**Project Overview:**

Our project, entitled "Catch Turtle All", is an interactive simulation developed in the Turtlesim environment using the Robotics Operating System 2 (ROS2). The core goal of the project is to simulate a master turtle that is able to autonomously navigate and "catch" other turtles that randomly appear in the environment.

**Project objectives:**

1. **Master Turtle Behaviour:** Our primary goal is to program master turtles to autonomously identify and navigate to the nearest turtle in the environment. This involves developing dynamic decision-making capabilities to adapt to changing conditions brought about by the constant generation of new sea turtles.
2. **Turtle Generation:** We aim to implement a system that generates a new turtle every 3 seconds at a random location, adding unpredictability and complexity to the simulation.
3. **Turtle Chain Formation:** Once a turtle has been captured, it should fit seamlessly into the chain formation, following directly behind the main turtle. This requires that the programmed sea turtles remain in close contact with the turtles in front of them to ensure that a coherent chain is formed.
4. **Simulation Control:** The entire simulation must be controlled via a startup file, allowing easy starting and stopping of the simulation and continuous operation of the nodes for sustained operation

**Component description:**

1. **Preparation before the project:** Assuming that we have installed the Ubuntu system of 22.04 and ros2 humble, the first step is to create a workspace and function pack: Run (1) mkdir -p~/ros2\_ws/src (2) cd ros2\_ws/src (3) ros2 pkg create --build-type ament python catch\_turtle. The second step is to put the written code in the ros2\_ws/src/catch\_turtle/catch\_turtle/ directory and make changes to setup.py and package.xml to configure the environment correctly. In the third step we compile the project package by running colcon build --packages-select catch\_turtle. Then you can run the code!
2. **Program component description:**

**2.1)** **Random turtle generation every three seconds:** In the code we use the spawn\_turtle and handle\_spawn functions to achieve this feature.

The spawn\_turtl function first creates a request object and randomly generates the position and orientation of the new tortoise. We ensure that the name given to the tortoise is unique by calculating the length in the turtle\_pose dictionary (which stores the position of each tortoise). Then we use the asynchronous service invocation and set the callback.

The handle\_spawn function is the callback that handles the response to generate the new turtle service. First, future.result() is called to obtain the result of service invocation, and the corresponding success is judged. For example, success: record the tortoise and generate diary information; Create subscribers for the new turtle (create callbacks through lambda expressions; Make sure the location information of different tortoises is handled correctly.) And the publisher (to whom the move command is sent); Update the follow dictionary and set follow goals for the new tortoise in the following\_dict dictionary so that the new tortoise can participate in dynamic interactions in the simulation environment.

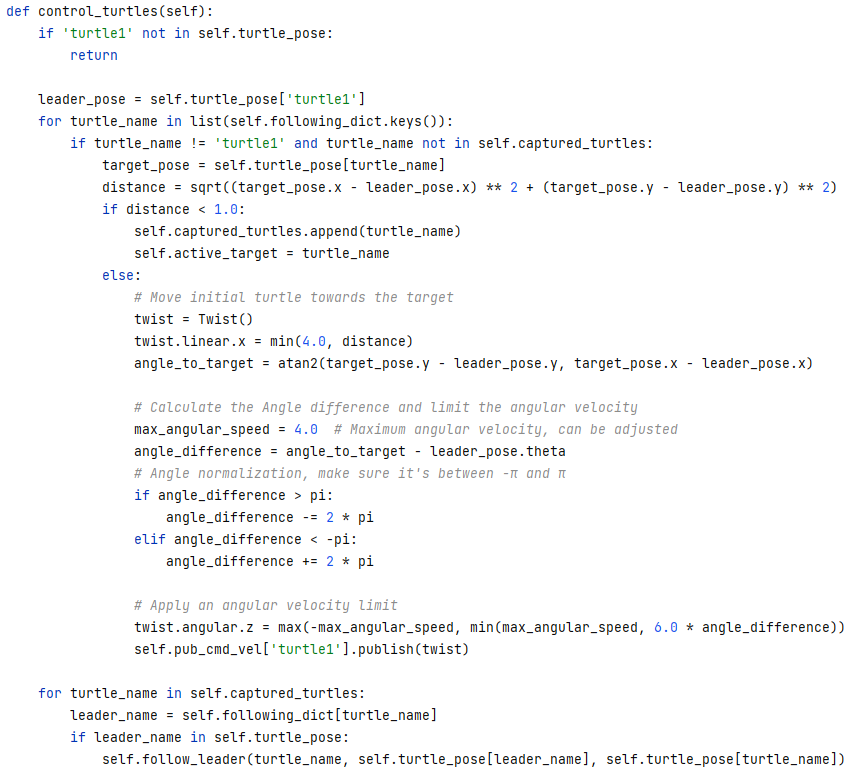
With these two functions, TurtleController can dynamically add new tortoises to the simulation environment and configure them with the necessary communication and control mechanisms so that these tortoises can be further controlled and managed.



**2.2)Catch turtles:** This function is mainly implemented by the control\_turtle function, the main purpose of this function is to periodically check and update the turtle's movement status, so that it can adjust accordingly to the position of other turtles. It examines the position relationship between each tortoise and its target tortoise to determine whether it needs to change speed or direction in order to achieve the following function.

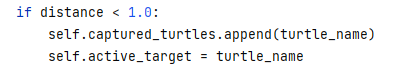
First get the position of the turtle by calling the turtle\_pose function. Then loop through the following dictionary of all turtles. For each turtle except 'turtle1' and those that have been captured, do the following. 1. Calculate the distance between the current turtle and the leader. If the distance is less than 1.0, the turtle has "caught" the target turtle, adding it to the catch list and updating the active target. Otherwise, create a Twist message that sets the line speed to the distance and the smaller value in 4.0 to get moderately close to the target. 2. Calculate and set the angular velocity. The Angle difference between the target direction and the current direction is calculated, the Angle difference is normalized to [−π,π], the angular velocity is set according to the Angle difference, and the maximum angular velocity is limited, and the Twist message is issued to control the movement of the main turtle. 3. Deal with the following behavior of captured turtles. For each captured turtle, find the one it should follow. If the location information of the lead tortoise is available, call the follow\_leader method to implement the specific following logic.

Through these steps, the control\_turtles function continuously adjusts the speed and direction of the tortoise to achieve dynamic following and interactive behavior. This function is the core of turtle behavior control in simulation environment, and realizes complex multi-turtle interaction logic.



**2.3)follow\_leader:** This function accepts three parameters: the name of the follower (turtle\_name), the location of the lead turtle (leader\_pose), and the current location of the follower (target\_pose). Its goal is to calculate the direction and speed at which the follower should move in order to stay a fixed distance behind the lead tortoise.

First of all, we explain the logic of turtle following. The method of following is not to let all turtles follow the original turtle but to have an increasing distance, because this method, although it seems that the turtles will form a queue in a straight line, it will not be a strict chain shape when turning. To solve this problem I used nested following logic where if the initial tortoise is close enough to its target (distance is less than 1.0) then the initial tortoise is considered to have "captured" its target and the captured tortoise becomes the next active following target. When a new tortoise is generated and captured by the original tortoise it is set to follow the current active\_target, that is, the last tortoise that was activated or approached.



**Control\_turtles**

Now that we know the logic, let's go back to this function. First, the function calculates the target position behind the forest head turtle, then calculates the distance and Angle following the turtle to the target position, sets the Twist message to control the turtle's movement, calculates the Angle difference and limits the angular speed, and finally issues the control command.

The core of the function follow\_leader is to calculate and implement a following strategy that allows the tortoise to maintain a specific position behind the leader tortoise. By calculating the position and Angle difference, and then adjusting the linear and angular velocity, this function effectively controls the tortoise to maintain formation and distance, and is a key part of achieving complex interactive behavior in the simulation environment.

1. **Project run:** After understanding the code, we can start to run the code. First CD ros2\_ws in terminal, run the source/opt/ros/company/setup. Bash, run the source install/local\_setup bash, run ros2 run turtlesim turtlesim\_node. Open another window and repeat the previous steps, replacing the last step with ros2 run catch\_turtle catch\_turtle

**Component's contribution to project goals:**

One down is our file directory, and I'll come back to the components' contributions to the project in order below

1. package.xml   
2. setup.cfg   
3. setup.py   
4. <catch\_turtle>/   
 ├── \_\_init\_\_.py  
 ├── catch\_turtle.py

1. The package.xml file is a metadata file for ROS2 packages, describing package information, dependencies, maintainers, etc. It contains the packages we need for our project. It plays a central role in the build system and dependency management, ensuring that packages can find other packages and dependencies they need.
2. the setup.cfg file contains configuration options for building and installing packages to ensure that Python nodes are properly installed and recognised
3. setup.py This is the Python setuptools build script that defines how to build and install Python modules, packages and services. Package dependencies are specified to ensure that all necessary dependencies are installed during the build process.
4. **catch\_turtle.py** This environment This code implements a ROS2 node TurtleController which controls the turtles in the Turtlesim environment to perform the following functions:
   1. **Generate turtles**: generates a new turtle every 3 seconds at a random location by calling the spawn\_turtle function.
   2. **control\_turtles**: the control\_turtles function is responsible for controlling the main turtle (turtle1) to move towards the nearest turtle and capture it. Captured turtles are added to a queue that follows the main turtle.
   3. **Maintain queue**: captured turtles (the captured\_turtles list) will follow the turtle in front of them via the follow\_leader function, keeping the queue coherent.
   4. **clear\_trails**: the service is called every 3 seconds to clear the turtles' trails via the clear\_trails function.
   5. **Subscribe and Publish**: the node subscribes to each turtle's position information (Pose message) and publishes control commands (Twist message) to control the turtle's speed and direction.
   6. **Service calls**: nodes use services to generate new turtles (Spawn service), clear trajectories (Empty service)
   7. **Timers**: ROS2 timers (create\_timer) are used to periodically perform the operations of generating turtles, controlling turtles and clearing trajectories.