In phase 1, we implemented the evolutionary strategy algorithm and configured the environment. To build the ros2 environment in Ubuntu 22.02, we used Docker. Below is the dockerfile necessary to create the container:

# Use maze example docker image as the base image

FROM haiderabbasi333/ros2-pathplanning-course:1

# Add ROS 2 Foxy repositories and GPG key

RUN apt-get update && apt-get install -y curl gnupg2 lsb-release \

&& curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | apt-key add - \

&& sh -c 'echo "deb [trusted=yes] http://packages.ros.org/ros2/ubuntu $(lsb\_release -cs) main" >

/etc/apt/sources.list.d/ros2-latest.list' \

&& apt-get update# Install required ROS 2 Foxy packages

RUN apt-get update && apt-get install -y \

ros-foxy-turtlebot3-gazebo \

ros-foxy-turtlebot3-teleop \

vim

# Set environment variable for TurtleBot3 model

ENV TURTLEBOT3\_MODEL=burger

# Source the ROS2 setup file every time the container runs

RUN echo "source /opt/ros/foxy/setup.bash" >> ~/.bashrc

RUN echo "export TURTLEBOT3\_MODEL=burger" >> ~/.bashrc

# Set the working directory inside the container

WORKDIR /root

# Default command to run the simulation when the container starts

CMD ["ros2", "launch", "turtlebot3\_gazebo", "turtlebot3\_world.launch.py"]



Then, run the following command to start the container

docker build -t my\_turtle\_bot:v1 Dockerfile

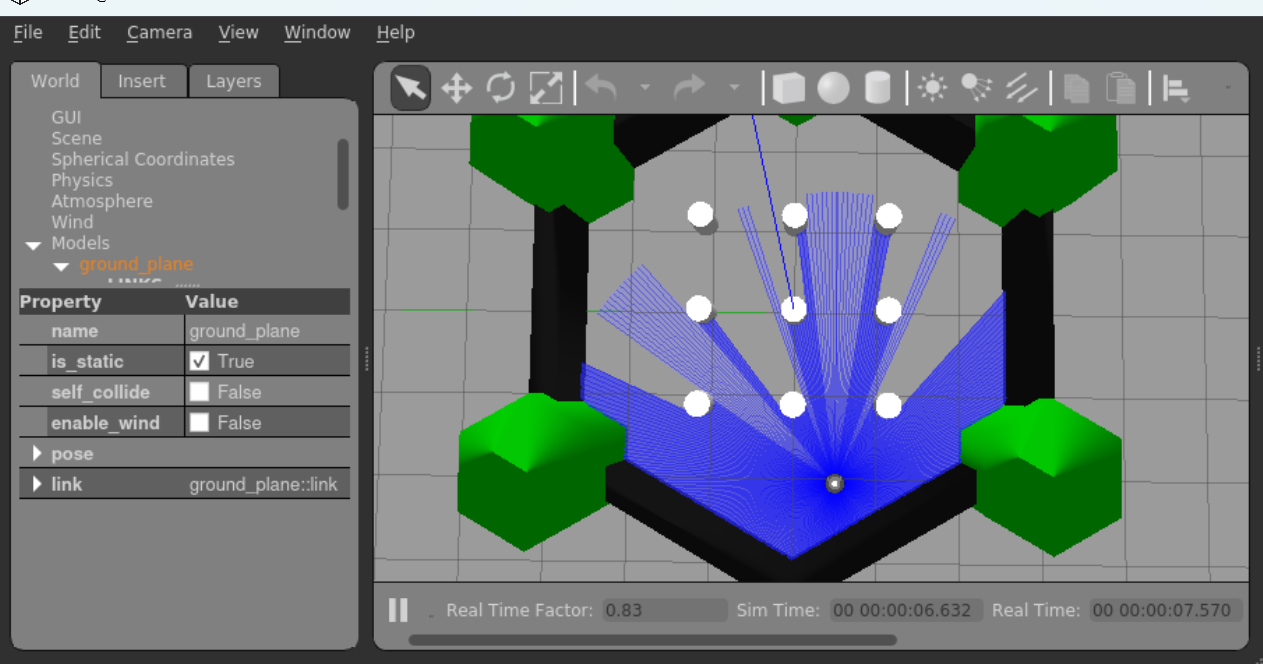


docker run -it --name r2\_turtle\_container -e DISPLAY=host.docker.internal:0.0

-e LIBGL\_ALWAYS\_INDIRECT=0 my\_turtle\_bot:v1 bash



Screenshot of environment is attached, we used gazebo world environment:



Up to the point, we have the scratch code and are still working on debugging. And the main functions and their purposes are illustrated below with screenshots.

The initialize\_population function sets up the initial generation by extracting from uniform distribution.

 def initialize\_population(self):

return [

{

'left\_speed': random.uniform(0, 1),

'right\_speed': random.uniform(0, 1)

} for \_ in range(self.population\_size)

]



The laser\_callback function gets the sensor output from the robot and calls the update\_behavior function. This function is used to divide the output from the sensor into 3 parts: left, front and right as the proposal says. And these three parts will decide which behavior the robot will take: turn left, right or go straight.

The evaluate\_fitness function calculates the fitness of each child in a generation. It mainly follows the policy we introduced in the proposal: heavy penalty on colliding with obstacles, and rewards depending on the time spent on walking and turning.

def evaluate\_fitness(self, front, left, right):

if front < 0.2:

self.scores[self.current\_individual] -= 10

else:

self.scores[self.current\_individual] += 1

if self.current\_individual >= self.population\_size - 1:

self.evolve\_population()

else:

self.current\_individual += 1



The evolve\_population function is the evolve process. We first pick the children who get the highest fitness calculated by evaluate\_fitness function. Then do the crossover on two random-pick children and append the result into a new generation list. And after implementing mutate on each child, the new generation list will be the next generation.

def evolve\_population(self):

top\_performers = np.argsort(self.scores)[-self.population\_size // 2:]

new\_population = [self.population[i] for i in top\_performers]

while len(new\_population) < self.population\_size:

parent1 = random.choice(new\_population)

parent2 = random.choice(new\_population)

child = self.crossover(parent1, parent2)

new\_population.append(child)

self.population = [self.mutate(ind) for ind in new\_population]

self.scores = [0 for \_ in range(self.population\_size)]

self.current\_individual = 0

self.current\_generation += 1



The crossover function shows the crossover step in the algorithm. We now simply just take the average of two parents’ speed, and in the further study we will try some more rigorous formula.

The mutate function shows the mutate step. We add the mutate in a form of gaussian distribution and set the threshold of each individual’s speed.

def crossover(self, parent1, parent2):

return {

'left\_speed': (parent1['left\_speed'] + parent2['left\_speed']) / 2,

'right\_speed': (parent1['right\_speed'] + parent2['right\_speed']) / 2

}

def mutate(self, individual):

if random.random() < self.mutation\_rate:

individual['left\_speed'] += random.gauss(0, 0.1)

individual['right\_speed'] += random.gauss(0, 0.1)

individual['left\_speed'] = min(max(individual['left\_speed'], 0), 1)

individual['right\_speed'] = min(max(individual['right\_speed'], 0), 1)

return individual



When testing the code, we saw that the robot went straight as no obstacle detected but collided into walls and lost control afterwards. We are still trying to figure out the reason.