Understanding ROS 2 Topics

Estimated time to completion: **1 hour**

3.7 How to mix Publishers and Subscribers

So far, you've created separate ROS 2 nodes for either publishing or subscribing to topics. However, in most real-world applications, your ROS 2 nodes will need to both publish and subscribe to topics. In this section, we will explore how to combine both publishers and subscribers within a single ROS 2 node.

For this, we'll work on a new mission for our Mars Rover!

NASA has tasked us with creating a prototype program capable of recognizing alien plants using the robot's camera.

Let's build an object detection system using both subscribers and publishers.

We need to develop a script that detects plants using the RGB camera sensor.

- Create a script named `plant_detector_node.py`.
- This script should subscribe to the image topic `/leo/camera/image_raw`.
- We will use an AI model developed by NASA's Space Biology Division to identify plants.
- Once a plant is detected, the program should publish a string message to the `/plant_detector` topic, indicating that a plant has been found.

- Let's create the file `plant_detector_node.py` inside the package `mars_rover_tasks`.
- You can create it through the IDE or through these commands:

Execute in Terminal #1

```
In [ ]:
cd ~/ros2_ws/src/mars_rover_tasks/mars_rover_tasks
touch plant_detector_node.py
```

- We now have to download the **AI Model and python script** provided by the **NASA's space biology division**. You can have a look at it if you want, but there is no need to understand the inner workings of this model in this course.

Execute in Terminal #1

```
In [ ]:
cd ~/ros2_ws/src/
git clone https://bitbucket.org/theconstructcore/basic_ro
cp basic_ros2_extra_files/plant_detector/plant_detector.p
```

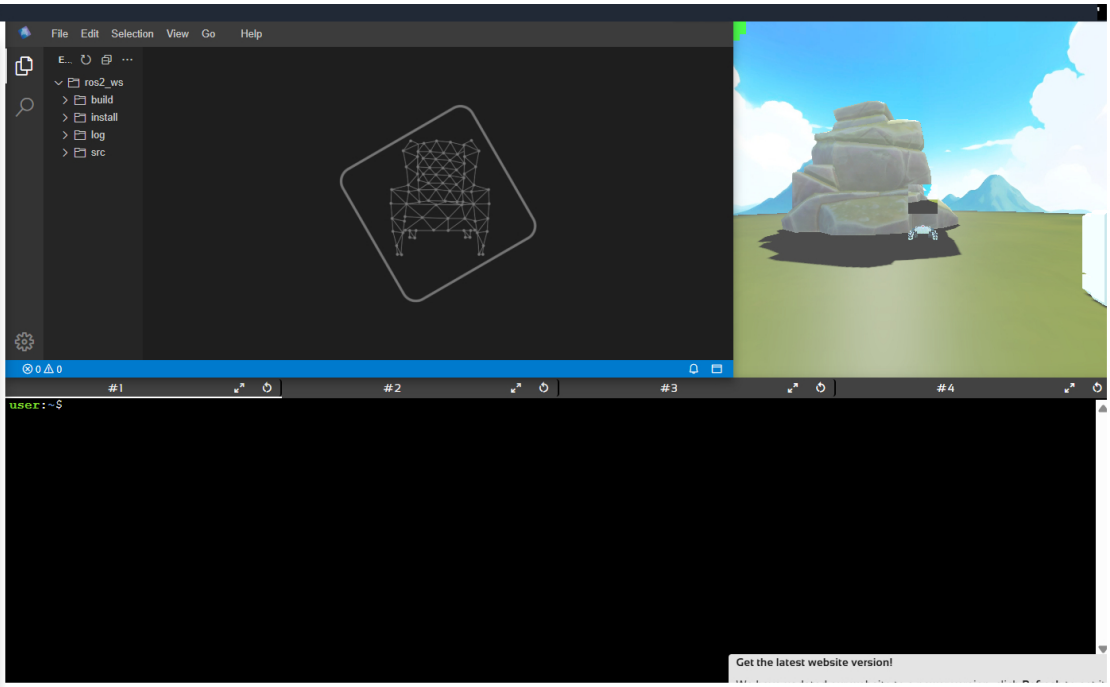
- Let's now create the code in the script `plant_detector_node.py` step by step:

Add the imports:

```
In [ ]:
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from std_msgs.msg import String  # Import String message
from cv_bridge import CvBridge
import cv2
from mars_rover_tasks.plant_detector import PlantDetector
```

- We add the old favorite: `rclpy`, `Node`
- Because we will publish a `topic` named `plant_detector` that uses `String` type `interface`, we need to import it.
- We import `cv`. This is the Python module version of **OpenCV**. We won't go into details in this course about how **OpenCV works**. You can have a look at this course if you are interested: OpenCV for Robotics.
- But it's important that you learn how to use `Image` `interface` `topics`, because they are a bit peculiar in ROS2, and one of the most used in robotics.
- The main idea is that **ROS2 images** `topics` have a slightly different format to **OpenCV images**, so we need this **CvBridge** to make the conversions.
- And finally we import the `plant_detector` script given by the **NASA space biology division** so that we can use the class `PlantDetector` to feed in **images** and output **if a plant was detected**.

Create the `init` method:



In [ ]:

```
class PlantDetectorNode(Node):
    def __init__(self):
        super().__init__('plant_detector_node')

        # Initialize the CvBridge
        self.bridge = CvBridge()

        # Initialize the PlantDetector
        path_to_model = "/home/user/ros2_ws/src/basic_ros
        self.plant_detector = PlantDetector(model_path=pa

        # Subscribe to the image topic
        self.subscription = self.create_subscription(
            Image,
            '/leo/camera/image_raw',
            self.image_callback,
            10)
        self.subscription  # prevent unused variable warn

        # Initialize the Publisher for plant detection re
        self.publisher_ = self.create_publisher(String, '
```

- We initialize this `CvBridge()` for the mentioned Image conversion.
- We initialize the `PlantDetector(model_path=path_to_model)` with the path to the **AI pretrained model** that you downloaded from the **git repository**.
- We create a `subscription` using `create_subscription` to the image `topic` `/leo/camera/image_raw`.
- We also create a `publisher` using `create_publisher` to a new `topic` named `/plant_detector` where we will publish the detections.
- As you can see in the `init` methods, we can start as many `Subscribers` and `Publishers` as we want.

Define the image `callback`

In [ ]:

```
def image_callback(self, msg):
        # Convert ROS Image message to OpenCV image
        cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")

        # Use the PlantDetector to make a prediction
        prediction = self.plant_detector.predict(cv2.cvtC

        # Determine the result message based on the predi
        if prediction > 0.5:
            result = f"Plant detected with confidence: {p
            self.get_logger().warning(result)
        else:
            result = f"No plant detected. Confidence: {1
            self.get_logger().info(result)

        # Publish the result as a String message
        msg = String()
        msg.data = result
        self.publisher_.publish(msg)
```

- We use the `image_callback` to process the **Image** and check if there are any **plants** in the image.
- For that, we use the `self.bridge.imgmsg_to_cv2(msg, "bgr8")` to convert from the `msg == ROS2 Image` to an **OpenCV image**.
- Then we give that image to the `plant_detector.predict()` method that inputs an **Image** and outputs the **confidence of plant detection**.
- If **confidence of plant detection** is higher than **50%**, we consider that it's a plant; If not, then we calculate the inverse to see how much confidence we know that it's **NOT a plant**.
- Once that is done, we just **Publish** the `result` variable string through the `publisher_.publish()` method.

Main method

In [ ]:

```
def main(args=None):
    rclpy.init(args=args)

    plant_detector_node = PlantDetectorNode()

    rclpy.spin(plant_detector_node)

    # Destroy the node explicitly (optional)
    plant_detector_node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

- Typical main initialization of the custom Node class `PlantDetectorNode`.

- At the end you should have something similar to this:

plant_detector_node.py

In [ ]:

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from std_msgs.msg import String  # Import String message
from cv_bridge import CvBridge
import cv2
from mars_rover_tasks.plant_detector import PlantDetector

class PlantDetectorNode(Node):
    def __init__(self):
        super().__init__('plant_detector_node')
```

```
        # Initialize the CvBridge
        self.bridge = CvBridge()

        # Initialize the PlantDetector
        path_to_model = "/home/user/ros2_ws/src/basic_ros
        self.plant_detector = PlantDetector(model_path=pa

        # Subscribe to the image topic
        self.subscription = self.create_subscription(
            Image,
            '/leo/camera/image_raw',
            self.image_callback,
            10)
        self.subscription  # prevent unused variable warn

        # Initialize the Publisher for plant detection re
        self.publisher_ = self.create_publisher(String, '

    def image_callback(self, msg):
        # Convert ROS Image message to OpenCV image
        cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")

        # Use the PlantDetector to make a prediction
        prediction = self.plant_detector.predict(cv2.cvtC

        # Determine the result message based on the predi
        if prediction > 0.5:
            result = f"Plant detected with confidence: {p
            self.get_logger().warning(result)
        else:
            result = f"No plant detected. Confidence: {1
            self.get_logger().info(result)

        # Publish the result as a String message
        msg = String()
        msg.data = result
        self.publisher_.publish(msg)

def main(args=None):
    rclpy.init(args=args)

    plant_detector_node = PlantDetectorNode()

    rclpy.spin(plant_detector_node)

    # Destroy the node explicitly (optional)
    plant_detector_node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

setup.py

In [ ]:

```
from setuptools import find_packages, setup
import os
from glob import glob

package_name = 'mars_rover_tasks'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launc
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='user',
    maintainer_email='user@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'subscriber_obstacle_detector_executable = ma
            'publish_mars_rover_move_executable = mars_ro
            'image_plant_detector_executable = mars_rover
        ],
    },
)
```

- We added this `image_plant_detector_executable` endpoint.

Build and Run

Execute in Terminal #1

In [ ]:

```
cd ~/ros2_ws
colcon build --packages-select mars_rover_tasks
source install/setup.bash
```

- And run it:

Execute in Terminal #1

- Start the `plant_detector_node`:

In [ ]:

```
cd ~/ros2_ws
source install/setup.bash
ros2 run mars_rover_tasks image_plant_detector_executable
```

Execute in Terminal #2

- Move the Mars rover around to test that when the camera sees the plant, it detects it.

In [ ]:

```
cd ~/ros2_ws
source install/setup.bash
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

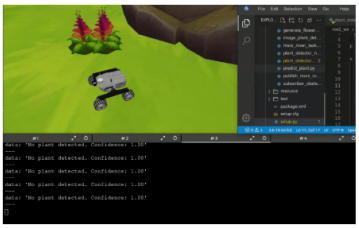
▶ Execute in Terminal #3

- `Echo` the `topic` `/plant_detector` to see the publication of these detections.
- This `topic` can now be used by other **ROS2 nodes** to use this detection information to do whatever you want.

In [ ]:

```
cd ~/ros2_ws
source install/setup.bash
ros2 topic echo /plant_detector
```

- You should experience something similar to this:



- NOTE -

- Here we have seen a `Subscriber` and a `Publisher` in the **SAME SCRIPT**.
- This is not always needed and is especially not recommended if we place too much code inside the `callback` of the `Subscriber`.
- At the end, it depends on how the code performs best.

- End NOTE -

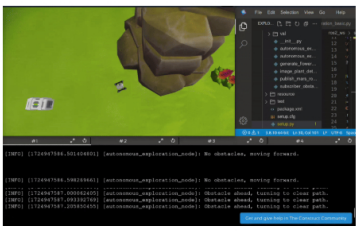
- Exercise 3.3 -

- Based on what we developed previously with the `subscriber_obstacle_detector.py` and the `publish_mars_rover_move.py`, we now need to create a script named `autonomous_exploration.py` that allows our Mars rover to explore its surroundings avoiding obstacles.
- Apply what we have seen on the `Subscriber` and `Publisher` combination to move around randomly, and when the laser detects an obstacle, move accordingly to avoid it.
- There are many strategies to do this, but the one we recommend is:
  - When you detect something in front, turn until it is no longer detected in front.
  - If detected on the sides, turn accordingly.
  - If no detection or detection on the **rear sections of the laser**, move forward.

- End of Exercise 3.3 -

- Expected Result -

- The Mars rover should detect the obstacles ahead and avoid them like so.



- Expected Result -

- RECOMMENDATION -

PLEASE TRY TO DO THE EXERCISE BEFORE LOOKING THE SOLUTION

- END RECOMMENDATION -

- Solution for Exercise 3.3 -

autonomous_exploration.py

In [ ]:

```python
#!/usr/bin/env python

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
from rclpy.qos import ReliabilityPolicy, QoSProfile

class AutonomousExplorationNode(Node):
    def __init__(self):
        super().__init__('autonomous_exploration_node')

        # Subscriber to LaserScan
        self.subscriber = self.create_subscription(
            LaserScan,
            '/laser_scan',
            self.laserscan_callback,
            QoSProfile(depth=10, reliability=ReliabilityP
        )

        # Publisher for movement commands
        self.publisher_ = self.create_publisher(Twist, '/

        # Initialize state variables
        self.turning = False
        self.turn_direction = -0.5  # Default to turning

        self.get_logger().info("Autonomous Exploration No

    def laserscan_callback(self, msg):
        # Define the sectors
        sectors = {
            "Right_Rear": (0, 33),
            "Right": (34, 66),
            "Front_Right": (67, 100),
            "Front_Left": (101, 133),
            "Left": (134, 166),
            "Left_Rear": (167, 199)
        }

        # Initialize the minimum distances for each secto
        min_distances = {key: float('inf') for key in sec

        # Find the minimum distance in each sector
        for sector, (start_idx, end_idx) in sectors.items
            # Ensure the index range is within bounds and
            if start_idx < len(msg.ranges) and end_idx <
                sector_ranges = msg.ranges[start_idx:end_
                if sector_ranges:
                    min_distances[sector] = min(sector_ra

        # Define the threshold for obstacle detection
        obstacle_threshold = 0.8  # meters

        # Determine detected obstacles
        detections = {sector: min_distance < obstacle_thr

        # Determine suggested action based on detection
        action = Twist()

        # If obstacles are detected in both front sectors
        if detections["Front_Left"] or detections["Front_
            if not self.turning:
                # Start turning if not already turning
                self.turning = True
                self.turn_direction = -0.5  # Turning rig
            action.angular.z = self.turn_direction  # Con
            self.get_logger().info('Obstacle ahead, turni
        else:
            self.turning = False  # Stop turning when the
            # Priority 2: Side detections
            if detections["Left"]:
                action.linear.x = 0.2  # Move forward slo
                action.angular.z = -0.3  # Slight right t
                self.get_logger().info('Obstacle on the l
            elif detections["Right"]:
                action.linear.x = 0.2  # Move forward slo
                action.angular.z = 0.3  # Slight left tur
                self.get_logger().info('Obstacle on the r
            # Priority 3: Rear detections
            elif detections["Right_Rear"]:
                action.linear.x = 0.3  # Move forward
                self.get_logger().info('Obstacle on the r
            elif detections["Left_Rear"]:
                action.linear.x = 0.3  # Move forward
                self.get_logger().info('Obstacle on the l
            else:
                action.linear.x = 0.5  # Move forward
                self.get_logger().info('No obstacles, mov

        # Publish the action command
        self.publisher_.publish(action)

def main(args=None):
    rclpy.init(args=args)
    node = AutonomousExplorationNode()
    rclpy.spin(node)
```

```
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

- Let's comment on the newest parts:

In [ ]:

```
# Initialize state variables
self.turning = False
self.turn_direction = -0.5  # Default to turning right
```

- First, you have to know which values make the robot turn **RIGHT** and which ones turn **LEFT**.
- A good way is to use the **teleoperation node** and the `ros2 topic echo /cmd_vel` to check which value corresponds to going left and which one to go right.
- In this Mars rover, **NEGATIVE angular values** are used to turn **RIGHT**
- In this Mars rover, **POSITIVE angular values** are used to turn **LEFT**

In [ ]:

```
if detections["Front_Left"] or detections["Front_Right"]:
    if not self.turning:
        # Start turning if not already turning
        self.turning = True
        self.turn_direction = -0.5  # Turning right
    action.angular.z = self.turn_direction  # Continue tu
    self.get_logger().info('Obstacle ahead, turning to cl
```

- This structure is meant to make the Mars rover start turning when it has an obstacle in front, and not stop until it doesn't detect anything in front. This is to avoid entering loops and getting stuck in movements that cancel each other, like turning left and then right and then left again...

setup.py

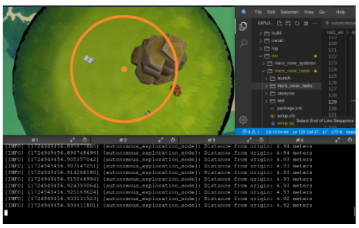
In [ ]:

```
from setuptools import find_packages, setup
import os
from glob import glob

package_name = 'mars_rover_tasks'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launc
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='user',
    maintainer_email='user@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'subscriber_obstacle_detector_executable = ma
            'publish_mars_rover_move_executable = mars_ro
            'image_plant_detector_executable = mars_rover
            'autonomous_exploration_executable = mars_rov

        ],
    },
)
```

- End of Solution for Exercise 3.3 -

- Exercise 3.4 -

- We need to make this `autonomous_exploration.py` totally autonomous.
- For that, we need the Mars rover to detect when it's too far away from the center of its exploration zone.
- Normally, we define a circle of a certain radius (**5.0 meters**), where we are going to look for life and take samples.



- Luckily, the Mars rover has an absolute **odometry system** that gives us its location respect to the place where it was turned on.
- Odometry has information of **orientation** and **position**.
- With the position, we can calculate the distance from the center.
- We can also know its orientation to calculate how much it has to turn to head to the center of the exploration area circle.
- Here is the code that shows how to do that:

In [ ]:

```
import math
import tf_transformations


...
self.current_position['x'] = msg.pose.pose.position.x
self.current_position['y'] = msg.pose.pose.position.y

# Calculate the distance from the origin (0,0)
self.distance_from_origin = math.sqrt(self.current_positi
self.get_logger().info(f'Distance from origin: {self.dist

# Calculate the yaw (orientation around the z-axis)
orientation_q = msg.pose.pose.orientation
orientation_list = [orientation_q.x, orientation_q.y, ori
(roll, pitch, self.yaw) = tf_transformations.euler_from_q

...

def return_to_origin(self):
    action = Twist()

    # Calculate the desired angle to the origin
    desired_yaw = math.atan2(-self.current_position['y'],

    # Calculate the difference between current yaw and de
    yaw_error = desired_yaw - self.yaw

    # Normalize the yaw error to the range [-pi, pi]
    yaw_error = (yaw_error + math.pi) % (2 * math.pi) - m

    # If the yaw error is significant, rotate towards the
    if abs(yaw_error) > 0.1:  # 0.1 radians threshold for
        action.angular.z = 0.5 if yaw_error > 0 else -0.5
        self.get_logger().info(f'Turning towards origin.
    else:
        # If oriented towards the origin, move forward
        action.linear.x = 0.5
        self.get_logger().info('Heading towards origin.')
```

- The `return_to_origin` is the method that you have to use to trigger the action of returning to the center until the distance to the center is lower than the radius of the exploration area.

- Remember how to get information about a `topic` :

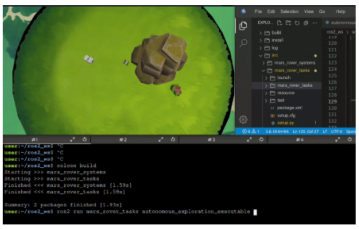
In [ ]:

```
ros2 topic info /odom
```

- End of Exercise 3.4 -

- Expected Result -

- The Mars rover should detect the obstacles but **also detect when it's exiting the 5-meter radius exploration circle and turn around.**



- Expected Result -

- RECOMMENDATION -

PLEASE TRY TO DO THE EXERCISE BEFORE LOOKING THE SOLUTION

- END RECOMMENDATION -

- Solution for Exercise 3.4 -

autonomous_exploration.py

In [ ]:

```
#!/usr/bin/env python

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from rclpy.qos import ReliabilityPolicy, QoSProfile
import math
import tf_transformations

class AutonomousExplorationNode(Node):
    def __init__(self):
        super().__init__('autonomous_exploration_node')

        # Subscriber to LaserScan
        self.subscriber_laser = self.create_subscription(
            LaserScan,
            '/laser_scan',
            self.laserscan_callback,
            QoSProfile(depth=10, reliability=ReliabilityP
        )

        # Subscriber to Odometry
        self.subscriber_odom = self.create_subscription(
            Odometry,
            '/odom',
            self.odom_callback,
            QoSProfile(depth=10, reliability=ReliabilityP
        )

        # Publisher for movement commands
        self.publisher_ = self.create_publisher(Twist, '/

        # Initialize state variables
        self.turning = False
        self.turn_direction = -0.5  # Default to turning
        self.distance_from_origin = 0.0
        self.returning_to_origin = False
        self.current_position = {'x': 0.0, 'y': 0.0}  # S
        self.yaw = 0.0  # Yaw angle of the robot

        self.get_logger().info("Autonomous Exploration No

    def laserscan_callback(self, msg):
        # If returning to the origin, skip normal explora
        if self.returning_to_origin:
            self.return_to_origin()
            return

        # Define the sectors
        sectors = {
            "Right_Rear": (0, 33),
            "Right": (34, 66),
            "Front_Right": (67, 100),
            "Front_Left": (101, 133),
            "Left": (134, 166),
            "Left_Rear": (167, 199)
        }

        # Initialize the minimum distances for each secto
        min_distances = {key: float('inf') for key in sec

        # Find the minimum distance in each sector
        for sector, (start_idx, end_idx) in sectors.items
            # Ensure the index range is within bounds and
            if start_idx < len(msg.ranges) and end_idx <
                sector_ranges = msg.ranges[start_idx:end_
                if sector_ranges:
                    min_distances[sector] = min(sector_ra

        # Define the threshold for obstacle detection
        obstacle_threshold = 0.8  # meters

        # Determine detected obstacles
        detections = {sector: min_distance < obstacle_thr

        # Determine suggested action based on detection
        action = Twist()

        # If obstacles are detected in both front sectors
        if detections["Front_Left"] or detections["Front_
            if not self.turning:
                # Start turning if not already turning
                self.turning = True
                self.turn_direction = -0.5  # Turning rig
            action.angular.z = self.turn_direction  # Con
            self.get_logger().info('Obstacle ahead, turni
        else:
            self.turning = False  # Stop turning when the
            # Priority 2: Side detections
            if detections["Left"]:
                action.linear.x = 0.2  # Move forward slo
                action.angular.z = -0.3  # Slight right t
                self.get_logger().info('Obstacle on the l
            elif detections["Right"]:
                action.linear.x = 0.2  # Move forward slo
                action.angular.z = 0.3  # Slight left tur
                self.get_logger().info('Obstacle on the r
            # Priority 3: Rear detections
            elif detections["Right_Rear"]:
                action.linear.x = 0.3  # Move forward
                self.get_logger().info('Obstacle on the r
            elif detections["Left_Rear"]:
                action.linear.x = 0.3  # Move forward
                self.get_logger().info('Obstacle on the l
            else:
                action.linear.x = 0.5  # Move forward
                self.get_logger().info('No obstacles, mov

        # Publish the action command
        self.publisher_.publish(action)

    def odom_callback(self, msg):
        # Extract the x, y coordinates from the odometry
        self.current_position['x'] = msg.pose.pose.positi
```

```
        self.current_position['y'] = msg.pose.pose.positi

        # Calculate the distance from the origin (0,0)
        self.distance_from_origin = math.sqrt(self.curren
        self.get_logger().info(f'Distance from origin: {s

        # Calculate the yaw (orientation around the z-axi
        orientation_q = msg.pose.pose.orientation
        orientation_list = [orientation_q.x, orientation_
        (roll, pitch, self.yaw) = tf_transformations.eule

        # If the distance exceeds PERIMETER_DIST meters,
        PERIMETER_DIST = 5.0
        if self.distance_from_origin > PERIMETER_DIST:
            self.returning_to_origin = True
        elif self.distance_from_origin <= PERIMETER_DIST
            # If the rover is back within 7 meters of the
            self.returning_to_origin = False
            self.get_logger().info('Within 7 meters of or

    def return_to_origin(self):
        action = Twist()

        # Calculate the desired angle to the origin
        desired_yaw = math.atan2(-self.current_position['

        # Calculate the difference between current yaw an
        yaw_error = desired_yaw - self.yaw

        # Normalize the yaw error to the range [-pi, pi]
        yaw_error = (yaw_error + math.pi) % (2 * math.pi)

        # If the yaw error is significant, rotate towards
        if abs(yaw_error) > 0.1:  # 0.1 radians threshold
            action.angular.z = 0.5 if yaw_error > 0 else
            self.get_logger().info(f'Turning towards orig
        else:
            # If oriented towards the origin, move forwar
            action.linear.x = 0.5
            self.get_logger().info('Heading towards origi

        # Publish the action command
        self.publisher_.publish(action)

def main(args=None):
    rclpy.init(args=args)
    node = AutonomousExplorationNode()
    rclpy.spin(node)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

setup.py

In [ ]:

```
from setuptools import find_packages, setup
import os
from glob import glob

package_name = 'mars_rover_tasks'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launc
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='user',
    maintainer_email='user@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'subscriber_obstacle_detector_executable = ma
            'publish_mars_rover_move_executable = mars_ro
            'image_plant_detector_executable = mars_rover
            'autonomous_exploration_executable = mars_rov

        ],
    },
)
```

- End of Solution for Exercise 3.4 -

24/09/2024