Object Detection/Avoidance

Problem Description:

Simulating a robot equipped with an ultrasonic sensor for object detection in order to perform collision avoidance. The objects are static for this case.

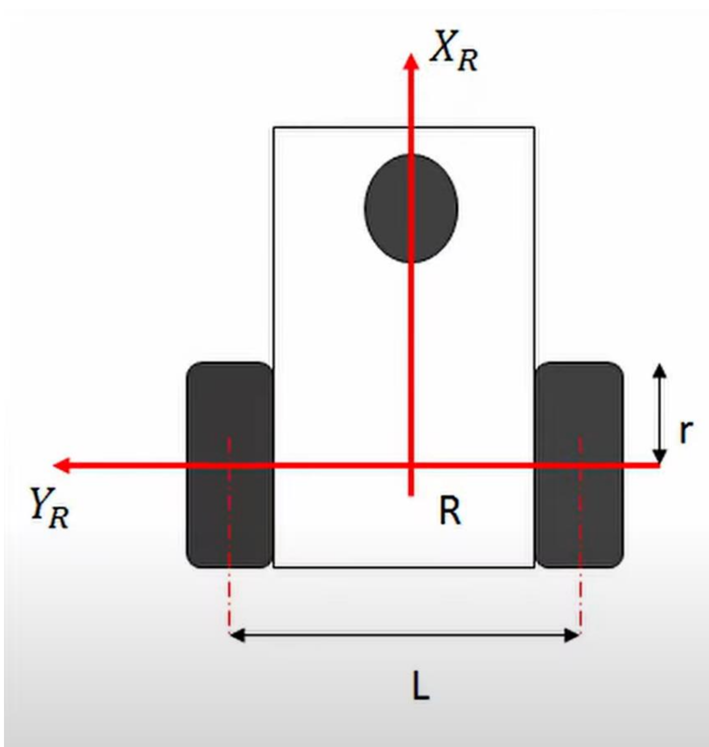
The kinematics of the model are as follows:

$$\dot{x} = \frac{(v_l + v_r)}{2} cos(\theta)$$

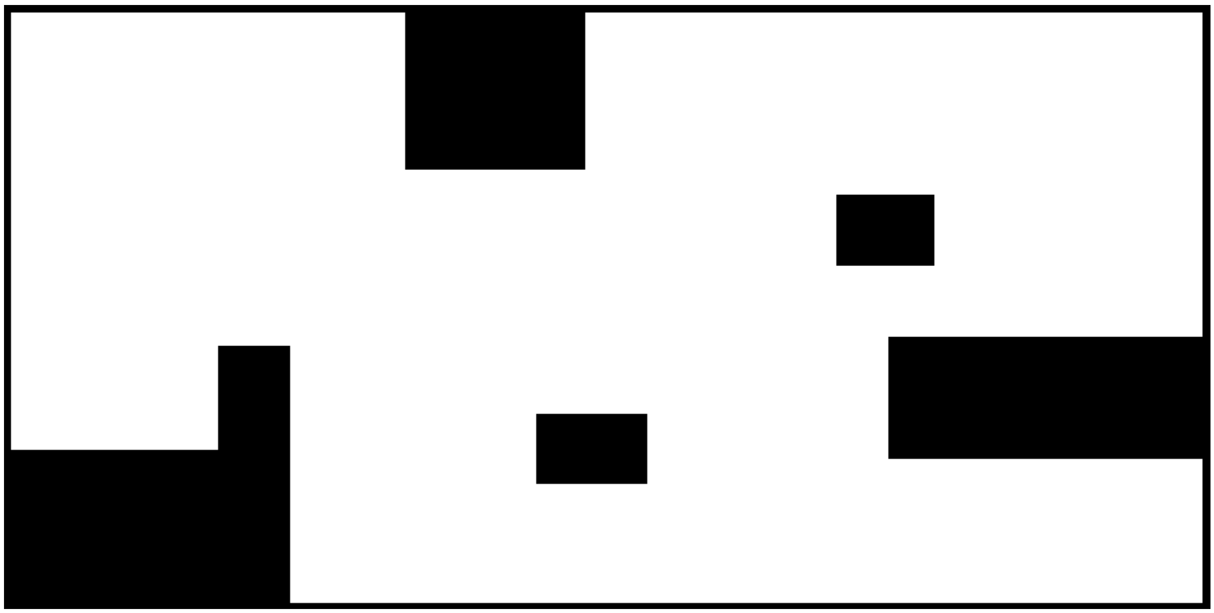
$$\dot{y} = \frac{(v_l + v_r)}{2} sin(\theta)$$

$$\dot{\theta} = \frac{(v_l - v_r)}{L}$$

Where v_l is the linear velocity of the left driving wheel, v_r is the linear velocity of the right driving wheel and L is the length between two wheels.



Environment map is created using Microsoft Paint as shown below:

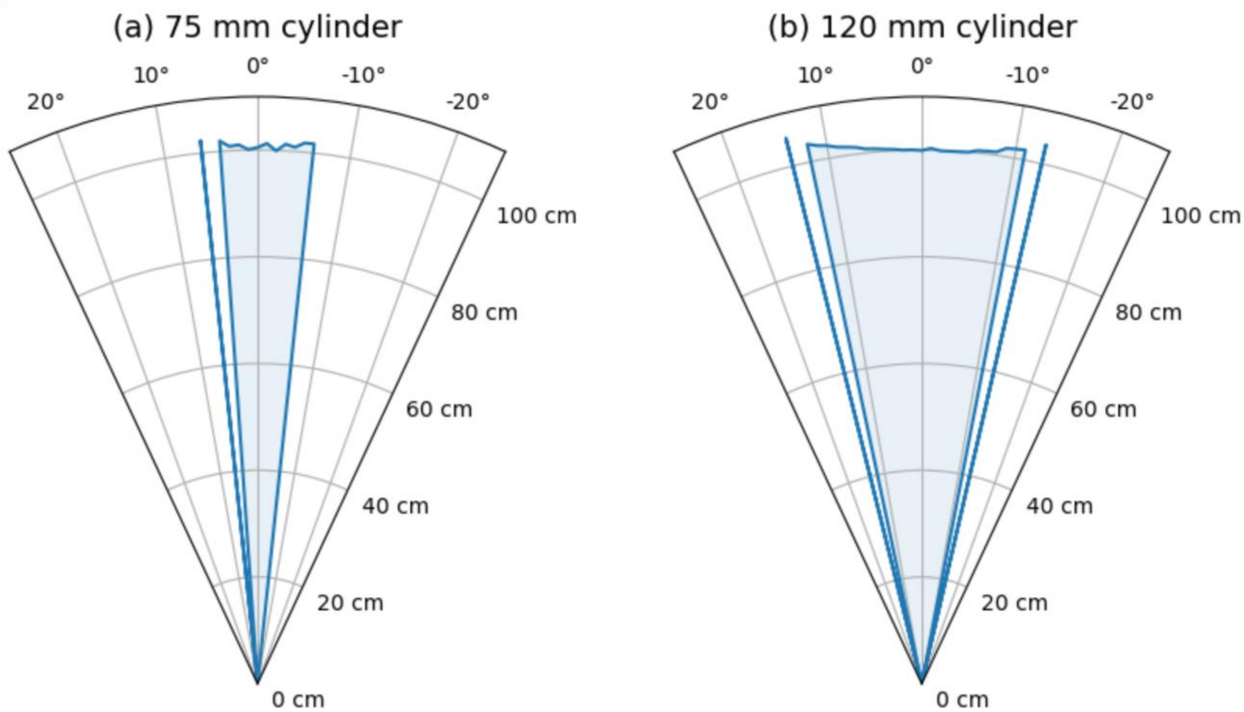


Robot Model:

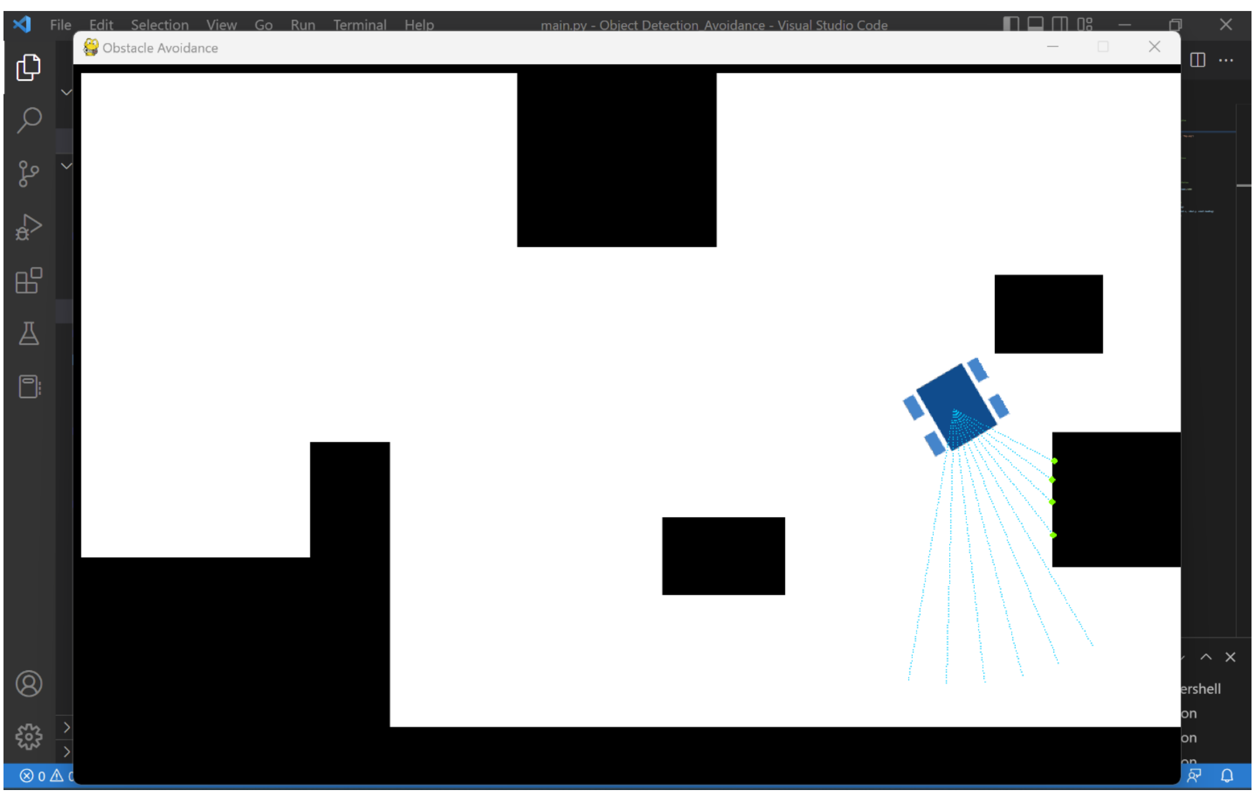
For robot code file, three classes are created named as:

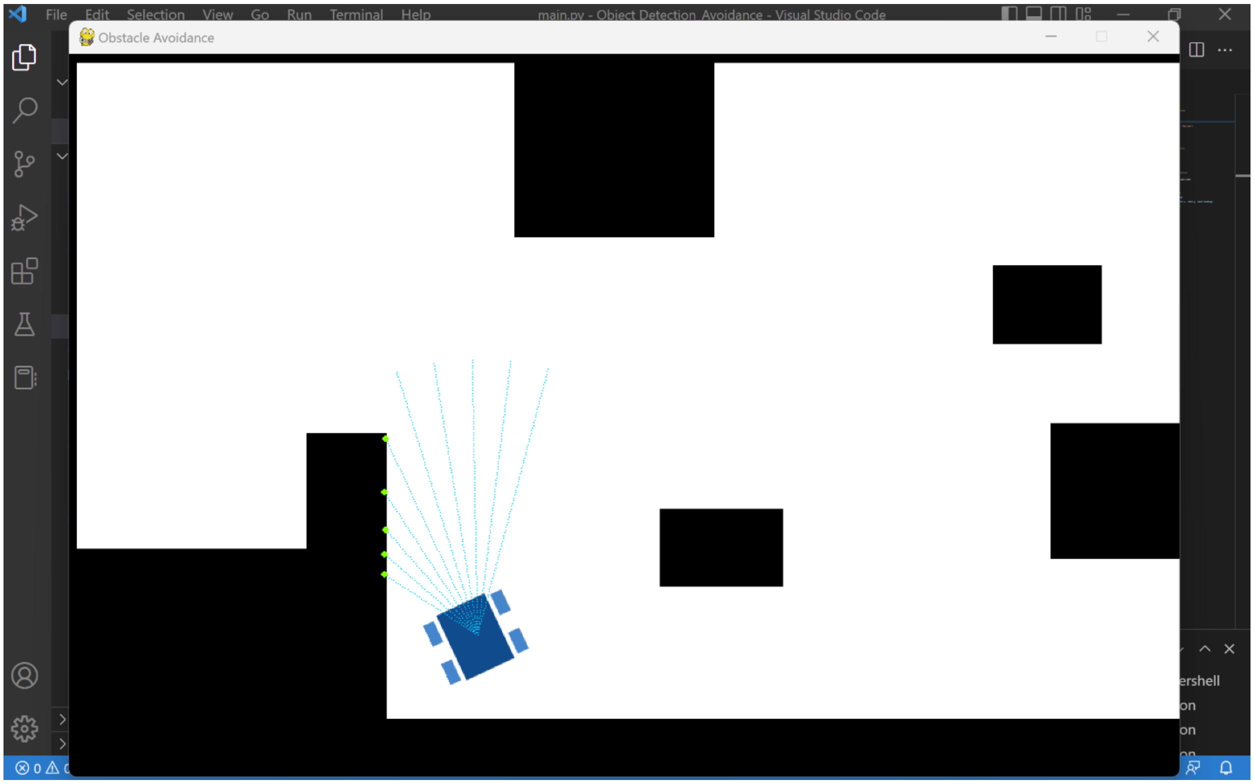
1. Robot
    - Kinematic models of the robot
2. Graphics
    - Scene visualization using Pygame module in Python
3. Ultrasonic
    - Represents the ultrasonic sensor given a 2D point cloud representing the face obstacles in the environment

Please note that ultrasonic sensor detection range is limited as shown below:



The final results:





Please see here for the GIF of the full simulation.

Code:

```
import os
import math
import numpy as np
import pygame


# clear = lambda: os.system('cls')  # On Windows System
# clear()

# calculating two distances using Euclidian method in environment
def distance(point1, point2):
    point1 = np.array(point1)
    point2 = np.array(point2)
    return np.linalg.norm(point1 - point2)

# defining robot class
class Robot:
    def __init__(self, startpos, width):

        # from meters to pixels
        self.m2p = 3779.52

        # robot dims
        self.w = width
        self.x = startpos[0]
        self.y = startpos[1]

        self.heading = 0

        # initializing the right and left wheel velocities
        self.vl = 0.01*self.m2p # m/s (1 cm/s)
        self.vr = 0.01*self.m2p # m/s

        self.max_speed = 0.02*self.m2p  # m/s
        self.min_speed = 0.01*self.m2p  # m/s

        # defining the minimum distance tht robot is allowed to get close to the
obstacle (this needs to be modified)
        self.min_obs_dist = 100
        self.count_down = 5 # seconds

    # obstacle avoidance function
    def avoid_obstacles(self, point_cloud, time_step):
```

```
        closest_obs = None
        dist = np.inf

        # conditions for the point cloud
        if len(point_cloud) > 1:
            # searching for the closest obstacle to the robot
            for point in point_cloud:
                if dist > distance([self.x, self.y], point):
                    dist = distance([self.x, self.y], point)
                    closest_obs = (point, dist)

            # if distance (between robot and obstacle) is less than minimum, move
backward
            if closest_obs[1] < self.min_obs_dist and self.count_down > 0:
                self.count_down -= time_step
                self.move_backward()
            # make sure we do not move back for ever
            else:
                # reset count down
                self.count_down = 5
                # move forward
                self.move_forward()

    # move backward function
    def move_backward(self):
        # making a circular trajectory backward move
        self.vr = -self.max_speed
        self.vl = -self.max_speed/2

    # move forward function
    def move_forward(self):
        self.vr = self.min_speed
        self.vl = self.min_speed

    # kinematics of the robot
    def kinematics(self, time_step):

        self.x += ((self.vl + self.vr)/2) * math.cos(self.heading) *time_step
        self.y -= ((self.vl + self.vr)/2) * math.sin(self.heading) *time_step

        self.heading += (self.vr - self.vl) / self.w * time_step

        # resetting the heading value to zero to avoid unwanted movement of the
robot
        if self.heading > 2*math.pi or self.heading < -2*math.pi:
```

```python
            self.heading = 0

        # setting vr and vl to min and max velocities
        self.vr = max(min(self.max_speed, self.vr), self.min_speed)
        self.vl = max(min(self.max_speed, self.vl), self.min_speed)


# defining graphics class
class Graphics:
    def __init__(self, dimensions, robot_img_path, map_img_path):
        pygame.init()

        # colors
        self.sensor_color = (127, 255, 0)
        self.black = (0, 0, 0)
        self.white = (255, 255, 255)
        self.green = (0, 255, 0)
        self.blue = (0, 0, 255)
        self.red = (255, 0, 0)
        self.yel = (255, 255, 0)

        # MAP section
        # map = pygame.display
        # load imgs
        self.robot = pygame.image.load(robot_img_path)
        self.map_img = pygame.image.load(map_img_path)

        # dimensions
        self.height, self.width = dimensions

        # window settings
        pygame.display.set_caption("Obstacle Avoidance")

        self.map = pygame.display.set_mode((self.width, self.height))
        self.map.blit(self.map_img, (0, 0))

        # self.map_img.blit(self.map_img, (0, 0))

        # display white on screen other than image
        # self.map_img.fill(self.white)

    # draw robot function
    def draw_robot(self, x, y, heading):
        rotated = pygame.transform.rotozoom(self.robot, math.degrees(heading), 1)
        rect = rotated.get_rect(center=(x, y))
```

```python
        self.map.blit(rotated, rect)

    # sensor data functiom
    def draw_sensor_data(self, point_cloud):
        for point in point_cloud:
            pygame.draw.circle(self.map, self.sensor_color, point, 3, 0)


# Ultrasonic Sensor Class
class Ultrasonic:
    def __init__(self, sensor_range, map):
        self.sensor_range = sensor_range
        self.map_width, self.map_height = pygame.display.get_surface().get_size()
        self.map = map
        self.obstacle_color = (0, 0, 0)

    # sensing close obstacles
    def sense_obstacles(self, x, y, heading):
        # empty list to save the obstacles
        obstacles = []
        x1, y1 = x, y

        # ultrasonic sensor angle (limited to 40 degrees)
        start_angle = heading - self.sensor_range[1]
        finish_angle = heading + self.sensor_range[1]

        for angle in np.linspace(start_angle, finish_angle, 10, False):
            x2 = x1 + self.sensor_range[0] * math.cos(angle)
            y2 = y1 - self.sensor_range[0] * math.sin(angle)

            for i in range(0, 100):
                u = i/100
                x = int(x2 * u + x1 * (1-u))
                y = int(y2 * u + y1 * (1-u))

                if 0 < x < self.map_width and 0 < y < self.map_height:

                    color = self.map.get_at((x, y))
                    self.map.set_at((x, y), (0, 208, 255))

                    # check if the color is black (obstacle)
                    if (color[0], color[1], color[2]) == self.obstacle_color:
                        obstacles.append([x, y])
                        break
```

```python
        return obstacles
```

```python
import os
import math
import numpy as np
import pygame
from Robot import Graphics, Robot, Ultrasonic


clear = lambda: os.system('cls')  # On Windows System
clear()

# required inputs for map
Mam_Dimensions = (650, 1000)

# creating the environment
gfx = Graphics(Mam_Dimensions, "Robot Model.png", "Map.png")

# required input for the robot
start = (200, 200)
robot = Robot(start, 0.01*3779.52)

# required input for ultrasonic sensor
sensor_range = 250, math.radians(40)
ultra_sonic = Ultrasonic(sensor_range, gfx.map)

# keep track of the time lapse beteen loop iterations
time_step = 0
last_time = pygame.time.get_ticks()

running = True

# simulation loop
while running:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            # if clicked quit terminate the simulation
            running = False

    time_step = (pygame.time.get_ticks() - last_time)/1000
    last_time = pygame.time.get_ticks()
```

```
    # draw the map
    gfx.map.blit(gfx.map_img, (0, 0))

    # move the robot using the kinematics defined
    robot.kinematics(time_step)
    gfx.draw_robot(robot.x, robot.y, robot.heading)
    # finding obstacle
    point_cloud = ultra_sonic.sense_obstacles(robot.x, robot.y, robot.heading)
    # avoiding collision with obstacle
    robot.avoid_obstacles(point_cloud, time_step)
    gfx.draw_sensor_data(point_cloud)
    #update the screen
    pygame.display.update()
```