ACW1 – SIMULATION OF ROBOTIC SYSTEMS

Wiktor Biesiadecki

551530 – Robotic systems and simulation

Dr P A Robinson

University of Hull

11/12/2025

Introduction

The goal of the assignment is to utilise learned skills and demonstrate proficiency in ROS2 by completing 4 tasks gradually increasing in complexity. Successful completion of each task means delivery of turtle sim simulation programmed in python, video demonstrating working solution and result as well as report combining detailed explanation of how each task was completed. For each example below I'm using ROS2 Humble running on Ubuntu 22.04.

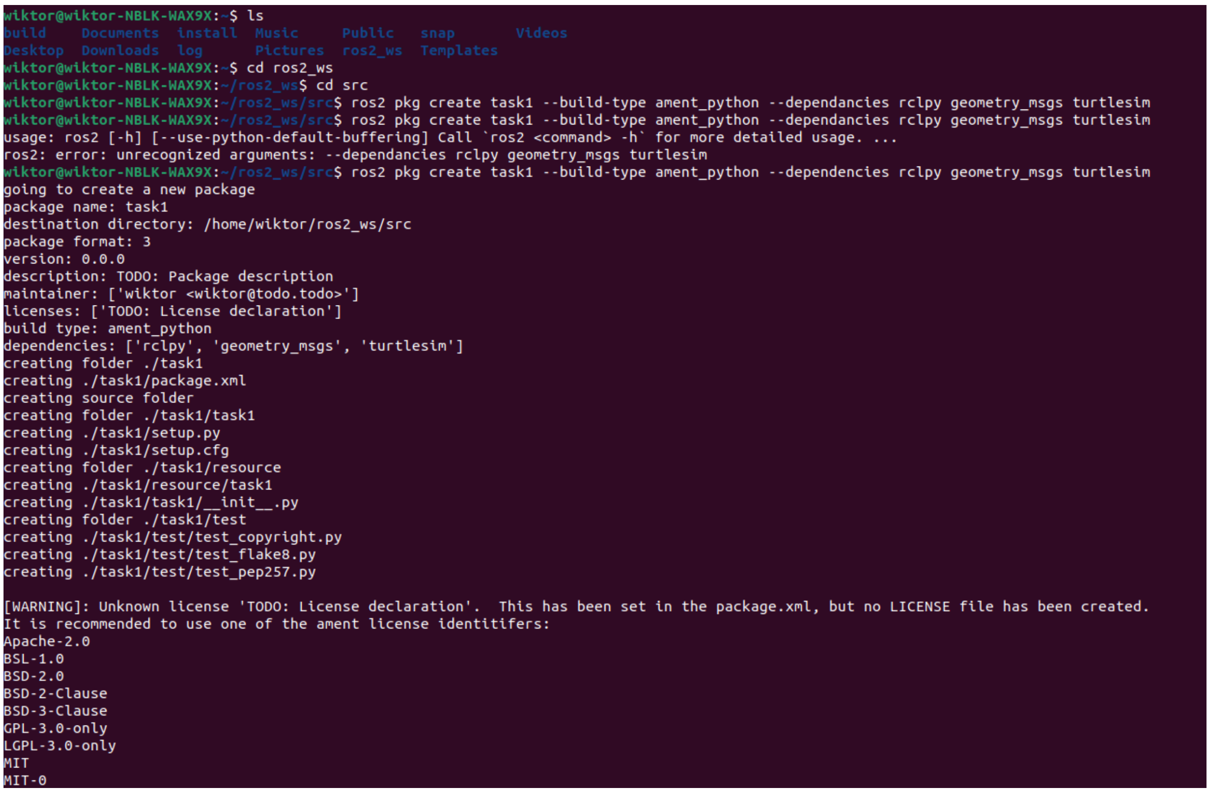
Task 1

Overview

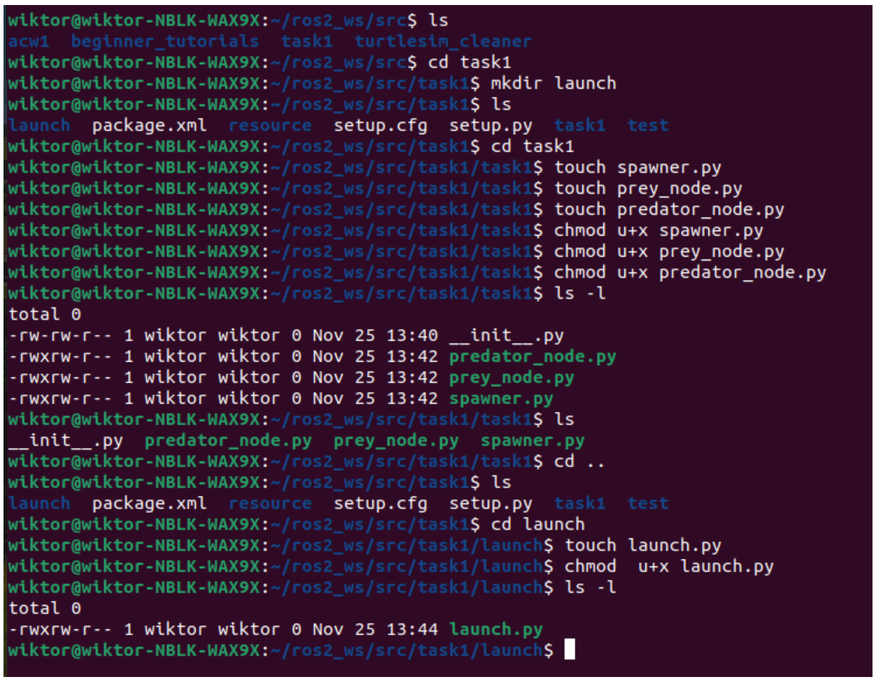
In first task my goal was to spawn two turtles in random positions – predator and prey. The prey will remain stationary, publishing its coordinates to prey_location topic. Predator will move to prey's position retrieving coordinates from prey_location with use of Euclidean distance and angle, without finding optimal path. Predator will then stop when it gets very close to the prey.

Implementation

First to achieve the objective I created new ros2 package in ros2_ws/src. I've called it task1. Next, I had to create files responsible for required behaviour of this program. They are spawner.py, predator_node.py, prey_node.py and launch.py in launch directory. I also made all these files executable using chmod u+x.

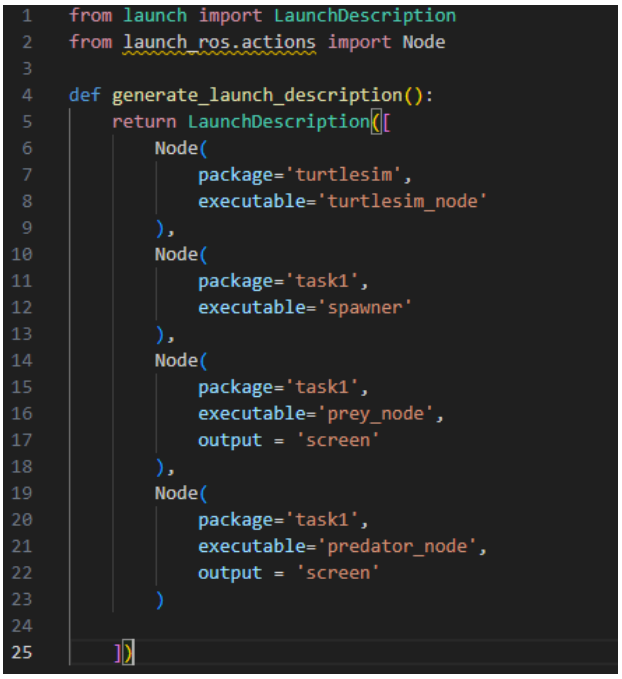


```
wiktor@wiktor-NBLK-WAX9X:~$ ls
build    Documents  install  Music     Public   snap       Videos
Desktop  Downloads  log      Pictures  ros2_ws  Templates
wiktor@wiktor-NBLK-WAX9X:~$ cd ros2_ws
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws$ cd src
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src$ ros2 pkg create task1 --build-type ament_python --dependancies rclpy geometry_msgs turtlesim
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src$ ros2 pkg create task1 --build-type ament_python --dependancies rclpy geometry_msgs turtlesim
usage: ros2 [-h] [--use-python-default-buffering] Call `ros2 <command> -h` for more detailed usage. ...
ros2: error: unrecognized arguments: --dependancies rclpy geometry_msgs turtlesim
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src$ ros2 pkg create task1 --build-type ament_python --dependencies rclpy geometry_msgs turtlesim
going to create a new package
package name: task1
destination directory: /home/wiktor/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['wiktor <wiktor@todo.todo>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: ['rclpy', 'geometry_msgs', 'turtlesim']
creating folder ./task1
creating ./task1/package.xml
creating source folder
creating folder ./task1/task1
creating ./task1/setup.py
creating ./task1/setup.cfg
creating folder ./task1/resource
creating ./task1/resource/task1
creating ./task1/task1/__init__.py
creating folder ./task1/test
creating ./task1/test/test_copyright.py
creating ./task1/test/test_flake8.py
creating ./task1/test/test_pep257.py

[WARNING]: Unknown license 'TODO: License declaration'.  This has been set in the package.xml, but no LICENSE file has been created.
It is recommended to use one of the ament license identitifers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```



```
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src$ ls
acw1  beginner_tutorials  task1  turtlesim_cleaner
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src$ cd task1
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1$ mkdir launch
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1$ ls
launch  package.xml  resource  setup.cfg  setup.py  task1  test
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1$ cd task1
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/task1$ touch spawner.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/task1$ touch prey_node.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/task1$ touch predator_node.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/task1$ chmod u+x spawner.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/task1$ chmod u+x prey_node.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/task1$ chmod u+x predator_node.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/task1$ ls -l
total 0
-rw-rw-r-- 1 wiktor wiktor 0 Nov 25 13:40 __init__.py
-rwxrw-r-- 1 wiktor wiktor 0 Nov 25 13:42 predator_node.py
-rwxrw-r-- 1 wiktor wiktor 0 Nov 25 13:42 prey_node.py
-rwxrw-r-- 1 wiktor wiktor 0 Nov 25 13:42 spawner.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/task1$ ls
__init__.py  predator_node.py  prey_node.py  spawner.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/task1$ cd ..
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1$ ls
launch  package.xml  resource  setup.cfg  setup.py  task1  test
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1$ cd launch
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/launch$ touch launch.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/launch$ chmod  u+x launch.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/launch$ ls -l
total 0
-rwxrw-r-- 1 wiktor wiktor 0 Nov 25 13:44 launch.py
wiktor@wiktor-NBLK-WAX9X:~/ros2_ws/src/task1/launch$ 
```

At this point I had to code relevant behaviour in each file. Purpose of launch.py is to launch multiple nodes necessary for this simulation and configure their communication settings. In this example launch.py file handles turtlesim_node which is environment program is running in, spawner, predator_node and_prey_node.

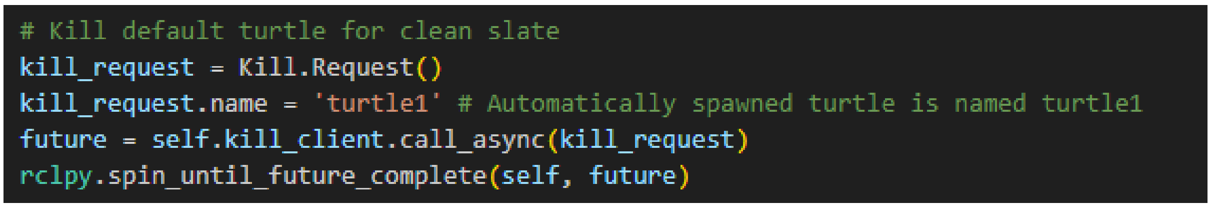


```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node'
        ),
        Node(
            package='task1',
            executable='spawner'
        ),
        Node(
            package='task1',
            executable='prey_node',
            output = 'screen'
        ),
        Node(
            package='task1',
            executable='predator_node',
            output = 'screen'
        )

    ])
```

Above we can see that launcher file launches 4 nodes used in this simulation. Each node has information about executable file launcher needs to look for as well as package where it's located. Predator and prey have additional output setting that allow them to display information in terminal when prey gets caught or escapes.

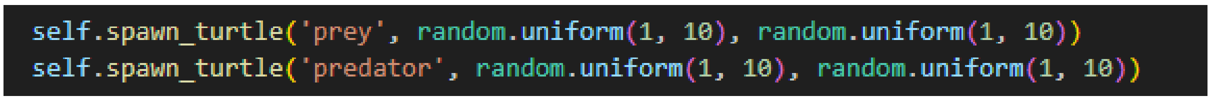
File spawner.py removes default turtle that always spawns in the middle and replaces it with predator and prey spawned at random locations.

Removing default turtle:



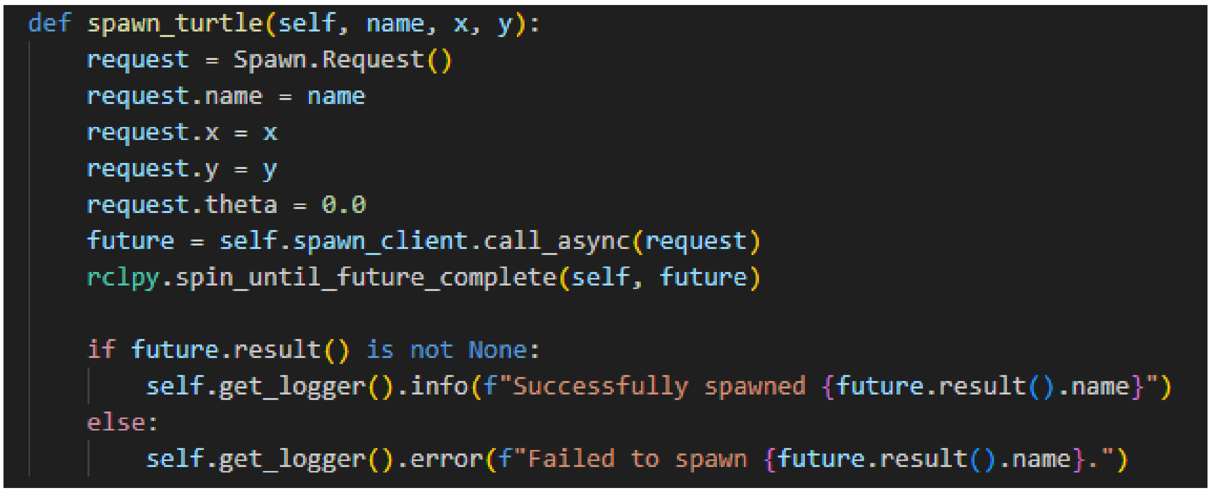
```
# Kill default turtle for clean slate
kill_request = Kill.Request()
kill_request.name = 'turtle1' # Automatically spawned turtle is named turtle1
future = self.kill_client.call_async(kill_request)
rclpy.spin_until_future_complete(self, future)
```

Spawning predator and prey:



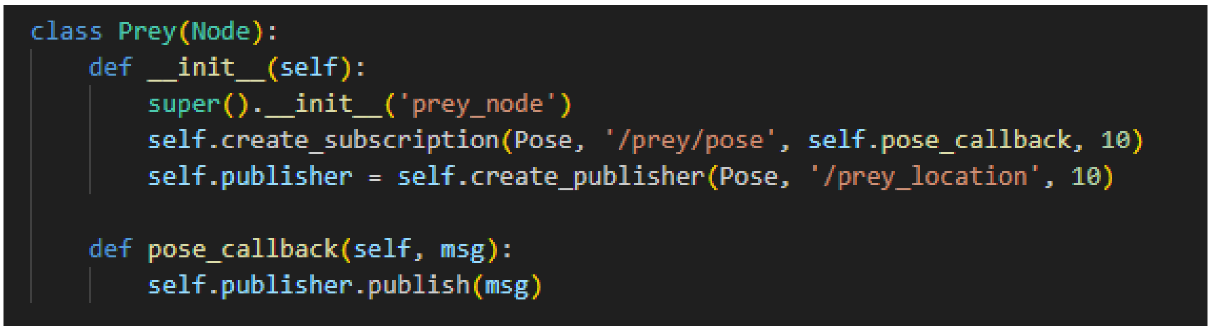
```
self.spawn_turtle('prey', random.uniform(1, 10), random.uniform(1, 10))
self.spawn_turtle('predator', random.uniform(1, 10), random.uniform(1, 10))
```

Helper function to spawn turtle:



```
def spawn_turtle(self, name, x, y):
    request = Spawn.Request()
    request.name = name
    request.x = x
    request.y = y
    request.theta = 0.0
    future = self.spawn_client.call_async(request)
    rclpy.spin_until_future_complete(self, future)

    if future.result() is not None:
        self.get_logger().info(f"Successfully spawned {future.result().name}")
    else:
        self.get_logger().error(f"Failed to spawn {future.result().name}.")
```

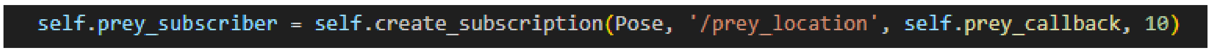
prey_node dictates behaviour of prey. In this example the file has very simple structure. Sole functionality of this file revolves around publishing location of prey.



```
class Prey(Node):
    def __init__(self):
        super().__init__('prey_node')
        self.create_subscription(Pose, '/prey/pose', self.pose_callback, 10)
        self.publisher = self.create_publisher(Pose, '/prey_location', 10)

    def pose_callback(self, msg):
        self.publisher.publish(msg)
```

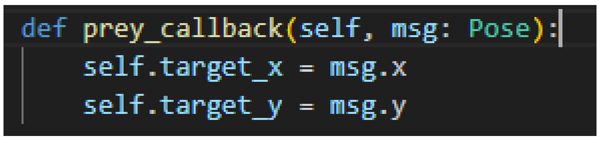
By far the most complex is predator_node. This node reads position of prey, then calculates Euclidean distance and angle between itself and prey and executes manoeuvre towards prey location, then stops upon reaching its coordinates.

Subscriber to prey’s location:



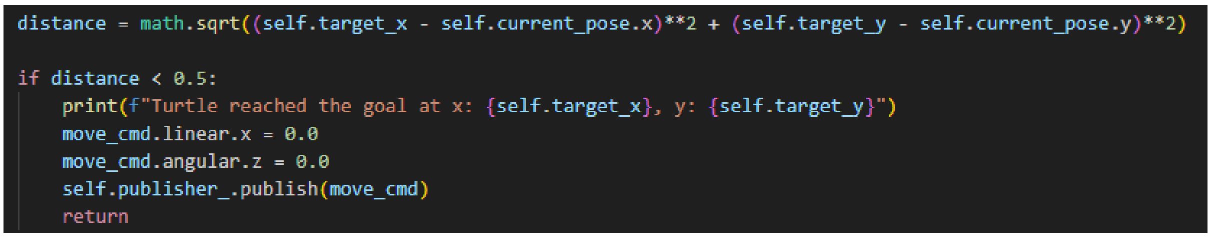
```
self.prey_subscriber = self.create_subscription(Pose, '/prey_location', self.prey_callback, 10)
```

Callback function to acquire prey’s coordinates:



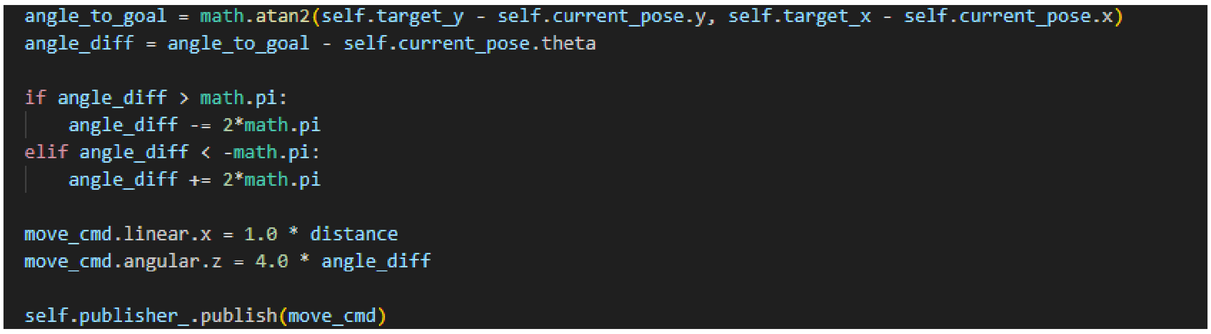
```
def prey_callback(self, msg: Pose):
    self.target_x = msg.x
    self.target_y = msg.y
```

Function move_turtle checks if prey was caught by calculating distance between predator and prey:



```
distance = math.sqrt((self.target_x - self.current_pose.x)**2 + (self.target_y - self.current_pose.y)**2)

if distance < 0.5:
    print(f"Turtle reached the goal at x: {self.target_x}, y: {self.target_y}")
    move_cmd.linear.x = 0.0
    move_cmd.angular.z = 0.0
    self.publisher_.publish(move_cmd)
    return
```

And if condition is not met it will calculate trajectory to the prey, and publish commands that predator will follow to reach prey’s position:



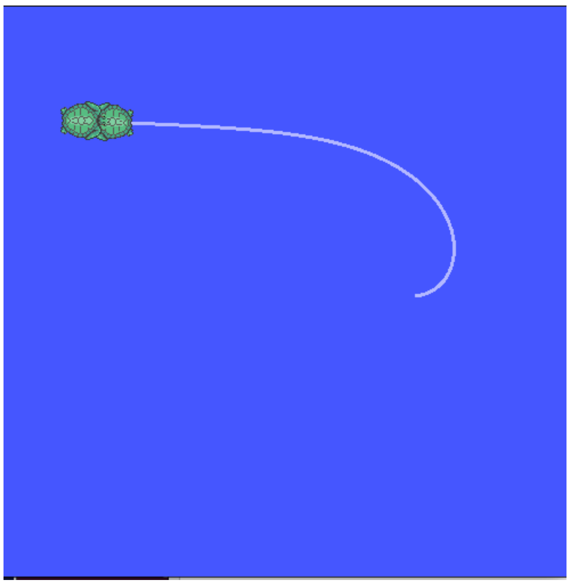
```
angle_to_goal = math.atan2(self.target_y - self.current_pose.y, self.target_x - self.current_pose.x)
angle_diff = angle_to_goal - self.current_pose.theta

if angle_diff > math.pi:
    angle_diff -= 2*math.pi
elif angle_diff < -math.pi:
    angle_diff += 2*math.pi

move_cmd.linear.x = 1.0 * distance
move_cmd.angular.z = 4.0 * angle_diff

self.publisher_.publish(move_cmd)
```

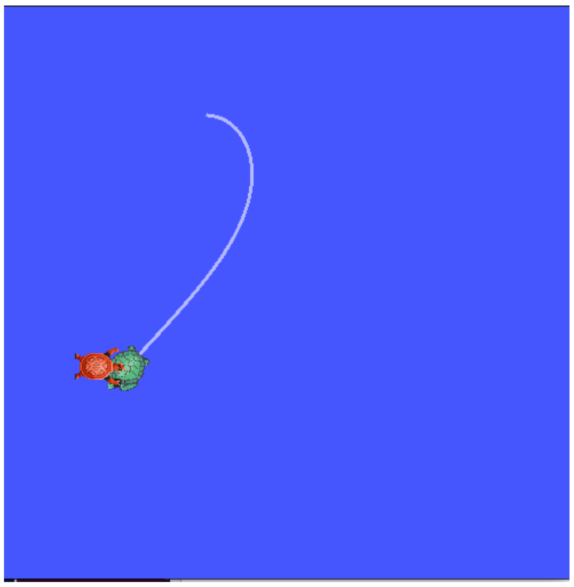
Full code for all files available in 551530_code_202421412.zip/taskl

Functionality

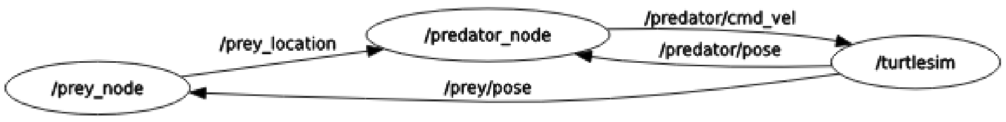
Upon start we can see that default turtle is not present and two other turtles are spawned in random places in available space. Shortly after we can see that predator makes adequate moves and adjusts its position to turn towards the prey and reach its location. Predator stops when reaching prey. In this example predator is successful 100% of times, as there’s nothing preventing him from founding the prey.

Examples of simulation running, note different starting positions of predator and prey each time simulation runs:





Using rqt_graph we can see relations between nodes of this program:



Task 2

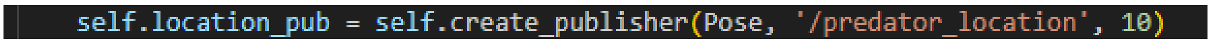
Overview

Second task is a modification of first therefore it will carry over a lot of the same code and functionality from task 1. Most notable difference was that prey had to read predator's location and try to get away from it, by calculating a path away from incoming threat. Second difference was that simulation would not only stop when predator reached pray but also when it wouldn't be able to do so in 60 seconds or less.

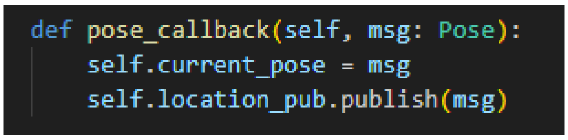
Implementation

As this task expands on functionality of task 1, in this section I will focus on highlighting differences and how they overall functionality of the simulation.

Predator_node now publishes its location to prey:



```
    self.location_pub = self.create_publisher(Pose, '/predator_location', 10)
```

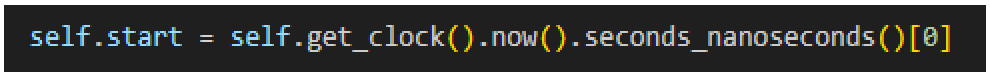


```
def pose_callback(self, msg: Pose):
    self.current_pose = msg
    self.location_pub.publish(msg)
```

The most notable changes happened in prey_node. Right now instead of remaining idle prey will try to actively evade predator while simultaneously avoiding colliding with walls.

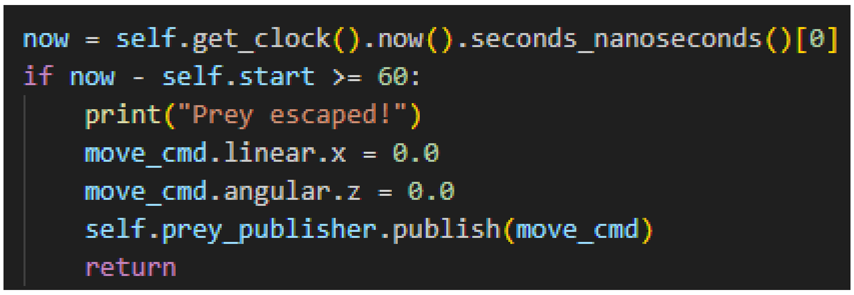
First to meet the requirements of this task 60s timer have been added.

Declaring variable start with current time:



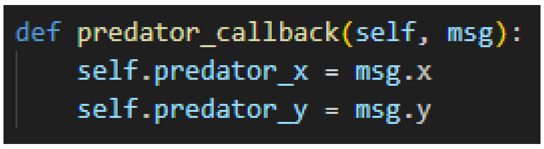
```
self.start = self.get_clock().now().seconds_nanoseconds()[0]
```

Checking when 60s passed, stopping turtle and ending simulation:



```
now = self.get_clock().now().seconds_nanoseconds()[0]
if now - self.start >= 60:
    print("Prey escaped!")
    move_cmd.linear.x = 0.0
    move_cmd.angular.z = 0.0
    self.prey_publisher.publish(move_cmd)
    return
```

Prey now has ability to move and does so in similar fashion as predator turtle. The main difference between the two is prey moves away from predator's position obtained by subscriber:



```
def predator_callback(self, msg):
    self.predator_x = msg.x
    self.predator_y = msg.y
```

```
dx = self.current_pose.x - self.predator_x
dy = self.current_pose.y - self.predator_y
```

```
escape_angle = math.atan2(dy, dx)
angle_diff = escape_angle - self.current_pose.theta

if angle_diff > math.pi:
    angle_diff -= 2 * math.pi
elif angle_diff < -math.pi:
    angle_diff += 2 * math.pi

move_cmd.linear.x = 2.0
move_cmd.angular.z = 4.0 * angle_diff
self.prey_publisher.publish(move_cmd)
```

Compared to task 1 linear velocity x is not scaling proportionally to distance as this resulted in turtles catching up really quickly from across the simulation area and then following each other at normal pace.

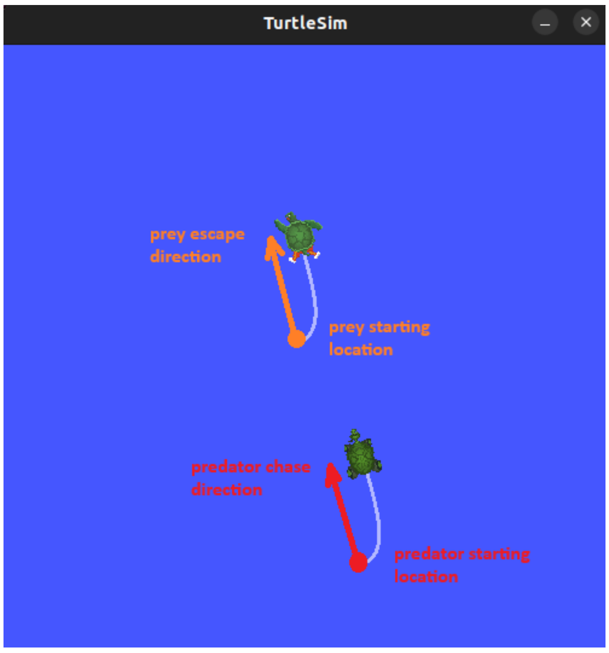
If prey comes too close to a wall components of vector are adjusted to turn away from it:

```
if self.current_pose.x <= 1.0 or self.current_pose.x >= 10.0:
    dx = -dx
if self.current_pose.y <= 1.0 or self.current_pose.y >= 10.0:
    dy = -dy
```

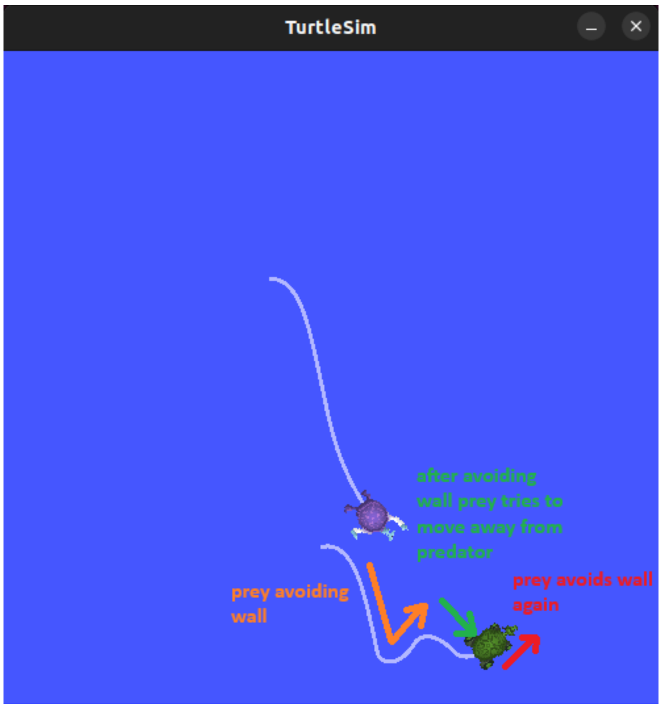
Full code for all files available in 551530_code_202421412.zip/task2

Functionality

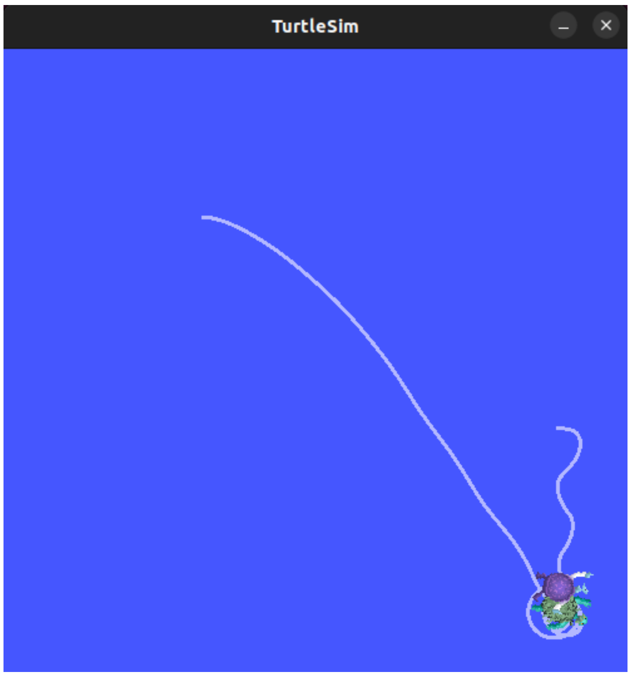
At the start of the simulation, we can see 2 turtles spawned at random locations, then immediately prey starts moving in the opposite location of predator.



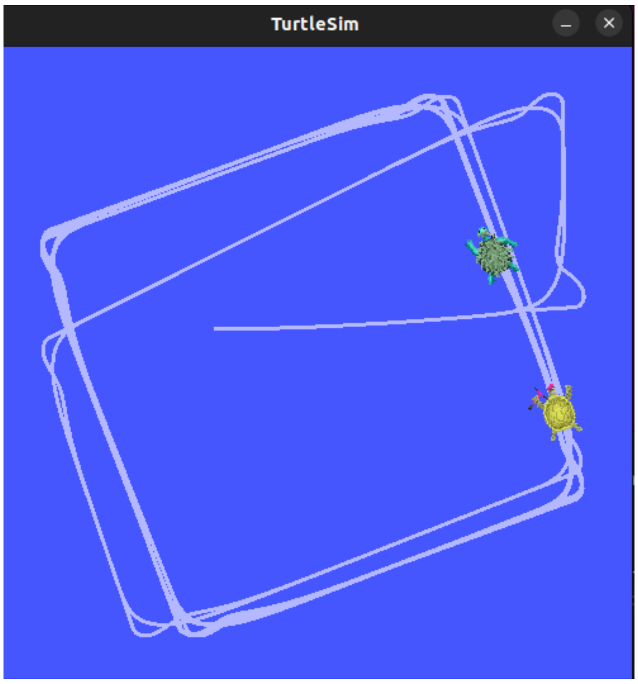
Prey will try to escape in opposite direction until it reaches the wall. Then it will bounce off the wall at opposite angle and continue in new direction. Sometimes this causes wavy move pattern as prey switches between avoiding wall and predator.



Simulation ends after 60s if predator is not able to catch the prey or when the prey is caught. Escape algorithm is not perfect in current implementation as prey doesn't always make the best choice to escape. Simplicity of this algorithm causes prey to be stuck in the corner moving in circles where predator can easily catch up.

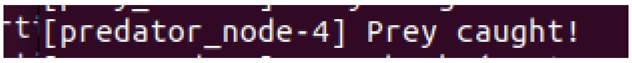


If they prey doesn't get stuck in the corner it has decent chances of escaping pursuit.



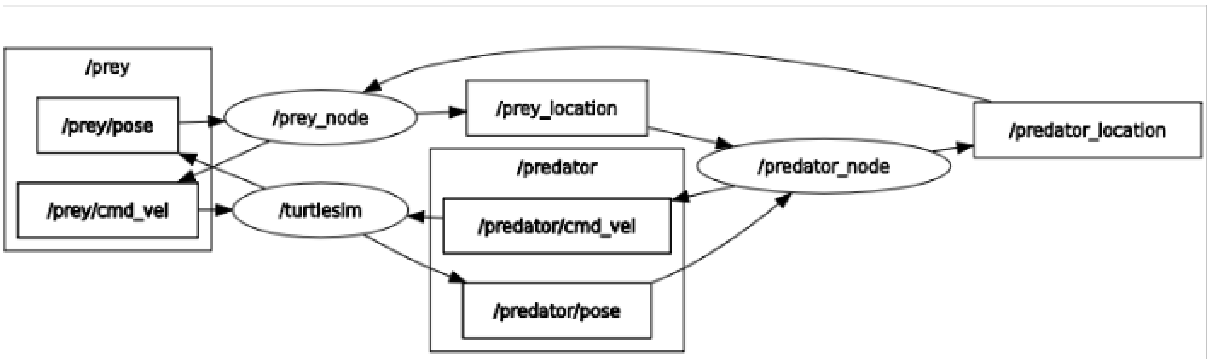
Relevant messages are displayed in terminal to notify user of the outcome.

```
Start [prey_node-3] Prey escaped!
```



```
t [predator_node-4] Prey caught!
```

Using rqt_graph we can see relations between nodes of this program:



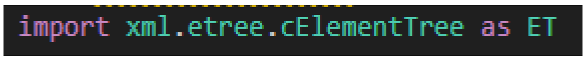
Task 3

Overview

Main goal of this task is to showcase the ability of simulation to store data in file, then load it during next run to use it to predict path. Code for this task is heavily based on task nr 2. Most notable change is added implementation for xml files processing. To better showcase desired learning behaviour of predator turtle I decided to spawn them in set locations. This way predator turtle gets ahead of prey and is able to catch it much faster. When random spawning points were still in place learning behaviour was counterproductive and was working against predator.

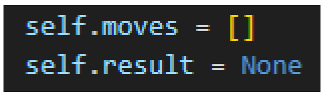
Implementation

Code used in this example bears a lot of similarities to task 2, but there are some key differences. They start with prey node which is not responsible for storing data from the run and saving it to file when simulation ends. To achieve this functionality first relevant imports for file manipulation are added.



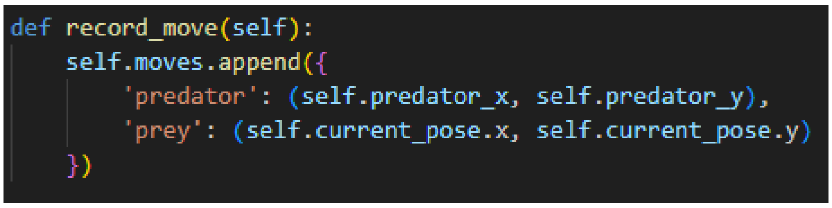
```python
import xml.etree.cElementTree as ET
```

Variables to store coordinates and result of the run.



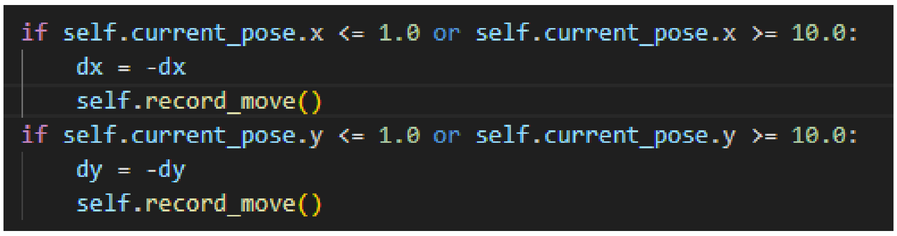
```python
self.moves = []
self.result = None
```

A function that appends coordinates to previously initialised list.



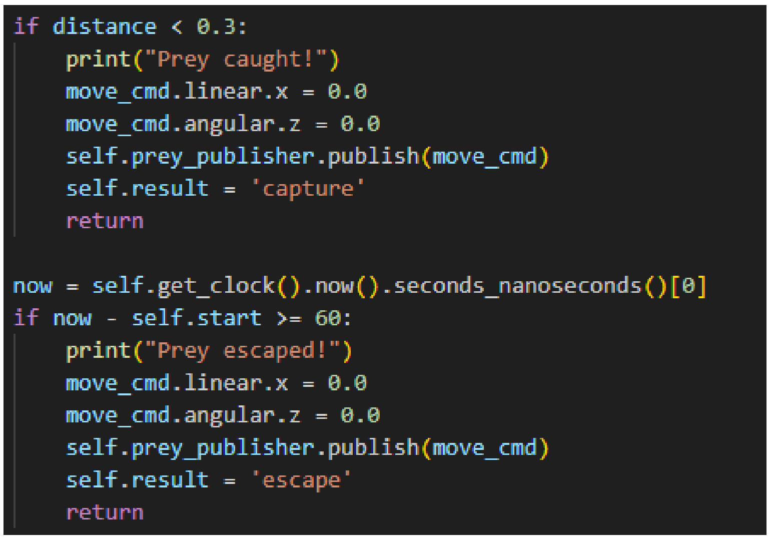
```python
def record_move(self):
    self.moves.append({
        'predator': (self.predator_x, self.predator_y),
        'prey': (self.current_pose.x, self.current_pose.y)
    })
```

This function is called every time prey makes turn to avoid walls.



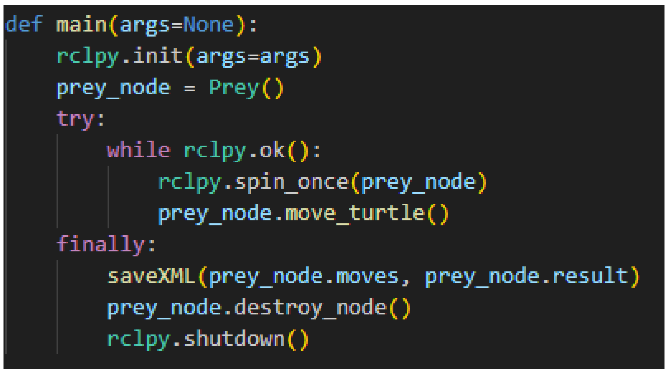
```python
if self.current_pose.x <= 1.0 or self.current_pose.x >= 10.0:
    dx = -dx
    self.record_move()
if self.current_pose.y <= 1.0 or self.current_pose.y >= 10.0:
    dy = -dy
    self.record_move()
```

On top of that functions responsible for checking win conditions now assign correct result outcome at the end of simulation.



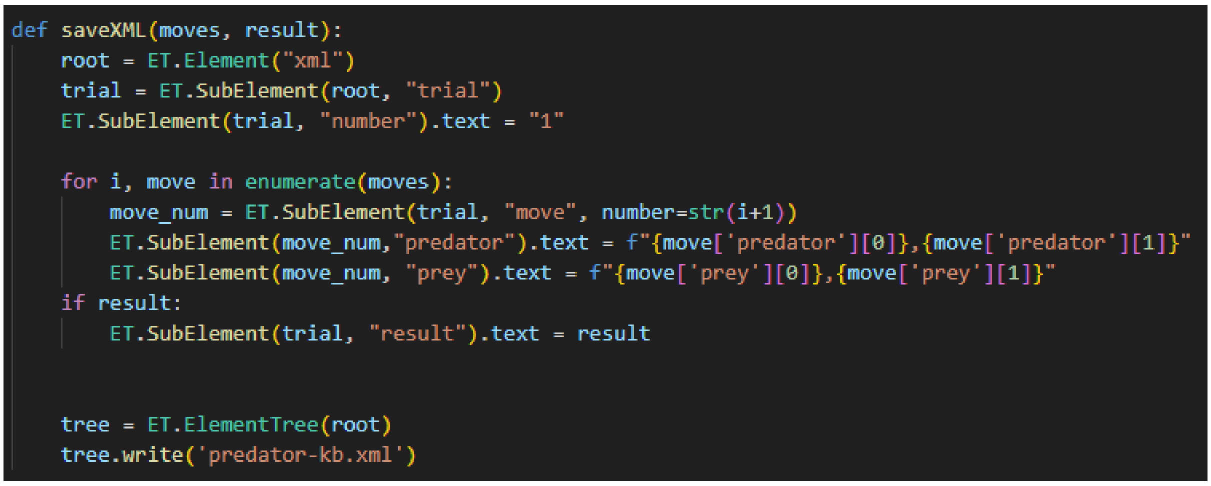
```
if distance < 0.3:
    print("Prey caught!")
    move_cmd.linear.x = 0.0
    move_cmd.angular.z = 0.0
    self.prey_publisher.publish(move_cmd)
    self.result = 'capture'
    return

now = self.get_clock().now().seconds_nanoseconds()[0]
if now - self.start >= 60:
    print("Prey escaped!")
    move_cmd.linear.x = 0.0
    move_cmd.angular.z = 0.0
    self.prey_publisher.publish(move_cmd)
    self.result = 'escape'
    return
```

When simulation is finished function responsible for saving xml file is called.



```
def main(args=None):
    rclpy.init(args=args)
    prey_node = Prey()
    try:
        while rclpy.ok():
            rclpy.spin_once(prey_node)
            prey_node.move_turtle()
    finally:
        saveXML(prey_node.moves, prey_node.result)
        prey_node.destroy_node()
        rclpy.shutdown()
```

This function enumerates each change of direction, saves data from moves list in desired format and saves result of the simulation.



```
def saveXML(moves, result):
    root = ET.Element("xml")
    trial = ET.SubElement(root, "trial")
    ET.SubElement(trial, "number").text = "1"

    for i, move in enumerate(moves):
        move_num = ET.SubElement(trial, "move", number=str(i+1))
        ET.SubElement(move_num,"predator").text = f"{move['predator'][0]},{move['predator'][1]}"
        ET.SubElement(move_num, "prey").text = f"{move['prey'][0]},{move['prey'][1]}"
    if result:
        ET.SubElement(trial, "result").text = result


    tree = ET.ElementTree(root)
    tree.write('predator-kb.xml')
```

Similarly, changes were applied to predator node as well.

Imports for file manipulation and reading file from drive.

```
from xml.dom import minidom
import os
```

New variables initialised to store coordinates of virtual target and Boolean value for condition when this target is reached, and predator can resume chase.

```
self.virtual_x = None
self.virtual_y = None

self.virtual_reached = False
```

Reading data from saved file. In this case code will read coordinates from move number 200.

```
if os.path.exists('predator-kb.xml'):
    xmldoc = minidom.parse('predator-kb.xml')
    move = xmldoc.getElementsByTagName("move")
    prey = move[200].getElementsByTagName("prey")[0]
    virtual_coordinates = prey.firstChild.data.split(',')
    self.virtual_x = float(virtual_coordinates[0])
    self.virtual_y = float(virtual_coordinates[1])
```

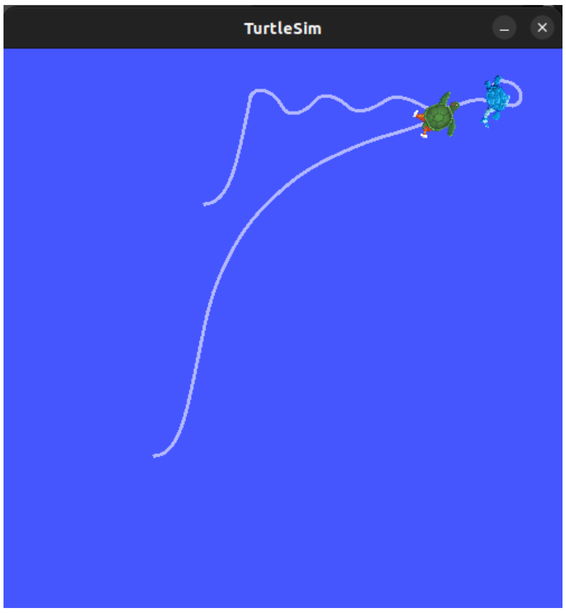
Move_turtle function was modified. First when virtual_reached is set to false this function will take coordinates of virtual target to follow. When that target is reached value will be set to True and function will take coordinates of prey instead.

```
def move_turtle(self):
    if not self.virtual_reached and self.virtual_x is not None and self.virtual_y is not None:
        target_x = self.virtual_x
        target_y = self.virtual_y
    elif self.target_x is not None and self.target_y is not None:
        target_x = self.target_x
        target_y = self.target_y
    else:
        return

    move_cmd = Twist()
    distance = math.sqrt((target_x - self.current_pose.x) ** 2 + (target_y - self.current_pose.y) ** 2)

    if not self.virtual_reached and distance < 0.3:
        self.virtual_reached = True
        return
```

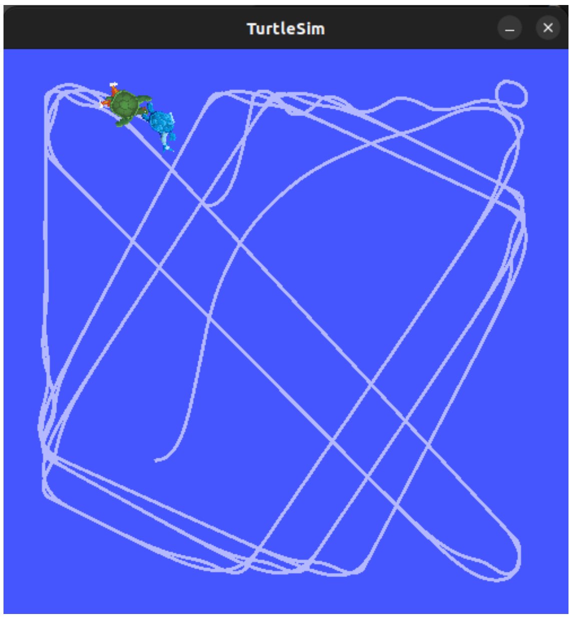
Full code for all files available in 551530_code_202421412.zip/task3

Functionality

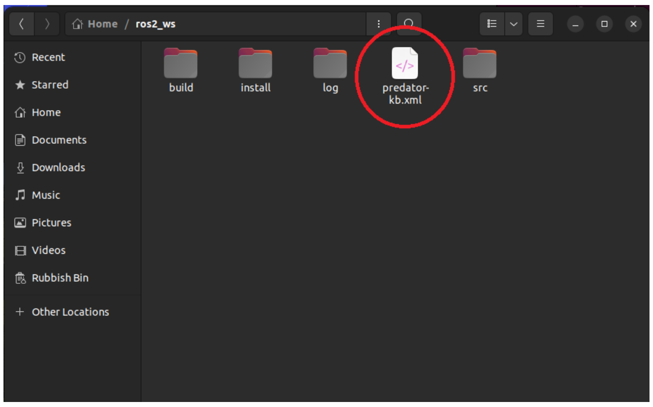
During first run simulation plays out similarly to task2. We can see prey escaping and predator following.



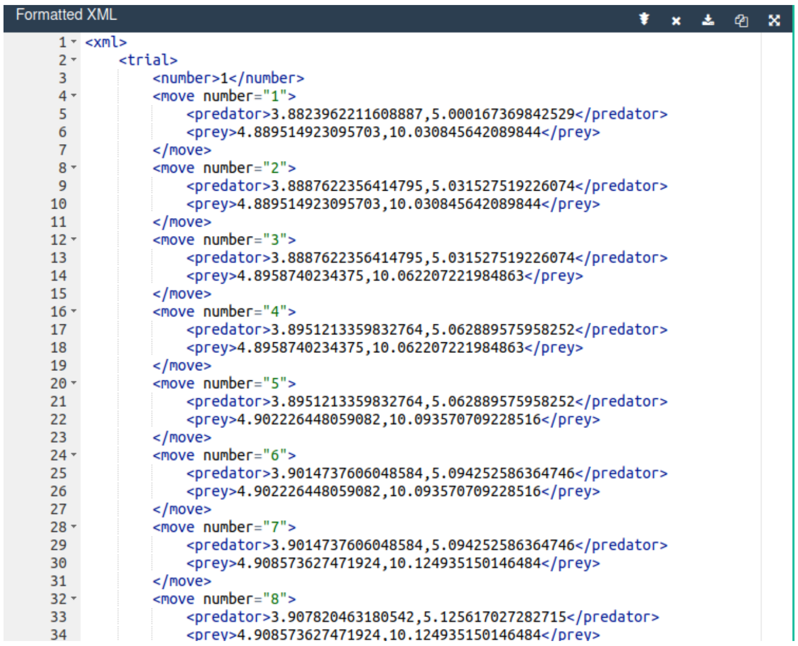
On first run predator follows prey in usual pattern and rest of the simulation plays out as usual. In this case it resulted in prey escaping, simulation ended after 60s.



After simulation finished .xml file was saved in main directory containing records of previous run.

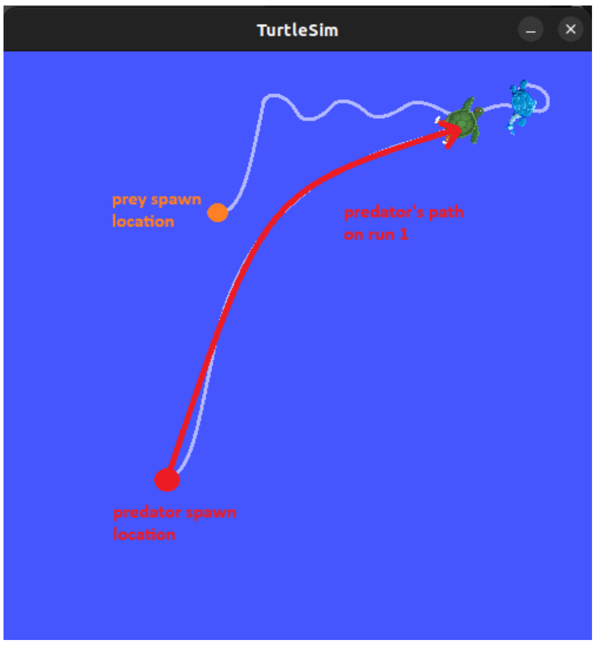


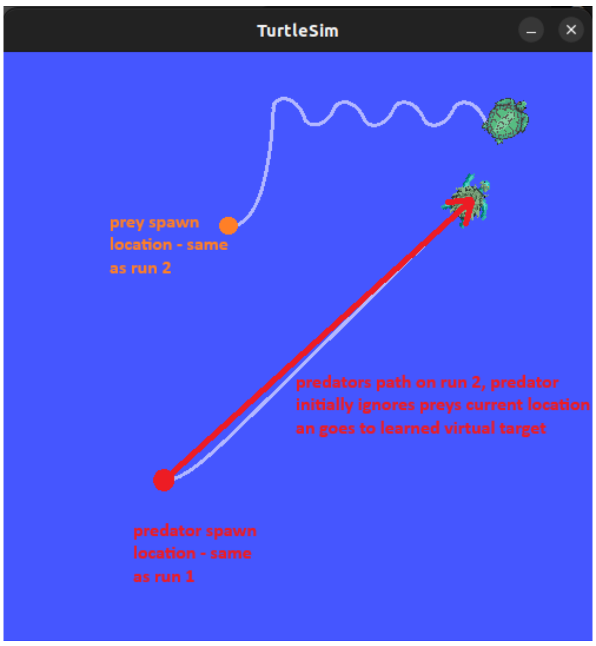
The structure of the file can be seen below.



```
<xml>
    <trial>
        <number>1</number>
        <move number="1">
            <predator>3.8823962211608887,5.000167369842529</predator>
            <prey>4.889514923095703,10.030845642089844</prey>
        </move>
        <move number="2">
            <predator>3.8887622356414795,5.031527519226074</predator>
            <prey>4.889514923095703,10.030845642089844</prey>
        </move>
        <move number="3">
            <predator>3.8887622356414795,5.031527519226074</predator>
            <prey>4.8958740234375,10.062207221984863</prey>
        </move>
        <move number="4">
            <predator>3.8951213359832764,5.062889575958252</predator>
            <prey>4.8958740234375,10.062207221984863</prey>
        </move>
        <move number="5">
            <predator>3.8951213359832764,5.062889575958252</predator>
            <prey>4.902226448059082,10.093570709228516</prey>
        </move>
        <move number="6">
            <predator>3.9014737606048584,5.094252586364746</predator>
            <prey>4.902226448059082,10.093570709228516</prey>
        </move>
        <move number="7">
            <predator>3.9014737606048584,5.094252586364746</predator>
            <prey>4.908573627471924,10.124935150146484</prey>
        </move>
        <move number="8">
            <predator>3.907820463180542,5.125617027282715</predator>
            <prey>4.908573627471924,10.124935150146484</prey>
```

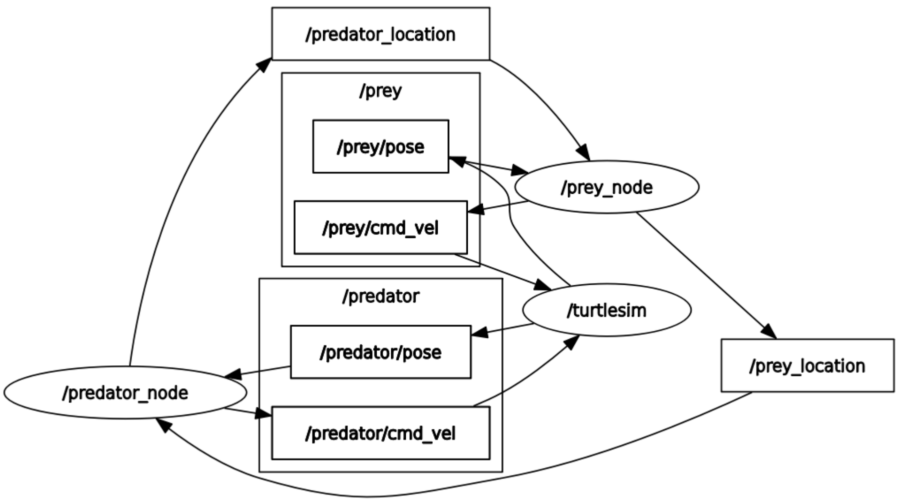
When we start the simulation again, we can clearly see that predator turtle takes different path at the beginning going to one of the saved locations of prey from previous run to intercept it. This is main reason why spawning point were fixed for this example to demonstrate effectiveness of this learning method. With random spawn points this method was mostly counterproductive.




TurtleSim
prey spawn location - same as run 2
predators path on run 2, predator initially ignores preys current location an goes to learned virtual target
predator spawn location - same as run 1

Run 2 resulted in much faster capture of prey. This approach works mostly well, but during testing on occasion prey can initially turn left to get away from predator due to changed trajectory in this scenario predator will initially go in the opposite direction of prey until it reaches virtual target, then resumes chase in correct direction.

Using rqt_graph we can see relations between nodes of this program:



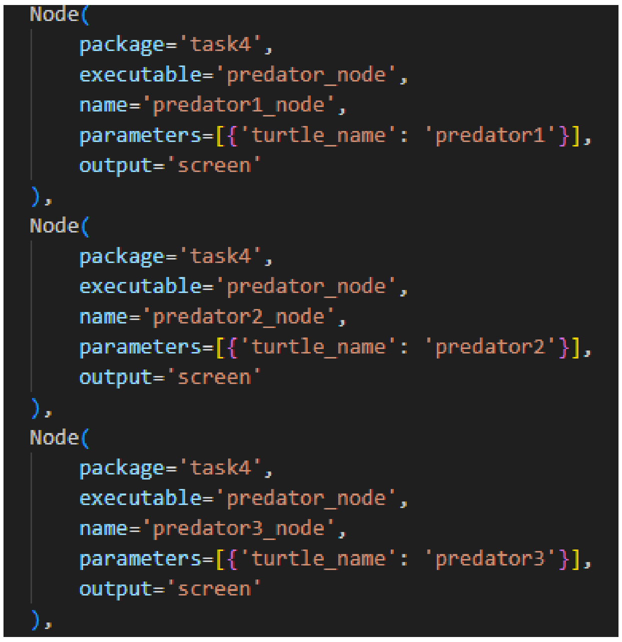
Task4

Overview

Main goal of this task is to create simulation where prey needs to escape multiple predators. Each predator acts independently to capture prey, who needs to avoid them. This task shares a lot of similarities with task 2, but prey needs to consider all 3 predators now in order to escape. These changes necessitated in relevant code modifications to spawn additional turtles, reworked subscriber-publisher system so that prey could get awareness of all predators, as well as changes to escape algorithm.

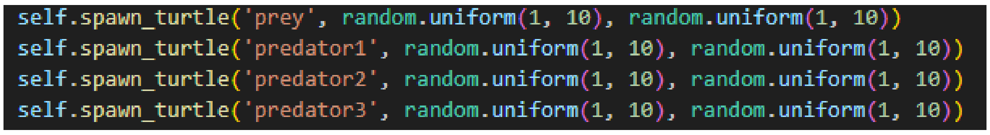
Implementation

Launcher file had to be modified to launch 3 separate predator nodes.



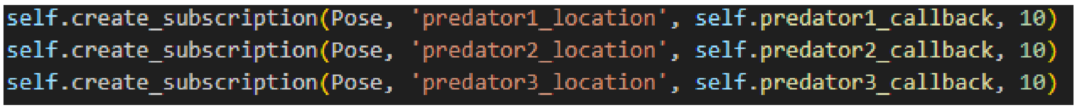
```
Node(
    package='task4',
    executable='predator_node',
    name='predator1_node',
    parameters=[{'turtle_name': 'predator1'}],
    output='screen'
),
Node(
    package='task4',
    executable='predator_node',
    name='predator2_node',
    parameters=[{'turtle_name': 'predator2'}],
    output='screen'
),
Node(
    package='task4',
    executable='predator_node',
    name='predator3_node',
    parameters=[{'turtle_name': 'predator3'}],
    output='screen'
),
```

As well as spawner to spawn 4 turtles – 1 prey and 3 predators.

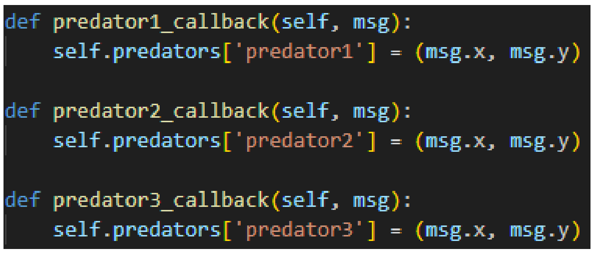


```
self.spawn_turtle('prey', random.uniform(1, 10), random.uniform(1, 10))
self.spawn_turtle('predator1', random.uniform(1, 10), random.uniform(1, 10))
self.spawn_turtle('predator2', random.uniform(1, 10), random.uniform(1, 10))
self.spawn_turtle('predator3', random.uniform(1, 10), random.uniform(1, 10))
```

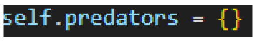
Predator node remained unchanged as its behaviour didn't need any modifications, and current implementation works perfectly fine when 3 nodes are running at the same time. Most notable changes were done to prey node. Right now, it subscribes to 3 predators individually, reads their positions with callback functions. This data is stored in dictionary.



```
self.create_subscription(Pose, 'predator1_location', self.predator1_callback, 10)
self.create_subscription(Pose, 'predator2_location', self.predator2_callback, 10)
self.create_subscription(Pose, 'predator3_location', self.predator3_callback, 10)
```

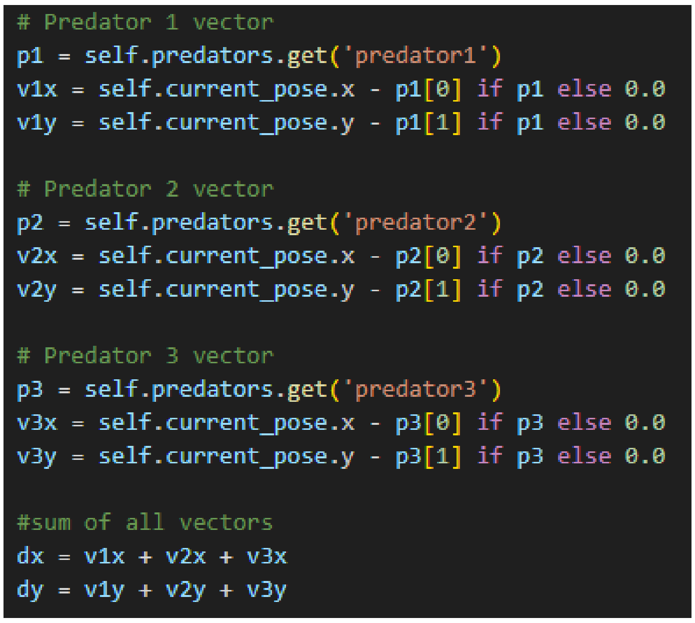


```python
def predator1_callback(self, msg):
    self.predators['predator1'] = (msg.x, msg.y)

def predator2_callback(self, msg):
    self.predators['predator2'] = (msg.x, msg.y)

def predator3_callback(self, msg):
    self.predators['predator3'] = (msg.x, msg.y)
```



```python
self.predators = {}
```

Prey similarly to previous tasks calculates vector from its position to a position of a predator, but in this example for each predator individually. Then sums them up to get vector away from the position of 3 predators simultaneously.



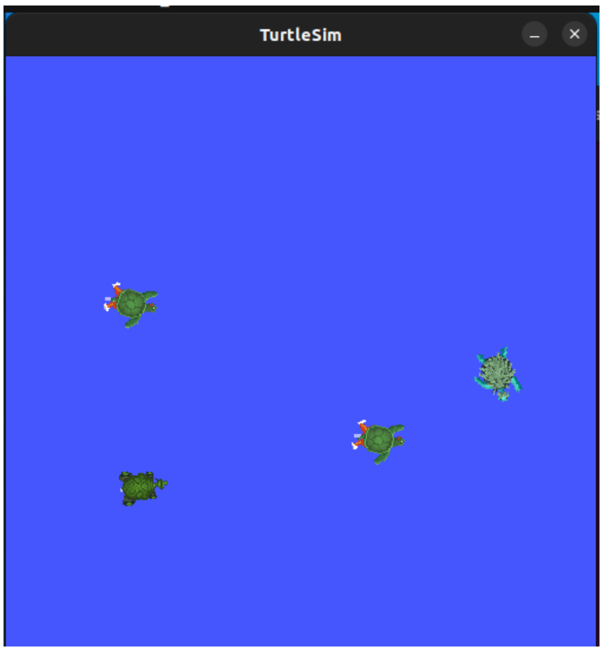
```python
# Predator 1 vector
p1 = self.predators.get('predator1')
v1x = self.current_pose.x - p1[0] if p1 else 0.0
v1y = self.current_pose.y - p1[1] if p1 else 0.0

# Predator 2 vector
p2 = self.predators.get('predator2')
v2x = self.current_pose.x - p2[0] if p2 else 0.0
v2y = self.current_pose.y - p2[1] if p2 else 0.0

# Predator 3 vector
p3 = self.predators.get('predator3')
v3x = self.current_pose.x - p3[0] if p3 else 0.0
v3y = self.current_pose.y - p3[1] if p3 else 0.0

#sum of all vectors
dx = v1x + v2x + v3x
dy = v1y + v2y + v3y
```

Components of this vector are then passed through wall avoiding logic like in previous examples before being published as a movement.

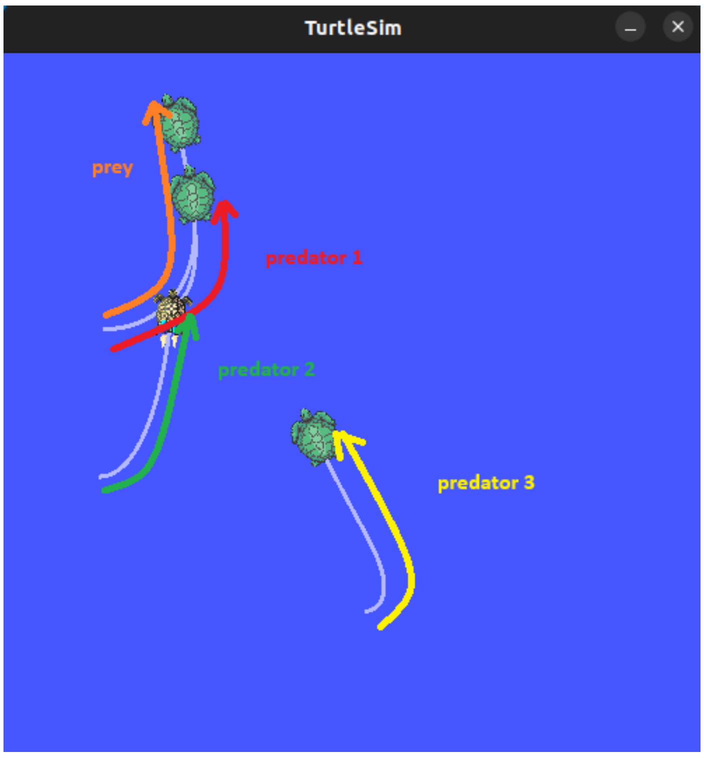
Full code for all files available in 551530_code_202421412.zip/task4

Functionality

At the beginning of the simulation, we can see 4 turtles spawned at different locations.



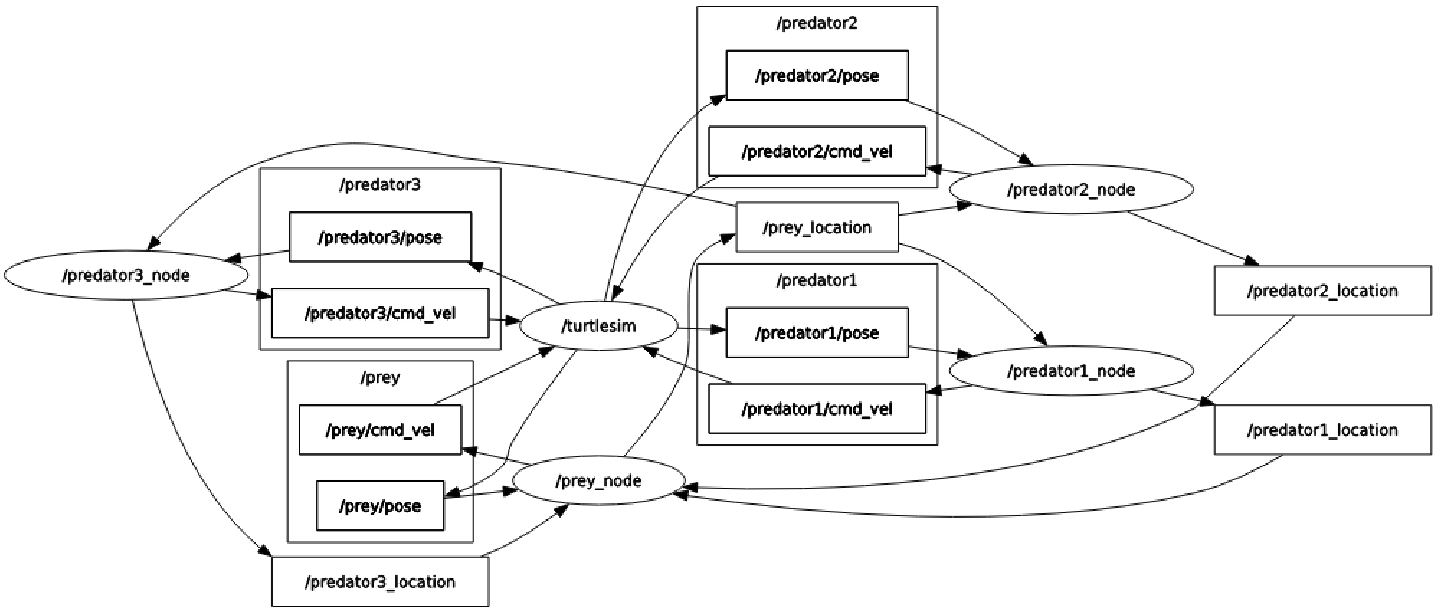
Then we can see that prey moves in direction away from all 3 predators.



The reast of the simulation plays out very similar to task nr 2, as predator's behaviour is exactly the same, they follow prey's coordinates. Prey is trying to escape, but it's very challenging due to constrains of available space and in most cases it's quickly caught.

Being based on code from task 2 it shares its problems especially prey getting stuck in corners but also introduces new ones. Sum of vectors method of escape isn't perfect it means prey will always move away from predators are. In this situation especially with confined space different tactic would be more beneficial. For example, trying to escape in between predators.

Using rqt_graph we can see relations between nodes of this program. In this example it looks much more complex than previous ones, but this only due to higher number of active nodes:



References:

ROS2 Documentation: Humble Launching nodes, 2025, URL: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Launching-Multiple-Nodes/Launching-Multiple-Nodes.html

ROS2 Documentation: Humble Understanding actions, 2025, URL: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html

ROS2 Documentation: Using parameters in a class (Python), 2025, URL: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Using-Parameters-In-A-Class-Python.html