# Detailed Project Report (DPR)

## Main.py

1. Imports

- import math
  - This imports Python's built-in math module, which provides access to mathematical functions like radians.

- from Project1 import Graphics, Robot, Ultrasonic
  - This imports the Graphics, Robot, and Ultrasonic classes from the Project1 module, which are likely custom classes defined in another part of your project.
- import pygame
  - This imports the pygame library, which is used for creating games and simulations. It provides functionalities for handling graphics, events, and more.

2. Initialization

- MAP_DIMENSIONS=(600,1200)
  - Defines the dimensions of the map as 600 pixels in height and 1200 pixels in width.
- gfx=Graphics(MAP_DIMENSIONS,'robot.png','wall.png')
  - Initializes an instance of the Graphics class with the specified map dimensions and image files for the robot and walls.
- start=(200,200)
  - Sets the starting position of the robot to coordinates (200, 200).
- robot=Robot(start,0.01*3779.52)
  - Initializes a Robot object with the starting position and a conversion factor (possibly from inches to pixels or some other unit).
- sensor_range=250,math.radians(40)
  - Sets the sensor range for the ultrasonic sensor: 250 units distance and 40 degrees field of view (converted to radians).
- ultra_sonic=Ultrasonic(sensor_range,gfx.map)
  - Initializes an Ultrasonic sensor object with the defined range and the map from the Graphics object.

3. Main Loop

- dt=0
  - Initializes the delta time variable, which will keep track of the time between frames.
- last_time=pygame.time.get_ticks()
  - Records the current time in milliseconds.
- running=True
  - Sets a flag to keep the main loop running.
- while running:
  - Starts the main loop of the simulation.

4. Event Handling

- for event in pygame.event.get():
  - Processes events from the event queue.
- if event.type==pygame.QUIT:
  - Checks if the event is a quit event (e.g., the user closes the window).

- running=False
  - Stops the loop if a quit event is detected.

5. Simulation Logic

- dt=(pygame.time.get_ticks()-last_time)/1000
  - Calculates the time elapsed since the last frame (delta time) in seconds.
- last_time=pygame.time.get_ticks()
  - Updates the last recorded time.
- gfx.map.blit(gfx.map_img,(0,0))
  - Draws the map image onto the map surface at position (0, 0).
- robot.kinematics(dt)
  - Updates the robot's position based on its kinematics model using the elapsed time dt.
- gfx.draw_robot(robot.x,robot.y,robot.heading)
  - Draws the robot at its current position and heading.
- point_cloud=ultra_sonic.sense_obstacles(robot.x,robot.y,robot.heading)
  - Uses the ultrasonic sensor to detect obstacles around the robot and returns the detected points (point cloud).
- robot.avoid_obstacles(point_cloud,dt)
  - Updates the robot's behavior to avoid obstacles based on the sensed data.
- gfx.draw_sensor_data(point_cloud)
  - Draws the sensor data (point cloud) onto the map.
- pygame.display.update()
  - Updates the display to reflect the new frame.

Summary

This script simulates a robot navigating within a defined map, using kinematics for movement and an ultrasonic sensor for obstacle detection and avoidance. It continuously updates the robot's position, detects obstacles, and renders the graphics in a loop until the user closes the simulation.

# Project1.py

## Imports

1. import pygame

This imports the pygame library, which is used for creating and managing graphics, handling events, and managing game logic.

2. import math

This imports the math module, which provides access to mathematical functions such as cos, sin, and radians.

3. import numpy as np

This imports the numpy library and gives it the alias np. Numpy is used for numerical operations and array manipulation.

## Utility Function

```
def distance(point1,point2):
    point1=np.array(point1)
    point2=np.array(point2)
    return np.linalg.norm(point1-point2)
```

This function calculates the Euclidean distance between two points. The points are converted to numpy arrays, and np.linalg.norm computes the distance between them.

## Robot Class

The Robot class represents a robot with methods to handle movement and obstacle avoidance.

```
class Robot:
    def __init__(self,startpos,width):
        self.m2p=3779.52
        self.w=width
        self.x=startpos[0]
        self.y=startpos[1]
        self.heading=0

        self.vl=0.01*self.m2p
        self.vr=0.01*self.m2p

        self.maxspeed=0.02*self.m2p
        self.minspeed=0.01*self.m2p
```

```
        self.min_obs_dist=100
        self.count_down=5
```

Initialization (__init__ method): Sets up the initial state of the robot, including its position (x, y), heading, speeds of the left and right wheels (vl, vr), maximum and minimum speeds, and obstacle avoidance parameters.

```
def avoid_obstacles(self,point_cloud,dt):
    closest_obs=None
    dist=np.inf
    if len(point_cloud)>1:
       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 closest_obs is not None and closest_obs[1] < self.min_obs_dist and
self.count_down > 0:
       self.count_down -= dt
       self.move_backward()
    else:
       self.count_down = 5
       self.move_forward()
```

Obstacle Avoidance (avoid_obstacles method): Determines the closest obstacle from the point cloud. If an obstacle is within the minimum distance, the robot moves backward; otherwise, it moves forward.

```
def move_backward(self):
   self.vr=-self.minspeed
   self.vl=-self.minspeed/2

def move_forward(self):
   self.vr=self.minspeed
   self.vl=self.minspeed
```

Movement Methods: Define how the robot moves forward and backward by setting the speeds of the left and right wheels.

```
def kinematics(self,dt):
    self.x+=((self.vl+self.vr)/2)*math.cos(self.heading)*dt
    self.y-=((self.vl+self.vr)/2)*math.sin(self.heading)*dt
    self.heading+=(self.vr-self.vl)/self.w*dt

    if self.heading>2*math.pi or self.heading<-2*math.pi:
       self.heading=0

    self.vr=max(min(self.maxspeed,self.vr),self.minspeed)
    self.vl=max(min(self.maxspeed,self.vl),self.minspeed)
```

Kinematics: Updates the robot's position and heading based on its current speed and heading. Ensures the robot's heading remains within a valid range and constrains the wheel speeds.

## Graphics Class

The Graphics class handles rendering the robot and sensor data on the map.

```
class Graphics:
    def __init__(self,dimentions,robot_img_path,map_img_path):
        pygame.init()
        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)


        self.robot=pygame.image.load(robot_img_path)
        self.map_img=pygame.image.load(map_img_path)
        self.height,self.width=dimentions

        pygame.display.set_caption("Obstacle Avoidance")
        self.map=pygame.display.set_mode((self.width,self.height))
        self.map.blit(self.map_img,(0,0))
```

Initialization: Sets up the graphical interface, loads images for the robot and the map, and initializes the display window.

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

def draw_sensor_data(self,point_cloud):
    for point in point_cloud:
        pygame.draw.circle(self.map,self.red,point,3,0)
```

Drawing Methods: draw_robot draws the robot at its current position and orientation. draw_sensor_data visualizes the detected obstacles as red circles.

## Ultrasonic Class

The Ultrasonic class simulates an ultrasonic sensor for obstacle detection.

```
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
```

Initialization: Sets up the sensor's range and the map dimensions.

```
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
    def sense_obstacles(self,x,y,heading):
        obstacles=[]
        x1,y1=x,y
        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))
                    if(color[0],color[1],color[2])==(0,0,0):
                        obstacles.append([x,y])
                        break
        return obstacles
```

Sensing Obstacles (sense_obstacles method): Simulates the ultrasonic sensor by casting rays in a range of angles around the robot's heading. Detects obstacles by checking for black pixels on the map and returns their positions.

## Summary

This script provides a complete framework for simulating a robot navigating and avoiding obstacles within a graphical map. The Robot class handles the robot's movement and obstacle avoidance logic, the Graphics class manages the visualization, and the Ultrasonic class simulates the sensor's detection of obstacles.