

OpenAI Gym Environment

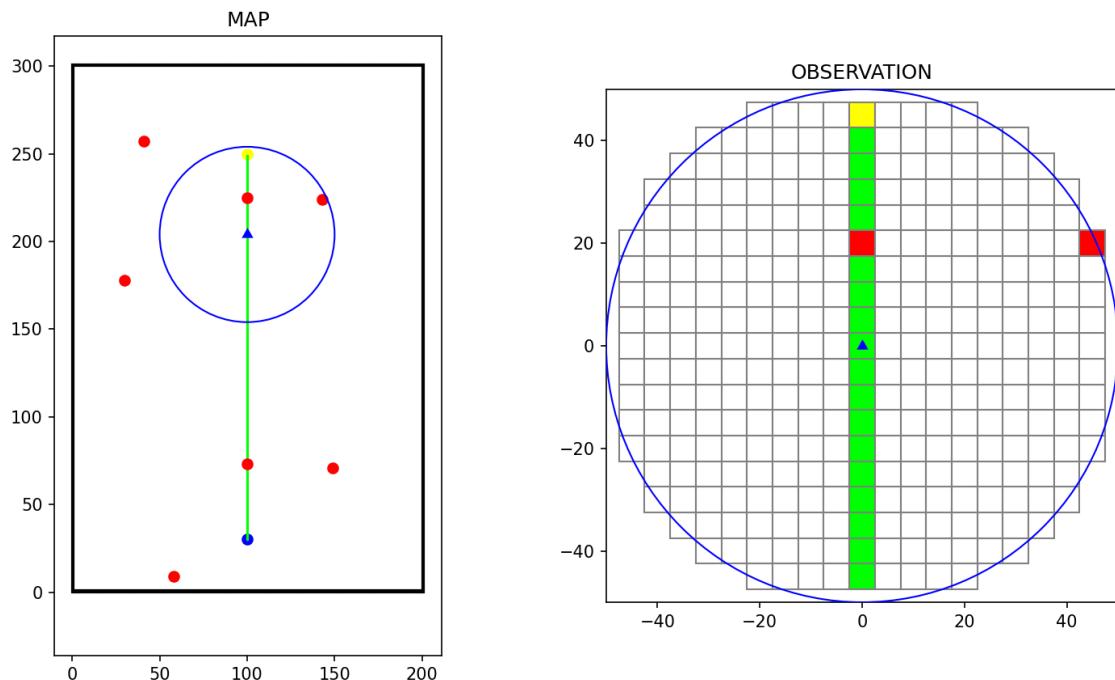
Environment overview

This Gym environment was created to train a Deep Reinforcement Learning (DRL) agent, an Autonomous Surface Vessel (ASV), to follow a path from the start point to end point, while avoiding obstacles along the way. The goal of the agent is to learn how to temporarily avoid obstacles (path replanning) and follow a global path (trajectory tracking) as closely as possible. For static obstacles, the agent will need to find the optimal path to avoid efficiently, and for dynamic obstacles, the agent will need to avoid moving obstacles following the marine traffic rules, The International Regulations for Preventing Collisions at Sea (COLREGs).

The following features are included in the environment:

- Initially, the 2D will have a boundary to prevent it from going off the map, a global path to follow, multiple obstacles and a goal point.
- Static obstacles are placed randomly around the map, with 2 random obstacles generated along the main path.
- Around the agent/ASV, a circle is generated to represent the observation range. The observation circle moves along with the agent and is updated at each timestep.
- The agent is represented as a triangular icon, with varying heading angle.

There will be 2 maps plotted on each side for observation: global map and local map. The global map consists of all the elements, along with the moving observation circle, this is the map for the user to view the behaviour of the agent. The local map only consists of the area inside the observation circle. At the beginning of each timestep, the local map is divided into grids, each grid has a state (obstacle / free space / path / goal point) and the agent is mapped in the middle. The state of each grid is updated continuously in relation to the position of the observation area in the global map.



This simulation of the ASV does not involve control parameters of maritime vehicles, such as thruster or propulsion. The agent is a modeless ASV, where it only moves by adjusting speed and/or heading angle.

1. Install and import dependencies

```
import numpy as np
import gymnasium as gym
from gymnasium import spaces
import matplotlib.pyplot as plt
```

2. Create Configuration File

```
# Define colours
BLACK = (0, 0, 0)
WHITE = (1, 1, 1)
RED = (1, 0, 0)
GREEN = (0, 1, 0)
YELLOW = (1, 1, 0)
BLUE = (0, 0, 1)
```

Firstly, we define colours to quickly change how we want to visualize the elements, such as red for obstacles, green for path, etc.

```
# Define map dimensions and start/goal points, number of static obstacles
WIDTH = 100
HEIGHT = 150
START = (50, 20)
GOAL = (50, 120)
NUM_STATIC_OBS = 5
```

Then we define the map dimension, location of the start and goal points (x, y) accordingly, and number of static obstacles around the map.

```
# Define observation radius and grid size
RADIUS = 100
SQUARE_SIZE = 10
```

Next, the parameters of the observation and grid size are defined. For the grids to be evenly divided in the circle, the diameter (or 2x radius) divided by square size must result in an even number.

```
# Define initial heading angle, turn rate and number of steps
INITIAL_HEADING = 90
TURN_RATE = 5
SPEED = 1
```

The initial heading angle is set as 90°, pointing upward towards the positive y-axis. The agent can move by adjusting heading angle (\pm turn rate) and speed. The turn rate and speed should be adjusted based on the dynamic of a real ASV.

```
# Define states
FREE_STATE = 0          # free space
PATH_STATE = 1           # path
COLLISION_STATE = 2      # obstacle or border
GOAL_STATE = 3            # goal point
```

Finally, the states are defined from 0 to 3 for the DRL agent to process the state at each timestep. Currently it has 4 states: on free space / on path / collision with boundary or obstacle / reached goal.

3. Initialize Gym Environment

```
class ASVEnv(gym.Env):
    metadata = {"render_modes": ["human"]}
    def __init__(self, render_mode = "human"):
        super(ASVEnv, self).__init__()
        self.render_mode = render_mode
```

The next step is to define the Gym environment class. Defining render mode is necessary for rendering a gym environment.

```
    self.width = WIDTH
    self.height = HEIGHT
    self.heading = INITIAL_HEADING
    self.turn_rate = TURN_RATE
    self.speed = SPEED
    self.start = START
    self.goal = GOAL
    self.radius = RADIUS
    self.grid_size = SQUARE_SIZE
    self.center_point = (0,0)
    self.step_taken = []
```

The map dimension, initial heading angle, turn rate and speed, start and goal coordinates, radius of the observation circle, grid size are initialized by taking the values from configuration.

The center point is set as (0, 0) to always set the agent in the middle of the local map.

The step_taken list will append every step the agent has taken to display the result.

```
    self.path = self.generate_path(self.start, self.goal)
    self.boundary = self.generate_border(self.width, self.height)
    self.goal_point = self.generate_goal(self.goal)
```

In every episode, the stationary elements are path, boundary and goal point. They do not change so we can initialize it in this function.

```
# Define action space and observation space
self.action_space = spaces.Discrete(3)
self.observation_space = spaces.Box(low=0, high=3, shape=(313,), dtype=np.int32)
```

The action space is defined as discrete actions, with 3 possible actions: turn left or turn right (by adjusting turn rate) or go straight (keep the current heading angle).

The observation space is defined by the range of possible states: lowest is 0 to highest is 3, which results in 4 different states can exist. The shape is a 1D array, each element is a grid consists of the coordinates and state, the value is the total number of grids in the local map, which depends on how the observation radius and square size are defined in the configuration. This will be explained further in the generate_grid and fill_grid functions.

4. Helper Functions

```
# Create a function that sets the priority of each state in case they overlap:  
# obstacle/collision > goal point > path > free space  
def get_priority_state(self, current_state, new_state):  
    if new_state == COLLISION_STATE:  
        return COLLISION_STATE  
    elif new_state == GOAL_STATE and current_state != COLLISION_STATE:  
        return GOAL_STATE  
    elif new_state == PATH_STATE and current_state not in (COLLISION_STATE, GOAL_STATE):  
        return PATH_STATE  
    elif current_state not in (COLLISION_STATE, GOAL_STATE, PATH_STATE):  
        return FREE_STATE  
    return current_state
```

The `get_priority_state` function is used for deciding the state of each grid in case there are elements at the same location (since the obstacles are generated randomly). It takes 2 arguments: the current state and the new state.

- If the new state is a boundary or an obstacle, that state is defined as `COLLISION_STATE`. This is to avoid the case where the obstacle is on the path, the agent will need to prioritize avoiding the obstacle, rather than staying on the path with a risk of collision.
- Else if the new state is a goal point and the current state is not an obstacle, the state is set as `GOAL_STATE`. For the case where the obstacle is generated near the edge of the goal area, if there is not obstacle, it can stay as goal point. However, if the path overlaps the goal, it will still be recognised as the goal.
- Else if the new state is a path, and the current state is either an obstacle or a goal, the state is set as a path. As stated above, both obstacles and goal point are prioritized over the path.
- Else if the current state is not a path or boundary/obstacles or goal, it is defined as free space, the lowest level of priority.

By calling the function, it will return a single value of the current state. In summary, the order or priority is: `COLLISION_STATE > GOAL_STATE > PATH_STATE > FREE_STATE`

```
# Create a function that generate grid coordinates (x,y) from global map  
def generate_grid(self, radius, square_size, center):  
    x = np.arange(-radius + square_size, radius, square_size)  
    y = np.arange(-radius + square_size, radius, square_size)  
    grid = []  
    for i in x:  
        for j in y:  
            if np.sqrt(i ** 2 + j ** 2) <= radius:  
                grid.append((center[0] + i, center[1] + j))  
    return grid
```

This function helps generating the collision grid inside the observation radius and is updated at every timestep. It takes 3 arguments: radius of the observation area, grid size, and location of the center point. Each grid coordinate (x, y) is generated from $(-\text{radius} + \text{grid size})$ to (radius) , with a step size equal to grid size. For example, with a radius of 100 and grid size of 10, we would have

$$x = (-90, -80, \dots, 80, 90) \quad y = (-90, -80, \dots, 80, 90)$$

Then, each (x, y) coordinate is iterated and checked if it is within the observation radius. If less than or equal to the radius, the grid coordinate is appended to the grid list.

$$\text{If } (x = 90, y = 90) \rightarrow \sqrt{90^2 + 90^2} = 127.28 > 100 \rightarrow \text{not append to grid}$$

$$\text{If } (x = 60, y = 50) \rightarrow \sqrt{60^2 + 50^2} = 78.1 < 100 \rightarrow \text{append to grid}$$

By applying this method, the grid points will be generated only within the observation area.

```
# Create a function that converts each point from the global map to a grid coordinate
def closest_multiple(self, n, square_size):
    return int((n + square_size / 2) // square_size) * square_size
```

The *closest_multiple* function converts a coordinate to the nearest multiple of the grid size. It takes a single coordinate value and the grid size as input arguments. For example, if there is an obstacle on the global map at point (32, 25), it will appear on the local map at the grid coordinate of (30, 40) and marked as COLLISION_STATE.

$$x = \text{int}((32 + 5) // 10) * 10 = 40$$

$$y = \text{int}((25 + 5) // 10) * 10 = 30$$

Even though at coordinate (33, 25) there is no obstacles, when convert to grid coordinate in the local map, it is still considered as coordinate (30, 40) and defined as an obstacle presented. Therefore, the smaller the grid size, the more accurate the local map is, but the trade-off is higher computational cost.

```
# Create a function to generate a dictionary, storing the grid coordinates and state
def fill_grid(self, objects, grid_size):
    grid_dict = {}
    for obj in objects:
        m = obj['x']
        n = obj['y']
        state = obj['state']

        m = self.closest_multiple(m, grid_size)
        n = self.closest_multiple(n, grid_size)

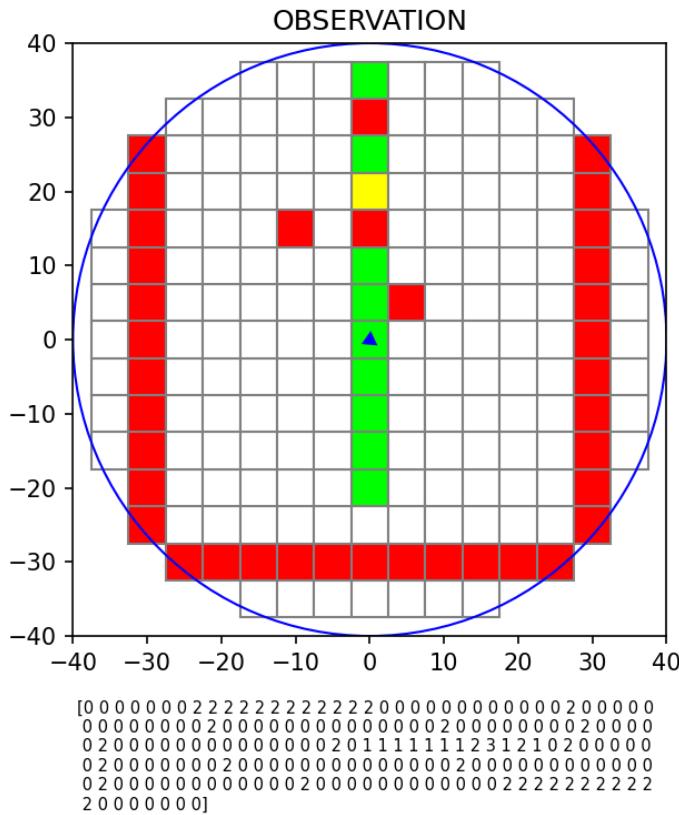
        if (m, n) not in grid_dict:
            grid_dict[(m, n)] = FREE_STATE

        grid_dict[(m, n)] = self.get_priority_state(grid_dict[(m, n)], state)
    return grid_dict
```

The *fill_grid* function takes objects and grid size as input arguments. The purpose of this function is to create a dictionary of (x, y, state) for each coordinate on the map.

Firstly, it extracts each object available in the global map, whether it's a path, goal point, or obstacles/boundary, along with its state. Each (x, y) coordinate then goes through the *closest_multiple* function to be converted into grid coordinates. For each grid coordinate not listed in the previous step, it is defined as a free space, otherwise, each grid is assigned to its state. Every grid coordinate are then stored in the dictionary *grid_dict*. Finally, the dictionary is passed through the *get_priority_state*

function to resolve the overlapping state of each grid (if a grid is both PATH_STATE and COLLISION_STATE, assign it as COLLISION_STATE).



The next three functions are additional helper functions for calculating the rewards in the next section.

```
def calculate_distance_to_path(self, position):
    path_x = [point['x'] for point in self.path]
    path_y = [point['y'] for point in self.path]
    min_distance = float('inf')
    for px, py in zip(path_x, path_y):
        distance = np.sqrt((position[0] - px) ** 2 + (position[1] - py) ** 2)
        if distance < min_distance:
            min_distance = distance
    return min_distance
```

This function calculates the distance between the current position to any point on the path. The purpose is to make the agent acknowledge how far it is from the path and try to get back as close as possible to the path. To calculate the distance, we use Euclidean distance between 2 points.

$$\min distance = \min \sqrt{(x - px)^2 + (y - py)^2}$$

Where (x, y) is the current position of the agent, and (px, py) is the closest point on the path.

```
def calculate_distance_to_goal(self, position):
    goal_x = [point['x'] for point in self.goal_point]
    goal_y = [point['y'] for point in self.goal_point]
```

```

min_distance = float('inf')
for px, py in zip(goal_x, goal_y):
    distance = np.sqrt((position[0] - px) ** 2 + (position[1] - py) ** 2)
    if distance < min_distance:
        min_distance = distance
return min_distance

```

Upon calling, this function returns the distance between current position and goal position. Similar to the previous function, it also uses Euclidean distance to calculate and update the distance between agent's position and goal point.

```

def calculate_distance_to_nearest_obstacle(self, position):
    x, y = position
    min_distance = float('inf')

    for (obstacle_x, obstacle_y), state in self.grid_dict.items():
        if state == COLLISION_STATE:
            distance = np.sqrt((x - obstacle_x)**2 + (y - obstacle_y)**2)
            if distance < min_distance:
                min_distance = distance
    return min_distance

```

The final helper function calculates the distance between agent's position and the nearest obstacle. The purpose of this function is to prevent the agent from approaching too close to the obstacle before avoiding it.

5. Map Generation

These are the main functions taking part in structuring the global map.

```

# Create a function to generate borders around the map
def generate_border(self, map_width, map_height):
    boundary = []
    for x in range(0, map_width + 1):
        boundary.append({'x': x, 'y': 0, 'state': COLLISION_STATE})
        boundary.append({'x': x, 'y': map_height, 'state': COLLISION_STATE})
    for y in range(0, map_height + 1):
        boundary.append({'x': 0, 'y': y, 'state': COLLISION_STATE})
        boundary.append({'x': map_width, 'y': y, 'state': COLLISION_STATE})
    return boundary

```

Firstly, we create a boundary enclosing the map. It includes points that connect the coordinates of the four corners: (0, 0), (0, map_width), (0, map_height), (map_width, map_height). The coordinates are then appended to *boundary*.

```

# Function that generate a path line (list/array of points)
def generate_path(self, start_point, goal_point):
    path = []
    num_points = goal_point[1] - start_point[1]      # straight vertical line
    for i in range(num_points):
        y = start_point[1] + i

```

```

    path.append({'x': start_point[0], 'y': y, 'state': PATH_STATE})
return path

```

The path is a straight vertical line connecting the start point and goal point. Therefore, we iterate through all the y-axis coordinates between start and goal points, while they share the same x-coordinate, we don't have to iterate through the x-axis. The path is then appended to *path*.

```

def generate_goal(self, goal_point):
    goal = []
    goal.append({'x': goal_point[0], 'y': goal_point[1], 'state': GOAL_STATE})
    return goal

```

For generating the goal point, we simply take the (x, y) coordinate of the goal and append it to *goal*.

```

# Create a function to generate static obstacles
def generate_static_obstacles(self, num_obs):
    obstacles = []
    # Generate random obstacles around the map
    for _ in range(num_obs):
        x = np.random.randint(0, self.width)
        y = np.random.randint(0, self.height)
        obstacles.append({'x': x, 'y': y, 'state': COLLISION_STATE})
    # Generate 2 random obstacles along the path
    for _ in range(2):
        x = self.start[0]
        y = np.random.randint(self.start[1] + 20, self.goal[1] - 20)
        obstacles.append({'x': x, 'y': y, 'state': COLLISION_STATE})
    return obstacles

```

When calling *generate_static_obstacles* function, we need to define the *num_obs* variable, which will generate a number of obstacles randomly within the map. However, since the obstacles are generated randomly, the agent may not encounter any obstacles on the path. So, we generate 2 additional obstacles always presented on the path randomly at each episode so that the agent can learn to avoid. The static obstacles coordinates are then appended to *obstacles*.

Note that after generating boundaries, obstacles, path, or goal point, they are appended to their dictionary with the format (x, y, state) for compatibility with the dictionary *grid_dict*.

6. Gym Environment Functions

These are the compulsory functions for the Gym library to recognize and process.

```

def reset(self, seed=None, options=None):
    self.obstacles = self.generate_static_obstacles(3, self.width, self.height)
    self.objects_environment = self.obstacles + self.path + self.boundary +
                                self.goal_point
    self.grid_dict = self.fill_grid(self.objects_environment, self.grid_size)
    self.grid = self.generate_grid(self.radius, self.grid_size, self.position)

    self.step_count = 0
    self.current_heading = self.heading

```

```

        self.current_speed = self.speed
        self.position = self.start
        self.done = False

    return self.get_observation(), {}

```

The reset function is called at the initialization, as well as at the beginning of each episode. Upon resetting, we generate obstacles, update *object_environment* and *grid_dict* dictionaries, generate grid coordinates. The *step_count*, *current_heading*, *current_speed*, and *current_position* are reset to their initial values. And the session is marked as not done. As required by OpenAI Gym library, the *reset* function returns the current observation *get_observation()*, which we will get to later, and *info*, which we don't have so we pass an empty dictionary.

```

def get_observation(self):
    current_pos = self.position
    new_grid = self.generate_grid(self.radius, self.grid_size, current_pos)
    observation = np.zeros(len(new_grid), dtype = np.int32)
    for idx, (x, y) in enumerate(new_grid):
        state = self.grid_dict.get((self.closest_multiple(x, self.grid_size),
                                    self.closest_multiple(y, self.grid_size)), FREE_STATE)
        observation[idx] = state
    return observation

```

The function *get_observation* returns the observation space at one instance. Firstly, we set the current position of the agent, then generate grid points around the observation area. The observation is initialized as a list of zeroes with the size equal to the number of grids. For each grid coordinate, its state is taken from the dictionary *grid_dict* and updated to the *observation* list. Therefore, we have a list of state for each grid point at one instance.

```

def step(self, action):
    if action == 0:      # go straight
        self.position = (self.position[0] + self.speed *
                          np.cos(np.radians(self.current_heading)),
                          self.position[1] + self.speed *
                          np.sin(np.radians(self.current_heading)))
    elif action == 1:    # turn left
        self.current_heading += self.turn_rate
        self.position = (self.position[0] + self.speed *
                          np.cos(np.radians(self.current_heading)),
                          self.position[1] + self.speed *
                          np.sin(np.radians(self.current_heading)))
    elif action == 2:    # turn right
        self.current_heading -= self.turn_rate
        self.position = (self.position[0] + self.speed *
                          np.cos(np.radians(self.current_heading)),
                          self.position[1] + self.speed *
                          np.sin(np.radians(self.current_heading)))
    self.step_count += 1

```

```

    self.step_taken.append((self.position[0], self.position[1]))
    reward = self.calculate_reward(self.position)
    terminated = self.check_done(self.position)
    observation = self.get_observation()

    return observation, reward, terminated, False, {}

```

The *step* function is a core function in a Gym environment. It is used to apply an action taken by the agent and update the agent's state, then calculate the reward, observation, and termination conditions. The function is called at every timestep during the agent's interaction with the environment.

With velocity/speed, heading angle and turning rate, the agent has 3 possible movements: move straight, turn left, or turn right. The agent's position in (x, y) is updated after each action by the following equation.

$$x_i = x_{i-1} + v \times \cos \theta_i$$

$$y_i = y_{i-1} + v \times \sin \theta_i$$

Where v is velocity and θ is heading angle. The heading angle is updated in each timestep by adding or subtracting the turning rate.

$$\theta_i = \theta_{i-1} \pm \text{turn rate}$$

Here is an example of updating the agent's position, where initially

$$x_{i-1} = 50 \text{ m} \quad y_{i-1} = 20 \text{ m} \quad v = 1 \text{ m/s} \quad \theta_{i-1} = 90^\circ \quad \text{turn rate} = 5^\circ$$

If the agent goes straight, the new position would be (50, 21)

$$\theta_i = \theta_{i-1}$$

$$x_i = 50 + 1 \times \cos 90^\circ = 50$$

$$y_i = 20 + 1 \times \sin 90^\circ = 21$$

If the agent turns left, the new position would be (49.91, 20.99)

$$\theta_i = 90^\circ + 5^\circ = 95^\circ$$

$$x_i = 50 + 1 \times \cos 95^\circ \approx 49.91$$

$$y_i = 20 + 1 \times \sin 95^\circ \approx 20.99$$

If the agent turns right, the new position would be (50.09, 20.99)

$$\theta_i = 90^\circ - 5^\circ = 85^\circ$$

$$x_i = 50 + 1 \times \cos 85^\circ \approx 50.09$$

$$y_i = 20 + 1 \times \sin 85^\circ \approx 20.99$$

After calculating the new position, the reward is calculated, termination condition is checked, and the new observation is updated. We will discuss these functions shortly.

```
def calculate_reward(self, position):
    x, y = position
    state = self.grid_dict.get((self.closest_multiple(x, self.grid_size),
                                self.closest_multiple(y, self.grid_size)), FREE_STATE)
    distance_to_path = self.calculate_distance_to_path(self.position)
    distance_to_goal = self.calculate_distance_to_goal(self.position)
    # Calculate the distance to the nearest obstacle
    nearest_obstacle_distance =
        self.calculate_distance_to_nearest_obstacle(self.position)

    # Set a threshold distance for significant penalty
    danger_zone_threshold = self.grid_size * 2

    reward = 0
    if state == COLLISION_STATE:
        reward -= 500
    elif state == GOAL_STATE:
        reward += 500
    elif state == PATH_STATE:
        reward += (10 - distance_to_goal*0.1)
    elif state == FREE_STATE:
        reward -= (1 + distance_to_path + distance_to_goal*0.1)

    # Add a penalty for being too close to an obstacle
    if nearest_obstacle_distance <= danger_zone_threshold:
        reward -= 1000 / nearest_obstacle_distance
    reward -= distance_to_goal*0.1

    return reward
```

```
def check_done(self, position):
    x, y = position
    state = self.grid_dict.get((self.closest_multiple(x, self.grid_size),
                                self.closest_multiple(y, self.grid_size)), FREE_STATE)

    # If the agent collide with obstacles or boundary
    if state == COLLISION_STATE:
        return True
    # If the agent reached goal
    elif state == GOAL_STATE:
        return True
    # If the total number of steps are 250 or above
    elif self.step_count >= 500:
        return True
    return False
```

```

def render(self, mode="human"):
    if mode == 'human':
        if not hasattr(self, 'fig'):
            self.fig, (self.ax1, self.ax2) = plt.subplots(1, 2, figsize=(12, 8))

            self.ax1.set_aspect('equal')
            self.ax1.set_title('MAP')
            self.ax1.set_xlim(-self.radius, self.width + self.radius)
            self.ax1.set_ylim(-self.radius, self.height + self.radius)

            self.ax2.set_aspect('equal')
            self.ax2.set_title('OBSERVATION')
            self.ax2.set_xlim(-self.radius, self.radius)
            self.ax2.set_ylim(-self.radius, self.radius)

            self.agent_1, = self.ax1.plot([], [], marker='^', color=BLUE)
            self.agent_2, = self.ax2.plot([], [], marker='^', color=BLUE)
            self.observation_horizon1 = plt.Circle(self.start, self.radius,
                                                    color=BLUE, fill=False)
            self.observation_horizon2 = plt.Circle((0, 0), self.radius,
                                                    color=BLUE, fill=False)
            self.ax1.add_patch(self.observation_horizon1)
            self.ax2.add_patch(self.observation_horizon2)

            self.ax1.plot(self.start[0], self.start[1], marker='o', color=BLUE)
            self.ax1.plot(self.goal[0], self.goal[1], marker='o', color=YELLOW)

        for obj in self.boundary:
            boundary_line = plt.Rectangle((obj['x'], obj['y']), 1, 1,
                                          edgecolor=BLACK, facecolor=BLACK)
            self.ax1.add_patch(boundary_line)

        path_x = [point['x'] for point in self.path]
        path_y = [point['y'] for point in self.path]
        self.ax1.plot(path_x, path_y, '--', color=GREEN)

        for obj in self.obstacles:
            self.ax1.plot(obj['x'], obj['y'], marker='o', color=RED)

        self.agent_1.set_data(self.position[0], self.position[1])
        self.agent_1.set_marker((3, 0, self.current_heading - 90))

        self.observation_horizon1.center = self.position

        new_grid = self.generate_grid(self.radius, self.grid_size, self.position)
        for rect in getattr(self, 'grid_patches', []):
            rect.remove()

```

```

        self.grid_patches = []

        for (cx, cy) in new_grid:
            state = self.grid_dict.get((self.closest_multiple(cx, self.grid_size),
                                         self.closest_multiple(cy, self.grid_size)), FREE_STATE)
            color = WHITE
            if state == COLLISION_STATE:
                color = RED
            elif state == PATH_STATE:
                color = GREEN
            elif state == GOAL_STATE:
                color = YELLOW
            rect = plt.Rectangle((cx - self.grid_size / 2 - self.position[0],
                                  cy - self.grid_size / 2 - self.position[1]),
                                 self.grid_size, self.grid_size,
                                 edgecolor='gray', facecolor=color)
            rect.set_zorder(1)
            self.grid_patches.append(rect)
            self.ax2.add_patch(rect)

            self.agent_2.set_data(0, 0)
            self.agent_2.set_marker((3, 0, self.current_heading - 90))
            self.agent_2.set_zorder(3)

            self.observation_horizon2.center = (0, 0)
            self.observation_horizon2.set_zorder(2)

            plt.draw()
            plt.pause(0.01)

```

7. Display Result

```

def display_path(self):
    # Plot the path taken
    fig, ax = plt.subplots(1,1, figsize=(8,8))
    ax.set_aspect("equal")
    ax.set_title("Steps Taken")
    ax.set_xlim(-self.radius, self.width + self.radius)
    ax.set_ylim(-self.radius, self.height + self.radius)
    ax.plot(self.start[0], self.start[1], marker='o', color=BLUE)
    ax.plot(self.goal[0], self.goal[1], marker='o', color=YELLOW)
    for obj in self.boundary:
        boundary_line = plt.Rectangle((obj['x'], obj['y']), 1, 1,
                                      edgecolor=BLACK, facecolor=BLACK)
        ax.add_patch(boundary_line)
    path_x = [point['x'] for point in self.path]

```

```

    path_y = [point['y'] for point in self.path]
    ax.plot(path_x, path_y, '-', color=GREEN)
    for obj in self.obstacles:
        ax.plot(obj['x'], obj['y'], marker='o', color=RED)
    ax.plot(self.position[0], self.position[1], marker='^', color=BLUE)
    step_x = [point[0] for point in self.step_taken]
    step_y = [point[1] for point in self.step_taken]
    ax.plot(step_x, step_y, '-', color=BLUE)
plt.show()

```

8. Test Environment

To check if the environment works as intended, we can test it by letting the agent executing random actions.

```

# Test the environment with random actions
if __name__ == '__main__':
    env = ASVEnv()
    obs = env.reset()

    print("Observation Space Shape", env.observation_space.shape)
    print("Sample observation", len(env.observation_space.sample()))

    print("Action Space Shape", env.action_space.n)
    print("Action Space Sample", env.action_space.sample())

    for _ in range(100): # Run for 100 steps or until done
        action = env.action_space.sample() # Take a random action
        obs, reward, done, truncated, info = env.step(action)
        env.render()
        if done:
            break

    env.display_path()
    env.close()

```

The first step is to assign an object to the environment class *ASVEnv*.

9. Train the Agent

10. Evaluate the Agent

11. Test the Agent

