American University of Sharjah

College of Engineering

Department of Computer Science and Engineering

Spring 2024

CMP 49410 - Intelligent Autonomous Robotics

Project 1

Submitted By:

Ahmad Alsaleh - @00093749

& Ahmed Alabd Aljabar - @00092885

& Omar Ibrahim - @00093225

& Yousef Irshaid - @00093447

Submitted To: Dr. Michel Pasquier

Submission Date: 17th March 2024

Question 1

a) In order to design the potential field model that would attract the robot towards the goal, we have utilized the GPS sensor and IMU sensor. Since the goal's coordinates are always known, which in our case are (1.136, 1.136), the robot's position from the GPS and the yaw (i.e., the orientation of the robot) from the IMU were used. The heading angle is thereby calculated by subtracting the coordinates of the goal from the robot's position and applying that to a tan inverse function that would calculate the angle between the robot's heading and the goal in radians.

```
heading_vector = self.goal - position
return np.arctan2(heading_vector[1], heading_vector[0])
```

Based on the heading angle and the robot's orientation, a proportional control output is calculated. This is done by first finding the error by subtracting the target angle (heading angle) from the current angle (orientation). Then, the error is adjusted by taking into account the circular nature of angles and converting the error to the range [-pi, pi]. This is done by taking the sine and cosine of the error and then using the tan inverse function to calculate the angle in the correct quadrant. Then, the resulting error is multiplied with a parameter, Kp, that would control the strength of the output.

```
error = target_angle - current_angle
# take into account the circular nature of angles by converting the error
to the range [-pi, pi]
error = np.arctan2(np.sin(error), np.cos(error))
# Calculate the proportional control output
return Kp * error
```

The proportional control output, then, is subtracted from the left motor's speed and added to the right motor's speed. If the robot needs to turn left, for example, the control output would be positive and hence the left motor's speed would be less than the right motor's speed. Conversely, if the robot needs to turn right, the control output would be negative which would increase the left speed and decrease the right speed. Note that post-processing of the speeds' values is done using the np.clip function to ensure that it is

within the range of the minimum and maximum speed, respectively. With raw speed we can control how fast the motors rotate, but in this simple case it is set to maximum speed.

```
left_speed = raw_speed - control_output
left_speed = np.clip(left_speed, -Controller.MAX_SPEED,
Controller.MAX_SPEED)

right_speed = raw_speed + control_output
right_speed = np.clip(right_speed, -Controller.MAX_SPEED,
Controller.MAX_SPEED)
```

Additionally, the robot is set to stop when it reaches the proximity of the goal, with a certain threshold defining how close the robot has to be before stopping.

```
position = self.get_gps_position()
distance_to_goal = np.linalg.norm(self.goal - position)
if distance_to_goal < self.distance_threshold:
     return 0, 0
```

Full Code:

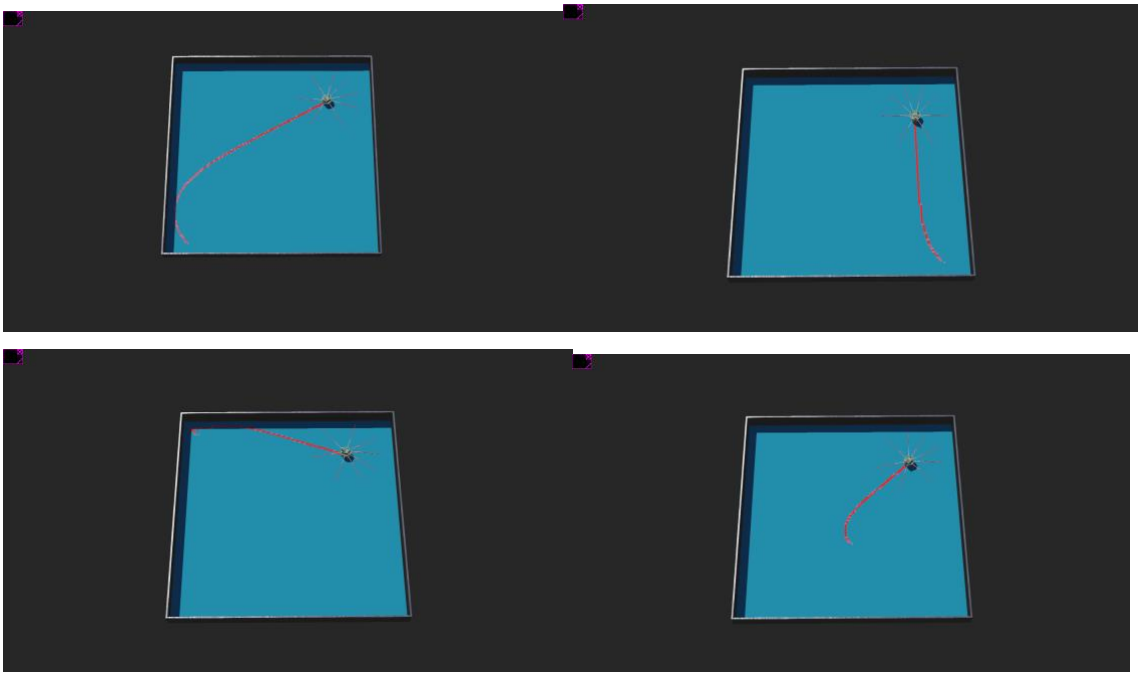
```
from controller import Robot
import numpy as np


class Controller(Robot):
    MAX_SPEED = 6

    def __init__(self, goal: np.ndarray, distance_threshold: float = 0.05) -> None:
        super().__init__()

        self.goal = goal
        self.distance_threshold = distance_threshold

        # initialize the devices of the robot

        self.timeStep = int(self.getBasicTimeStep())

        self.pen = self.getDevice("pen")

        self.gps = self.getDevice("gps")
        self.gps.enable(self.timeStep)
```

```
        self.imu = self.getDevice("IMU")
        self.imu.enable(self.timeStep)
        self.distance_sensors = [self.getDevice(f'ds{i}') for i in range(8)]

        for ds in self.distance_sensors:
            ds.enable(self.timeStep)

        self.left_motor = self.getDevice("left wheel motor")
        self.right_motor = self.getDevice("right wheel motor")
        for motor in [self.left_motor, self.right_motor]:
            motor.setPosition(float("inf"))
            motor.setVelocity(0.0)

    def get_gps_position(self) -> np.ndarray:
        """Returns the (x, y) position of the robot from the GPS device."""
        return np.array(self.gps.getValues())[:2]

    def get_orientation(self) -> np.float64:
        """Returns the yaw angle of the robot in radians"""
        _, _, yaw = self.imu.getRollPitchYaw()
        return np.float64(yaw)

    def get_heading(self, position: np.ndarray) -> np.float64:
        """Returns the angle between the robot's heading and the goal in radians"""
        heading_vector = self.goal - position
        return np.arctan2(heading_vector[1], heading_vector[0])

    def get_proportional_control(self, target_angle, current_angle, Kp=0.9):
        """Returns the proportional control output for a given target and current value."""
        error = target_angle - current_angle

        # take into account the circular nature of angles by converting the error to the
range [-pi, pi]
        error = np.arctan2(np.sin(error), np.cos(error))

        # Calculate the proportional control output
        return Kp * error

    def map(self, value, from_min, from_max, to_min, to_max):
        """Maps a value from the range [`from_min`, `from_max`] to the range [`to_min`,
`to_max`]."""
        return (value - from_min) * (to_max - to_min) / (from_max - from_min) + to_min

    def get_distances(self):
        return np.array([ds.getValue() for ds in self.distance_sensors])

    def get_motors_speeds(
        self, heading_angle: float, orientation: float
```

```
    ) -> tuple[float, float]:
        """Computes the left and right motor speeds based on the heading angle and the
robot's orientation."""

        # stop if too close to goal
        position = self.get_gps_position()
        distance_to_goal = np.linalg.norm(self.goal - position)
        if distance_to_goal < self.distance_threshold:
            return 0, 0

        raw_speed = Controller.MAX_SPEED
        control_output = self.get_proportional_control(heading_angle, orientation)

        left_speed = raw_speed - control_output
        left_speed = np.clip(left_speed, -Controller.MAX_SPEED, Controller.MAX_SPEED)

        right_speed = raw_speed + control_output
        right_speed = np.clip(right_speed, -Controller.MAX_SPEED, Controller.MAX_SPEED)

        return left_speed, right_speed

    def run(self) -> None:
        while self.step(self.timeStep) != -1:
            position = self.get_gps_position()
            heading_angle = self.get_heading(position)
            orientation = self.get_orientation()

            left_speed, right_speed = self.get_motors_speeds(heading_angle, orientation)
            self.left_motor.setVelocity(left_speed)
            self.right_motor.setVelocity(right_speed)

            print(
                f"Position: {position}\tOrientation: {orientation}\tHeading angle:
{heading_angle}"
            )

controller = Controller(goal=np.array([1.136, 1.136]))
controller.run()
```

b) Experiments with the APF designed in (a) for the map in Figure 3 are shown below. The Kp was refined from 0.25 (initial) to 0.9 as the control output was not strong enough to simulate the robot turning. With that change, the robot finds its way to the goal from any position on the map.

While the robot always stops near the goal, the precision of the location of the goal is not perfect. This is because a certain threshold is used to determine the proximity in which the robot has to stop. This parameter has been modified (experimented with) and it was found that if it is too small it would make the robot overshoot the goal, causing the robot to go in infinite circles around the goal. The optimal threshold was empirically found to be 0.05. Additionally, it is clear (in the third image below), that the robot has no sense of obstacles with it hitting the arena wall on its way to the goal. The robot needs the repulsive potential field behavior to be modeled for it to avoid hitting obstacles such as the wall.

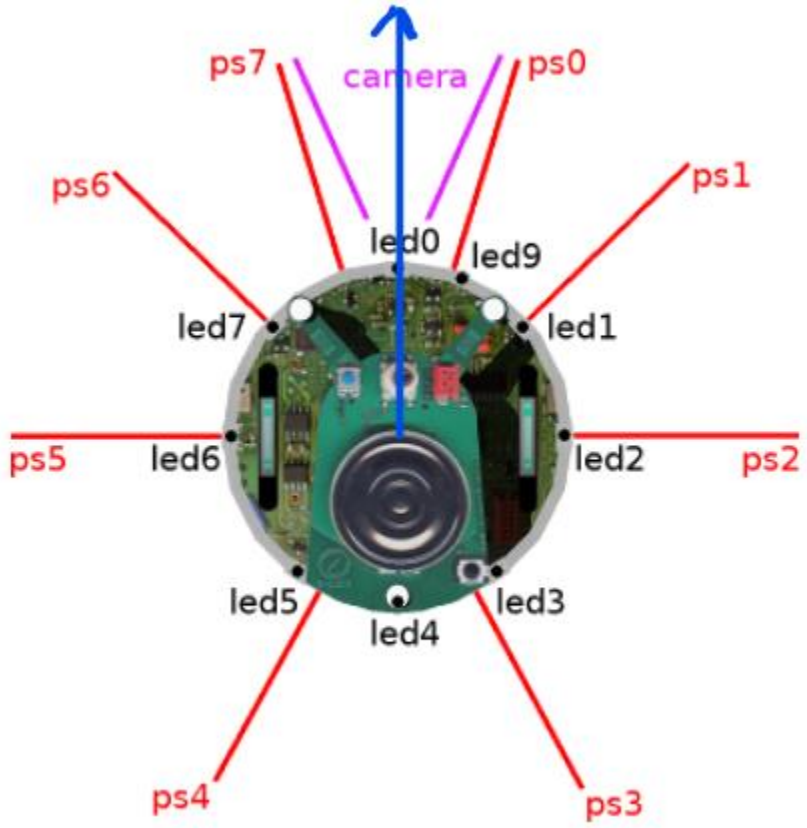


Question 2

a, b, c) To design an APF model to avoid collision with the arena walls, barrels, and any other obstacle, readings from the 8 distance sensors on the robot are used to calculate a repulsive force vector moving away from the walls to prevent collision. To elaborate, the sensors on the robot are placed in known fixed orientations (Webots documentation: GCTronic' e-puck) which can be used to convert the sensor's relative angle to an absolute angle using the following mapping:

```
RELATIVE_ANGLES_OF_DISTANCE_SENSORS = np.array(
        [
```

```
            1.27 - np.pi / 2,
            0.77 - np.pi / 2,
            -np.pi / 2,
            -(2 * np.pi - 5.21) - np.pi / 2,
            4.21 - np.pi / 2,
            np.pi / 2,
            2.37 - np.pi / 2,
            1.87 - np.pi / 2,
        ]
    )
```



These angles are the angles of the distance sensor placements relative to the robot's orientation (the blue vector in the figure above). This was done so that by doing simple addition we get the angles of the distance sensors at any given orientation. By doing so, we can get the angles of the repulsive vectors pointing ***towards*** the wall/obstacle for each sensor. We can also obtain the distance to the wall or the obstacle using the distance sensor. Therefore, given the angle of the distances to obstacles/walls and the magnitude of the distance, we can find the (x,y) components of the repulsive vector with:

$(x, y) = (D \cdot cos(\theta), D \cdot sin(\theta))$ where D is the distance.

```
def get_vector_components(self, magnitude: float, angle: float) -> np.ndarray:
        """Calculate the x and y components of the vector"""
        x = magnitude * np.cos(angle)
        y = magnitude * np.sin(angle)
        return np.array([x, y])
```

The angles of the repulsive forces are calculated as follows:

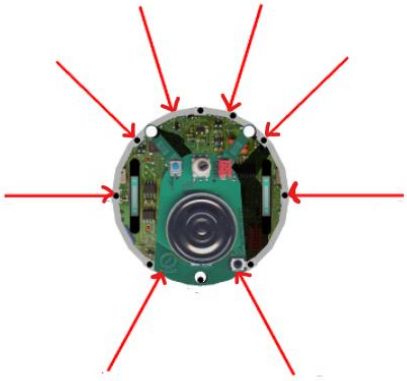
```
sensors_angles = (
            self.get_robot_angle() + Controller.RELATIVE_ANGLES_OF_DISTANCE_SENSORS
        )
```

After calculating the angles, the magnitude of the distance returned by the sensors from the lookup table $(0 \rightarrow 1000)$ is scaled to $(\text{max\_magnitude} \rightarrow 0)$, where max_magnitude is the max magnitude of an individual repulsive vector, after experimentation max_magnitude of 4 seemed to work as expected:

```
distances = self.map(self.get_distances(), 0, 1000, max_magnitude, 0)
```

Following that, the inverted (repulsive) sum of the vectors is returned to be added with the other forces, i.e. attraction to the goal in this case.

```
return -np.sum(
    [
        self.get_vector_components(distance, angle)
        for distance, angle in zip(distances, sensors_angles)
    ],
    axis=0,
)
```



What we effectively compute at the end of this function is the sum of the red vectors in the figure above. It should be noted that the dropoff function is linear drop off since we simply scale down the distance reading from the distance sensors, which is already linear from 0 → 1000. It will look like the following:

*Dropoff = -0.004*distance_reading + 4*

where 4 is max_magnitude from before and distance_reading is the sensor value from 0 → 1000.

Overall the repulsive force code is the following:

```
def get_total_repulsive_force(self, max_magnitude=4) -> np.ndarray:
        """Return the repulsive forces relative to the obstacles distances as 2D vectors."""
        sensors_angles = (
            self.get_robot_angle() + Controller.RELATIVE_ANGLES_OF_DISTANCE_SENSORS
        )
        distances = self.map(self.get_distances(), 0, 1000, max_magnitude, 0)

        return -np.sum(
            [
                self.get_vector_components(distance, angle)
                for distance, angle in zip(distances, sensors_angles)
            ],
            axis=0,
        )
```

This total repulsive force is then summed with the attractive force obtained simply by subtracting the goal position from the current position:

```
def get_total_force(self):
        return self.get_total_repulsive_force() + self.get_robot_heading()
```

In this part the raw speed of the motor is controlled by the distance from the goal so that it moves faster further away and slower near the goal:

```
raw_speed = self.map(
            distance_to_goal, 0, self.initial_distance_to_goal, 0, Controller.MAX_SPEED
        )
```

The entire code for this part is below:

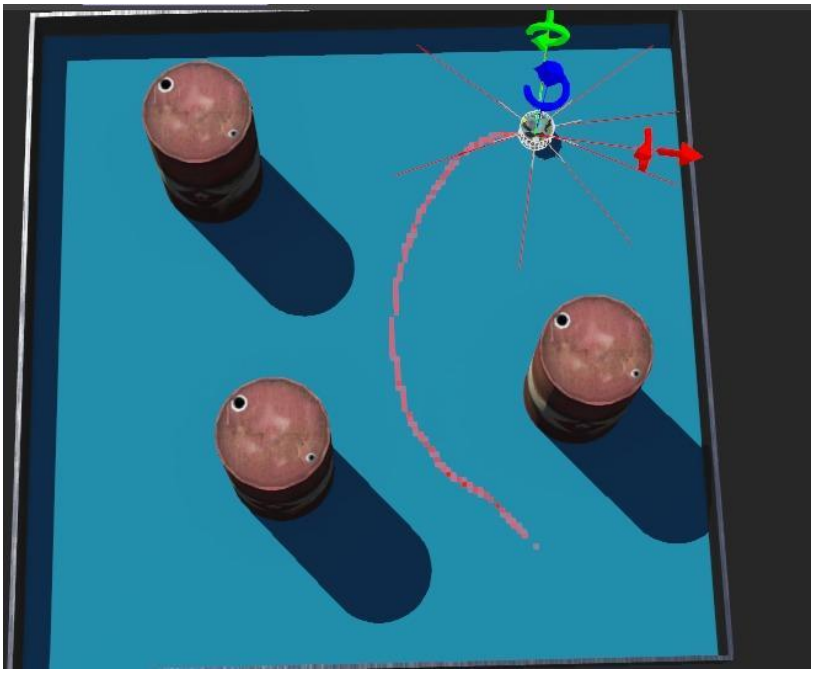
```
class Controller(Robot):
    MAX_SPEED = 6.2
    RELATIVE_ANGLES_OF_DISTANCE_SENSORS = np.array(
        [
            1.27 - np.pi / 2,
            0.77 - np.pi / 2,
            -np.pi / 2,
            -(2 * np.pi - 5.21) - np.pi / 2,
            4.21 - np.pi / 2,
            np.pi / 2,
            2.37 - np.pi / 2,
            1.87 - np.pi / 2,
        ]
    )

    def __init__(
        self, goal_position: np.ndarray, distance_threshold: float = 0.07
    ) -> None:
        super().__init__()

        self.goal_position = goal_position
        self.distance_threshold = distance_threshold

        # initialize the devices of the robot

        self.timeStep = int(self.getBasicTimeStep())

        self.pen = self.getDevice("pen")

        self.gps = self.getDevice("gps")
        self.gps.enable(self.timeStep)

        self.imu = self.getDevice("IMU")
        self.imu.enable(self.timeStep)

        self.distance_sensors = [self.getDevice(f"ds{i}") for i in range(8)]
        for ds in self.distance_sensors:
            ds.enable(self.timeStep)
```

```
        self.left_motor = self.getDevice("left wheel motor")
        self.right_motor = self.getDevice("right wheel motor")
        for motor in [self.left_motor, self.right_motor]:
            motor.setPosition(float("inf"))
            motor.setVelocity(0.0)

    def get_robot_position(self) -> np.ndarray:
        """Returns the (x, y) position of the robot from the GPS device."""
        return np.array(self.gps.getValues())[:2]

    def get_robot_angle(self) -> np.float64:
        """Returns the yaw angle of the robot in radians"""
        _, _, yaw = self.imu.getRollPitchYaw()
        return np.float64(yaw)

    def get_robot_heading(self) -> Tuple[np.ndarray, np.float64]:
        """Calculates the heading vector from the current position to the goal
position."""
        return self.goal_position - self.get_robot_position()

    def filter_angle(self, angle):
        """Returns the proportional control output for a given target and current
value."""
        # changing the range of the angle to the range [-pi, pi]
        angle = np.arctan2(np.sin(angle), np.cos(angle))

        front_distance = min(self.get_distances()[[0, 7]])
        if front_distance < 200:
            filter_amount = self.map(front_distance, 0, 200, 12, 0.9)
        else:
            filter_amount = 0.9

        return filter_amount * angle

    def map(self, value, from_lower, from_higher, to_lower, to_higher):
        """Maps a value from the range [`from_lower`, `from_higher`] to the range
[`to_lower`, `to_higher`]."""
        mapped_value = (value - from_lower) * (to_higher - to_lower) / (
            from_higher - from_lower
        ) + to_lower
        return np.clip(mapped_value, min(to_lower, to_higher), max(to_lower,
to_higher))

    def get_vector_components(self, magnitude: float, angle: float) -> np.ndarray:
        """Calculate the x and y components of the vector"""
        x = magnitude * np.cos(angle)
        y = magnitude * np.sin(angle)
        return np.array([x, y])
```

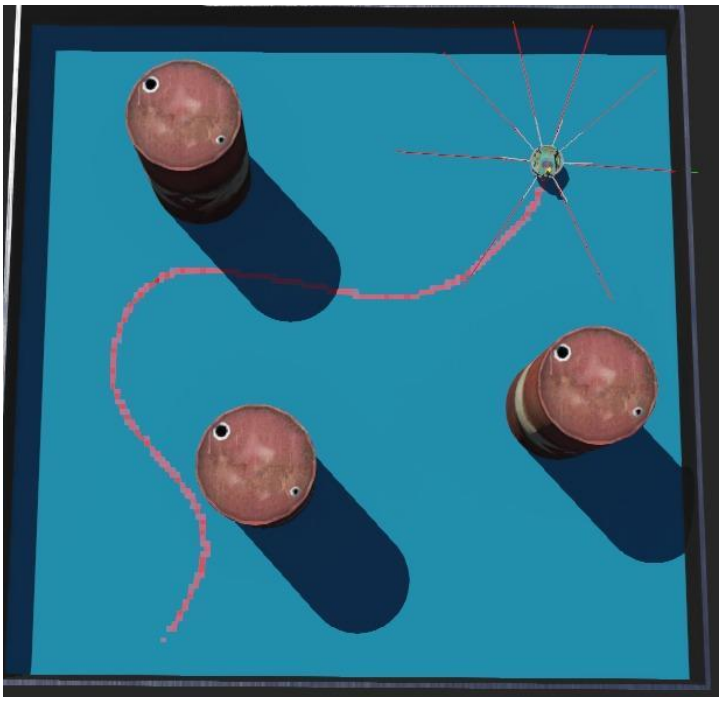
```python
    def get_distances(self) -> np.ndarray:
        return np.array([ds.getValue() for ds in self.distance_sensors])

    def get_total_repulsive_force(self, max_magnitude=4) -> np.ndarray:
        """Return the repulsive forces relative to the obstacles distances as 2D
vectors."""
        sensors_angles = (
            self.get_robot_angle() + Controller.RELATIVE_ANGLES_OF_DISTANCE_SENSORS
        )
        distances = self.map(self.get_distances(), 0, 1000, max_magnitude, 0)

        return -np.sum(
            [
                self.get_vector_components(distance, angle)
                for distance, angle in zip(distances, sensors_angles)
            ],
            axis=0,
        )

    def get_total_force(self):
        return self.get_total_repulsive_force() + self.get_robot_heading()

    def get_motors_speeds(
        self, distance_to_goal: float, total_force: np.ndarray
    ) -> tuple[float, float]:
        """Computes the left and right motor speeds based on the heading angle and
the robot's orientation."""
        # stop if too close to goal
        if distance_to_goal <= self.distance_threshold:
            return 0, 0

        raw_speed = self.map(
            distance_to_goal, 0, self.initial_distance_to_goal, 0,
Controller.MAX_SPEED
        )

        target_angle = np.arctan2(total_force[1], total_force[0])
        angle_difference = target_angle - self.get_robot_angle()
        angle_difference = self.filter_angle(angle_difference)

        left_speed = raw_speed - angle_difference
        left_speed = np.clip(left_speed, -Controller.MAX_SPEED, Controller.MAX_SPEED)

        right_speed = raw_speed + angle_difference
        right_speed = np.clip(right_speed, -Controller.MAX_SPEED,
Controller.MAX_SPEED)
```

```
        return left_speed, right_speed

    def get_initial_distance_to_goal(self) -> float:
        self.step(self.timeStep)  # needed to enable gps
        return np.linalg.norm(self.goal_position - self.get_robot_position())

    def get_distance_to_goal(self):
        return np.linalg.norm(self.goal_position - self.get_robot_position())

    def set_motors_speeds(self, left_speed: float, right_speed: float) -> None:
        self.left_motor.setVelocity(left_speed)
        self.right_motor.setVelocity(right_speed)

    def run(self) -> None:
        self.initial_distance_to_goal = self.get_initial_distance_to_goal()

        while self.step(self.timeStep) != -1:
            distance_to_goal = self.get_distance_to_goal()

            total_force = self.get_total_force()

            left_speed, right_speed = self.get_motors_speeds(
                distance_to_goal, total_force
            )
            self.set_motors_speeds(left_speed, right_speed)

            if distance_to_goal <= self.distance_threshold:
                print("Goal reached!")
                break


controller = Controller(goal_position=np.array([1.136, 1.136]))
controller.run()
```

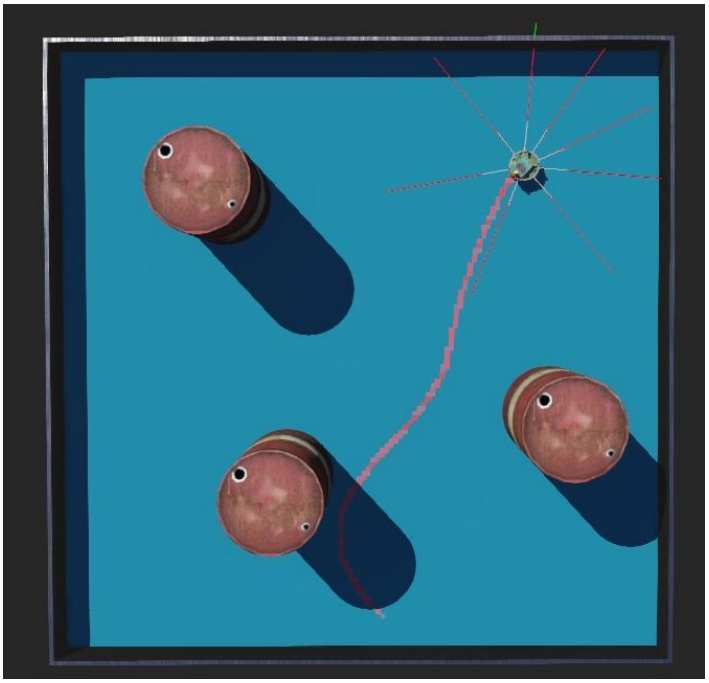
The experiments are as follows:



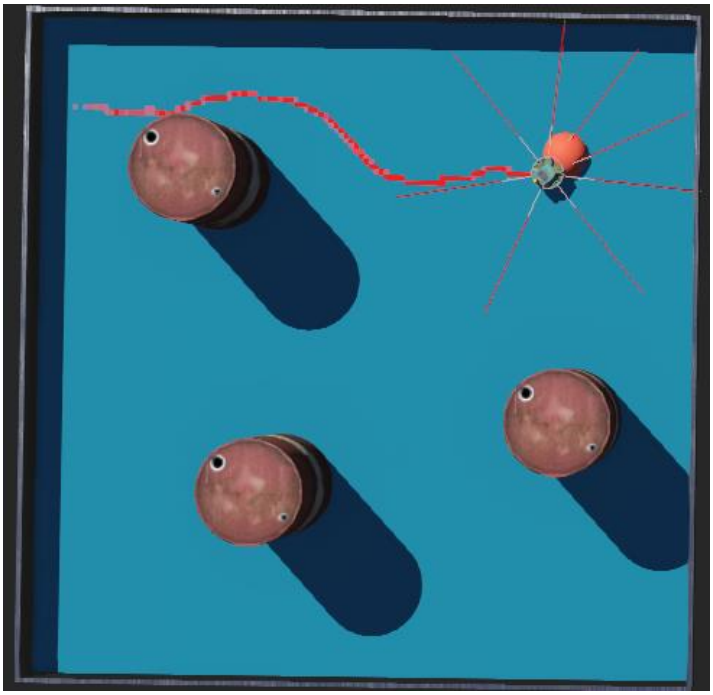
The robot was successfully able to move between the obstacles and reach the goal from the bottom right corner.



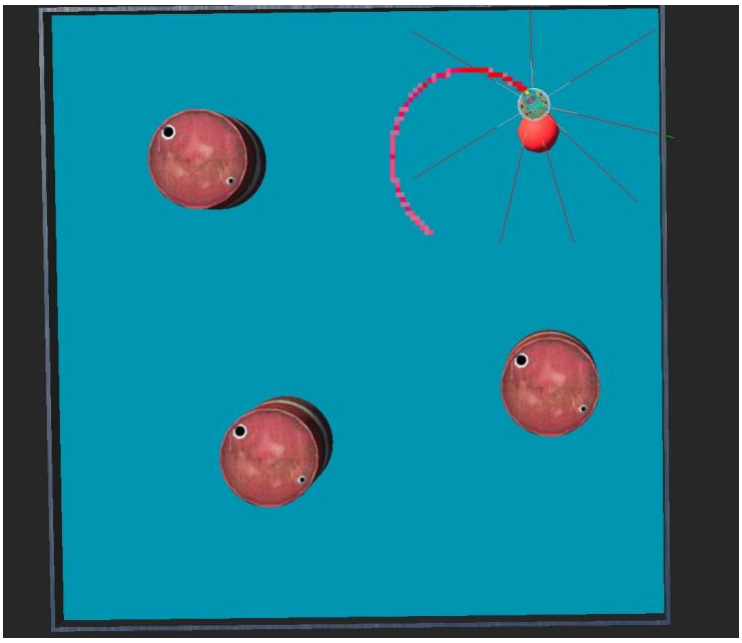
The robot was again able to avoid the obstacle in the bottom left position and reach the goal.



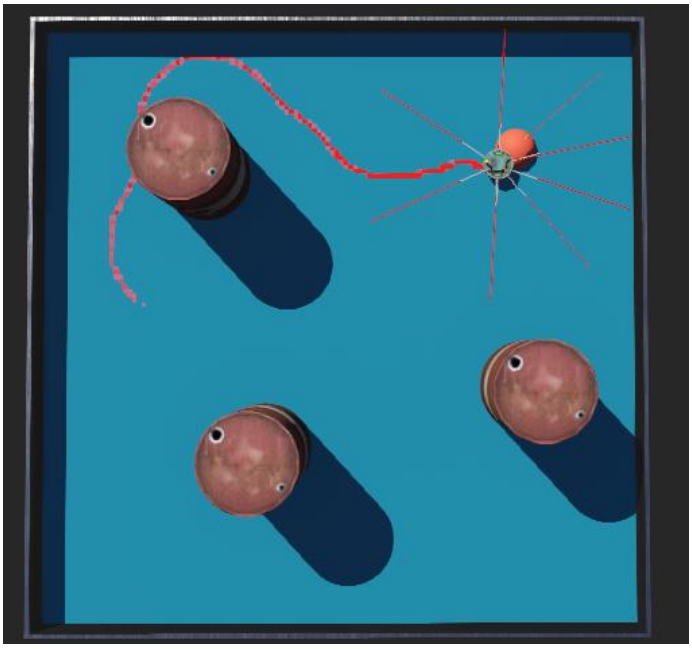
Again it avoided obstacles when it was oriented directly to face it.

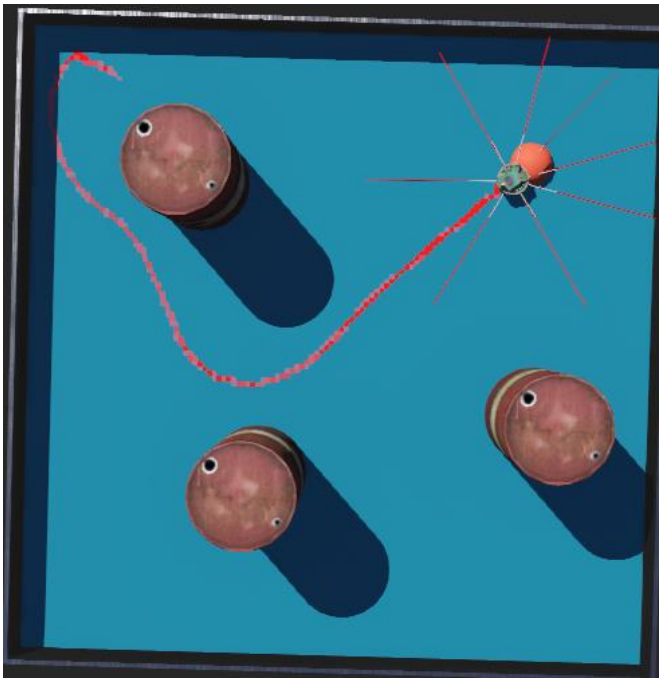


The robot successfully reached the goal from the top left corner avoiding the obstacle and wall.



The attractive behavior still works as intended. The robot rotated itself to reach the goal. The red object at the goal location is not a physical obstacle, it’s an indicator of the goal.

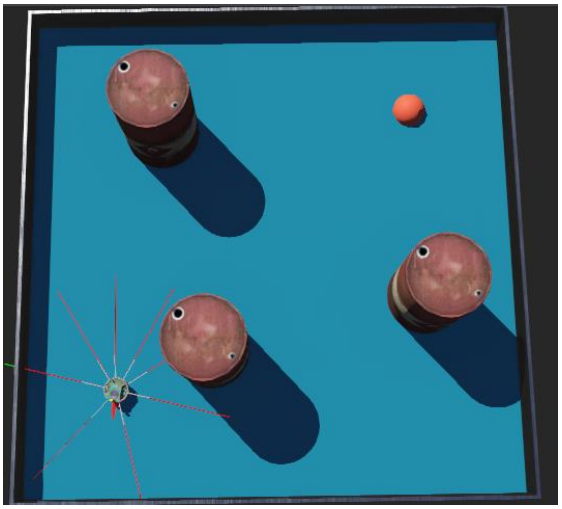




The robot was able to avoid the wall when placed too close to it without collision.

d) With our initial implementation where the motor speed is controlled by the distance to the goal, even if the robot reaches a local minima, (vector sum = 0) it will still move bypassing that position. However, if we set the motor speed to be controlled by the magnitude of the vector sum, then we found that position as shown below. The robot will not be moving indefinitely because the motor speeds will be 0 rad/sec. This happens because the repulsive forces cancel out the attractive force.

```
raw_speed = self.map(
            np.linalg.norm(total_force), 0, 4, 0, Controller.MAX_SPEED
        )
```



Our initial solution kind of solves this problem but it is not the ideal solution. This is because if the robot is stuck in a loop, it will still have no way to exit that loop. So even if our solution fixes the local minima, it still suffers from the looping problem. To properly solve the local minima and looping problems, we can add random noise to the motor speeds. This will ensure that the robot is pushed from the local minima to again move towards the goal even when the forces add up to 0. The disadvantage of this solution, however, is that it introduces unpredictability to the system. It might lead to erratic movements and behaviors that are difficult to anticipate and control. It might also lead the robot to overcorrect itself, which leads to instability in the robot's path.

e) In order to make the robot move towards the goal as fast as possible (while still being safe enough to avoid hitting into obstacles), two changes were made. First, the maximum magnitude for the repulsive forces was decreased from 4 to 3 to make the robot take riskier turns that save time.

```
def get_total_repulsive_force(self, max_magnitude=3) -> np.ndarray:
        """Return the repulsive forces relative to the obstacles
distances as 2D vectors."""
        sensors_angles = (
            self.get_robot_angle() +
Controller.RELATIVE_ANGLES_OF_DISTANCE_SENSORS
        )
        distances = self.map(self.get_distances(), 0, 1000,
max_magnitude, 0)
```

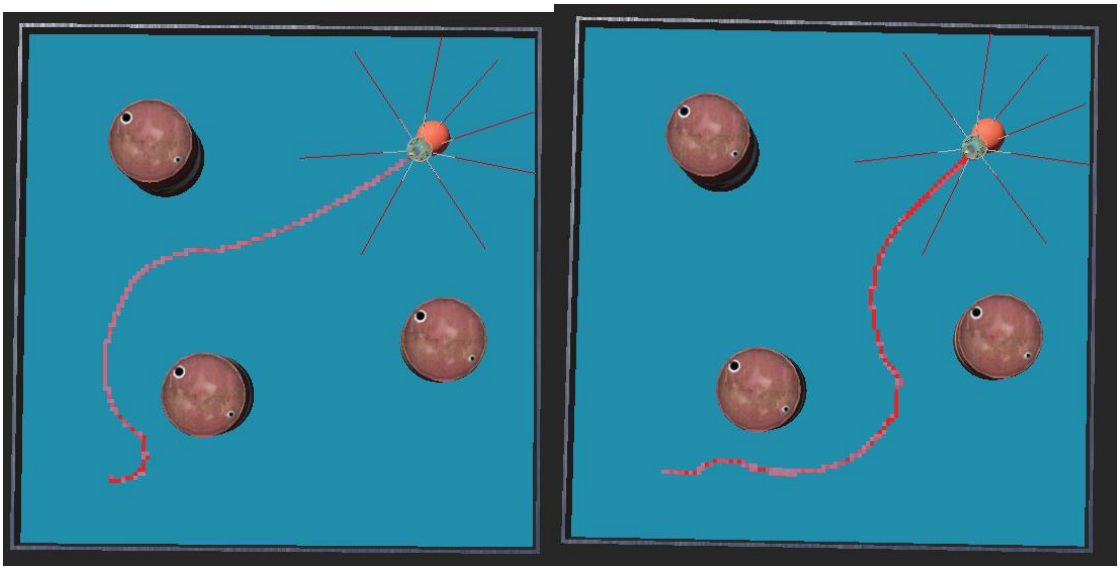
```
        return -np.sum(
            [
                self.get_vector_components(distance, angle)
                for distance, angle in zip(distances, sensors_angles)
            ],
            axis=0,
        )
```

Secondly, the speed was made to be proportional to the angle representing the difference between the robot's orientation and the target angle instead of being proportional to the distance to the goal. To elaborate, the robot's speed was initially related to how close it was to the goal, the further, the faster, and the closer the slower. By making the speed proportional to the difference in angles, the robot slows down when it's turning (for safety) since its orientation is different from the target one, and speeds up when it's on a straight line to the goal since the difference in angles becomes *approximately* zero.

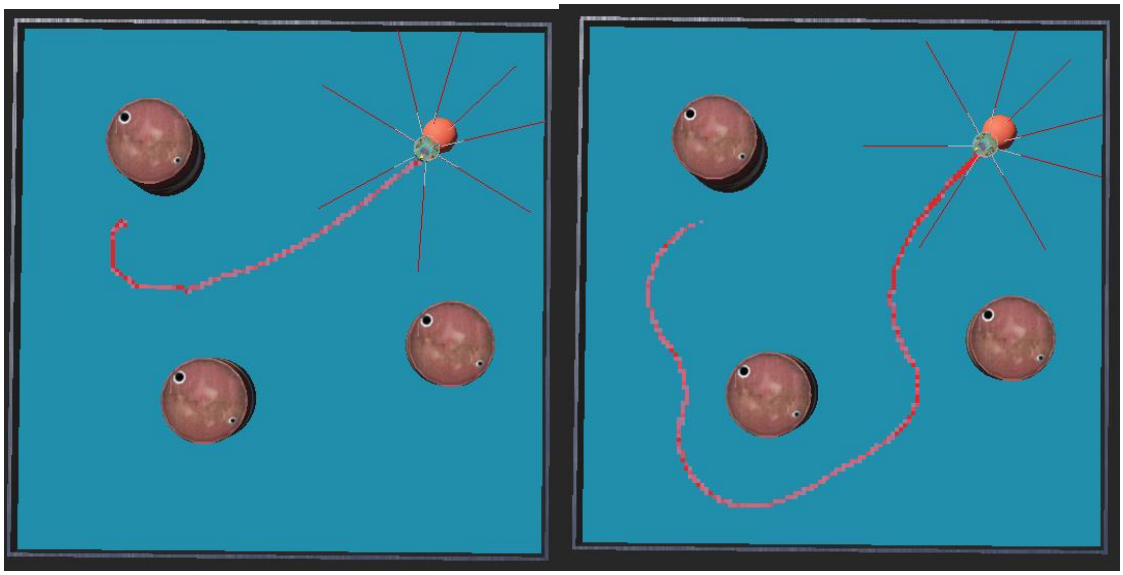
```
def get_motors_speeds(
    self, distance_to_goal: float, total_force: np.ndarray
) -> tuple[float, float]:
    """Computes the left and right motor speeds based on the heading
angle and the robot's orientation."""
    # stop if too close to goal
    if distance_to_goal <= self.distance_threshold:
        return 0, 0

    target_angle = np.arctan2(total_force[1], total_force[0])
    angle_difference = target_angle - self.get_robot_angle()
    angle_difference = self.filter_angle(angle_difference)

    raw_speed = self.map(angle_difference, np.pi / 2, 0, 0,
Controller.MAX_SPEED)

    left_speed = raw_speed - angle_difference
    left_speed = np.clip(left_speed, -Controller.MAX_SPEED,
Controller.MAX_SPEED)
```

```
    right_speed = raw_speed + angle_difference
    right_speed = np.clip(right_speed, -Controller.MAX_SPEED,
Controller.MAX_SPEED)

    return left_speed, right_speed
```

The two figures below illustrate the fast behavior (left) and safe behavior (right) in two different starting positions.



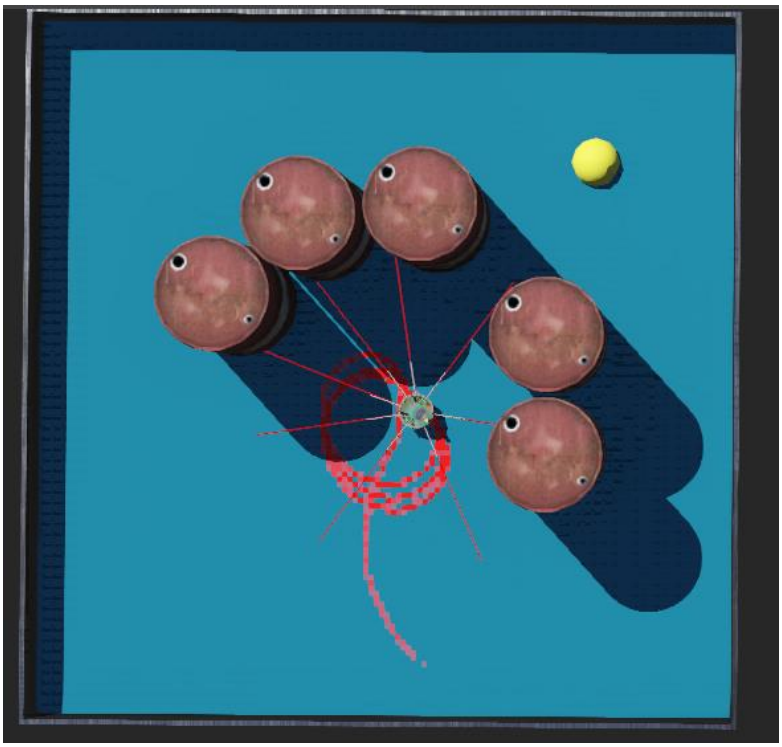
In this scenario, it can be seen that the fast robot takes a riskier, faster turn. Furthermore, the thickness of the traced path drawn by the pen (thicker is slower) indicates that the fast robot slows down when turning and speeds up when in a straight line, while tAe robot slows down drastically the closer it gets to the goal.



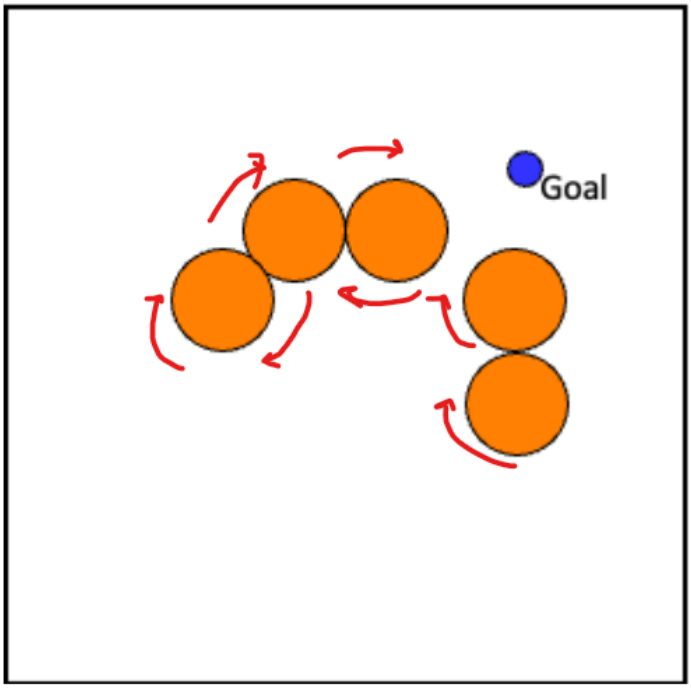
Similar behavior can be seen this second scenario as well.

Question 3

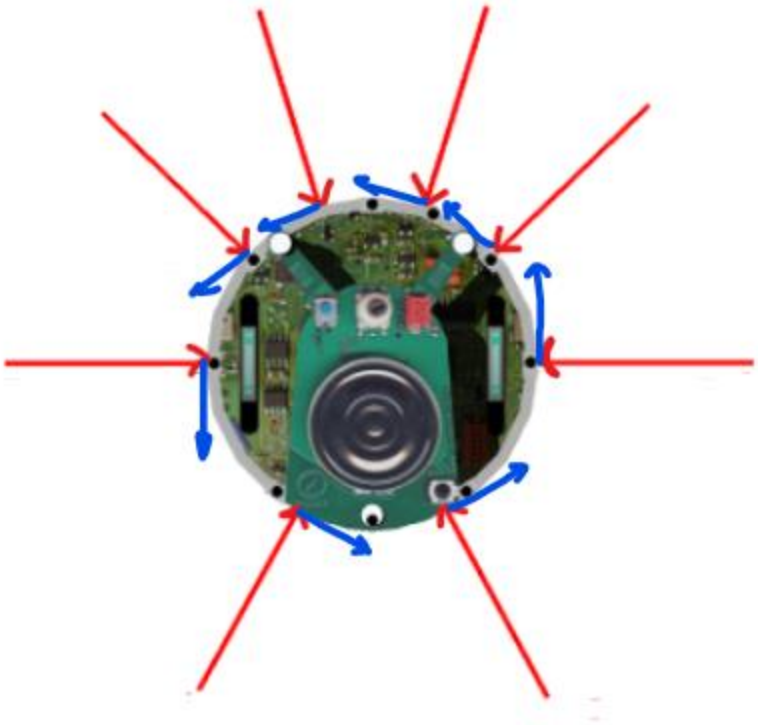
a) The issue that would arise in this map is that if the robot is placed in between the barrels, it will be stuck in a loop there. This is because the attractive forces tell it to go into the obstacles, the obstacles push the robot around other obstacles, which prevents the robot from escaping the area. This is demonstrated in the figure below. To solve this problem, we implement a tangential behavior, as it will help the robot circumvent around the obstacles.



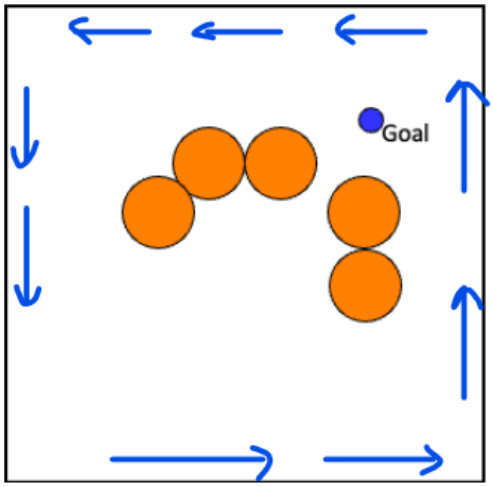
The tangential vectors should be clockwise as shown below (this also makes the robot reach the goal in the next question). The same linear dropoffs were used from the repulsive forces.



This is implemented by simply adding pi/2 to the angle of the distance vector as shown in the blue vectors below:



However, we will use the tangential behavior along with the repulsive behavior to ensure the robot does not collide with the obstacles. The problem with this design is that since the distance sensors have no way of knowing the type of obstacle it is sensing, it will still activate the tangential behavior for the walls, and it will look like the following:



Therefore, as per the design, the robot will hug the wall when it senses it and prevent the robot from reaching the goal.

b) The tangential behavior looks identical to the repulsive behavior in code. Only the negative sign was removed and pi/2 was added to each distance vector angle (clockwise):

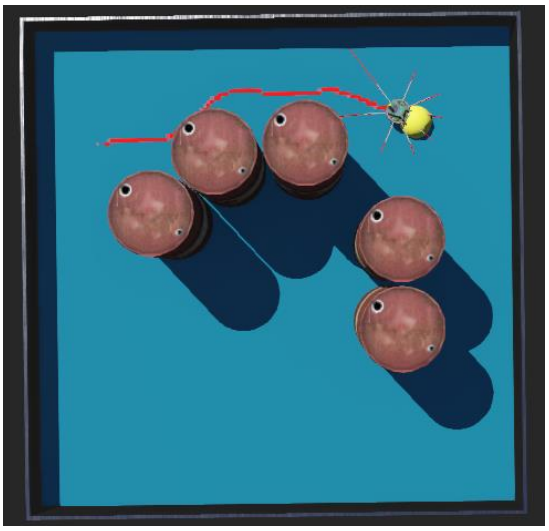
```
def get_total_tangential_force(self, sensors : list = list(range(8)), max_magnitude=4) ->
np.ndarray:
        """Return the clock-wise tangential forces relative to the obstacles
            distances as 2D vectors."""
        sensors_angles = (
            self.get_robot_angle() + Controller.RELATIVE_ANGLES_OF_DISTANCE_SENSORS
        )
        sensors_angles = sensors_angles[sensors]
        distances = self.map(self.get_distances(), 0, 1000, max_magnitude, 0)
        distances = distances[sensors]
        return np.sum(
            [
                self.get_vector_components(distance, angle + np.pi/2)
                for distance, angle in zip(distances, sensors_angles)
            ],
            axis=0,
        )
```

The combination of the three behaviors is described below:

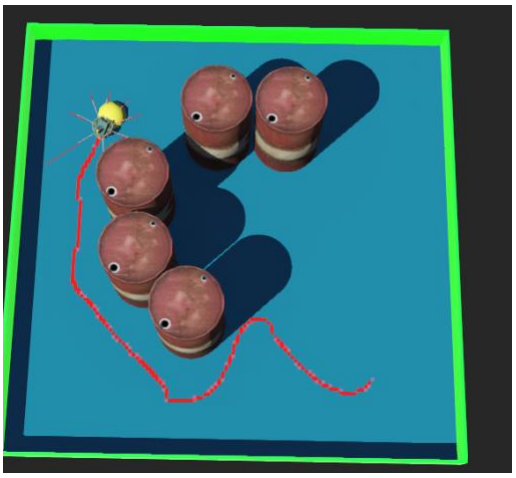
```
def get_total_force(self):
        print('tan:', (tan:=self.get_total_tangential_force(max_magnitude= 8)))
        print('rep:', (rep:=self.get_total_repulsive_force(sensors=[2,5],max_magnitude=0.5)))
        print('att:', (att:=self.get_robot_heading(max_magnitude=0.01)))
        return tan + rep + att
```

The sensors argument in the repulsive force function was added to activate that behavior from those specific sensors. In this case, only ds2 and ds5 (the horizontal ones) repel the robot. This argument defaults to list(range(8)), meaning it will activate for all distance sensors. The tangential behavior is activated for all the sensors. Also, the max_magnitude argument on all the behaviors specifies the maximum vector magnitude for that behavior. In this case we set a high value of 8 for the tangential, and the other behaviors were weakened (0.5 repulsive and 0.01 attractive). These values were determined after many experiments and this combination seems to work correctly. Also the range of the distance sensors were reduced to prevent the robot from sensing the wall strongly and hence hug it. The two sensors on the back ds3 and ds4 were kept longer than the other because in some cases the robot drifts away from the tangential behavior around the obstacle too early due to the attractive force, so these two sensors would keep the

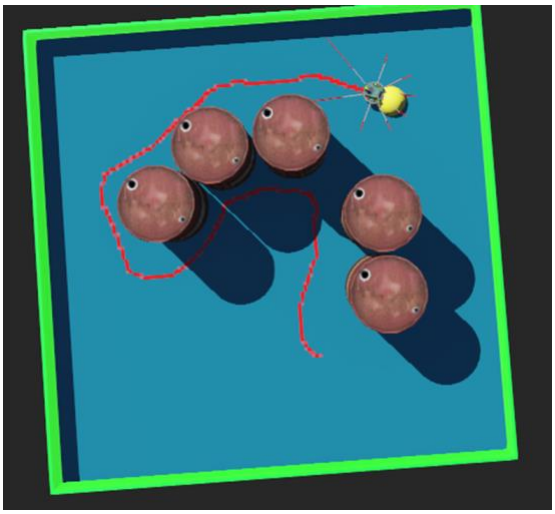
tangential behaviors active long enough as not to drift away affected by the attraction. The experiments are summarized below:



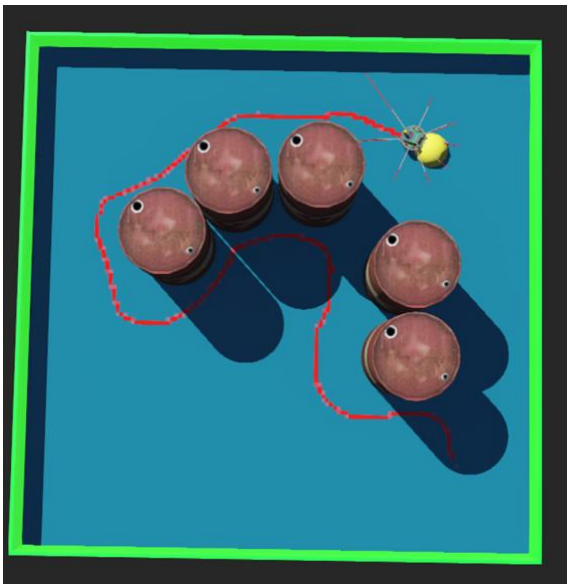
The robot follows the tangential vectors from the obstacles when placed in the top left corner.



The robot correctly follows the obstacles around when faced by them and reaches the goal when placed in the bottom left.

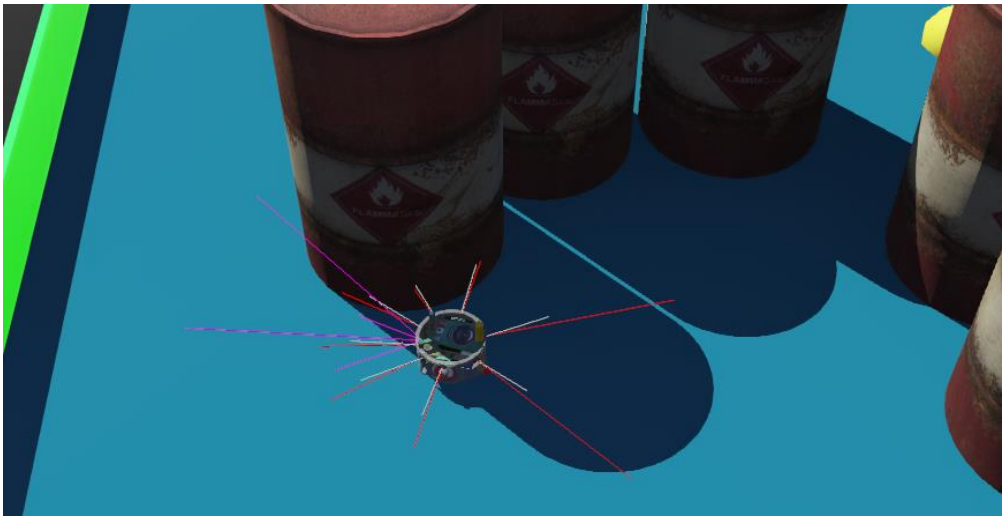


The robot correctly escapes the obstacles area and circumvents to reach the goal when placed in between the obstacles without collision.

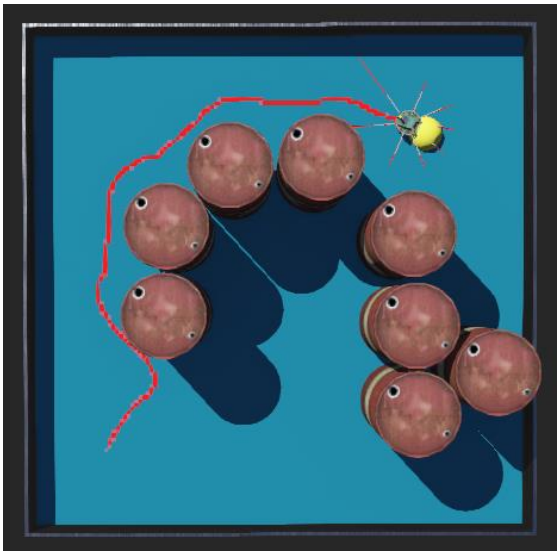


In a similar manner, the robot reaches the goal from the bottom right corner.

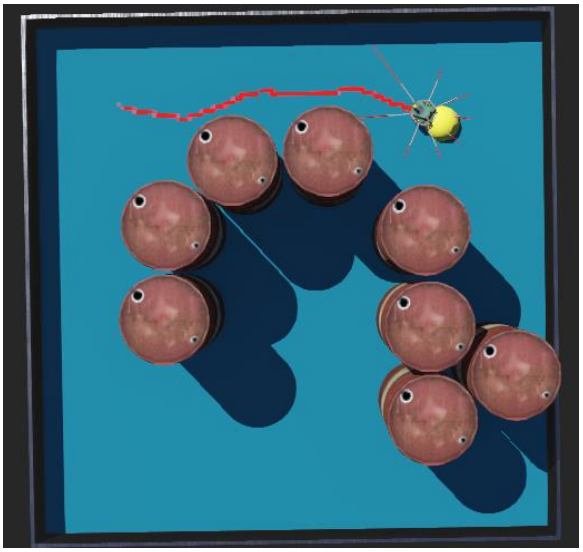
Note that we tried to use the camera to detect the green wall and hence stop the tangential behavior when facing it, but it was buggy and removed. If the obstacle is still near the obstacle but facing the wall, we would want the tangential behavior to stay active, but the camera detects the wall and deactivates it. This mainly happens in the following orientation (The purple rays are the camera view points):



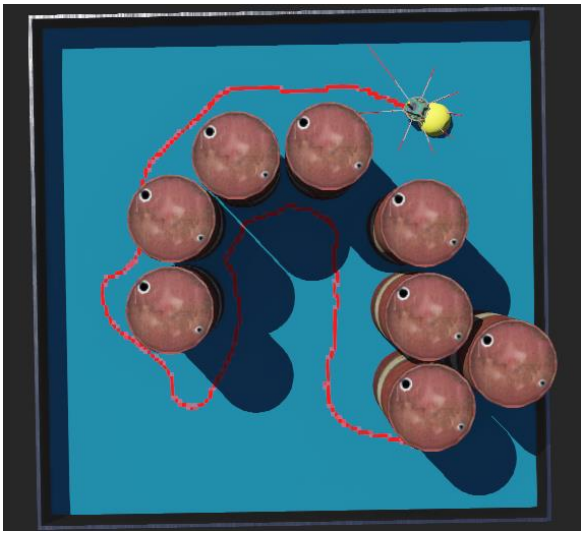
c)



The robot correctly moves to the goal as it did earlier from the bottom left corner.



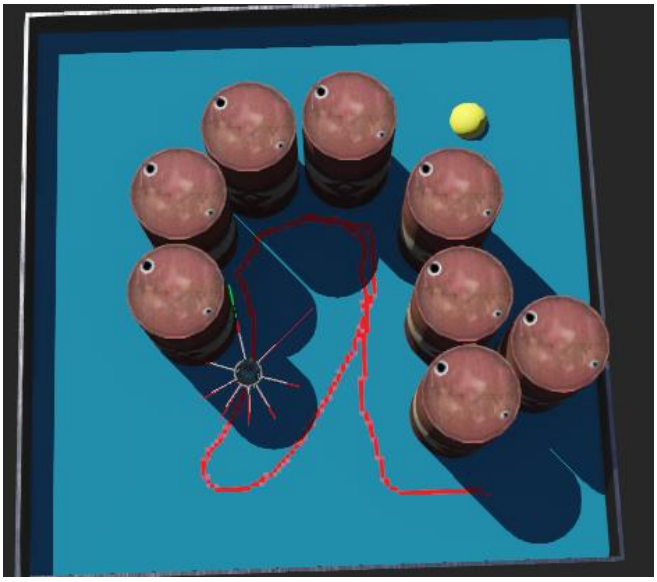
Again, as expected the robot reaches the goal from the top left corner.



The bottom left corner worked by slight modifications had to be done for it to work. The max_magnitude like so:

```
def get_total_force(self):
        print('tan:', (tan:=self.get_total_tangential_force(max_magnitude= 12)))
        print('rep:', (rep:=self.get_total_repulsive_force(sensors=[2,5],max_magnitude=1)))
        print('att:', (att:=self.get_robot_heading(max_magnitude=0.001)))
        return tan + rep + att
```

Otherwise the robot would drift away from the last obstacle and try to reach the goal like so:



We also noticed that in this case, the robot collides with the obstacles because it doesn't have time to avoid them correctly, so our model is far from perfect. In order to implement a far better model, we would first tackle the wall vs barrel to only activate the tangential vectors when the obstacle is only a barrel. Also a far better design would cleverly detect which direction of the tangential is the best (clockwise or anticlockwise) to reach the goal with the least distance moved regardless of the robot or goal positions.

The entire code for Question 3 is below:

```
from typing import Tuple
from controller import Robot
import numpy as np


class Controller(Robot):
    MAX_SPEED = 6.2
    RELATIVE_ANGLES_OF_DISTANCE_SENSORS = np.array(
        [
            1.27 - np.pi / 2,
            0.77 - np.pi / 2,
            -np.pi / 2,
            -(2 * np.pi - 5.21) - np.pi / 2,
            4.21 - np.pi / 2,
            np.pi / 2,
            2.37 - np.pi / 2,
            1.87 - np.pi / 2,
        ]
```

```
    )

    def __init__(
        self, goal_position: np.ndarray, distance_threshold: float = 0.07
    ) -> None:
        super().__init__()

        self.goal_position = goal_position
        self.distance_threshold = distance_threshold

        # initialize the devices of the robot

        self.timeStep = int(self.getBasicTimeStep())

        self.pen = self.getDevice("pen")

        self.gps = self.getDevice("gps")
        self.gps.enable(self.timeStep)

        self.imu = self.getDevice("IMU")
        self.imu.enable(self.timeStep)

        self.distance_sensors = [self.getDevice(f"ds{i}") for i in range(8)]
        for ds in self.distance_sensors:
            ds.enable(self.timeStep)

        self.left_motor = self.getDevice("left wheel motor")
        self.right_motor = self.getDevice("right wheel motor")
        for motor in [self.left_motor, self.right_motor]:
            motor.setPosition(float("inf"))
            motor.setVelocity(0.0)

    def get_robot_position(self) -> np.ndarray:
        """Returns the (x, y) position of the robot from the GPS device."""
        return np.array(self.gps.getValues())[:2]

    def get_robot_angle(self) -> np.float64:
        """Returns the yaw angle of the robot in radians"""
        _, _, yaw = self.imu.getRollPitchYaw()
        return np.float64(yaw)

    def get_robot_heading(self, max_magnitude = 0.5) -> Tuple[np.ndarray, np.float64]:
        """Calculates the heading vector from the current position to the goal position."""
        heading = self.goal_position - self.get_robot_position()
```

```
        return (max_magnitude/np.linalg.norm(heading)) * (heading)

    def filter_angle(self, angle):
        """Returns the proportional control output for a given target and current value."""
        # changing the range of the angle to the range [-pi, pi]
        angle = np.arctan2(np.sin(angle), np.cos(angle))

        front_distance = min(self.get_distances()[[0, 7]])
        if front_distance < 200:
            filter_amount = self.map(front_distance, 0, 200, 12, 0.9)
        else:
            filter_amount = 0.9

        return filter_amount * angle

    def map(self, value, from_lower, from_higher, to_lower, to_higher):
        """Maps a value from the range [`from_lower`, `from_higher`] to the range [`to_lower`, `to_higher`]."""
        mapped_value = (value - from_lower) * (to_higher - to_lower) / (
            from_higher - from_lower
        ) + to_lower
        return np.clip(mapped_value, min(to_lower, to_higher), max(to_lower, to_higher))

    def get_vector_components(self, magnitude: float, angle: float) -> np.ndarray:
        """Calculate the x and y components of the vector"""
        x = magnitude * np.cos(angle)
        y = magnitude * np.sin(angle)
        return np.array([x, y])

    def get_distances(self) -> np.ndarray:
        return np.array([ds.getValue() for ds in self.distance_sensors])

    def get_total_repulsive_force(self, sensors : list = list(range(8)), max_magnitude=4) -> np.ndarray:
        """Return the repulsive forces relative to the obstacles distances as 2D vectors."""
        sensors_angles = (
            self.get_robot_angle() + Controller.RELATIVE_ANGLES_OF_DISTANCE_SENSORS
        )
        sensors_angles = sensors_angles[sensors]
        distances = self.map(self.get_distances(), 0, 1000, max_magnitude, 0)
        distances = distances[sensors]
        return -np.sum(
            [
                self.get_vector_components(distance, angle)
```

```
                for distance, angle in zip(distances, sensors_angles)
            ],
            axis=0,
        )

    def get_total_tangential_force(self, sensors : list = list(range(8)), max_magnitude=4) ->
np.ndarray:
        """Return the clock-wise tangential forces relative to the obstacles
            distances as 2D vectors."""
        sensors_angles = (
            self.get_robot_angle() + Controller.RELATIVE_ANGLES_OF_DISTANCE_SENSORS
        )
        sensors_angles = sensors_angles[sensors]
        distances = self.map(self.get_distances(), 0, 1000, max_magnitude, 0)
        distances = distances[sensors]
        return np.sum(
            [
                self.get_vector_components(distance, angle + np.pi/2)
                for distance, angle in zip(distances, sensors_angles)
            ],
            axis=0,
        )

    def get_total_force(self):
        print('tan:', (tan:=self.get_total_tangential_force(max_magnitude= 10)))
        print('rep:', (rep:=self.get_total_repulsive_force(sensors=[2,5],max_magnitude=0.5)))
        print('att:', (att:=self.get_robot_heading(max_magnitude=0.001)))
        return tan + rep + att

    def get_motors_speeds(
        self, distance_to_goal: float, total_force: np.ndarray
    ) -> tuple[float, float]:
        """Computes the left and right motor speeds based on the heading angle and the robot's
orientation."""
        # stop if too close to goal
        if distance_to_goal <= self.distance_threshold:
            return 0, 0

        raw_speed = self.map(
            distance_to_goal, 0, self.initial_distance_to_goal, 0, Controller.MAX_SPEED/2
        )

        target_angle = np.arctan2(total_force[1], total_force[0])
        angle_difference = target_angle - self.get_robot_angle()
```

```
        angle_difference = self.filter_angle(angle_difference)

        left_speed = raw_speed - angle_difference
        left_speed = np.clip(left_speed, -Controller.MAX_SPEED, Controller.MAX_SPEED)

        right_speed = raw_speed + angle_difference
        right_speed = np.clip(right_speed, -Controller.MAX_SPEED, Controller.MAX_SPEED)

        return left_speed, right_speed

    def get_initial_distance_to_goal(self) -> float:
        self.step(self.timeStep)  # needed to enable gps
        return np.linalg.norm(self.goal_position - self.get_robot_position())

    def get_distance_to_goal(self):
        return np.linalg.norm(self.goal_position - self.get_robot_position())

    def set_motors_speeds(self, left_speed: float, right_speed: float) -> None:
        self.left_motor.setVelocity(left_speed)
        self.right_motor.setVelocity(right_speed)

    def run(self) -> None:
        self.initial_distance_to_goal = self.get_initial_distance_to_goal()

        while self.step(self.timeStep) != -1:
            distance_to_goal = self.get_distance_to_goal()

            total_force = self.get_total_force()
            print('total:',total_force)
            left_speed, right_speed = self.get_motors_speeds(
                distance_to_goal, total_force
            )
            self.set_motors_speeds(left_speed, right_speed)

            if distance_to_goal <= self.distance_threshold:
                print("Goal reached!")
                break


controller = Controller(goal_position=np.array([1.136, 1.136]))
controller.run()
```