connections are enabled on both devices. This way you shall be able to connect with the IP addresses and your password.

For the Pi 3, TeamViewer does not work, therefore, we shall use SSH in order to run the commands. This is enabled by default but just in case if it is not you can follow this tutorial to check:`https://phoenixnap.com/kb/ssh-to-connect-to-remote-server-linux-or-windows`. With the SSH enabled all you need to know is the host name and username. By default it should be ubuntu (and the password as well). For the hostname type the following command in the terminal.

```
hostname -I
```

This should return an IP address. In the PC terminal you then can type:

```
ssh <user name>@<ip address>
```

After that enter the password for the user. The ssh might ask you to trust the host and you should put yes. Then the text on the terminal should change to show the new user: *insert picture here* It is recommended to shut down the RPi manually so to do that from the terminal you can use the following command:

```
sudo shutdown -h now
```

The final step is to use chrony sync to synchronize the clocks on all devices.

For chrony sync this tutorial was used: `https://answers.ros.org/question/312757/how-to-synchronize-two-pc/` Make sure that you do not repeat lines, therefore, check if a line is there before adding them. On the Pi(s) include:

```
server master_ip_address iburst
driftfile /var/lib/chrony.drift
logdir /var/log/chrony
log measurements statistics tracking
makestep 1 3
rtcsync         \#real time clock sync
```

And on the PC:

```
driftfile /var/lib/chrony.drift
local stratum 8
manual
allow master_ip_address
smoothtime 400 0.01
```

Now you are ready to run the real thing.

3 Behavior Trees Simulation

It is preferable to start with the behavior trees in simulation to develop a basic understanding of how the nodes operate. This is where we started the practical part of the project after having done some research. First some basic trees were tested and then we made a simple one for controlling the rover that we shall explain here.

To begin, it is recommended that you read the book Behavior Trees and AI: An Introduction and then this tutorial I have previously prepared: `https://uccl0-my.sharepoint.com/personal/jadrinadeandrade_uc_cl/Documents/BehaviorTreesTutorial%20(1).pdf?login_hint=jadrinadeandrade%40uc.cl`. Once you have done so you will have sufficient understanding to code this more complex behavior tree. First, let us look at our custom behavior, you will find it the file extrabehaviors.py.

3.1 Simple Two Step Tree

```
class MoveClient(py_trees.behaviour.Behaviour):
"""
Behavior that includes an action client to send goals to robot.
Takes a point from its list of goals and sends that point when
ticked.
At every tick after it shall return running until it reaches
the
goal (Sucess) or the goal is aborted (Failure). Goal can be
preempted which will set behavior to invalid.

Args:
    name (:obj:`str`): name of the behaviour, must be in format
 MoveRobot # where #
    is the index of the goal (home is the exception)

Attributes:
    num (:obj:`int`): index of selected goal
    pose_seq (:obj:`Pose`): list of poses representing goals
    goal_s (:obj:`str`): string version of goal as (x, y)
    client (:obj:`SimpleActionClient`) : simple action client
that will send goals to movebase

"""

def __init__(self, name):
    super(MoveClient, self).__init__(name=name)
    # Get goal index
    if (name.split(" ")[1] == "Home"):
        self.num = 3
    else:
        self.num = int(name.split(" ")[1]) - 1

    # Prepare list of goals
    points_seq = [1,1,0,1,5,0,3,3,0,0,0,0]
```

```
        self.goal_s = "(" + str(points_seq[self.num*3]) + "," + str
    (points_seq[self.num*3+ 1]) +")"
        yaweulerangles_seq = [90,0,180,0]
        quat_seq = list()
        self.pose_seq = list()
        for yawangle in yaweulerangles_seq:
            quat_seq.append(Quaternion(*(quaternion_from_euler(0,
    0, yawangle*math.pi/180, axes='sxyz'))))
        n = 3
        points = [points_seq[i:i+n] for i in range(0, len(
    points_seq), n)]
        for point in points:
            self.pose_seq.append(Pose(Point(*point),quat_seq[n-3]))
            n += 1

        # Initialize the MoveBase Action Client
        self.client = actionlib.SimpleActionClient('move_base',
    MoveBaseAction)
        rospy.loginfo("Waiting for move_base action server...")
        wait = self.client.wait_for_server(rospy.Duration(5.0))
        if not wait:
            rospy.logerr("Action server not available!")
            rospy.signal_shutdown("Action server not available!")
            return
        rospy.loginfo("Connected to move base server")
        rospy.loginfo("Starting goals achievements ...")

    def setup(self, timeout):
        # Create the goal based on the pose
        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = "map"
        goal.target_pose.header.stamp = rospy.Time.now()
        goal.target_pose.pose = self.pose_seq[self.num]
        self.goal = goal
        # Always begin with not having sent the goal
        self.goal_sent = False
        # Set up publisher to send robot status
        self.publisher = rospy.Publisher("/status", std_msgs.String
    , queue_size=10, latch=True)
        return True

    def initialise(self):
        # When ticked for the first time since S or F, goal hasnt
    been sent
        self.goal_sent = False

    def update(self):
        # Only send goal if it has not been sent yet
        if (not self.goal_sent):
            self.client.send_goal(self.goal)
            self.goal_sent = True
        state = self.client.get_state()
        if state == GoalStatus.SUCCEEDED:
            # Send message of arrival and return SUCCESS
            self.publisher.publish(std_msgs.String("Arrived at " +
    str(self.goal_s)))
            return py_trees.common.Status.SUCCESS
```

```
        elif state == GoalStatus.ACTIVE or state == GoalStatus.
    PENDING:
            # Send message of moving and return RUNNING
            self.publisher.publish(std_msgs.String("Moving to " +
    str(self.goal_s)))
            return py_trees.common.Status.RUNNING
        else:
            # Anything goes wrong return FAILURE
            print(state)
            return py_trees.common.Status.FAILURE

    def terminate(self, new_status):
        # If interrupted send message to inform user
        state = self.client.get_state()
        if state == GoalStatus.ACTIVE:
            self.publisher.publish(std_msgs.String("Goal was
    preempted"))
```

This behavior is in charge of controlling the robot and sending the MoveBaseAction server one goal to get the robot to navigate there. If the robot arrives it returns Success, if the goal is aborted then it returns Failure. Otherwise it returns Running. We shall now break down the code part by part.

```
    def __init__(self, name):
        super(MoveClient, self).__init__(name=name)
        # Get goal index
        if (name.split(" ")[1] == "Home"):
            self.num = 3
        else:
            self.num = int(name.split(" ")[1]) - 1

        # Prepare list of goals
        points_seq = [1,1,0,1,5,0,3,3,0,0,0,0]
        self.goal_s = "(" + str(points_seq[self.num*3]) + "," + str
    (points_seq[self.num*3+ 1]) +")"
        yaweulerangles_seq = [90,0,180,0]
        quat_seq = list()
        self.pose_seq = list()
        for yawangle in yaweulerangles_seq:
            quat_seq.append(Quaternion(*(quaternion_from_euler(0,
    0, yawangle*math.pi/180, axes='sxyz'))))
        n = 3
        points = [points_seq[i:i+n] for i in range(0, len(
    points_seq), n)]
        for point in points:
            self.pose_seq.append(Pose(Point(*point),quat_seq[n-3]))
            n += 1

        # Initialize the MoveBase Action Client
        self.client = actionlib.SimpleActionClient('move_base',
    MoveBaseAction)
        rospy.loginfo("Waiting for move_base action server...")
        wait = self.client.wait_for_server(rospy.Duration(5.0))
        if not wait:
            rospy.logerr("Action server not available!")
            rospy.signal_shutdown("Action server not available!")
            return
```

```
        rospy.loginfo("Connected to move base server")
        rospy.loginfo("Starting goals achievements ...")
```

When we initialize the node we first call its super class then from its name we shall get the goal that corresponds to this node. In our example there are two points and the third one represents home or (0,0) on our map. Each node contains the list of all three points and then transforms this list into poses so that they can be sent to the MoveBaseAction Server. We initialize the client and wait until the server responds.

```
    def setup(self, timeout):
        # Create the goal based on the pose
        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = "map"
        goal.target_pose.header.stamp = rospy.Time.now()
        goal.target_pose.pose = self.pose_seq[self.num]
        self.goal = goal
        # Always begin with not having sent the goal
        self.goal_sent = False
        # Set up publisher to send robot status
        self.publisher = rospy.Publisher("/status", std_msgs.String
    , queue_size=10, latch=True)
        return True
```

Set up is only done once so we create the goal that will be sent and we create our boolean that will let us now if we've sent the goal yet or not. We also set up our publisher which will communicate with the dashboard to display the status of the robot.

```
    def initialise(self):
        # When ticked for the first time since S or F, goal hasnt
    been sent
        self.goal_sent = False
```

The function initialise is called the first time a node is ticked and at the first tick after returning Success or Failure. Therefore, here is where we shall reset the goal send failure so that whenever the node is ticked it resends the goal to the robot.

```
    def update(self):
        # Only send goal if it has not been sent yet
        if (not self.goal_sent):
            self.client.send_goal(self.goal)
            self.goal_sent = True
        state = self.client.get_state()
        if state == GoalStatus.SUCCEEDED:
            # Send message of arrival and return SUCCESS
            self.publisher.publish(std_msgs.String("Arrived at " +
    str(self.goal_s)))
            return py_trees.common.Status.SUCCESS
        elif state == GoalStatus.ACTIVE or state == GoalStatus.
    PENDING:
            # Send message of moving and return RUNNING
            self.publisher.publish(std_msgs.String("Moving to " +
    str(self.goal_s)))
            return py_trees.common.Status.RUNNING
```

```
        else:
            # Anything goes wrong return FAILURE
            print(state)
            return py_trees.common.Status.FAILURE
```

Update is called every time the node is ticked. Therefore, this is where we will check our robot's progress towards the goal. First we check if we have sent the goal yet. If we have not then we send it and toggle the variable. Then we check the state of our robot and if the state shows the robot has arrived the node shall return Success. If the status is active or pending which means the robot is on its way, then we return running and finally if the status is anything else, like aborted we return Failure.

```
    def terminate(self, new_status):
        # If interrupted send message to inform user
        state = self.client.get_state()
        if state == GoalStatus.ACTIVE:
            self.publisher.publish(std_msgs.String("Goal was
    preempted"))
```

Finally we have the terminate function which is called when the node is interrupted. In this example this will happen when we call the robot home and we shall use this function to display on the dashboard that the goal was preempted.

Now that we have examined our behavior we can take a look at our tree. We create the tree in the function create_root in our main script which is myfirstree.py.

```
    def create_root():
    """
    Creates the different behaviors and composites and joins them
    together
    in tree structure.

    root --> sequence --> gohome2bb, behaviorbb
         --> fallback --> go home  --> gohome?, moverobothome
                      --> sequence --> moverobot1, moverobot2
    """

    root = py_trees.composites.Parallel("Root")
    sequence = py_trees.composites.Sequence("Sequence")
    sequence2 = py_trees.composites.Sequence("Sequence")
    gh_seq = py_trees.composites.Sequence("Go Home")
    fallback = py_trees.composites.Selector("Fallback")
    moveclient = MoveClient(name="MoveRobot 1")
    moveclient2 = MoveClient(name="MoveRobot 2")
    moveclienth = MoveClient(name="MoveRobot Home")

    # Gets message about what robot is doing to bb
    behavior2bb = py_trees_ros.subscribers.EventToBlackboard(
        name="Behavior2BB",
        topic_name="/dashboard/behavior",
        variable_name="curr_behavior"
    )

    # Gets message to gohome to bb, if true robot must go home
```

```
    gohome2bb = py_trees_ros.subscribers.EventToBlackboard(
        name="GoHome2BB",
        topic_name="/dashboard/gohome",
        variable_name="gohome"
    )

    # If variable gohome is true robot is sent home
    is_gohome_requested = py_trees.blackboard.
    CheckBlackboardVariable(
        name="GoHome?",
        variable_name='gohome',
        expected_value=True
    )

    sequence2.add_children([gohome2bb, behavior2bb])

    sequence.add_children([moveclient, moveclient2])
    gh_seq.add_children([is_gohome_requested, moveclienth])
    fallback.add_children([gh_seq, sequence])
    root.add_children([sequence2, fallback])
    return root
```

First we create all the necessary nodes and then we join them in the correct order. The first Sequence node has two children, the nodes that shall pass our variables to the blackboard. We use our custom behavior MoveClient in two nodes in a Sequence so that the robot goes to one point and then the other. After that we make a Sequence that has a node that checks if the go home button has been pressed and if so it makes the robot go home. We then join these two last Sequences in a Fallback node, with the go home Sequence first because we want that behavior to have priority. Finally our root has two children, the updating Sequence and the Fallback previously mentioned.

In order to communicate with the robot, we use a dashboard.

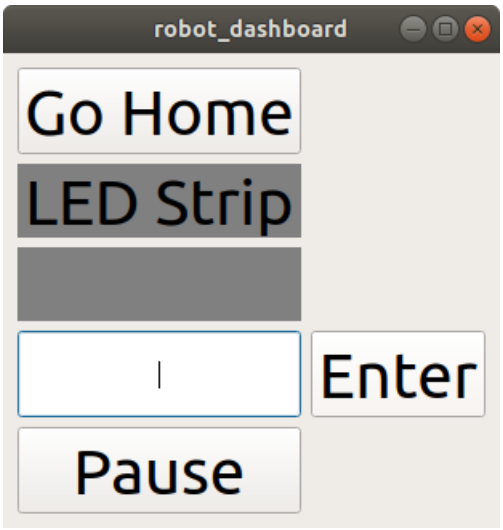

Figure 2: Robot dashboard

This dashboard contains a button to send the robot home, to pause and to send the chosen path. To do this there is a text box where one can type the number of the path. Path 0 is the path home and there are by default paths 1 and 2 already configured. For this tree you can only use the default paths, path 1 should always be kept as it causes the robot to move slightly back and forth giving it a chance to orient itself on the map making future navigation easier. As the robot passes each point on its path it will switch the color of the square in the dashboard to indicate its progress.

You can view the code for the dashboard in the robot_dashboard file.

Now we need a launch file to run the behavior tree. Use mylaunch as a base, we only need to use three nodes, the py qt trees to visualize our trees, our dashboard to access our button and the info, and finally one node that actually runs the tree. Now you can run the behavior tree with the following commands.

```
roslaunch curio_navigation curio_navigation.launch
roslaunch py_trees_ros mylaunch.launch
```

You should see the following windows: First Gazebo and RViz should show up with the rover model.

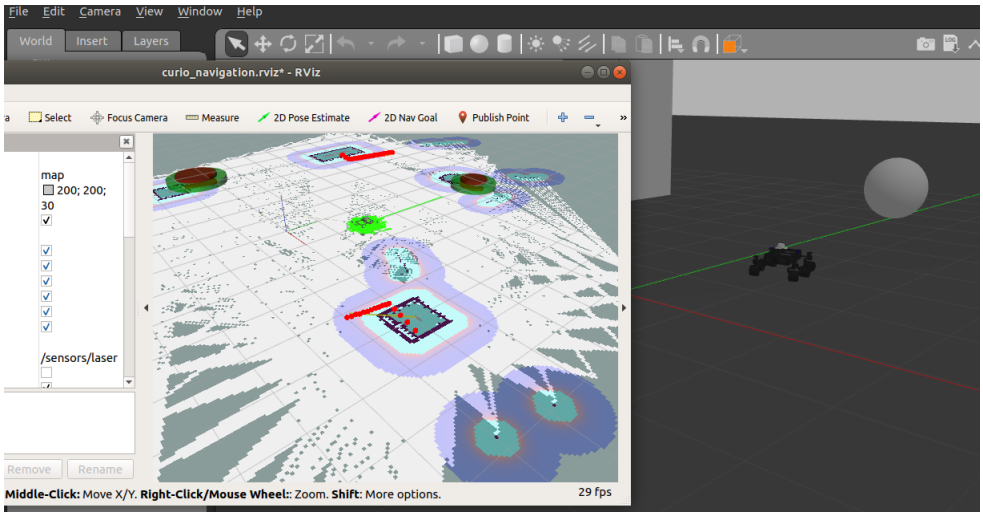

Figure 3: Gazebo and RViz window

Then the py_trees windows should show up with the behavior tree and the dashboard. You might have to select the tree from the drop down menu.

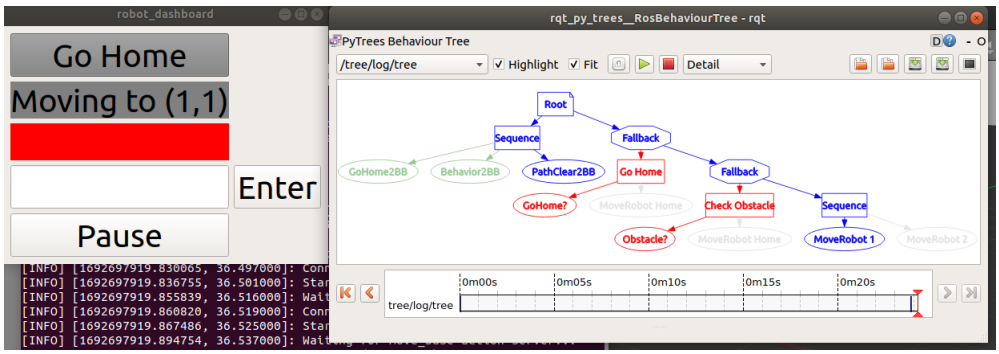

Figure 4: Dashboard and behavior tree

Play around with this example and then we can move on to our final tree. If you need help knowing how to use the dashboard please refer to section 3.3.

3.2 Final Tree

First let us examine the custom behaviors we have made. You will find these in the file mybehaviors.py. For our final tree we don't want our robot to just go to a single point we want it to follow a path and control which path it takes. So our first behavior is ChangePath.

```
    class ChangePath(py_trees.behaviour.Behaviour):
```

```
    """
    Behavior causes the robot to change the path taken.
    Returns SUCCESS if we find  a good path, FAILURE if
    we cycle through all paths and none work.

    Args:
        name (:obj:`str`): name of the behaviour

    Attributes:
        blackboard (:obj:`Blackboard`): Blackboard for behavior
    tree communication
        n_path (:obj:`int`): number of paths
        curr_path (:obj:`int`): id of the current path
        client (:obj:`Publisher`) : ROS publisher that alerts
    movebase client

    """
    def __init__(self, name):
        super(ChangePath, self).__init__(name=name)
        # Initialize blackboard with its variables
        self.blackboard = py_trees.blackboard.Blackboard()
        # Command to use a certain path
        self.blackboard.command = "None"
        # Current path feedback
        self.blackboard.curr_path = 1

    def setup(self, timeout):
        # Set up publisher to let movebase client know when to
    switch paths
        self.publisher = rospy.Publisher("move_base/switch_path",
    std_msgs.String, queue_size=10, latch=True)
        # Path 0 is always home so start at 1
        self.curr_path =1
        # Currently we have 3 paths
        self.n_path = 3
        return True

    def initialise(self):
        # Whenever ticked we want it to check all paths
        # Therefore start with 1
        self.curr_path = 1

    def update(self):
        if self.blackboard.command != "None":
            self.curr_path = int(self.blackboard.command)
            # if len(str(self.blackboard.command)) == 1 :
            #     self.curr_path = int(self.blackboard.command)
            # else:
            #     self.publisher.publish(self.blackboard.command)
            #     self.blackboard.command = "None"
            #     return py_trees.common.Status.SUCCESS
            self.blackboard.command = "None"
        else:
            if self.n_path > self.curr_path + 1:
                self.curr_path += 1
            else:
                rospy.loginfo("No paths left to try")
```

```
                return py_trees.common.Status.FAILURE
        self.publisher.publish(std_msgs.String(str(self.curr_path))
    )
        self.blackboard.curr_path = self.curr_path
        return py_trees.common.Status.SUCCESS
```

Upon creation the node's variables are created, then on initialization the current path is set to 1, so that it can cycle through the paths. Then when ticked the node checks if it has received a command, if it does it sets the current path to the chosen path. Then it resets the command. If there is no command it just chooses the next path if there are any paths left. Finally the current path is sent to the MoveClient node and the blackboard is updated. The node then returns SUCCESS.

Next we have the GoHome behavior.

```
class GoHome(py_trees.behaviour.Behaviour):
    """
    Behavior causes the robot to change the path to go home.

    Args:
        name (:obj:`str`): name of the behaviour

    Attributes:
        publisher (:obj:`Publisher`) : ROS publisher that alerts
    movebase node to switch paths
        blackboard (:obj:`Blackboard`): Blackboard for behavior
    tree communication
        curr_path (:obj:`int`): id of the current path

    """
    def __init__(self, name):
        super(GoHome, self).__init__(name=name)

    def setup(self, timeout):
        # Set up the publisher that will send command to switch
    path
        self.publisher = rospy.Publisher("move_base/switch_path",
    std_msgs.String, queue_size=10, latch=True)
        # Set up blackboard to communicate with other nodes
        self.blackboard = py_trees.blackboard.Blackboard()
        # Home path is always the first
        self.curr_path = 0
        return True

    def update(self):
        # Current path is now going home
        self.blackboard.curr_path = self.curr_path
        self.publisher.publish(std_msgs.String(str(self.curr_path))
    )
        return py_trees.common.Status.SUCCESS
```

When set up we establish the publisher and the blackboard. The way home will always be the first path on the list so we set the current path variable to 0. When ticked the node simply updates the blackboard and sends the message to the MoveClient before returning SUCCESS.

After that we have the FollowPath behavior that basically checks the progress of the robot as it moves through the path and updates the dashboard accordingly. We added that the robot would continue running if its status was canceled as we use this for pausing.

```
class FollowPath(py_trees.behaviour.Behaviour):
    """
    Behavior keeps the robot following the path.

    Args:
        name (:obj:`str`): name of the behaviour

    Attributes:
        blackboard (:obj:`Blackboard`): Blackboard for behavior
    tree communication


    """
    def __init__(self, name):
        super(FollowPath, self).__init__(name=name)

    def setup(self, timeout):
        # Set up blackboard to communicate with other nodes
        self.blackboard = py_trees.blackboard.Blackboard()
        # Start with a pending state
        self.blackboard.state = GoalStatus.PENDING
        return True

     def update(self):
        # Based on the current state I have succeeded, failed or am
     currently
        # following the path
        state = int(self.blackboard.state)
        #print(state)
        #rospy.sleep(0)
        if state == GoalStatus.SUCCEEDED:
            return py_trees.common.Status.SUCCESS
        elif state == GoalStatus.ACTIVE or state == GoalStatus.
    PENDING or state == 2:
            return py_trees.common.Status.RUNNING
        else:
            return py_trees.common.Status.FAILURE

    def terminate(self, new_status):
        # If I stopped when my goal was active
        # there was an interruption
        state = int(self.blackboard.state)
        if state == GoalStatus.ACTIVE:
            rospy.loginfo("Goal was preempted")
```

Finally our last behavior is Pause which sends the number -1 to the MoveBase client which will interpret this as a pause signal and cancel the goal.

```
class Pause(py_trees.behaviour.Behaviour):
    """
    Behavior causes the robot to pause. It will resume the previous
    path when unpaused.
```

```
    Args:
        name (:obj:`str`): name of the behaviour

    Attributes:
        publisher (:obj:`Publisher`) : ROS publisher that alerts
    movebase node to pause
        blackboard (:obj:`Blackboard`): Blackboard for behavior
    tree communication
        curr_path (:obj:`int`): id of the current path
        paused (:obj:`boolean`): if we have paused or not


    """
    def __init__(self, name):
        super(Pause, self).__init__(name=name)

    def setup(self, timeout):
        # Set up publisher
        self.publisher = rospy.Publisher("move_base/switch_path",
    std_msgs.String, queue_size=10, latch=True)
        # This will let the movebase client know to pause
        self.curr_path = -1
        # To save the current path when we pause we use the
    blackboard
        self.blackboard = py_trees.blackboard.Blackboard()
        # To know if we have paused or not we use a blackboard var
        self.blackboard.pause = False
        self.paused = False
        return True

    def initialise(self):
        # At the start we have not send the goal
        self.goal_sent = False

    def update(self):
        # When ticked if not paused it fails
        if not self.blackboard.pause:
            return py_trees.common.Status.FAILURE
        # If the pause signal has not been send yet then send it
        # After sending toggle the booleans
        if not self.goal_sent:
            if not self.paused:
                self.publisher.publish(std_msgs.String(str(self.
    curr_path)))
                self.paused = True
            else:
                self.publisher.publish(std_msgs.String(str(self.
    blackboard.curr_path)))
                self.paused = False
            self.goal_sent = True
        return py_trees.common.Status.RUNNING
```

We save the current path so that when the button is clicked again, the robot continues its trajectory along the path. In the tree we also used the predefined behavior Idle so that when the robot has aborted the goal it just waits until it receives a new command. These behaviors are all used in the code follow-

path.py, it is similar to the one showed above so we will not repeat it. The code itself is well commented so reading it should be sufficient to understand what is happening.

First the MoveBase client was directly in the behavior tree but we thought it best to not charge the node too much and that it would be good practice to learn how to make separate ROS nodes that were able to communicate with each other. Then on the second iteration we replaced the behavior with one that would communicate with that node instead and then change its status based on the messages it received. We set up a second move base node based on a tutorial which was then extended in order to include desired functionalities such as being able to switch paths and pause. The code for this node is in follow_path.py. There are several comments explaining the code so we will only review a couple of functions inside the main class, mainly the ones that were added to the base presented in this tutorial: `https://hotblackrobotics.github.io/en/blog/2018/01/29/seq-goals-py/`.

```
def switch_path(self, data):
    """
    Callback function that switched the current path based on the
    info received from
    the topic
    Attributes:
        data (:obj:`msg`): message containing the index of chosen
    path

    """
    # Get the current path
    self.pose_seq = self.paths[self.curr_path]
    # If we had gone home and now are going to another path reset
    the goal count
    if self.curr_path == 0 and int(data.data) != 0:
        self.goal_cnt = 0
    # Set the new path
    self.curr_path = int(data.data)
    # If there is no path with that index dont do anything and send
     a message
    if self.curr_path not in self.paths.keys() and self.curr_path >
     0:
        rospy.loginfo("No such path")
        self.curr_path = 0
        return
    # If we were told to pause cancel the goal
    if self.curr_path == -1:
        self.client.cancel_goal()
        self.curr_path = 1
    else:
        # If you are sent home reset goal count
        if self.curr_path == 0:
            self.goal_cnt = 0
        # If not go to the equivalent point on the chosen path
        if self.goal_cnt< len(self.pose_seq):
            self.pose_seq = self.paths[self.curr_path]
            next_goal = MoveBaseGoal()
```

```
            next_goal.target_pose.header.frame_id = "map"
            next_goal.target_pose.header.stamp = rospy.Time.now()
            next_goal.target_pose.pose = self.pose_seq[self.
    goal_cnt]
            rospy.loginfo("Switching to path " + data.data)
            self.goal_s = "(" + str(self.points_paths[self.
    curr_path][self.goal_cnt*3]) + "," + str(self.points_paths[self
    .curr_path][self.goal_cnt*3+ 1]) +")"
            rospy.loginfo("Sending goal pose "+str(self.goal_cnt+1)
    +" to Action Server")
            rospy.loginfo(str(self.pose_seq[self.goal_cnt]))
            # Send new goal to dashboard
            self.publisher.publish(String("Moving to " + str(self.
    goal_s)))
            self.client.send_goal(next_goal, self.done_cb, self.
    active_cb, self.feedback_cb)
```

This function is in charge of receiving the command to switch to a different path through the "move_base/switch_path" topic. If the current path is set to 0 but the next path is not then the robot was sent home before so we reset the goal count. Then we check to make sure the path does exist. Then if the path is -1 then that means it is the pause signal and therefore, instead of setting a new goal we cancel it. The current path must be set to something else after to avoid errors. If neither of these cases is met then we continue to send the new goal to the server. This is done the same way as in the old behavior.

```
    def add_point(self, data):
        # Add a new point to the last path on our list
        # Get point from the data
        point = [round(data.point.x,1), round(data.point.y,1), 0]
        # Angle is always 0 because we dont care
        quat = Quaternion(*(quaternion_from_euler(0, 0, 0*math.pi
    /180, axes='sxyz')))
        # If we are starting a new path we need to create the empty
     lists
        if len(self.paths.keys()) < self.ind + 1:
            self.paths[self.ind] = []
            self.points_paths.append([])
        # Add first the point to the list of poses
        self.paths[self.ind].append(Pose(Point(*point),quat))
        # Then add it to the list of points
        for p in point:
            self.points_paths[self.ind].append(p)
        # If our path is complete (it has three points)
        if len(self.paths[self.ind]) == 3:
            # Increase index to start new path and print out all
    paths
            self.ind += 1
            rospy.loginfo(self.points_paths)
```

In order to make new paths easily we have this function that is connected to RViz's "/clicked_point" topic. These points can be sent by clicking on the Publish Point tool in the upper right on RViz and then clicking a point on the map. Make sure you click on a point that is clear if not the planner will abort the goal. The function will take a point and assign it angle 0. Then if

we completed our last path we have to make new empty lists to accommodate the new one. Then we add the pose to the list and the point to our list. We determine that a path is complete if it has three points, technically you can still send the robot on a path that has less points but never more. This could be edited in the future to be more flexible. We already have two default paths so when you click points in RViz the paths created will have numbers from 3 and so forth.

In order to detect if the path was clear or not we also used another ROS package, obstacle detector. A obstacle detector node is added in the curio navigation launch file. In the file obstaclesdist.py you can take a look at how we process the obstacles outputted by the node and calculate the distances and angles to see if there is an obstacle in the robot's path. When working with the real robot this script was then expanded to also include the time since the last position was given (this is updated when the robot moves a certain amount). If the robot has not moved enough to update its position in around 10 seconds and there is an obstacle close by it is probably stuck and needs help. Be sure to make sure you are using the correct local plan topic in the program or the obstacles wont be detected. In the simulation this might be DWAPlannerROS or TrajectoryPlannerROS. This is set near the end of the file.

3.3 Usage

The main way of interacting with the robot is the dashboard.

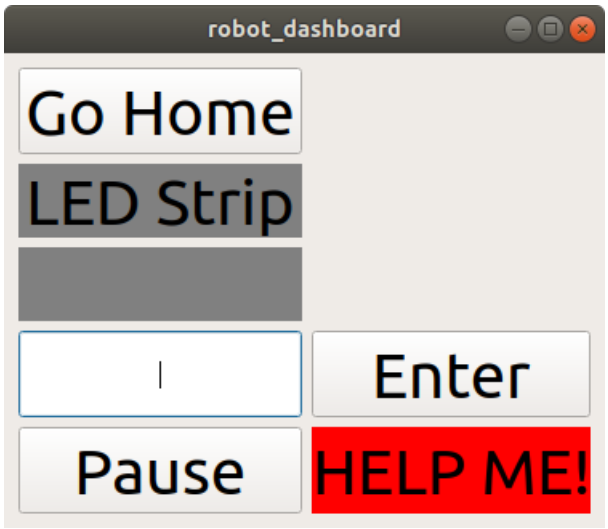

Figure 5: Dashboard complete

On the dashboard there are three buttons that can be used. The button labeled "Go Home" will change the path of the robot to the path home. Then

second button is on the fourth row on the right and says "Enter" clicking it will submit the text in the text box. This text must be a number or nothing will happen. When selecting a new path the robot will go to the equivalent point from the path it was following. That means that if it had passed through point 1 and is heading to point 2 in path 1 and you tell it to switch to path 3 then it will head to point 2 in path 3. If the new path does not have an equivalent point because it is shorter than the current path then the robot will not follow the next path. This was done thinking that all paths would be alternatives of each other but would lead to the same place. It should probably be changed in the future. The only exceptions to this are when a path is finished and after the robot is sent home as this will cause the goal count to reset and the robot will start the new path form the beginning. The third button is right below it and it says "Pause" and it will stop the robot no matter what it is doing. When the pause button is pressed again (you'll see the status change on the label on the second row) the path is resumed. Note that if you try to change the path while the robot is paused the change will not happen as that part of the tree is blocked. You can only switch paths when the robot is not paused. The two labels (dark grey boxes) will let you know the status of the robot. This first will let you know where the robot is headed, if it has arrived and if the robot is paused that status will be displayed. The second label changes color every time a goal is reached on the path. Finally when in the physical version it will probably be useful to know when the robot if the robot is stuck. So there is a final label next to the Pause button that will blink red and display the message "HELP ME!" to let you know the robot is in need of assistance.

Now you can run the program in the simulation executing the following commands, each in a separate terminal tab.

```
roslaunch curio_navigation curio_navigation.launch
roslaunch py_trees_ros followpath.launch
rosrun simple_navigation_goals obstaclesdist.py
roslaunch simple_navigation_goals movebase_seq.launch
```

Once you are satisfied you can move on to implementing the real life version.

3.4 Custom Recovery Behavior

One thing that is not vital to the project but still useful to learn is the inclusion of custom recovery behaviors. Recovery behaviors need to have a very specific format and a wrapper so instead of writing one from scratch we decided to use another recovery behavior as a template, especially since it was one that already did something similiar to what was needed. First we downloaded the package steer_drive_ros, this contained the behavior in a subpackage stepback_and_steerturn_recovery. If you do not want to modify it you can simply download it like this.

```
sudo apt install ros-melodic-steer-drive-ros
```

Since we wanted to modify it we downloaded directly from GitHub into our catkin workspace. You can edit the files directly, we edited the file steer_drive_ros/stepback_and_steerturn_recove

We took away the steering part of the behavior so that the robot would merely back up instead. In the future this behavior could be refined even further. Then you must indicate that you want to use that recovery behavior instead of the default. To do this create a new yaml file in curio_navigation/config and paste the following code.

```
recovery_behaviors:
  - name: 'stepback_and_steerturn_recovery'
    type: 'stepback_and_steerturn_recovery/
    StepBackAndSteerTurnRecovery'

step_back_and_max_steer_recovery:
    steering_timeout    : 5.0
```

Save this file as recovery_behaviors.yaml. Finally you need to edit move_base.launch and add the following line to the move base node:

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

Now you can use your custom behavior.

4 Moving to real life demo

The files needed for the real life demo are slightly different than from the simulation. We'll show screenshots of the important folders.

The most important is the curio_ws/src folder which should look like this:

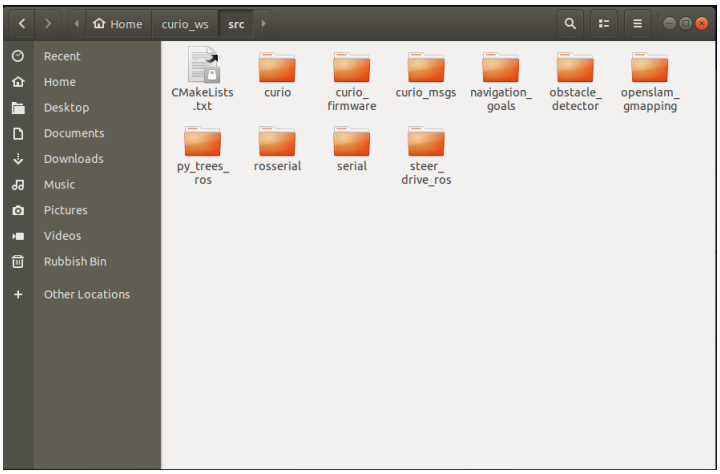

Figure 6: Contents of curio_ws/src

This means you have all the packages needed from GitHub. Then inside the curio/curio_navigation/launch folder:

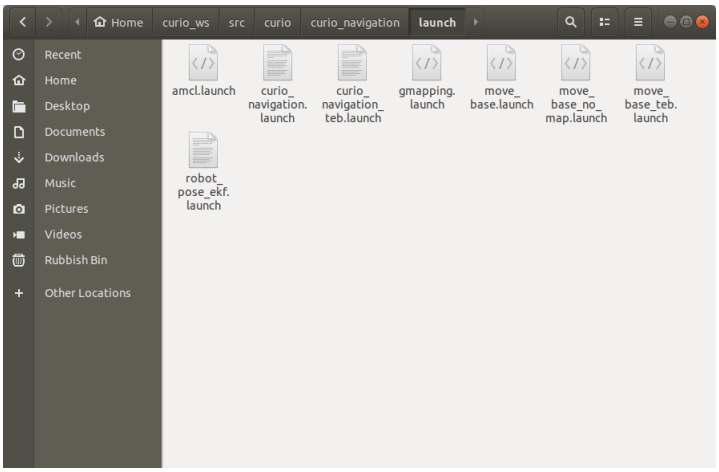

Figure 7: Curio navigation folder

Most yaml files with parameter are within curio/curio_navigation/config:

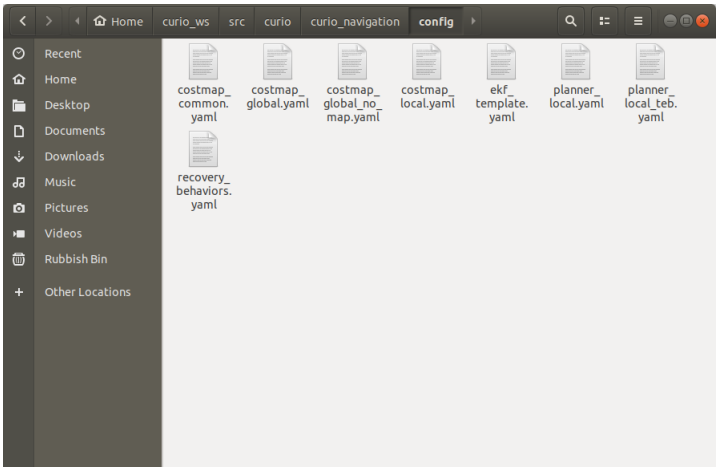

Figure 8: Config folder

Then in navigation_goals we must have these two files in the src folder.

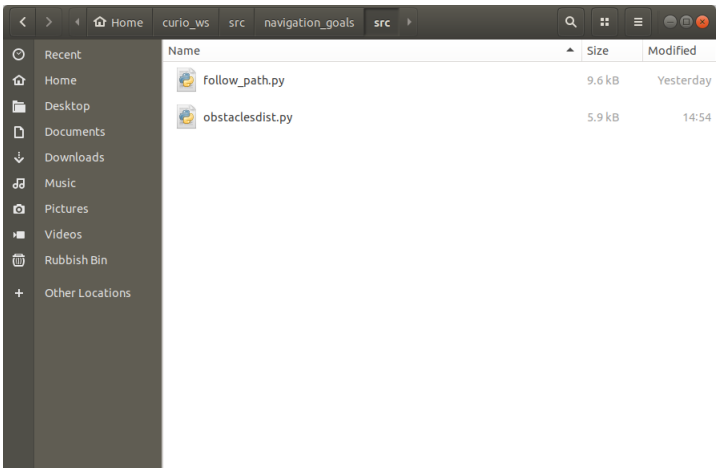

Figure 9: Navigation goals src folder

And in the launch folder:

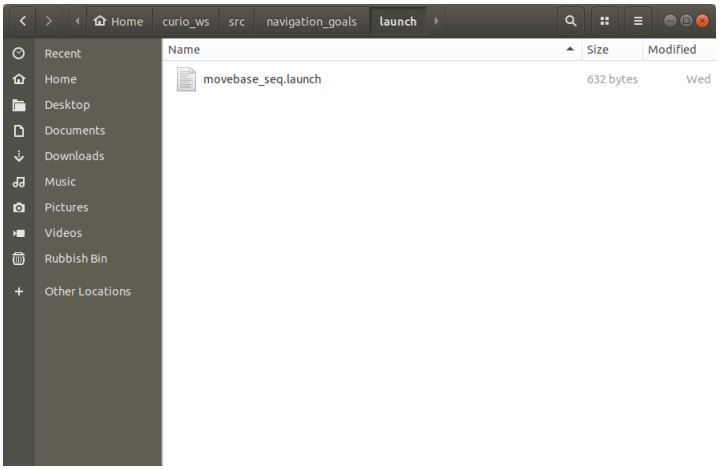

Figure 10: Navigation goals launch files

To launch the behavior trees we'll need these files in py_trees_ros/launch:

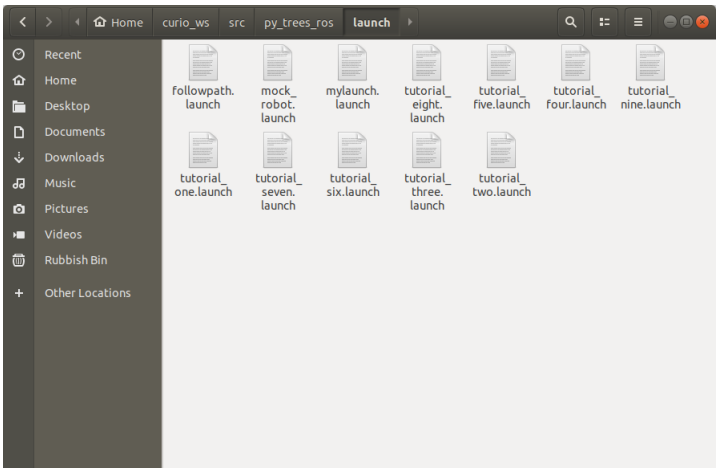

Figure 11: Py Trees launch files

And in py_trees_ros/scripts:

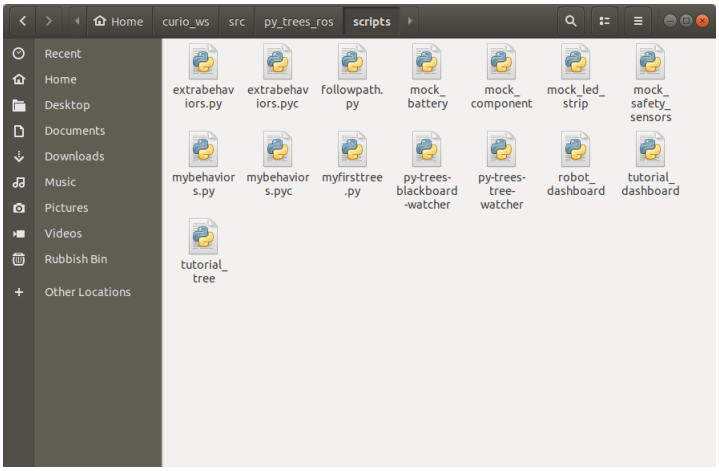

Figure 12: Py trees scripts

You also have to make some edit to the files to make sure you are using real world information. First go to curio_navigation.launch and comment out the block of code commented gazebo, this will make the robot use its own laser's information. Also after you make your own map of the room you will have to put its name in the first tag instead of the default shapes world.

Then for move_base.launch find the line that says remap and replace it with:

```
<remap from="cmd_vel" to="/cmd_vel"/>
```

Before you start working with the behavior tree and the real rover, it is recommended to just run navigation first and fine-tune that. I followed this guide: `https://kaiyuzheng.me/documents/navguide.pdf` . Really what seems to impact most is sim-time, as it allowed the robot more time to calculate its path and caused it to get stuck less. The odometry must be fine tuned as well, follow the procedure in this guide:. Based on that you should be able to change the wheel-radius multiplier. I changed the other multiplier but performance only seemed to worsen so leave it as it is. Another parameter that requires modification is the costmap inflation radius.

4.1 TEB

In the simulation we used the basic local planner provided by the ROS navigation stack because the environment was simple and did not require much intelligence. For the real life system we switched to TEB local planner since it proved to be the more resourceful planner as it got stuck less and was able to avoid obstacles not seen the map. In order to use TEB you must install it and then make some edits to the curio navigation launch file and the move base launch file. First install it with (and install gmapping while you are at it):

```
sudo apt install ros-melodic-teb-local-planner
sudo apt install ros-melodic-gmapping
```

Next create a copy of move_base.launch and name it move_base_teb.launch. Add the following lines after the line that says remap:

```
<param name="base_local_planner" value="teb_local_planner/TebLocalPlannerROS" />
<param name="controller_frequency" value="5.0" />
```

Then create a copy of curio_navigation.launch and name it curio_navigation_teb.launch. Inside this file where you load move_base.launch replace the file name with move_base_teb.launch Now you are good to go.

4.2 Debugging

The jitter problem was partly reduced by installing an IMU and also tuning certain parameters. At one point the local costmap was rotated, which was fixed by switching the global_frame parameter from odom to map. We then changed the odom_alpha parameter in the amcl to reflect a less noisy odometry, the first 4 were set to 0.005. Then we lowered the estimated noise in the laser by lowering laser_z_hit to 0.5. We decreased the confidence in the wheel odometry by increasing all the diagonal covariance values in ekf_template.yaml by 0.01. This decreased the jitter but there was still some left. As the rover navigated its wheels oscillated quite a bit, therefore, we lowered the acceleration limit in the theta direction to 0.05 and in the y direction to 0.2. This seemed to reduce the jitter considerably. The velocity in the x axis was set to 0.37 both backwards and forwards. This was done in planner_local_teb.yaml. We also increase the weights for forward drive, turning radius and shortest path which helped the robot not back up as much. Forward drive was then increased again later.

We did further experimenting with the dynamic obstacles and the obstacle and inflation distance but this just seemed to decrease performance. The next day we turned on the via points and set their weight to 5 and their minimum separation to 1 m. This helps the robot stick closer to the global path. Then we increased the yaw tolerance to avoid any shuffling when arriving to the goal.

Then in order to cross narrow hallways more smoothly, we decreased the minimum obstacle distance to 0.3 and the inflation distance to 0.4. There seemed to be a jitter due to the frequent update of the map so we decreased the value of linear update in the gmapping node. We also decreased the acceleration limit to 0.25. Finally we decreased the covariance slightly to trust the odometry less. Another parameter that seemed to improve performance was enabling the odometry differential in the EKF.

We also found that the frequency of messages being sent over a topic to the blackboard could affect if they were being received or not. A higher frequency would mean some messages were getting lost on the way. This was a problem for messages such as the one sent after arriving at a goal as it is only sent once. The size of the queues was also important as it needed to be big enough to allow some delay but not too big as seen in `http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers#Choosing_a_good_queue_size`.

4.3 New behavior

In this part we added a new behavior and a new label to the dashboard. This is so that when the robot is stuck, for example a doorway that was previously open is now closed, a human can come along and clear its path.

```
    class Ask4Help(py_trees.behaviour.Behaviour):
    """
    Behavior causes the robot to flash a button on its dashboard
    asking for help.
    When the path is clear again it should resume its normal path.

    Args:
        name (:obj:`str`): name of the behaviour

    Attributes:
        publisher (:obj:`Publisher`) : ROS publisher that alerts
    movebase node to pause
        blackboard (:obj:`Blackboard`): Blackboard for behavior
    tree communication
        curr_path (:obj:`int`): id of the current path
        goal_sent (:obj:`boolean`): if we have sent the signal or
    not


    """
    def __init__(self, name):
        super(Ask4Help, self).__init__(name=name)

    def setup(self, timeout):
```

```
        # Set up publisher
        self.publisher = rospy.Publisher("/helpme", std_msgs.String
    , queue_size=10, latch=True)
        self.blackboard = py_trees.blackboard.Blackboard()
        # To know if the robot is stuck or not we use a blackboard
    var
        self.blackboard.pathclear = True
        self.publisher_p = rospy.Publisher("move_base/switch_path",
     std_msgs.String, queue_size=10, latch=True)
        # This will let the movebase client know to pause
        self.curr_path = -1
        self.paused = False
        return True

    def initialise(self):
        # At the start we have not send the goal
        self.goal_sent = False

    def update(self):
        # If the signal has not been sent yet then send it
        # After sending toggle the booleans
        if int(self.blackboard.state) == 3:
            return py_trees.common.Status.FAILURE
        # Variable is opposite
        # If clear then thank the person and return Failure
        if not self.blackboard.pathclear:
            self.publisher.publish(std_msgs.String("Thanks!"))
            self.publisher_p.publish(std_msgs.String(str(self.
    blackboard.curr_path)))
            return py_trees.common.Status.FAILURE
        # If not clear then ask for help and stay running
        if not self.goal_sent:
                self.publisher.publish(std_msgs.String("HELPME"))
                self.publisher_p.publish(std_msgs.String(str(self.
    curr_path)))
                self.goal_sent = True

        return py_trees.common.Status.RUNNING
```

On setup, the behavior initializes the publishers and blackboard variables that are required. When the robot asks for help it also pauses, that is why it must also signal the switchpath topic. We only send the signal once so we have to set up a goal sent variable on initialization. Then on every update the node will check if the path has been cleared yet. If it has it will send a thank you message to display on the dashboard. After that it will resume its path. If the path has not been cleared then the node will send the signal if it has not yet and if it has it will stay running.

4.4 Running the rover

To run the rover follow this procedure: First turn on the router and make sure the master PC is connected to the asus router. Open a terminal and run the following command:

```
roscore
```

After that unplug the Arduino from the Raspberry Pi and turn on the Curio rover. Give it a couple of minutes to turn on. Then open TeamViewer and connect to the Pi 4. Plug the Arduino in again and then unplug and plug in the laser. Open the terminal and run the following command:

```
roslaunch curio_bringup curio_robot.launch arduino_port:=/dev/
ttyACM0 laser_scan_port:=/dev/ttyUSB0
```

If there are errors relating to the laser or an operation time out, unplug and plug the laser in again before executing the node again. Go back to the master PC, open a new terminal tab, and run:

```
roslaunch curio_navigation gmapping.launch
```

Open a new tab in the terminal and execute the command to connect to the Pi 3 via SSH (yours might look different):

```
ssh ubuntu@192.168.50.103
```

Enter the password and then run:

```
roslaunch imu_bbno055 imu.launch
```

Then go back to the master PC and run each of these commands in a separate terminal or window (the map name might be different).

```
rosrun map_server map_server ~/Documents/play_room3.yaml
roslaunch curio_navigation robot_pose_ekf.launch
roslaunch curio_navigation curio_navigation_teb.launch
rosrun navigation_goals obstaclesdist.py
roslaunch py_trees_ros followpath.launch
roslaunch navigation_goals movebase_seq.launch
```

Note: If you have yet to install the IMU and implement the EKF just skip those steps. Everything should work you will just have some jitter.

For how to use the dashboard and RViz please refer back to section 3.3.

4.5 Installing the IMU

At this point the code for the behavior tree was working but the jitter was causing some undesired effects. Therefore, to make the odometry better we decided to install an IMU. The process of wiring we got from the web and the set up is described below:

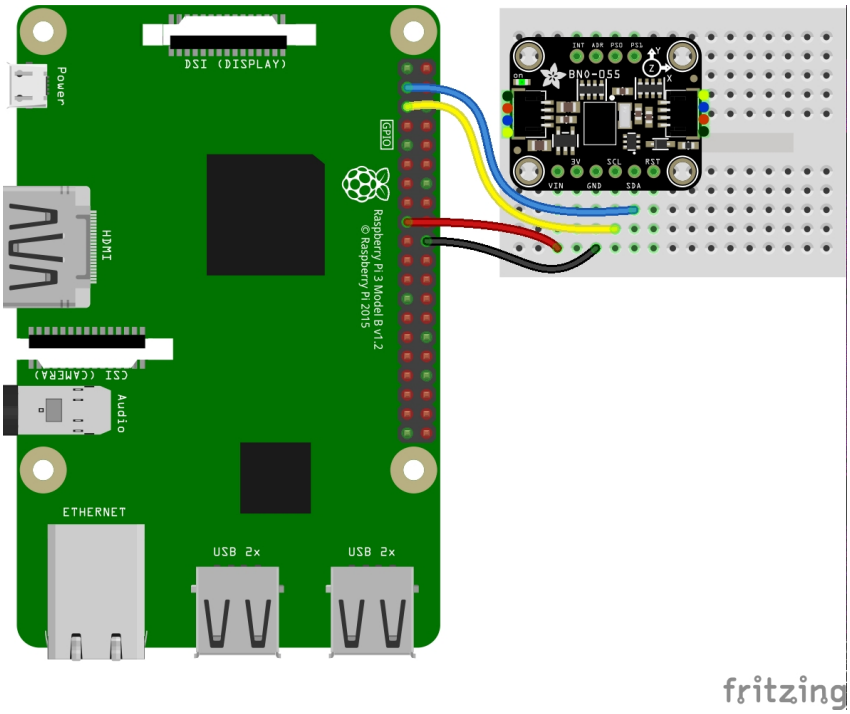

Figure 13: Wiring of IMU and RPi

Now the internet has differing views on whether I2C or UART should be used when combining this sensor and a Raspberry Pi. The website says to use UART to avoid clock stretching but we found that it was still possible to use I2C and just lower the baudrate. Therefore, make sure you use the wiring diagram shown above as this will guarantee you are using I2C mode.

Refer back to section 2.3 and make sure you've done all the steps in order to fully set up the RPi3 before you continue with the installation

On the RPi 3 you must install the package for running the IMU. Create a catkin workspace or navigate to the one you already have and type the following commands:

```
cd ~/catkin_ws/src
git clone https://github.com/dheera/ros-imu-bno055.git
cd ~/catkin_ws
catkin_make
```

Now you should be able to run the IMU.

```
roslaunch imu_bno055 imu.launch
```

If you get an error saying that the chip ID is incorrect try these two things. First run the following command:

```
sudo i2cdetect -r -y 1
```

If no number appears in the grid, check your wiring. If a 29 appears then go to the launch file and change the number 40 to 41. If a 28 appears then the problem is probably due to clock stretching. Follow `https://learn.adafruit.com/circuitpython-on-raspberrypi-linux/i2c-clock-stretching` to change the baudrate to something lower like 5000 Hz. If this fails as well switch the cables being used to connect the IMU and the RPi.

Now that you have the IMU working we need to implement the Extended Kalman Filter. To do this read section 15 in Ciaran Fahy's report. You will also need to update the URDF file so that the IMU's location in relation to the rest of the robot is known. In robot_pose_ekf.laucnh edit the first line and make sure the first three arguments represent the translation from the center of the robot to the imu in x, y, z. The x axes is along the vertical of the robot and is positive towards the laser, the y axis is the horizontal and is positive towards the laser and the z axis is positive above the robot. After that go to the Pi4 find the file curio_base/launch/base_controller.launch and add the following file inside the curio base node.

```
<remap from="odom" to="odom_raw">
```

This will allow us to first filter our odom into the EKF before using it. Make sure that once you implement the extended Kalman filter that you remove the transform being published by the controller by commenting line 1372 in base_controller.py (in the Pi 4).