UOW MALAYSIA KDU PENANG UNIVERSITY COLLEGE

PART OF THE UNIVERSITY OF WOLLONGONG AUSTRALIA GLOBAL NETWORK

ASSIGNMENT COVER PAGE

Validated by UNIVERSITY OF LINCOLN UNITED KINGDOM

Programme	**Course Code and Title**
Bachelor of Computer Science (Hons)	CAI3034N Autonomous Mobile Robotics
Student's name / student's id	**Lecturer's name**
CHAN SEOW FEN / 0207368	Dr. Ooi Woi Seng

Date issued	**Submission Deadline**	**Indicative Weighting**
Week 4 (24/2/2025)	Week 8 (24/3/2025)	30%
Assignment 1 title	Robot Chase and Prediction	

This assessment assesses the following course learning outcomes

# as in Course Guide	**UOWM KDU Penang University College Learning Outcome**
CLO3	Implement intelligent control strategies, by programming autonomous mobile robots to perform complex tasks in dynamic environments including obstacle avoidance, planning and navigation, robotic mapping and self-localisation.(C3, PLO3)
# as in Course Guide	**University of Lincoln Learning Outcome**
CLO3	Implement intelligent control strategies, by programming autonomous mobile robots to perform complex tasks in dynamic environments including obstacle avoidance, planning and navigation, robotic mapping and self-localisation.(C3, PLO3)

Student's declaration

I certify that the work submitted for this assignment is my own and research sources are fully acknowledged.

Student's signature:

Submission Date:

24/3/25

Doc2.docx

ORIGINALITY REPORT

0%	0%	0%	0%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		

Table of Contents

Table of Figures

1.0 Introduction

1.1 Background and Motivation

In the field of modern technological development, autonomous mobile robotics has become increasingly significant, with the various applications from search and rescue operations to warehouse automation and security surveillance. The pursuit-evasion problems particularly representing an important area of study within this field. For elaboration, it combines elements of artificial intelligence (AI), control theory, and game theory (Zhou et al., 2021). The movement prediction of dynamic targets and efficient pursuit of the mobile robots is crucial in many real-world applications. To illustrate that, the robots are able to anticipate the future positions of moving targets through the incorporation of prediction-based pursuit algorithms, thus enabling more efficient interception as compared to simple reactive following behaviours. Hence, the foundation of numerous robotic applications, such as wildlife tracking systems, sport-playing robots, and automated security patrols is formed by these capabilities (Qu et al., 2025). Moreover, the mathematics underlying such prediction methods often draws from estimation theory, with the employment of techniques such as Kalman filtering or simpler linear extrapolation models to make the target trajectories forecasting. While it is rather simplified compared to physical robots, simulating with TurtleSim environment in ROS (Robot Operating System), which is an excellent platform for the pursuit algorithms demonstration and testing in a controlled, visual manner is still valuable. To justify that, the core challenges that characterise real-world robotic applications such as sensing, prediction, movement control, and coordination is still retained by the TurtleSim. In essence, by utilising the TurtleSim, the complexities of hardware integration are indeed removed, however the fundamental algorithmic challenges are still maintained.

1.2 Project Scope and Objectives

It is aimed to achieve the implementation of a multi-agent chase simulation within the ROS TurtleSim environments in this project. For elaboration, it involves a predator turtle applying prediction-based movement strategies and actively chasing and capturing multiple prey turtles. The fundamental concept in autonomous mobile robotics is demonstrated by the simulation, which including target prediction, pursuit strategies, and the coordination of multiple agents. The primary objectives of this implementation consist of:

1. Developing a predator turtle capable of intelligently pursuing and capturing multiple prey turtles
2. Implementing position prediction algorithms to anticipate prey movement
3. Creating natural movement behaviours for prey turtles
4. Developing a sticking behaviour where captured prey follow the predator in formation
5. Implementing collision avoidance systems for both predator and prey

The documentation will be structured in the way of beginning with an introduction to the project background and objectives, followed by describing the system architecture, and explaining the development and implementation of the algorithm, subsequently showcasing the result and discussing the findings. Furthermore, the documentation also provided with the analysis and evaluation to identify limitation and potential improvements, followed by challenges encountered and solutions, then finally with a conclusion to sum up the entire project.

2.0 System Architecture

2.1 ROS Package Structure

The TurtleSim chase simulation implementation is through a ROS package named turtle_chase. The package structure with the directory organisation as shown in *Figure 1* aligned with standard ROS conventions.

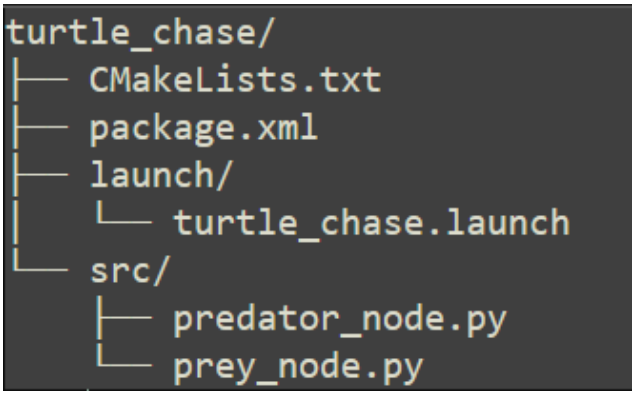


Figure 1 ROS Package Structure

The package depends on core ROS packages including rospy for Python-based ROS programming, geometry_msgs for standard message types related to geometric primitives, and turtlesim for the simulation environment itself.

2.2 Node Configuration

A distributed, node-based approach if followed by the system architecture, which consistent with ROS design principles. There are 2 primary custom nodes consisted by the simulation, which is the predator node (predator_node.py), and prey node (prey_node.py). The predator node will control the predator turtle (turtle1) while implementing prediction, chase and capture behaviours. The pose information from all turtles will be subscribed by this node, and it will also publish velocity commands to control the predator and any captured prey. On the other hand, the prey node responsible for managing multiple prey turtles, while implementing spawn functionality, random movement patterns, avoiding predator and collisions with other prey turtles. Separate data structures for each prey turtle are created by this node while sharing common behavioural logic. A publisher-subscriber pattern is adopted for the communication between nodes. For illustration, each turtle will publish its pose information and subscribe to velocity commands. This decoupling of information producers and consumers is a central feature of ROS, thus allowing for modular system design and ease of expansion. A state information about which prey have been caught is maintained by the predator node, utilising ROS parameters to communicate this information globally. Therefore, system-wide awareness of the simulation state is enabled through this approach without requiring predator and prey nodes direct communicating.

2.3 Launch File Setup

A ROS launch file (turtle_chase.launch) is utilised to configure and start all required nodes simultaneously. Following functions are performed by the launch file:

1. Initialises the TurtleSim environment by launching the turtlesim_node
2. Configures initial positions for each prey turtle using ROS parameters
3. Launches the prey node to spawn and control multiple prey turtles
4. Launches the predator node to control the predator turtle

The initial positions of prey turtles are spread across the simulation through the launch file in order to create an engaging chase scenario. The positioning strategy have been illustrated through the parameter configuration section:

```
node pkg="turtle_chase" type="prey_node.py" name="prey_node" output="screen">
        <!-- Initial position for turtle2: top left corner -->
        <param name="turtle2_x" value="2.0"/>
        <param name="turtle2_y" value="8.0"/>

        <!-- Initial position for turtle3: bottom right corner -->
        <param name="turtle3_x" value="8.0"/>
        <param name="turtle3_y" value="2.0"/>

        <!-- Initial position for turtle4: top right corner -->
        <param name="turtle4_x" value="9.0"/>
        <param name="turtle4_y" value="9.0"/>

        <!-- Initial position for turtle5: center of the arena -->
        <param name="turtle5_x" value="5.0"/>
        <param name="turtle5_y" value="5.0"/>
    </node>
```

Figure 2 Partial Code - Launch File Parameter Configuration

It is ensured that prey turtles are distributed across the entire simulation area through this configuration, thus creating a challenging scenario for the predator turtle. An easy adjustment of initial positions without modifying the node source code have been achieved through the use of named parameters, which follows the best practices for ROS configuration management.

3.0 Algorithm Development and Implementation

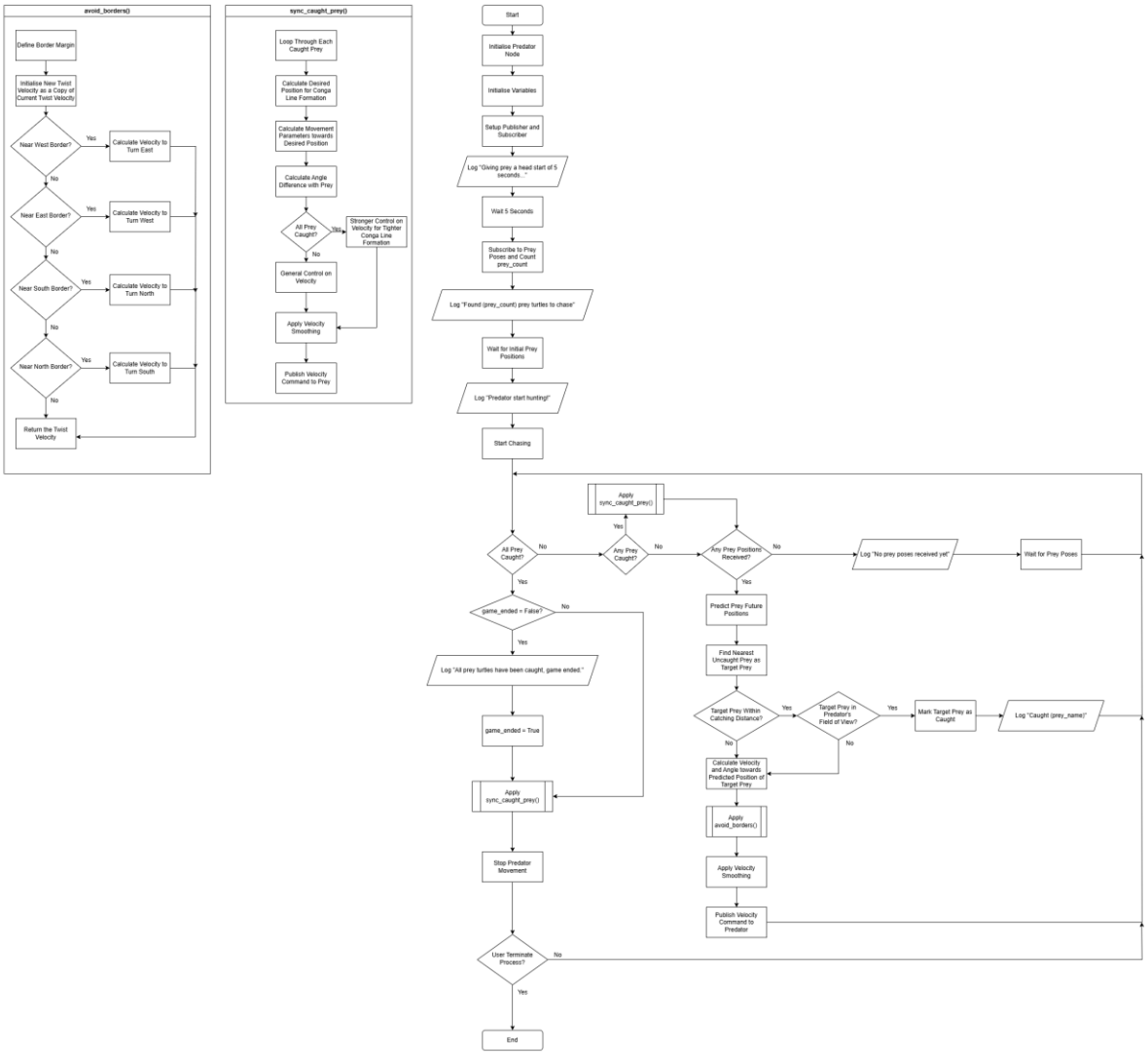


Figure 3 Simplified Flowchart for Predator Algorithm

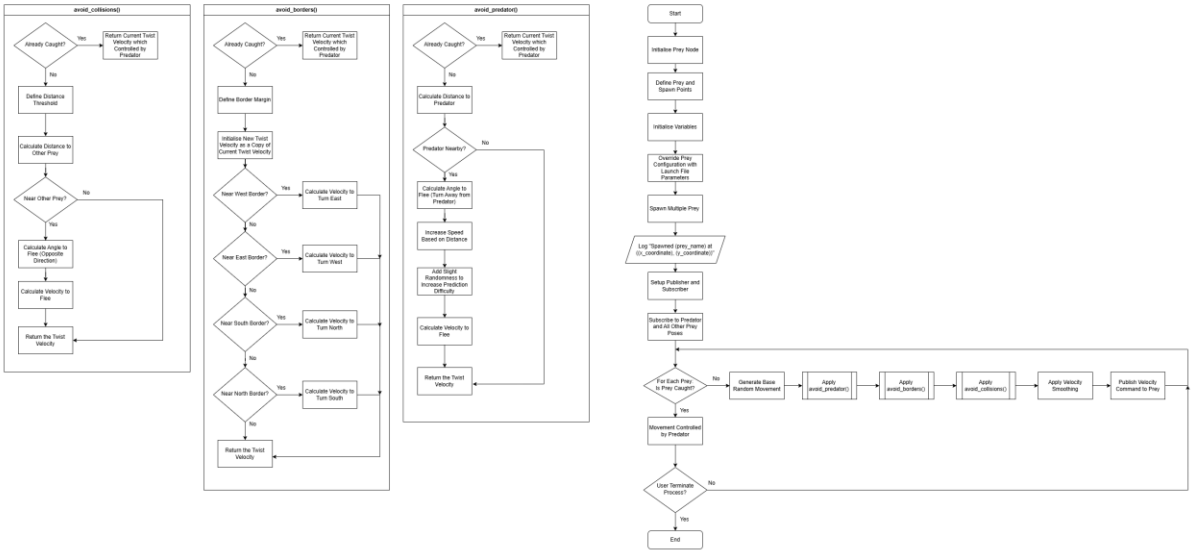


Figure 4 Simplified Flowchart for Prey Algorithm

The simplified flowchart for both Predator and Prey implementation is illustrated in *Figure 3* and *Figure 4* to showcase the basic logic of the algorithms.

3.1 Predator Turtle Implementation

3.1.1 Position Prediction Algorithm

The predator's efficient hunting capability is founded by the prey movement prediction ability. To illustrate that, rather than simply chasing the current position of prey, which would likely end up as constantly trailing behind, instead, the predator anticipates where prey will locate at in the near future. This position prediction is implemented in the predict_position() method of the PredatorTurtle class:

```python
def predict_position(self, prey_pose):
    """Predict a prey's future position based on its current velocity.

    Uses a simple linear prediction based on current velocity vector.

    :param prey_pose: Current pose of the prey
    :return: Tuple (x, y) of predicted position
    """
    # Time horizon for prediction (in seconds)
    time_step = 0.5

    # Calculate predicted position using current velocity and direction
    pred_x = prey_pose.x + prey_pose.linear_velocity * math.cos(prey_pose.theta) * time_step
    pred_y = prey_pose.y + prey_pose.linear_velocity * math.sin(prey_pose.theta) * time_step

    return pred_x, pred_y
```

Figure 5 Partial Code – Predator's Position Prediction

A linear extrapolation model based on the prey's current velocity vector is employed by this prediction algorithm. The prediction uses:

1. Current position (x, y) of the prey
2. Current linear velocity magnitude
3. Current orientation (theta)
4. A time horizon parameter (0.5 seconds)

The expected future position of the prey is calculated by the algorithm through the projection of the current velocity vector forward in time, utilising basic kinematic equations. The reason of applying such linear prediction models is due to the fact that it works well for short time horizons when acceleration is relatively constant, thus it is a reasonable assumption for the TurtleSim environment. In corresponding to that, 0.5 seconds was chosen as the time horizon parameter as it is obtained through experimentation in balancing prediction accuracy with system response time. To illustrate that, shorter horizons can make the predator more reactive but less efficient in pursuing. On the contrary, longer horizons might result in inaccurate predictions due to prey direction changes.

3.1.2 Chase Strategy

The implementation of predator’s chase strategy is under the chase() method, which act as the main control loop. Briefly, the strategy follows these key steps:

1. Identify and track all prey turtles in the environment
2. Determine which prey are uncaught
3. Predict the future position of each uncaught prey
4. Identify the nearest uncaught prey as the current target
5. Calculate velocity commands to move toward the predicted position
6. Adjust velocity for border avoidance and smoothness
7. Check if any prey is within capture range

The nearest uncaught prey is identified by the target selection logic based on the distance to its predicted position:

```python
# Find nearest uncaught prey (using predicted position)
target = None
min_dist = float('inf')
for prey_name, pose in self.prey_poses.items():
    if prey_name not in self.caught_prey:
        # Predict where this prey will be
        pred_x, pred_y = self.predict_position(pose)
        # Calculate distance to predicted position
        dist = self.distance(self.pose.x, self.pose.y, pred_x, pred_y)
        # Keep track of nearest prey
        if dist < min_dist:
            min_dist = dist
            target = (prey_name, pred_x, pred_y)
```

Figure 6 Partial Code - Predator's Target Selection

Efficiency is prioritised by this approach through focusing on the prey that can be caught most quickly at the moment. A proportional control is used on the velocity control toward the predicted position. To illustrate that, the linear velocity is proportional to the distance, which is capped at a maximum value, and the angular velocity is proportional to the angle difference:

```python
# Calculate velocity towards predicted prey position
vel = Twist()

# Calculate angle towards predicted position
angle = math.atan2(pred_y - self.pose.y, pred_x - self.pose.x)

# Calculate angle difference (normalised to -pi to pi range)
angle_diff = (angle - self.pose.theta) % (2 * math.pi)
if angle_diff > math.pi:
    angle_diff -= 2 * math.pi

# Set linear velocity proportional to distance (capped at 1.2)
vel.linear.x = min(1.2, 1.0 * min_dist)

# Set angular velocity proportional to angle difference
vel.angular.z = 3.0 * angle_diff
```

Figure 7 Partial Code - Predator's Velocity Control towards Predicted Position

This proportional control approach is a simplified version of the more general Proportional-Integral-Derivative (PID) control often used in robotics (Li, 2023). The proportional component alone offered sufficient control accuracy while ensuring simplicity of the implementation in this context.

3.1.3 Prey Capture Logic

For the prey capture logic, it is founded on the proximity detection with the combination of field of view (FOV) constraints. A prey is considered caught when:

1. It is within a specific distance threshold (0.6 units)
2. It is within the predator's forward FOV (±90 degrees)

This logic is implemented as follows:

```python
# Check if prey is within catching distance
current_dist = self.distance(self.pose.x, self.pose.y, current_prey_pose.x, current_prey_pose.y)
if current_dist < 0.6:  # Catching distance threshold
    # Calculate angle to prey
    angle_to_prey = math.atan2(current_prey_pose.y - self.pose.y, current_prey_pose.x - self.pose.x)

    # Check if prey is within predator's field of view
    angle_diff = abs((angle_to_prey - self.pose.theta) % (2 * math.pi))
    if angle_diff > math.pi:
        angle_diff = 2 * math.pi - angle_diff

    # Can only catch prey if they're in front of the predator (within 90 degree field of view)
    if angle_diff < math.pi / 2:
        # Mark prey as caught
        self.caught_prey.append(prey_name)
        rospy.set_param('/caught/{}'.format(prey_name), True)
        rospy.loginfo("Caught {}".format(prey_name))
```

Figure 8 Partial Code - Predator's Prey Capturing

A little realism is added to the simulation through this approach of mandating the predator to face its prey during capture, which is similar to many predators in nature that orient themselves toward their target to execute a successful capture. Therefore, a more realistic pursuit behaviour is implemented through the FOV constraint, which rather consider capture by simply approaching the prey, it requires the predator to position itself appropriately. Following that, when a prey is caught, its status is updated in two ways:

1. Added to the predator's internal caught_prey list
2. Updated in the ROS parameter server, making this information available to all nodes

3.1.4 Sticking Behaviour Implementation

Figure 9 Example of Conga Line (Cohen, 2011)

One of the most visually interesting aspects of the simulation is the sticking behaviour, where caught prey follows the predator in a conga line formation as shown in the figure above. This behaviour is implemented in the sync_caught_prey() method:

```python
def sync_caught_prey(self, all_caught=False):
    """Synchronise caught prey to follow the predator in a line.

    Makes caught prey turtles follow behind the predator in a 'conga line'.

    :param all_caught: True if all prey are caught, enabling more precise control
    """
    # Loop through each caught prey in order
    for i, prey_name in enumerate(self.caught_prey):
        # Skip if we don't have a publisher for this prey
        if prey_name not in self.caught_pubs:
            continue

        # Initialise velocity command for this prey
        follow_vel = Twist()

        # Calculate desired position: each prey should be positioned
        # behind the predator, spaced by index in the caught list
        distance_behind = 0.35 * (i + 1)  # Each prey spaced 0.35 units apart

        # Calculate target position behind predator based on predator's orientation
        desired_x = self.pose.x - distance_behind * math.cos(self.pose.theta)
        desired_y = self.pose.y - distance_behind * math.sin(self.pose.theta)
```

Figure 10 Partial Code - Predator's Caught Prey Synchronisation for Sticking Behaviour

The desired position for each caught prey is calculated by the method based on:

1. The predator's current position
2. The predator's orientation
3. The prey's position in the capture sequence (first caught follows directly behind, second caught follows the first, etc.)

The spacing of 0.35 units between caught prey was determined experimentally to present a visually appealing formation while avoiding the caught prey bunch together. This approach to formation control is similar to the leader-follower paradigm described by Hirata-Acosta et al., (2021), which having follower agents maintaining specific relative positions to the leader agent.

A proportional control is incorporated in the method as the guidance for each prey move toward its desired position:

```
# Calculate movement parameters towards desired position
dist = self.distance(prey_pose.x, prey_pose.y, desired_x, desired_y)
angle = math.atan2(desired_y - prey_pose.y, desired_x - prey_pose.x)

# Calculate angle difference (normalised to -pi to pi range)
angle_diff = (angle - prey_pose.theta) % (2 * math.pi)
if angle_diff > math.pi:
    angle_diff -= 2 * math.pi

# Adjust velocity based on whether all prey are caught
# When all prey are caught, more precise control is used
if all_caught:
    # Stronger control when all prey are caught for tighter formation
    follow_vel.linear.x = 4.0 * dist
    follow_vel.angular.z = 5.0 * angle_diff

    # Stop when very close to desired position
    if dist < 0.1:
        follow_vel.linear.x = 0.0
        follow_vel.angular.z = 0.0
else:
    # More relaxed control when still hunting
    follow_vel.linear.x = 2.5 * dist
    follow_vel.angular.z = 3.5 * angle_diff
```

Figure 11 Partial Code - Predator's Caught Prey Position Control

When the game completed, which is all prey have been caught, it is implemented a tighter control parameters in order to create a more precise formation. Hence, this adaptive mechanism allows for providing visual cue to differentiate predator and prey in a conga line while it is actively hunting, and all of the turtles stick together precisely after the game ended.

3.1.5 Border Avoidance

To prevent the predator from colliding with the simulation boundaries, the implementation includes a border avoidance behaviour in the avoid_borders() method:

```python
def avoid_borders(self, vel):
    """Adjust velocity to prevent hitting simulation borders.

    Checks proximity to borders and adjusts direction to avoid collisions.

    :param vel: Current Twist velocity
    :return: Adjusted Twist velocity
    """
    # Distance from border to start avoidance behaviour
    border_margin = 1.0 # Smaller margin than prey as the predator's primary goal is pursuit


    # Create a new velocity message to avoid modifying the input
    adjusted_vel = Twist()
    adjusted_vel.linear.x = vel.linear.x
    adjusted_vel.angular.z = vel.angular.z

    # Check proximity to left border (west)
    if self.pose.x < 1 + border_margin:
        target_angle = 0  # Turn east (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 2.0 * angle_diff # Higher angular velocity scaling than prey for quicker turns during pursuit.
        adjusted_vel.linear.x = 0.5  # Reduce speed during avoidance, greater speed reduction than prey to ensure sharper turns and prevent overshooting

    # Check proximity to right border (east)
    elif self.pose.x > 10 - border_margin:
        target_angle = math.pi  # Turn west (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 2.0 * angle_diff
        adjusted_vel.linear.x = 0.5

    # Check proximity to bottom border (south)
    elif self.pose.y < 1 + border_margin:
        target_angle = math.pi / 2  # Turn north (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 2.0 * angle_diff
        adjusted_vel.linear.x = 0.5

    # Check proximity to top border (north)
    elif self.pose.y > 10 - border_margin:
        target_angle = 3 * math.pi / 2  # Turn south (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 2.0 * angle_diff
        adjusted_vel.linear.x = 0.5

    return adjusted_vel
```

Figure 12 Partial Code - Predator's Border Avoidance

The algorithm works in the way of actively checking the predator's proximity to each of the four simulation boundaries (West, East, South, North), and adjust its velocity when it is near to the defined margin of 1.0 units. When the border avoidance mechanism is triggered, the predator:

1. Sets a target angle pointing away from the boundary
2. Calculates the angle difference between current orientation and target angle
3. Sets angular velocity proportional to this angle difference
4. Reduces linear velocity to ensure smooth turning

A natural-looking avoidance behaviour is created through this approach, while ensuring the predator remains within the simulation area.

3.2 Prey Turtle Implementation

3.2.1 Random Movement Generation

A base random movement pattern is exhibited by the prey turtles to create relatively unpredictable targets for the predator. This behaviour is implemented in the move_once() method of the PreyTurtle class:

```
# Base movement: constant speed with slight random turning
vel = Twist()
vel.linear.x = 0.7
vel.angular.z = random.uniform(-0.2, 0.2)
```

Figure 13 Partial Code - Prey's Base Movement

The random movement consists of:

1. A constant forward velocity (0.7 units per second)
2. A small random angular velocity (between -0.2 and 0.2 radians per second)

A motion that is relatively unpredictable but smooth is created by this approach, thus making they prey slightly more challenging to catch without using the approach that might introduce erratic movement, which will seem unnatural.

3.2.2 Predator Avoidance

In order to create more challenging and realistic simulations, the prey behaviour have introduced a predator avoidance mechanism in the avoid_predator() method:

```python
def avoid_predator(self, vel):
    """Adjust velocity to flee from the predator if nearby.

    Increases speed and steers away when predator gets close.

    :param vel: Current Twist velocity
    :return: Adjusted Twist velocity
    """
    # Don't avoid predator if already caught (controlled by predator)
    if self.caught:
        return vel

    # Check if we have the predator's position
    if 'turtle1' in self.other_poses:
        predator = self.other_poses['turtle1']

        # Calculate distance to predator
        dist = self.distance(self.pose.x, self.pose.y, predator.x, predator.y)

        # If predator is nearby (within 3.0 units), flee
        if dist < 3.0:
            # Calculate angle to predator
            angle_to_pred = math.atan2(predator.y - self.pose.y, predator.x - self.pose.x)

            # Calculate angle to flee (opposite direction)
            flee_angle = (angle_to_pred + math.pi) % (2 * math.pi)

            # Calculate angle difference (normalised to -pi to pi range)
            angle_diff = (flee_angle - self.pose.theta) % (2 * math.pi)
            if angle_diff > math.pi:
                angle_diff -= 2 * math.pi

            # Set velocity to flee (turn away from predator)
            vel.angular.z = 2.0 * angle_diff

            # Speed up when closer to predator (max 1.5)
            vel.linear.x = min(1.5, 1.0 + (3.0 - dist) * 0.3)

            # Add slight randomness to make prey harder to predict
            vel.angular.z += random.uniform(-0.1, 0.1)

    return vel
```

Figure 14 Partial Code - Prey's Predator Avoidance

The predator avoidance behaviour is triggered when the predator approach within 3.0 units of a prey turtle. When it is activated, the prey:

1. Calculates the direction directly away from the predator
2. Turns to face this direction
3. Increases its speed based on how close the predator is
4. Adds slight randomness to its turning to make its movement less predictable

Hence, a more dynamic chase scenario is created by this behaviour implementation due to the fact that prey actively try to escape when the predator approaches. Moreover, a small amount of randomness in the angular velocity is applied in the implementation, thus making the prey's movement less predictable for the predator's linear extrapolation algorithm. This approach is similar to the concept of protean behaviour, which is observed in the natural predator-prey systems. To elaborate that, unpredictable movements is used by the prey to confuse predators (Richardson et al., 2018).

3.2.3 Border Avoidance Strategy

Similar to the predator, a border avoidance is implemented in prey turtles to avoid colliding the simulation boundaries. The implementation in the avoid_borders() method follows the same general approach as the predator's border avoidance but with adjusted parameters:

```python
def avoid_borders(self, vel):
    """Adjust velocity to stay within simulation borders.

    Steers away from borders when getting too close.

    :param vel: Current Twist velocity
    :return: Adjusted Twist velocity
    """
    # Don't avoid borders if already caught (controlled by predator)
    if self.caught:
        return vel

    # Distance threshold from border to trigger avoidance
    border_margin = 1.5 #Larger margin than prey gives more reaction time for evasion.

    # Create a new velocity message to avoid modifying the input
    adjusted_vel = Twist()
    adjusted_vel.linear.x = vel.linear.x
    adjusted_vel.angular.z = vel.angular.z

    # Check proximity to left border (west)
    if self.pose.x < border_margin:
        target_angle = 0  # Turn east (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 1.0 * angle_diff # Prey turns less aggressively than predator, smoother turns beneficial for evasion.
        adjusted_vel.linear.x = 0.8 # Reduce speed during avoidance, smaller speed reduction than predator to maintain some momentum for escape.

    # Check proximity to right border (east)
    elif self.pose.x > 11.0 - border_margin:
        target_angle = math.pi  # Turn west (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 1.0 * angle_diff
        adjusted_vel.linear.x = 0.8

    # Check proximity to bottom border (south)
    elif self.pose.y < border_margin:
        target_angle = math.pi / 2  # Turn north (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 1.0 * angle_diff
        adjusted_vel.linear.x = 0.8

    # Check proximity to top border (north)
    elif self.pose.y > 11.0 - border_margin:
        target_angle = 3 * math.pi / 2  # Turn south (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 1.0 * angle_diff
        adjusted_vel.linear.x = 0.8

    return adjusted_vel
```

Figure 15 Prey's Border Avoidance

For justification, a larger threshold of 1.5 units is used by the prey as compared to the predator with 1.0 unit is to make the prey to start avoiding borders earlier. By this design, more cautious behaviour for prey is created, which is appropriate and reasonable for their role as evaders rather than pursuers. Another key difference in the implementation detail is that when the prey is caught, the border avoidance mechanism is disabled due to the fact that they should be controlled directly by the predator node. Therefore, the conflicts between the prey's autonomous behaviours and the predator's formation control for caught prey is prevented through this implementation.

3.2.4 Collision Avoidance with Other Prey

In order to prevent collisions between prey turtles occurring, a collision avoidance mechanism is implemented in the avoid_collisions() method:

```python
def avoid_collisions(self, vel):
    """Adjust velocity to avoid collisions with other prey turtles.

    Only avoids uncaught prey - no need to avoid caught prey or predator.

    :param vel: Current Twist velocity
    :return: Adjusted Twist velocity
    """
    # Don't avoid collisions if already caught (controlled by predator)
    if self.caught:
        return vel

    # Distance threshold for collision avoidance
    avoidance_distance = 1.0

    # Check distance to each other turtle
    for other_name, other_pose in self.other_poses.items():
        # Skip predator (turtle1) and caught prey
        if other_name == 'turtle1' or rospy.get_param('/caught/{}'.format(other_name), False):
            continue

        # Calculate distance to other turtle
        dist = self.distance(self.pose.x, self.pose.y, other_pose.x, other_pose.y)

        # If too close, steer away
        if dist < avoidance_distance:
            # Calculate angle to other turtle
            angle_to_other = math.atan2(other_pose.y - self.pose.y, other_pose.x - self.pose.x)

            # Calculate angle to flee (opposite direction)
            flee_angle = (angle_to_other + math.pi) % (2 * math.pi)

            # Calculate angle difference (normalised to -pi to pi range)
            angle_diff = (flee_angle - self.pose.theta) % (2 * math.pi)
            if angle_diff > math.pi:
                angle_diff -= 2 * math.pi

            # Set velocity to steer away
            vel.angular.z = 1.5 * angle_diff
            vel.linear.x = 0.8

    return vel
```

Figure 16 Partial Code - Collisions Avoidance

It works on the basis of constantly checking its distance to each other uncaught prey and immediate triggering the avoidance behaviour if they approach within 1.0 units of each other. Similar to the predator avoidance approach, the avoidance strategy calculates a direction away from the other prey and adjust velocity to steer in that direction.

Important to note that the implementation only avoid colliding other uncaught prey, while ignoring the predator as it is handled by the predator avoidance method and already caught prey that is controlled by the predator. In essence, it is ensured that the movement behaviour remains appropriate to the turtle's current state through this selective avoidance.

4.0 Results and Discussion

The experiment has conducted on both versions of the prey with predator avoidance and without predator avoidance. The video evidence is presented in **9.2 Appendix 2 – Video Evidence**. It has clearly demonstrated that, by adding predator avoidance mechanisms in the prey turtle's implementation, the simulation has increased in terms of interestingness and challenging for the predator to catch the prey. Hence, the discussion below will focus on the results of the implementation with predator avoidance.

4.1 System Initialisation and Setup

As shown in the figure below, after launching the simulation with the turtle_chase.launch file, the TurtleSim environment have been initialised by the system, following by the spawning of the predator turtle (turtle1) at the default centre position.

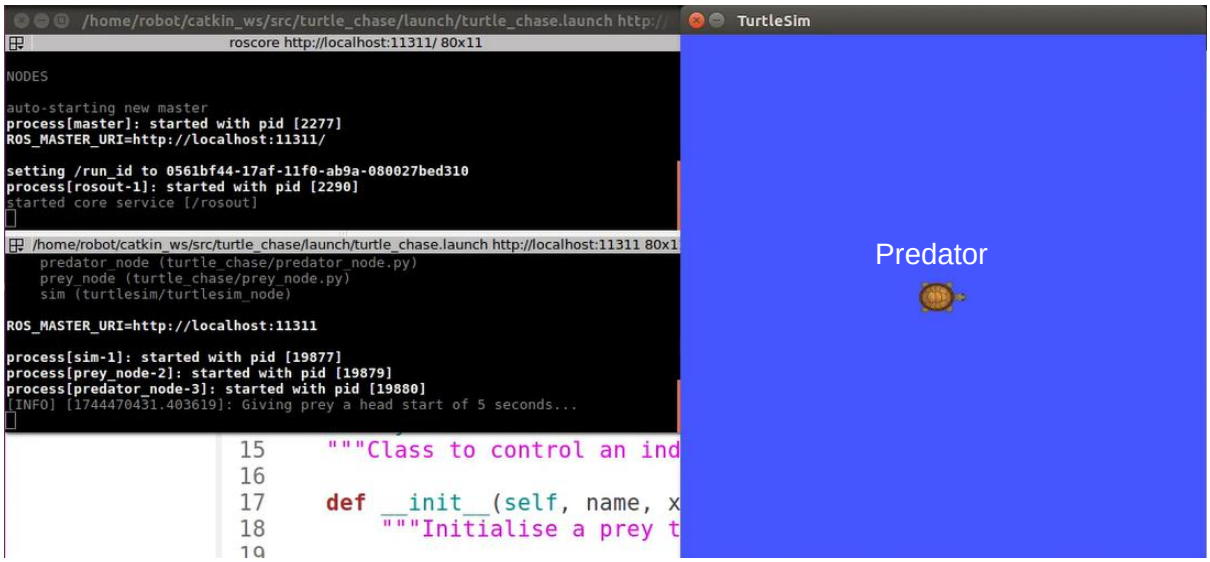


Figure 17 After Launching the Launch File

As presented in the figure below, immediately after that, 4 prey turtles (turtle2, turtle3, turtle4, turtle5) are spawned at their corresponding positions as specified in the launch file parameters.

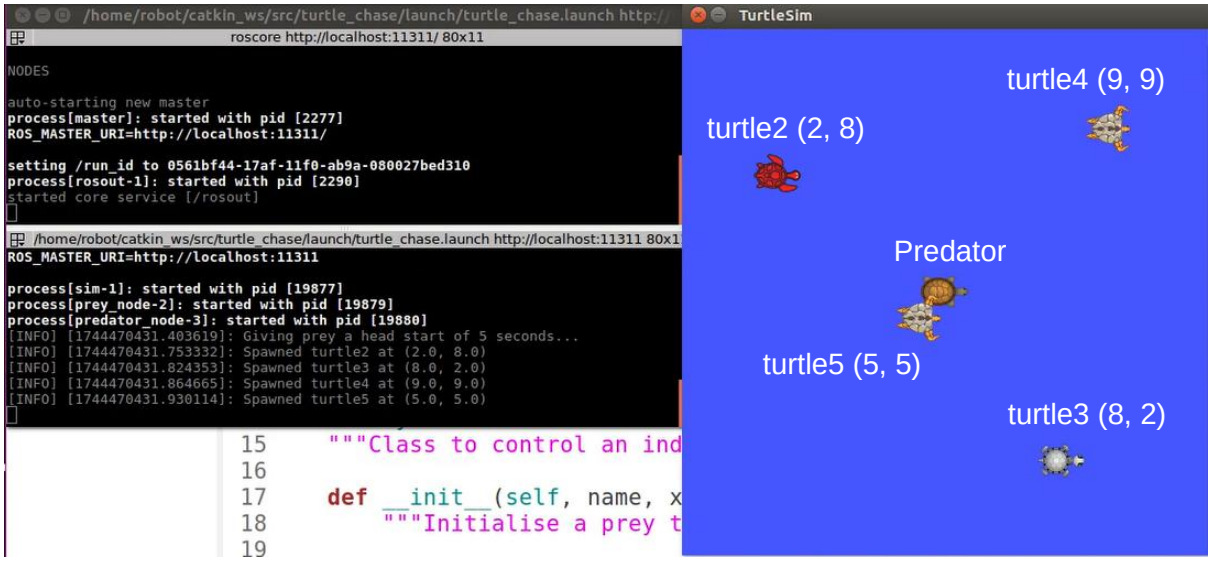


Figure 18 Multiple Prey Spawned

An easy adjustment of initial positions of the prey turtles without node source code modification is achieved by this parameter-based configuration approach. Hence, software engineering practices and ROS conventions for managing parameters are demonstrated by this separation of configuration from implementation.

4.2 Predator Performance

4.2.1 Prediction Accuracy

Good accuracy is demonstrated by the predator's position prediction algorithm for predicting short-term prey movement. To elaborate that, when prey is moving in relatively straight line, the linear extrapolation model works particularly well to capture its position ahead as demonstrated by the figures below. This is common when the prey is not actively avoiding the predator or borders.

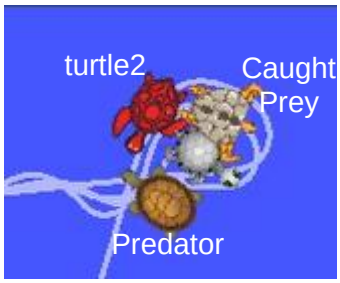


Figure 19 Prey Move in Relatively Straight Line

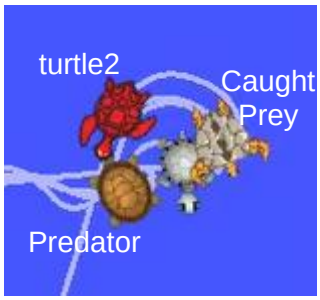


Figure 20 Predator Predict its Movement and Move Towards it

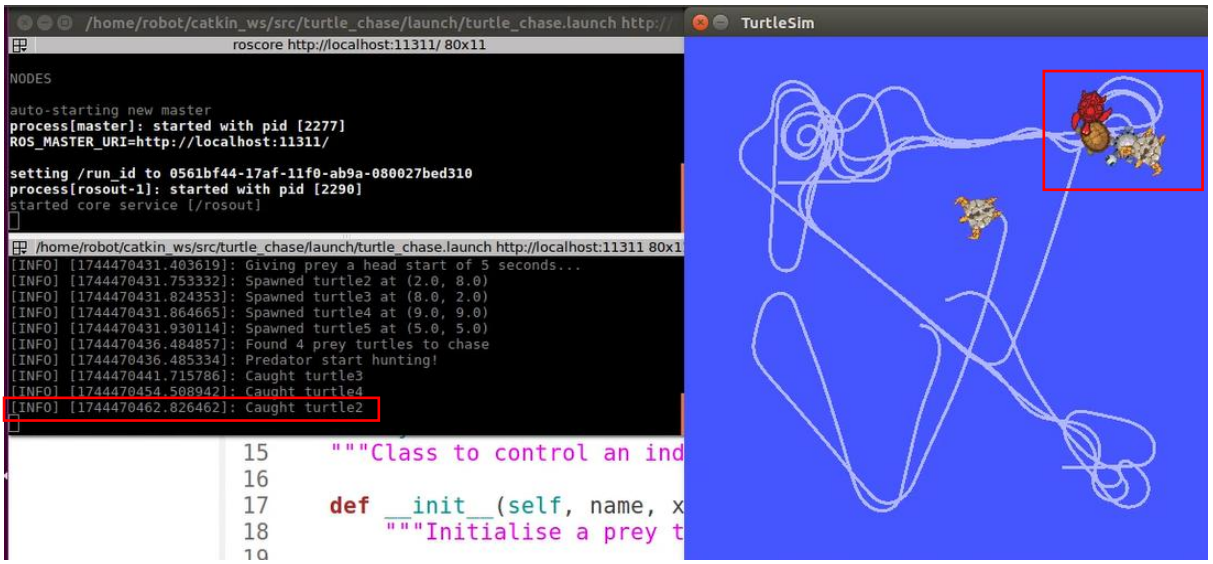


Figure 21 Prey Caught by the Predator

Moreover, a good balance between prediction range and accuracy is also provided by the time horizon of 0.5 seconds. This value was determined through experimentation, as it is observed that longer horizons resulted in reduction of predictions accuracy as the prey will change direction frequently, while shorter horizons led to the reduction of the prediction advantage over simple pursuit.

Furthermore, the effectiveness of the prediction algorithm is most apparent when chasing prey that is moving at high speeds, which when it is fleeing from the predator. Therefore, if aiming for current position in these cases, it would constantly be trailing behind the prey, thus prediction allows the predator to intercept the prey more efficiently as presented in the figures below.

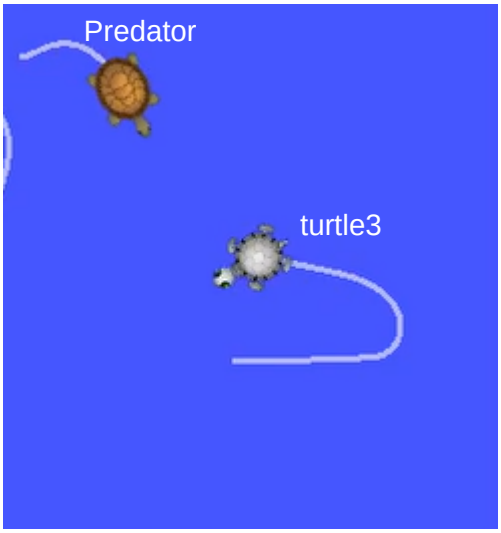


Figure 22 Prey Detected Predator

Figure 23 Prey Flee Away from Predator

Figure 24 Predator Anticipates Prey's Future Position

Figure 25 Predator Move Towards Prey's Future Position

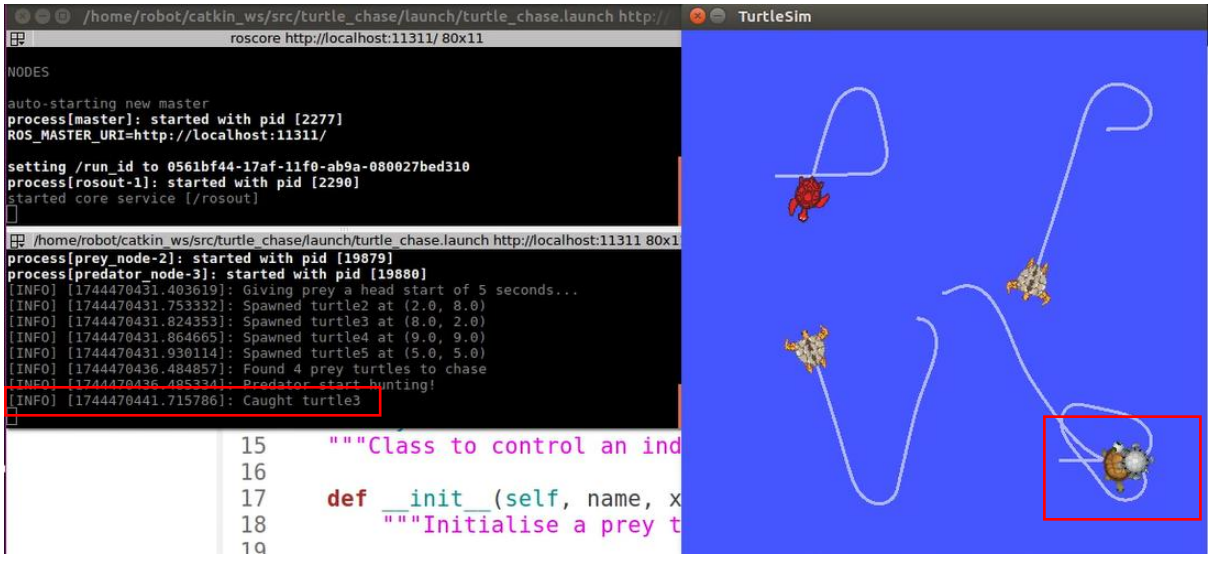


Figure 26 Predator Caught Prey

4.2.2 Chase Efficiency

Efficient pursuit of prey turtles is demonstrated through the predator's chase behaviour. For illustration, it is ensured that the predator does not waste time pursuing distant prey when closer options are available through the algorithm of selecting nearest prey as target based on the predicted positions. Hence, a natural-looking hunting behaviour that is appropriate for this scenario is created by this greedy approach, as demonstrated in the screenshots below.

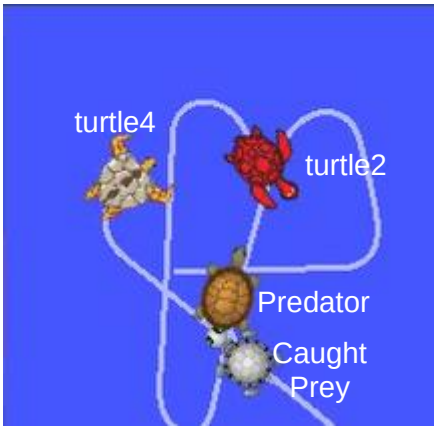


Figure 27 Predator Initially Targeting turtle2

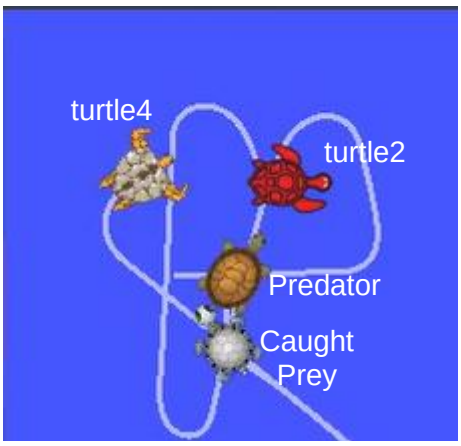


Figure 28 Predator Moving Towards turtle2

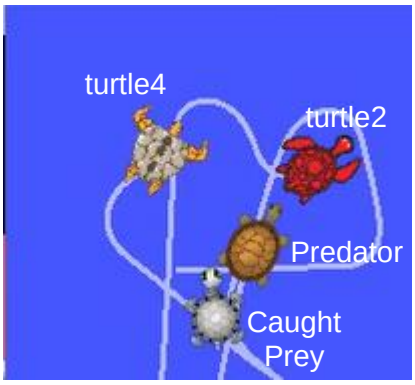


Figure 29 turtle2 Flee Away

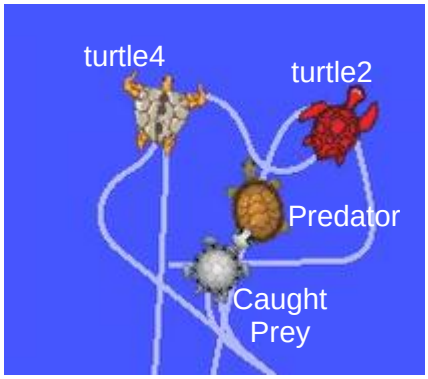


Figure 30 Predator Predicted turtle4's Future Position Having Shorter Distance than turtle2's Future Position

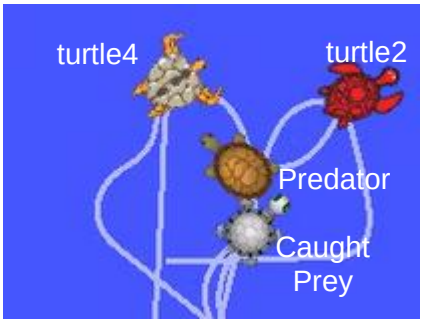


Figure 31 Predator Change Target to turtle4

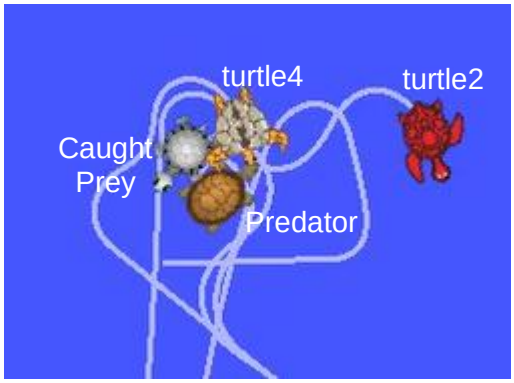


Figure 32 Predator Move Towards Anticipated turtle4 Future Position

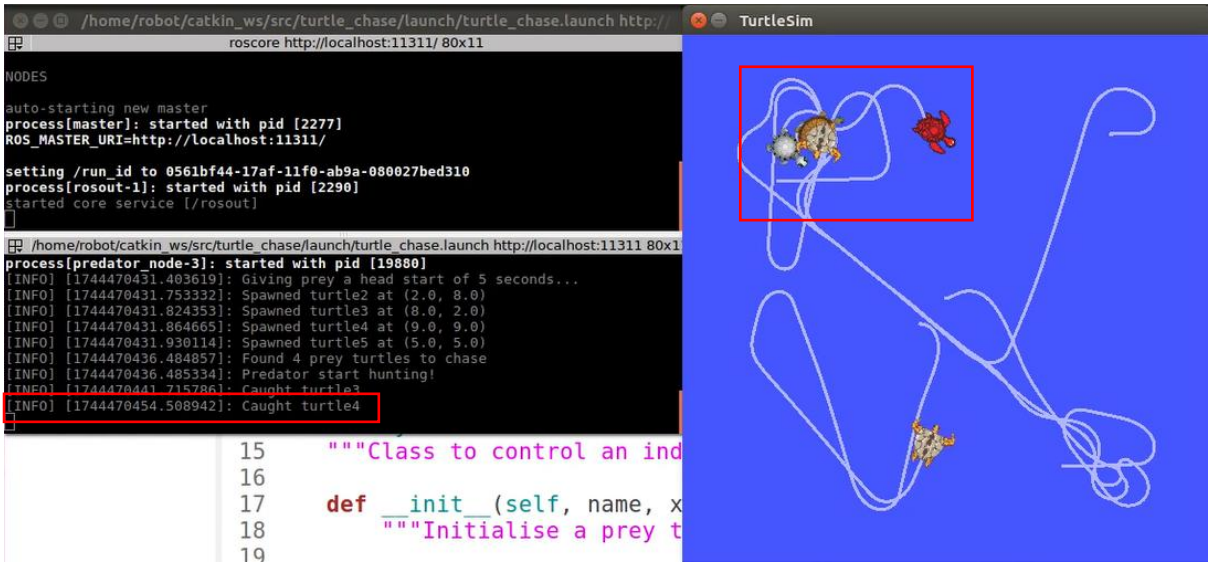


Figure 33 Predator Caught turtle4

Moreover, oscillations and jerkiness that might occur with binary control decisions is eliminated through the proportional control for velocity commands. To elaborate that, smooth movement toward the target is produced through applying linear velocity proportional to distance and angular velocity proportional to the angle difference. The predator's movement quality is even further enhanced by the smooth_velocity() method, which apply exponential smoothing to the velocity commands as shown in *Figure 34*.

```python
def smooth_velocity(self, target_vel):
    """Smooth velocity commands to reduce jerkiness.

    Uses exponential smoothing to blend new target velocity with previous velocity.

    :param target_vel: Desired Twist velocity
    :return: Smoothed Twist velocity
    """
    # Smoothing factor (0 to 1, higher = less smoothing)
    alpha = 0.3

    # Apply exponential smoothing to linear and angular velocities
    self.last_vel.linear.x = alpha * target_vel.linear.x + (1 - alpha) * self.last_vel.linear.x
    self.last_vel.angular.z = alpha * target_vel.angular.z + (1 - alpha) * self.last_vel.angular.z

    return self.last_vel
```

Figure 34 Predator's Velocity Smoothing

The smoothing factor, alpha which tuned to 0.3 created a good balance between responsiveness and smoothness. To illustrate that, the predator is able to react quickly enough to prey movements, while avoid abrupt changes in velocity, which would appear unnatural.

4.2.3 Sticking Behaviour Results

The sticking behaviour, which caught prey stick to each other and follow the predator in a "conga line" formation, creates a visually appealing demonstration of the predator's success. It is ensured by the implementation that each caught prey maintaining position that directly behind the turtle in front of it while having equal spacing throughout the formation. The

sequence is arranged according to the caught sequence, where first caught prey stay right after the predator and newly caught prey stay at the end of the line as shown in the screenshot below.

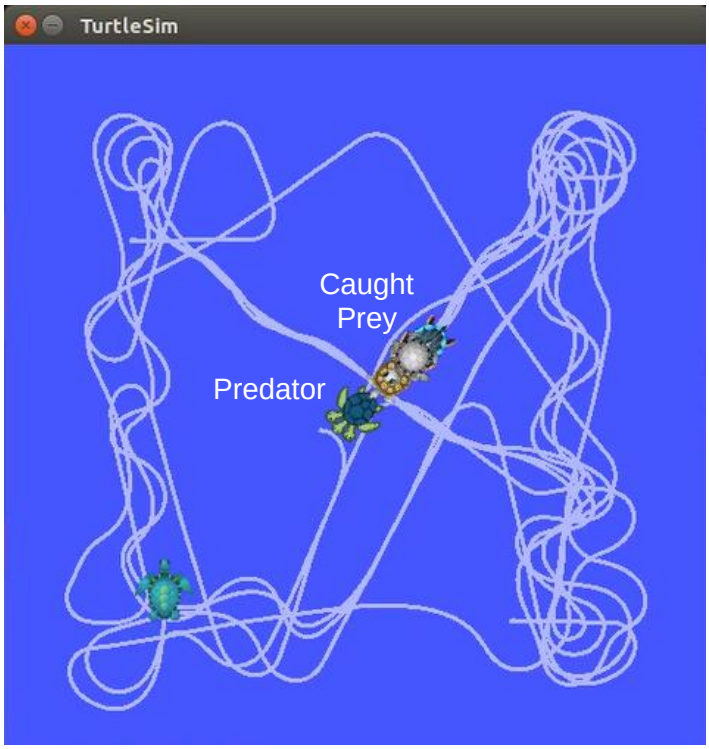


Figure 35 Caught Prey Stick in a Line Behind Predator

The control parameters follow an adaptive method that based on whether all prey have been caught. For example, the formation control becomes more precise when all prey are captured as shown in the figure below.

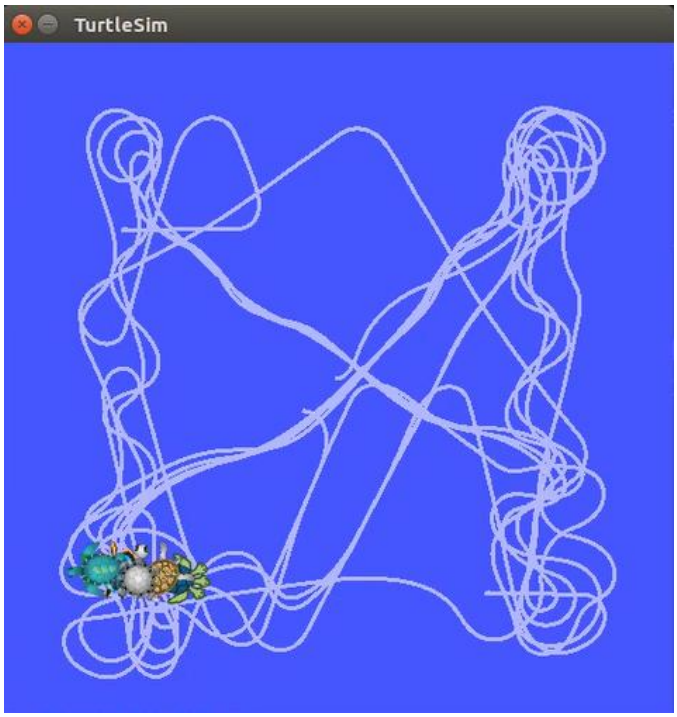


Figure 36 Line Formation After All Prey Have Been Caught

The hunting efficiency during the active phase of the simulation is prioritised by this adaptive behaviour. It is then transitioned to precise formation control when all prey are caught. It resulted in smooth, coordinated movement of all turtles in the final state. Moreover, the integrity of the formation is maintained even during rapid turns or changes in the predator's velocity, which is achieved through the proportional control system, where relying on each prey's velocity adjustments based on its corresponding distance from the desired position.

4.3 Prey Behaviour Analysis

4.3.1 Movement Patterns

A combination of random movement, border avoidance and collision avoidance mechanisms that exhibited by the prey turtles have created natural-looking movement patterns that mimic the animal behaviour in the real world. The paths that are unpredictable but smooth are created through the base random movement with slight directional changes as shown in the screenshots below, thus making the prey challenging targets for the predator.

Figure 37 Prey Moving in Straight Line

Figure 38 Prey Randomly Change Direction

Furthermore, the prey is ensured to remain within the simulation boundaries through the border avoidance mechanism. Interesting interactions when prey approach corners are presented due to greater border threshold to be implemented for the prey than the predator which prey with the margin of 1.5 units while predator with the margin of 1.0 unit. Therefore, this design results in prey to begin turning away from borders earlier and providing more time for the prey to react and flee away. The border avoidance is demonstrated in the screenshots below.

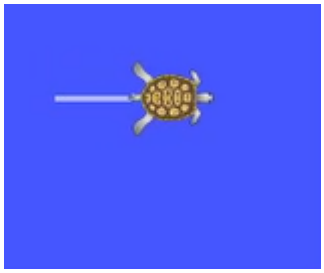


Figure 39 Prey Approaching Corner

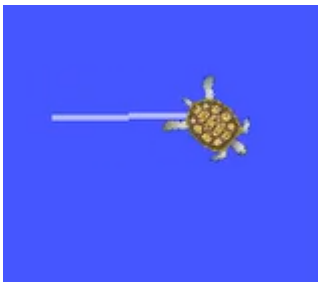


Figure 40 Prey Turning Away from Corner

Figure 41 Prey Escape from Corner Without Colliding the Border

Lastly, the collision avoidance between prey turtles have prevented them from bunching together, which would make them easier targets for the predator to catch multiple prey turtles at once. Therefore, a more challenging scenarios for the predator have been created through the separation between prey maintained by this behaviour. Moreover, it also demonstrates a more realistic simulation where real-life animal will avoid colliding each other. The collision avoidance is presented in the figures below.

Figure 42 Two Prey is Going to Collide Together

Figure 43 Both Prey Turn Away from Each Other

Figure 44 Collision Have Been Avoided

4.3.2 Predator Avoidance Effectiveness

The challenge and realism of the simulation have been significantly enhanced by the predator avoidance mechanism. When the predator approaches the prey within 3.0 units, the prey has accelerated and turned away from the predator. Hence, a dynamic chase scenario where prey actively try to escape rather than blindly moving have been created. Two key features which including the speed increases as the predator gets closer and slight amount of randomness in movement patterns which make it less predictable have significantly increased the predator avoidance effectiveness. The predator avoidance behaviour is demonstrated in the screenshots below.

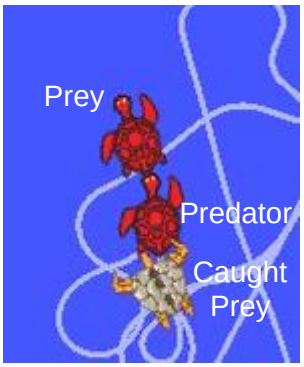


Figure 45 Predator Approaching Prey

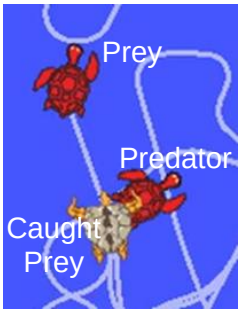


Figure 46 Prey Speed up and Flee Away

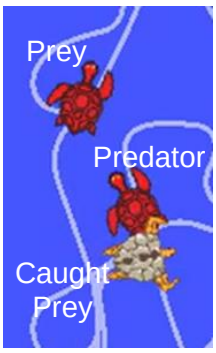


Figure 47 Prey Currently Facing Right

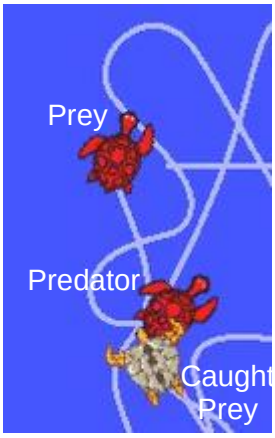


Figure 48 Predator Predict The Future Position on Right Side Hence Turning Right

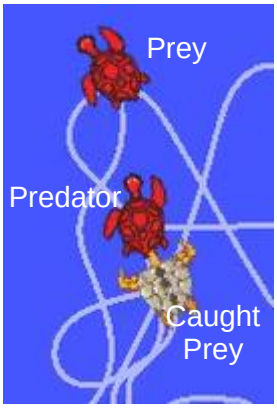


Figure 49 Prey Randomly Turn to Left Thus Increasing the Prediction Difficulty

In short, flee behaviour that is both realistic and challenging for the predator have been created by this combination, whereas speed increase provides prey a chance to escape when the predator approaching, while randomness induce difficulty for the predator's prediction algorithm to forecast the prey's position accurately. While compared to prey without predator avoidance, a longer chase times and more dynamic interactions have been observed. This is due to the fact that prey without predator avoidance would move randomly regardless of the predator proximity, which resulted in easier capture by the predator. On the contrary, a more balanced system where prey have temporarily reasonable chance to escape is created through this implementation of predator avoidance.

4.4 System-wide Performance

A robust and engaging predator-prey dynamics is demonstrated by the overall performance of the system. To illustrate that, begin from the initial configuration with 4 prey positioned across the environment, followed by predator turtle successfully captured all prey turtles over time, with the average hunting completion time that vary according on random factors in prey movement, while with sticking behaviour, collision avoidance, and border avoidance implemented successfully. Moreover, stability across different initial conditions have been provided by the program. To elaborate that, as shown in *Figure 50*, the system adapts appropriately without requiring modification in source code when the prey spawning positions are modified through the launch file parameters. Therefore, this configurability enables the simulation to be tested on different scenarios.

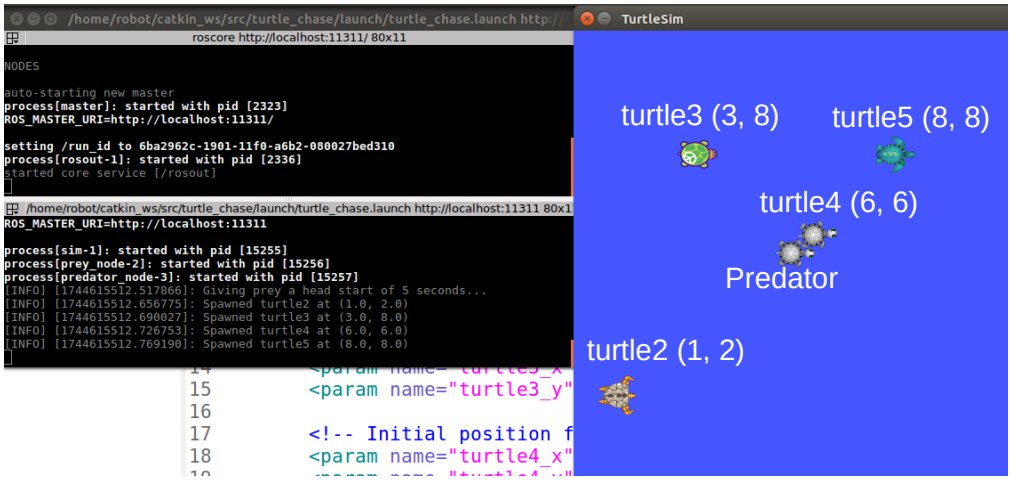


Figure 50 Modify Prey Spawn Points Through Launch File

Moreover, the system robustness has been greatly enhanced through the inclusion of error handling throughout the implementation. For example, the predator node gracefully handles missing prey information:

Figure 51 Error Handling - Missing Prey Poses

Similarly, the error handling for service calls is included in the prey initialisation:

```
try:
    # Call the spawn service to create this turtle
    spawn = rospy.ServiceProxy('/spawn', Spawn)
    spawn(x, y, theta, name)
    rospy.loginfo("Spawned {} at ({}, {})".format(name, x, y))

    # Set ROS parameters to track this prey
    rospy.set_param('/prey/{}'.format(name), True)
    rospy.set_param('/caught/{}'.format(name), False)
except rospy.ServiceException as e:
    rospy.logerr("Failed to spawn {}: {}".format(name, e))
    return
```

Figure 52 Error Handling - Failed Prey Initialisation

In essence, the system is ensured to be able for recovering from potential issues like communication delays or service failures through these error handling, thus contributing to the overall stability of the simulation.

5.0 Analysis and Evaluation

Even though the system successfully fulfilled the objectives, however, there is a limitation that is worth noting. For example, the effectiveness of the algorithm in complex settings. Currently, the prey position prediction algorithm utilises a simple linear extrapolation model, though its efficient, however, it has limited accuracy when it's applied in the environment where prey rapidly changing directions. Hence, in order to enhance the system in future iterations, advanced prediction algorithms such as Kalman filtering could be incorporated to significantly improve the predator's ability to anticipate prey movement, particularly in the context of having prey with complex movement patterns. In brief, Kalman filtering work in the basis of prediction and correction, which means taking into account of the inaccuracies observed in the previous prediction and make corrections or update to provide best estimation, thus increasing the accuracy (Kim & Bang, 2019). Moreover, the system could also apply machine learning integration. For elaboration, the predator could develop more effective pursuit strategies based on experience by utilising reinforcement learning. Similarly, evasion strategies that adapt to the predator's hunting behaviours could be learned by prey (Kober et al., 2013). Last but not least, in order to make the simulation more interesting, multi-predator extensions could be considered. For example, a cooperative hunting strategies can be explored by extending the system to support multiple predators. To elaborate that, in order to realise it, a coordination mechanisms between predators and target assignment algorithms need to be developed as demonstrated by Hu et al., (2021).

6.0 Challenges Encountered and Solutions

Throughout the discovering and implementing process, several challenges were encountered. To illustrate that, the first issue encountered is regarding the ROS communication synchronisation. Initially, before receiving the positions of all prey, the predator would directly begin its hunt, which led to inefficient initial behaviour such as incorrectly determining target prey. This issue is then resolved by introducing a delay after initialisation, as well as implementing the code to wait for prey poses. Moreover, another significant issue faced is due to the smoothing challenges of the turtles' movement. For elaboration, jerky movement in both predator and prey are observed during the early implementations. This issue was then addressed by the implementation of smooth_velocity() method using exponential smoothing, which resulted in significant improvement of the movement quality. In addition, another major issue encountered is regarding the parameter tuning complexity.

Extensive experimentation is required in finding the right balance of the parameters for behaviours including predator avoidance, formation control which stick in a conga line, and prediction horizons. The final solution involving the creation of modular implementation thus offering ease and convenience to allow the parameters to be adjusted and tested independently.

7.0 Conclusion

To sum up, autonomous mobile robotics concepts have successfully demonstrated through this turtle chase and prediction simulation in a visually engaging way. All project objectives including creating multiple prey with random movement while ensuring no collisions with border and other prey, a predator with predictive chasing capabilities, and a sticking behaviour where caught prey follow the predator have been fulfilled by the implementation. In corresponding to that, the hunting efficiency is greatly enhanced by the predator's position prediction algorithm which instead of simply reacting to current target positions, it moves towards the anticipated target movement. Therefore, the important concept in autonomous robotics have been reflected in this approach, which emphasise the value of predictive models for dynamic environments interaction. While on the other hand, the combination of random movement with predator avoidance, border avoidance, and collision avoidance of the prey's mechanism implementation have created a challenging and realistic scenario for the simulation. Hence, the principle of the behaviour-based robotics which stated that complex actions emerge from the interaction of simpler behavioural modules have been demonstrated by these layered behaviours (Dogar et al., 2008). Last but not least, the formation control concepts have been showcases by the sticking behaviour, which a cohesive group movement is created by maintaining the caught prey in specific relative positions. On top of that, the value of context-aware parameter adjustment in robotics systems is also highlighted by the adaptive control parameters, which increase the preciseness when all prey is caught, and the game ended. In essence, an effective demonstration of autonomous mobile robotics principles, such as prediction-based pursuit, behaviour-based control, and formation management have been demonstrated by the system. Easy extension and modification of the system have been enabled by its modular implementation, thus founding a valuable platform to further exploring the robotics concepts.

8.0 References

Cohen, A. (2011, October 26). *Conga Line*. The Independent Florida Alligator. https://www.alligator.org/multimedia/5a0bcc6b186f7c9abe07a0d567d1c2b3

Dogar, M. R., Emre Ugur, Sahin, E., & Cakmak, M. (2008). Using learned affordances for robotic behavior development. *OpenMETU (Middle East Technical University)*, 3802–3807. https://doi.org/10.1109/robot.2008.4543794

Hirata-Acosta, J., Pliego-Jiménez, J., César Cruz-Hernádez, & Martínez-Clark, R. (2021). Leader-Follower Formation Control of Wheeled Mobile Robots without Attitude Measurements. *Applied Sciences*, *11*(12), 5639–5639. https://doi.org/10.3390/app11125639

Hu, R., Tan, N., & Ni, F. (2021). A New Scheme for Cooperative Hunting Tasks with Multiple Targets in Dynamic Environments. *2021 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 1816–1822. https://doi.org/10.1109/robio54168.2021.9739257

Kim, Y., & Bang, H. (2019). Introduction to Kalman Filter and Its Applications. *Introduction and Implementations of the Kalman Filter*. https://doi.org/10.5772/intechopen.80600

Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, *32*(11), 1238–1274. https://doi.org/10.1177/0278364913495721

Li, J. (2023). Analysis And Comparison of Different Tuning Method of PID Control in Robot Manipulator. *Highlights in Science Engineering and Technology*, *71*, 28–36. https://doi.org/10.54097/hset.v71i.12373

Qu, X., Zeng, L., Qu, S., Long, F., & Zhang, R. (2025). An Overview of Recent Advances in Pursuit–Evasion Games with Unmanned Surface Vehicles. *Journal of Marine Science and Engineering*, *13*(3), 458. https://doi.org/10.3390/jmse13030458

Richardson, G., Dickinson, P., Oliver, & Pike, T. W. (2018). Unpredictable movement as an anti-predator strategy. *Proceedings of the Royal Society B Biological Sciences*, *285*(1885), 20181112–20181112. https://doi.org/10.1098/rspb.2018.1112

9.0 Appendices

9.1 Appendix 1 – Copy of Source Code

9.1.1 predator_node.py

```python
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import math
import time

"""Predator node for TurtleSim chase simulation.

Controls the predator turtle (turtle1) to chase and catch prey turtles.
"""

class PredatorTurtle:
    """Class to manage the predator turtle's behaviour."""

    def __init__(self):
        """Initialise the PredatorTurtle node.

        Sets up the ROS node, initialises variables, and configures publishers/subscribers.
        """
        # Initialise the ROS node with name 'predator_node'
        rospy.init_node('predator_node')

        # State variables
        self.game_ended = False  # Flag to indicate if all prey are caught
        self.name = 'turtle1'    # Name of the predator turtle
        self.pose = Pose()       # Current pose of the predator
        self.prey_poses = {}     # Dictionary of prey names to their poses
        self.caught_prey = []    # List of caught prey names

        # Velocity smoothing variables
        self.last_vel = Twist()  # Last velocity for smoothing predator movement
        self.last_vel_prey = {}  # Last velocities for smoothing caught prey movements

        # Publisher for predator velocity commands
        self.pub = rospy.Publisher('/{}/cmd_vel'.format(self.name), Twist, queue_size=10)

        # Subscriber for predator's pose updates
        rospy.Subscriber('/{}/pose'.format(self.name), Pose, self.pose_callback)

        # Set control loop rate to 10Hz
        self.rate = rospy.Rate(10)

        # Dictionary to store publishers for controlling caught prey
        self.caught_pubs = {}

        # Initialise 'caught' parameters for all possible prey (turtle2 to turtle5)
        # These parameters will be used to track which prey have been caught
        for i in range(2, 6):  # Match the 4 prey from the launch file
            prey_name = 'turtle{}'.format(i)
            if not rospy.has_param('/caught/{}'.format(prey_name)):
                rospy.set_param('/caught/{}'.format(prey_name), False)

        # Delay to give prey a head start before predator begins hunting
        rospy.loginfo("Giving prey a head start of 5 seconds...")
        time.sleep(5)

```

```python
def pose_callback(self, data):
    """Update the predator's current pose.

    :param data: Pose message from the topic
    """
    # Store the received pose data
    self.pose = data

def prey_pose_callback(self, data, prey_name):
    """Update the pose of a specific prey turtle.

    :param data: Pose message from the topic
    :param prey_name: Name of the prey turtle
    """
    # Store the received pose data for the specific prey
    self.prey_poses[prey_name] = data

def predict_position(self, prey_pose):
    """Predict a prey's future position based on its current velocity.

    Uses a simple linear prediction based on current velocity vector.

    :param prey_pose: Current pose of the prey
    :return: Tuple (x, y) of predicted position
    """
    # Time horizon for prediction (in seconds)
    time_step = 0.5

    # Calculate predicted position using current velocity and direction
    pred_x = prey_pose.x + prey_pose.linear_velocity * math.cos(prey_pose.theta) * time_step
    pred_y = prey_pose.y + prey_pose.linear_velocity * math.sin(prey_pose.theta) * time_step

    return pred_x, pred_y

def distance(self, x1, y1, x2, y2):
    """Calculate Euclidean distance between two points.

    :param x1, y1: Coordinates of the first point
    :param x2, y2: Coordinates of the second point
    :return: Distance between the points
    """
    # Standard Euclidean distance formula
    return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

def smooth_velocity(self, target_vel):
    """Smooth velocity commands to reduce jerkiness.

    Uses exponential smoothing to blend new target velocity with previous velocity.

    :param target_vel: Desired Twist velocity
    :return: Smoothed Twist velocity
    """
    # Smoothing factor (0 to 1, higher = less smoothing)
    alpha = 0.3

    # Apply exponential smoothing to linear and angular velocities
    self.last_vel.linear.x = alpha * target_vel.linear.x + (1 - alpha) * self.last_vel.linear.x
    self.last_vel.angular.z = alpha * target_vel.angular.z + (1 - alpha) * self.last_vel.angular.z

    return self.last_vel
```

```python

def avoid_borders(self, vel):
    """Adjust velocity to prevent hitting simulation borders.

    Checks proximity to borders and adjusts direction to avoid collisions.

    :param vel: Current Twist velocity
    :return: Adjusted Twist velocity
    """
    # Distance from border to start avoidance behaviour
    border_margin = 1.0 # Smaller margin than prey as the predator's primary goal is pursuit


    # Create a new velocity message to avoid modifying the input
    adjusted_vel = Twist()
    adjusted_vel.linear.x = vel.linear.x
    adjusted_vel.angular.z = vel.angular.z

    # Check proximity to left border (west)
    if self.pose.x < 1 + border_margin:
        target_angle = 0  # Turn east (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 2.0 * angle_diff # Higher angular velocity scaling than prey for quicker turns during pursuit.
        adjusted_vel.linear.x = 0.5  # Reduce speed during avoidance, greater speed reduction than prey to ensure sharper turns and prevent overshooting

    # Check proximity to right border (east)
    elif self.pose.x > 10 - border_margin:
        target_angle = math.pi  # Turn west (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 2.0 * angle_diff
        adjusted_vel.linear.x = 0.5

    # Check proximity to bottom border (south)
    elif self.pose.y < 1 + border_margin:
        target_angle = math.pi / 2  # Turn north (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 2.0 * angle_diff
        adjusted_vel.linear.x = 0.5

    # Check proximity to top border (north)
    elif self.pose.y > 10 - border_margin:
        target_angle = 3 * math.pi / 2  # Turn south (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 2.0 * angle_diff
        adjusted_vel.linear.x = 0.5

    return adjusted_vel
```

```python

def sync_caught_prey(self, all_caught=False):
    """Synchronise caught prey to follow the predator in a line.

    Makes caught prey turtles follow behind the predator in a 'conga line'.

    :param all_caught: True if all prey are caught, enabling more precise control
    """
    # Loop through each caught prey in order
    for i, prey_name in enumerate(self.caught_prey):
        # Skip if we don't have a publisher for this prey
        if prey_name not in self.caught_pubs:
            continue

        # Initialise velocity command for this prey
        follow_vel = Twist()

        # Calculate desired position: each prey should be positioned
        # behind the predator, spaced by index in the caught list
        distance_behind = 0.35 * (i + 1)  # Each prey spaced 0.35 units apart

        # Calculate target position behind predator based on predator's orientation
        desired_x = self.pose.x - distance_behind * math.cos(self.pose.theta)
        desired_y = self.pose.y - distance_behind * math.sin(self.pose.theta)

        # Get current prey position (if available)
        prey_pose = self.prey_poses.get(prey_name)
        if not prey_pose:
            continue

        # Calculate movement parameters towards desired position
        dist = self.distance(prey_pose.x, prey_pose.y, desired_x, desired_y)
        angle = math.atan2(desired_y - prey_pose.y, desired_x - prey_pose.x)

        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (angle - prey_pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi

        # Adjust velocity based on whether all prey are caught
        # When all prey are caught, more precise control is used
        if all_caught:
            # Stronger control when all prey are caught for tighter formation
            follow_vel.linear.x = 4.0 * dist
            follow_vel.angular.z = 5.0 * angle_diff

            # Stop when very close to desired position
            if dist < 0.1:
                follow_vel.linear.x = 0.0
                follow_vel.angular.z = 0.0
        else:
            # More relaxed control when still hunting
            follow_vel.linear.x = 2.5 * dist
            follow_vel.angular.z = 3.5 * angle_diff

        # Apply velocity smoothing to reduce jerkiness
        if prey_name in self.last_vel_prey:
            alpha = 0.4  # Smoothing factor
            self.last_vel_prey[prey_name].linear.x = alpha * follow_vel.linear.x + (1 - alpha) * self.last_vel_prey[prey_name].linear.x
            self.last_vel_prey[prey_name].angular.z = alpha * follow_vel.angular.z + (1 - alpha) * self.last_vel_prey[prey_name].angular.z
            follow_vel = self.last_vel_prey[prey_name]
        else:
            # Initialise smoothing for new caught prey
            self.last_vel_prey[prey_name] = follow_vel

        # Send command to the prey
        self.caught_pubs[prey_name].publish(follow_vel)
```

```python

def chase(self):
    """Execute the main chase loop to pursue and catch prey.

    This is the primary method that runs continuously to:
    1. Track prey positions
    2. Chase the nearest uncaught prey
    3. Capture prey when close enough
    4. Make caught prey follow in a line
    """
    # Reset velocity smoothing for caught prey
    self.last_vel_prey = {}

    # Step 1: Subscribe to all prey poses dynamically
    prey_count = 0
    for i in range(2, 6):  # Support 4 prey (turtle2 to turtle5)
        prey_name = 'turtle{}'.format(i)
        try:
            # Check if this prey exists in parameters
            if rospy.has_param('/prey/{}'.format(prey_name)):
                prey_count += 1
                # Create a callback closure to capture the prey name
                def create_callback(name):
                    return lambda msg: self.prey_pose_callback(msg, name)

                # Subscribe to this prey's pose topic
                rospy.Subscriber('/{}/pose'.format(prey_name), Pose, create_callback(prey_name))

                # Create publisher for controlling this prey (when caught)
                if prey_name not in self.caught_pubs:
                    self.caught_pubs[prey_name] = rospy.Publisher('/{}/cmd_vel'.format(prey_name), Twist, queue_size=10)
        except rospy.ROSException as e:
            rospy.logwarn("Error subscribing to {}: {}".format(prey_name, e))
            continue

    rospy.loginfo("Found {} prey turtles to chase".format(prey_count))
```

```python
# Step 2: Wait for initial prey positions before starting chase
timeout = rospy.Time.now() + rospy.Duration(5.0)
while len(self.prey_poses) < prey_count and rospy.Time.now() < timeout:
    rospy.sleep(0.1)

# Warn if not all prey positions were detected
if len(self.prey_poses) < prey_count:
    rospy.logwarn("Could only detect {} prey positions".format(len(self.prey_poses)))

rospy.loginfo("Predator start hunting!")

# Step 3: Main chase loop
while not rospy.is_shutdown():
    # Check if all prey are caught
    all_caught = len(self.prey_poses) > 0 and all(prey_name in self.caught_prey for prey_name in self.prey_poses)

    # If all prey are caught and not already marked as game ended
    if all_caught and self.caught_prey and not self.game_ended:
        rospy.loginfo("All prey turtles have been caught, game ended.")
        self.game_ended = True

    # Synchronise caught prey positions to follow predator
    if self.caught_prey:
        self.sync_caught_prey(all_caught)

    # If all prey are caught, stop the predator
    if all_caught and self.caught_prey:
        self.pub.publish(Twist())  # Zero velocity
        self.rate.sleep()
        continue

    # Wait if no prey poses have been received yet
    if not self.prey_poses:
        rospy.loginfo("No prey poses received yet")
        rospy.sleep(1.0)
        continue

    # Find nearest uncaught prey (using predicted position)
    target = None
    min_dist = float('inf')
    for prey_name, pose in self.prey_poses.items():
        if prey_name not in self.caught_prey:
            # Predict where this prey will be
            pred_x, pred_y = self.predict_position(pose)
            # Calculate distance to predicted position
            dist = self.distance(self.pose.x, self.pose.y, pred_x, pred_y)
            # Keep track of nearest prey
            if dist < min_dist:
                min_dist = dist
                target = (prey_name, pred_x, pred_y)

```

```
331
332 # If we found a target prey
333 if target:
334     prey_name, pred_x, pred_y = target
335     current_prey_pose = self.prey_poses[prey_name]
336
337     # Check if prey is within catching distance
338     current_dist = self.distance(self.pose.x, self.pose.y, current_prey_pose.x, current_prey_pose.y)
339     if current_dist < 0.6:  # Catching distance threshold
340         # Calculate angle to prey
341         angle_to_prey = math.atan2(current_prey_pose.y - self.pose.y, current_prey_pose.x - self.pose.x)
342
343         # Check if prey is within predator's field of view
344         angle_diff = abs((angle_to_prey - self.pose.theta) % (2 * math.pi))
345         if angle_diff > math.pi:
346             angle_diff = 2 * math.pi - angle_diff
347
348         # Can only catch prey if they're in front of the predator (within 90 degree field of view)
349         if angle_diff < math.pi / 2:
350             # Mark prey as caught
351             self.caught_prey.append(prey_name)
352             rospy.set_param('/caught/{}'.format(prey_name), True)
353             rospy.loginfo("Caught {}".format(prey_name))
354
355             # Initialise velocity smoothing for caught prey
356             self.last_vel_prey[prey_name] = Twist()
357         continue
358
359     # Calculate velocity towards predicted prey position
360     vel = Twist()
361
362     # Calculate angle towards predicted position
363     angle = math.atan2(pred_y - self.pose.y, pred_x - self.pose.x)
364
365     # Calculate angle difference (normalised to -pi to pi range)
366     angle_diff = (angle - self.pose.theta) % (2 * math.pi)
367     if angle_diff > math.pi:
368         angle_diff -= 2 * math.pi
369
370     # Set linear velocity proportional to distance (capped at 1.2)
371     vel.linear.x = min(1.2, 1.0 * min_dist)
372
373     # Set angular velocity proportional to angle difference
374     vel.angular.z = 3.0 * angle_diff
375
376     # Apply border avoidance and velocity smoothing
377     vel = self.avoid_borders(vel)
378     vel = self.smooth_velocity(vel)
379
380     # Send velocity command to predator
381     self.pub.publish(vel)
382
383 # Maintain loop rate
384 self.rate.sleep()
385
```

```
386 if __name__ == '__main__':
387     # Create predator instance
388     predator = PredatorTurtle()
389     try:
390         # Start chase loop
391         predator.chase()
392     except rospy.ROSInterruptException:
393         pass
```

9.1.2 prey_node.py

```python
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from turtlesim.srv import Spawn
import random
import math

"""Prey node for TurtleSim chase simulation.

Manages multiple prey turtles to avoid the predator and navigate the environment.
"""

class PreyTurtle:
    """Class to control an individual prey turtle."""

    def __init__(self, name, x, y, theta):
        """Initialise a prey turtle with a starting position.

        :param name: Name of the turtle (e.g., 'turtle2')
        :param x: Initial x-coordinate
        :param y: Initial y-coordinate
        :param theta: Initial orientation in radians
        """
        # Store the turtle's name
        self.name = name

        # Initialise state variables
        self.pose = Pose()          # Current pose of this turtle
        self.other_poses = {}       # Poses of other turtles (including predator)
        self.caught = False         # Flag indicating if caught by predator
        self.last_vel = Twist()     # Last velocity for smoothing

        # Override initial position with launch file parameters
        x = rospy.get_param('/prey_node/{}_x'.format(name), x)
        y = rospy.get_param('/prey_node/{}_y'.format(name), y)

        # Spawn the turtle at the initial position
        rospy.wait_for_service('/spawn')  # Wait for spawn service to be available
        try:
            # Call the spawn service to create this turtle
            spawn = rospy.ServiceProxy('/spawn', Spawn)
            spawn(x, y, theta, name)
            rospy.loginfo("Spawned {} at ({}, {})".format(name, x, y))

            # Set ROS parameters to track this prey
            rospy.set_param('/prey/{}'.format(name), True)
            rospy.set_param('/caught/{}'.format(name), False)
        except rospy.ServiceException as e:
            rospy.logerr("Failed to spawn {}: {}".format(name, e))
            return

        # Set up publisher for sending velocity commands to this turtle
        self.pub = rospy.Publisher('/{}/cmd_vel'.format(name), Twist, queue_size=10)

        # Subscribe to this turtle's pose updates
        rospy.Subscriber('/{}/pose'.format(name), Pose, self.pose_callback)

        # Subscribe to poses of all other turtles (for collision avoidance)
        for i in range(1, 6):  # turtle1 (predator) to turtle5
            other_name = 'turtle{}'.format(i)
            if other_name != self.name:
                try:
                    # Create a callback closure to capture the other turtle's name
                    def create_callback(name):
                        return lambda msg: self.other_pose_callback(msg, name)

                    # Subscribe to other turtle's pose topic
                    rospy.Subscriber('/{}/pose'.format(other_name), Pose, create_callback(other_name))
                except rospy.ROSException as e:
                    rospy.logwarn("Error subscribing to {}: {}".format(other_name, e))
```

```python
def pose_callback(self, data):
    """Update this turtle's pose and caught status.

    :param data: Pose message from the topic
    """
    # Store the received pose data
    self.pose = data

    # Update caught status from ROS parameter
    self.caught = rospy.get_param('/caught/{}'.format(self.name), False)

def other_pose_callback(self, data, other_name):
    """Update the pose of another turtle.

    :param data: Pose message from the topic
    :param other_name: Name of the other turtle
    """
    # Store the received pose data for the specific turtle
    self.other_poses[other_name] = data

def distance(self, x1, y1, x2, y2):
    """Calculate Euclidean distance between two points.

    :param x1, y1: Coordinates of the first point
    :param x2, y2: Coordinates of the second point
    :return: Distance between the points
    """
    # Standard Euclidean distance formula
    return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

def smooth_velocity(self, target_vel):
    """Smooth velocity commands to reduce jerkiness.

    Uses exponential smoothing to blend new target velocity with previous velocity.

    :param target_vel: Desired Twist velocity
    :return: Smoothed Twist velocity
    """
    # Smoothing factor
    alpha = 0.3

    # Apply exponential smoothing to linear and angular velocities
    self.last_vel.linear.x = alpha * target_vel.linear.x + (1 - alpha) * self.last_vel.linear.x
    self.last_vel.angular.z = alpha * target_vel.angular.z + (1 - alpha) * self.last_vel.angular.z

    return self.last_vel
```

```python
def avoid_collisions(self, vel):
    """Adjust velocity to avoid collisions with other prey turtles.

    Only avoids uncaught prey - no need to avoid caught prey or predator.

    :param vel: Current Twist velocity
    :return: Adjusted Twist velocity
    """
    # Don't avoid collisions if already caught (controlled by predator)
    if self.caught:
        return vel

    # Distance threshold for collision avoidance
    avoidance_distance = 1.0

    # Check distance to each other turtle
    for other_name, other_pose in self.other_poses.items():
        # Skip predator (turtle1) and caught prey
        if other_name == 'turtle1' or rospy.get_param('/caught/{}'.format(other_name), False):
            continue

        # Calculate distance to other turtle
        dist = self.distance(self.pose.x, self.pose.y, other_pose.x, other_pose.y)

        # If too close, steer away
        if dist < avoidance_distance:
            # Calculate angle to other turtle
            angle_to_other = math.atan2(other_pose.y - self.pose.y, other_pose.x - self.pose.x)

            # Calculate angle to flee (opposite direction)
            flee_angle = (angle_to_other + math.pi) % (2 * math.pi)

            # Calculate angle difference (normalised to -pi to pi range)
            angle_diff = (flee_angle - self.pose.theta) % (2 * math.pi)
            if angle_diff > math.pi:
                angle_diff -= 2 * math.pi

            # Set velocity to steer away
            vel.angular.z = 1.5 * angle_diff
            vel.linear.x = 0.8

    return vel
```

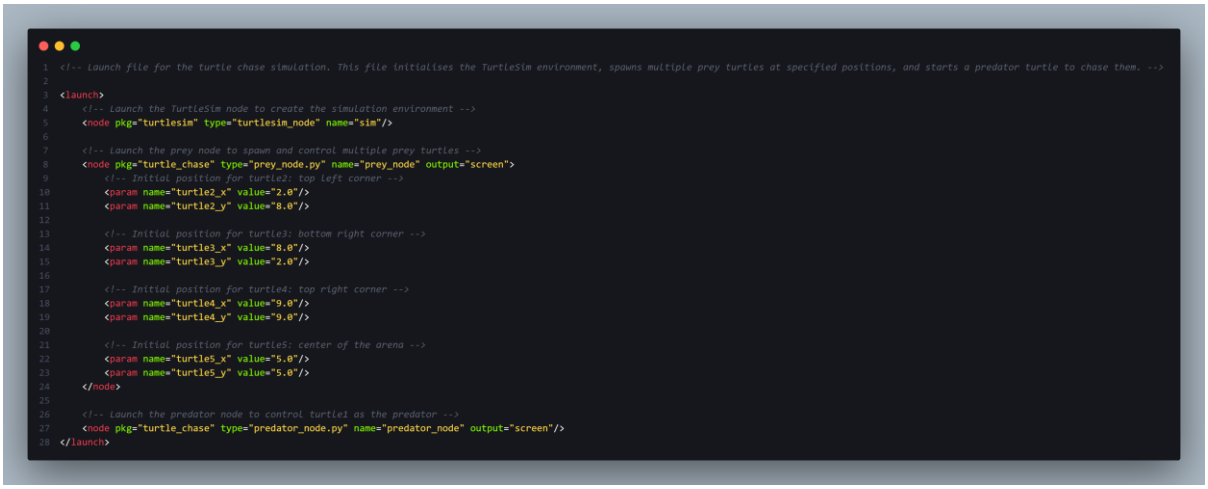
```python

def avoid_borders(self, vel):
    """Adjust velocity to stay within simulation borders.

    Steers away from borders when getting too close.

    :param vel: Current Twist velocity
    :return: Adjusted Twist velocity
    """
    # Don't avoid borders if already caught (controlled by predator)
    if self.caught:
        return vel

    # Distance threshold from border to trigger avoidance
    border_margin = 1.5 #Larger margin than prey gives more reaction time for evasion.

    # Create a new velocity message to avoid modifying the input
    adjusted_vel = Twist()
    adjusted_vel.linear.x = vel.linear.x
    adjusted_vel.angular.z = vel.angular.z

    # Check proximity to left border (west)
    if self.pose.x < border_margin:
        target_angle = 0  # Turn east (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 1.0 * angle_diff # Prey turns less aggressively than predator, smoother turns beneficial for evasion.
        adjusted_vel.linear.x = 0.8 # Reduce speed during avoidance, smaller speed reduction than predator to maintain some momentum for escape.

    # Check proximity to right border (east)
    elif self.pose.x > 11.0 - border_margin:
        target_angle = math.pi  # Turn west (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 1.0 * angle_diff
        adjusted_vel.linear.x = 0.8

    # Check proximity to bottom border (south)
    elif self.pose.y < border_margin:
        target_angle = math.pi / 2  # Turn north (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 1.0 * angle_diff
        adjusted_vel.linear.x = 0.8

    # Check proximity to top border (north)
    elif self.pose.y > 11.0 - border_margin:
        target_angle = 3 * math.pi / 2  # Turn south (away from border)
        # Calculate angle difference (normalised to -pi to pi range)
        angle_diff = (target_angle - self.pose.theta) % (2 * math.pi)
        if angle_diff > math.pi:
            angle_diff -= 2 * math.pi
        adjusted_vel.angular.z = 1.0 * angle_diff
        adjusted_vel.linear.x = 0.8

    return adjusted_vel
```

```python

def avoid_predator(self, vel):
    """Adjust velocity to flee from the predator if nearby.

    Increases speed and steers away when predator gets close.

    :param vel: Current Twist velocity
    :return: Adjusted Twist velocity
    """
    # Don't avoid predator if already caught (controlled by predator)
    if self.caught:
        return vel

    # Check if we have the predator's position
    if 'turtle1' in self.other_poses:
        predator = self.other_poses['turtle1']

        # Calculate distance to predator
        dist = self.distance(self.pose.x, self.pose.y, predator.x, predator.y)

        # If predator is nearby (within 3.0 units), flee
        if dist < 3.0:
            # Calculate angle to predator
            angle_to_pred = math.atan2(predator.y - self.pose.y, predator.x - self.pose.x)

            # Calculate angle to flee (opposite direction)
            flee_angle = (angle_to_pred + math.pi) % (2 * math.pi)

            # Calculate angle difference (normalised to -pi to pi range)
            angle_diff = (flee_angle - self.pose.theta) % (2 * math.pi)
            if angle_diff > math.pi:
                angle_diff -= 2 * math.pi

            # Set velocity to flee (turn away from predator)
            vel.angular.z = 2.0 * angle_diff

            # Speed up when closer to predator (max 1.5)
            vel.linear.x = min(1.5, 1.0 + (3.0 - dist) * 0.3)

            # Add slight randomness to make prey harder to predict
            vel.angular.z += random.uniform(-0.1, 0.1)

    return vel
```

```python

def move_once(self):
    """Compute and publish a single velocity command for this turtle.

    Combines different behaviours to determine prey movement:
    1. Default random movement
    2. Predator avoidance (highest priority)
    3. Border avoidance
    4. Collision avoidance with other prey
    """
    # If caught, don't control movement (predator will control)
    if self.caught:
        return

    # Base movement: constant speed with slight random turning
    vel = Twist()
    vel.linear.x = 0.7
    vel.angular.z = random.uniform(-0.2, 0.2)

    # Apply avoidance behaviours in priority order
    vel = self.avoid_predator(vel)   # 1. Flee from predator (highest priority)
    vel = self.avoid_borders(vel)    # 2. Stay within borders
    vel = self.avoid_collisions(vel) # 3. Avoid other prey

    # Smooth and publish the velocity
    vel = self.smooth_velocity(vel)
    self.pub.publish(vel)
```

```python

class PreyNode:
    """Class to manage multiple prey turtles."""

    def __init__(self, prey_configs):
        """Initialise the PreyNode with a list of prey configurations.

        :param prey_configs: List of tuples (name, x, y, theta) for each prey
        """
        # Create PreyTurtle instances for each configuration
        self.preys = [PreyTurtle(name, x, y, theta) for name, x, y, theta in prey_configs]

        # Set control loop rate to 10Hz
        self.rate = rospy.Rate(10)

    def run(self):
        """Run the main loop to control all prey turtles."""
        while not rospy.is_shutdown():
            # Update movement for each prey turtle
            for prey in self.preys:
                prey.move_once()

            # Maintain loop rate
            self.rate.sleep()

if __name__ == '__main__':
    # Initialise the ROS node
    rospy.init_node('prey_node')

    # Default configurations (these will typically be overridden by launch file)
    prey_configs = [
        ('turtle2', 0.0, 0.0, 0.0),
        ('turtle3', 0.0, 0.0, 0.0),
        ('turtle4', 0.0, 0.0, 0.0),
        ('turtle5', 0.0, 0.0, 0.0)
    ]

    # Create prey node instance with the specified configurations
    prey_node = PreyNode(prey_configs)

    try:
        # Start the main control loop
        prey_node.run()
    except rospy.ROSInterruptException:
        pass
```

9.1.3 turtle_chase.launch



```
<!-- Launch file for the turtle chase simulation. This file initialises the TurtleSim environment, spawns multiple prey turtles at specified positions, and starts a predator turtle to chase them. -->

<launch>
    <!-- Launch the TurtleSim node to create the simulation environment -->
    <node pkg="turtlesim" type="turtlesim_node" name="sim"/>

    <!-- Launch the prey node to spawn and control multiple prey turtles -->
    <node pkg="turtle_chase" type="prey_node.py" name="prey_node" output="screen">
        <!-- Initial position for turtle2: top left corner -->
        <param name="turtle2_x" value="2.0"/>
        <param name="turtle2_y" value="8.0"/>

        <!-- Initial position for turtle3: bottom right corner -->
        <param name="turtle3_x" value="8.0"/>
        <param name="turtle3_y" value="2.0"/>

        <!-- Initial position for turtle4: top right corner -->
        <param name="turtle4_x" value="9.0"/>
        <param name="turtle4_y" value="9.0"/>

        <!-- Initial position for turtle5: center of the arena -->
        <param name="turtle5_x" value="5.0"/>
        <param name="turtle5_y" value="5.0"/>
    </node>

    <!-- Launch the predator node to control turtle1 as the predator -->
    <node pkg="turtle_chase" type="predator_node.py" name="predator_node" output="screen"/>
</launch>
```

9.2 Appendix 2 – Video Evidence

9.2.1 Prey without Predator Avoidance

https://uowmalaysia-my.sharepoint.com/:v:/g/personal/0207368_student_uow_edu_my/EXQn9Pbbx5hBmsRsVErkAPABemLSCWlc_wr3KaqByX3EJQ?nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJPbmVEcml2ZUZvckJ1c2luZXNzIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsVmlld0lkZSI6InZpZXciLCJyZWZlcnJhbFZpZXciOiJNeUZpbGVzTGlua0NvcHkifX0&e=GfKwl9

9.2.2 Prey with Predator Avoidance

https://uowmalaysia-my.sharepoint.com/:v:/g/personal/0207368_student_uow_edu_my/EW7jZsjRcydBsMhZGT6NLG4B1cdrRdyCwIvYxJduR7zpMQ?nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJPbmVEcml2ZUZvckJ1c2luZXNzIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsVmlld0lkZSI6InZpZXciLCJyZWZlcnJhbFZpZXciOiJNeUZpbGVzTGlua0NvcHkifX0&e=XjWfHz

REPORT COMPONENT (100%)

CAI3034N Autonomous Mobile Robotics
MARKING RUBRIC
ASSIGNMENT 1
Assignment Weighting (30%)

LEARNING OUTCOME	MARKING CRITERIA	SCALE					
		Fail (0-49)	**3rd Class (50-59)**	**2nd Lower Class (60-69)**	**2nd Upper Class (70-79)**	**1st Class (80-100)**	**YOUR MARKS/COMMENTS**
CO3: Implement intelligent control strategies, by programming autonomous mobile robots to perform complex tasks in dynamic environments including obstacle avoidance, planning and navigation, robotic mapping and self-localisation.(C3, PLO3)	Algorithm development (40%)	• ROS package is created but without launch file. Neither publisher nor subscriber nodes are created. • Fail to create and program multiple turtles to move. • The predator turtle fails to catch turtles, with the prediction and chasing logic not working.	• ROS package is created but without launch file. Only publisher node is created. • Multiple turtles are created to move but fail to avoid collision with borders and among the turtles. • The predator turtle catches some other turtles but struggles with predictions or sticking to them.	• ROS package and launch file are created. Publisher and subscriber nodes are created but with major flaws. • Multiple turtles are created to move across the environment but fail to avoid collision with the borders or among the turtles. • The predator turtle catches most other turtles, with minor issues in prediction or movement.	• ROS package and launch file are created. Publisher and subscriber nodes are created but with minor error. • Multiple turtles are created to move across the environment and avoid collision with borders and among the turtles but with minor error. • The predator turtle catches all other turtles, with minor issues in prediction or movement.	• ROS package and launch file are created. Publisher and subscriber nodes are created with no error. • Multiple turtles are created to move across the environment, and avoid collision with borders and among the turtles with no error. • The predator turtle catches all other turtles, sticking to each and chasing the next with smooth, accurate predictions.	
	Documentation (50%)	• Content is inaccurate. Information is incomplete, inaccurate, or not presented in a logical order, making it difficult to follow. • Do not provide details about techniques used in navigation, path prediction, and collision avoidance. • No results and discussion.	• Content is either questionable or incomplete. Information is not presented in a logical order, making it difficult to follow. • Little explanation on the techniques used in navigation, path prediction, and collision avoidance. • Results are presented but poorly discussed.	• Content is accurate but some required information is missing and/or not presented in a logical order, making it difficult to follow. • Reasonable explanation on the techniques used in navigation, path prediction, and collision avoidance. • Results are presented with reasonable discussion.	• Content is accurate but some required information is missing and/or not presented in a logical order, but it is still generally easy to follow. • Good explanation on the techniques used in navigation, path prediction, and collision avoidance. • Results are presented with good discussion.	• Content is accurate and all required information is presented in a logical order. • Excellence explanation on the techniques used in navigation, path prediction, and collision avoidance. • Results and discussion are very well presented which give the reader important information that goes beyond the obvious or predictable.	
	Code quality (10%)	• Very poor program structure and without code comments.	• Poor program structure but with some code comments.	• Clear program structure and appropriate comments.	• The program code is well structured and commented.	• The program code is efficient, well structured, and commented.	
						Overall score (100%)	