OpenAI Gym Environment

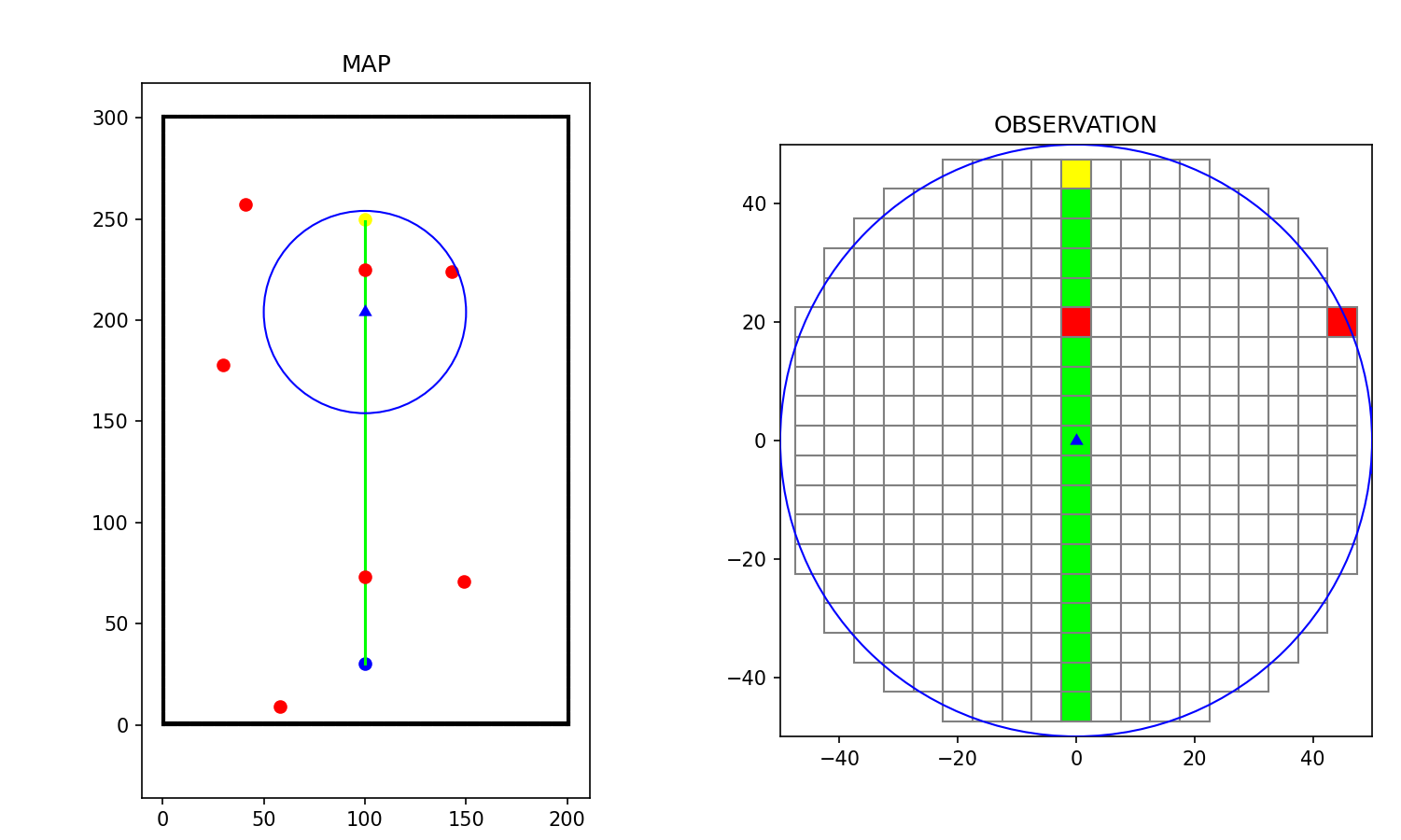
**Environment overview**

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 much 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

1. 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 90o, pointing upward towards the positive y-axis. The agent can move by adjusting heading angle ( 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.

1. 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.

1. 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

    # Create a function that converts each point from the global map to a grid coordinate

    def closest\_multiple(self, n, mult):

        return int((n + mult / 2) // mult) \* mult

# 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

1. 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

   # 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

    def generate\_goal(self, goal\_point):

        goal = []

        goal.append({'x': goal\_point[0], 'y': goal\_point[1], 'state': GOAL\_STATE})

        return goal

   # Create a function to generate static obstacles

    def generate\_static\_obstacles(self, num\_obs, map\_width, map\_height):

        obstacles = []

        # Generate random obstacles around the map

        for \_ in range(num\_obs):

            x = np.random.randint(0, map\_width)

            y = np.random.randint(0, map\_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

1. 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.step\_count = 0

        self.current\_heading = self.heading

        self.current\_speed = self.speed

        self.position = self.start

        self.done = False

        self.grid = self.generate\_grid(self.radius, self.grid\_size, self.position)

        return self.get\_observation(), {}

    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

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, {}

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

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

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

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)

1. 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()

1. Test Environment

# 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()

1. Train the Agent
2. Evaluate the Agent
3. Test the Agent