|

|

## Task - 1

***Aim-***

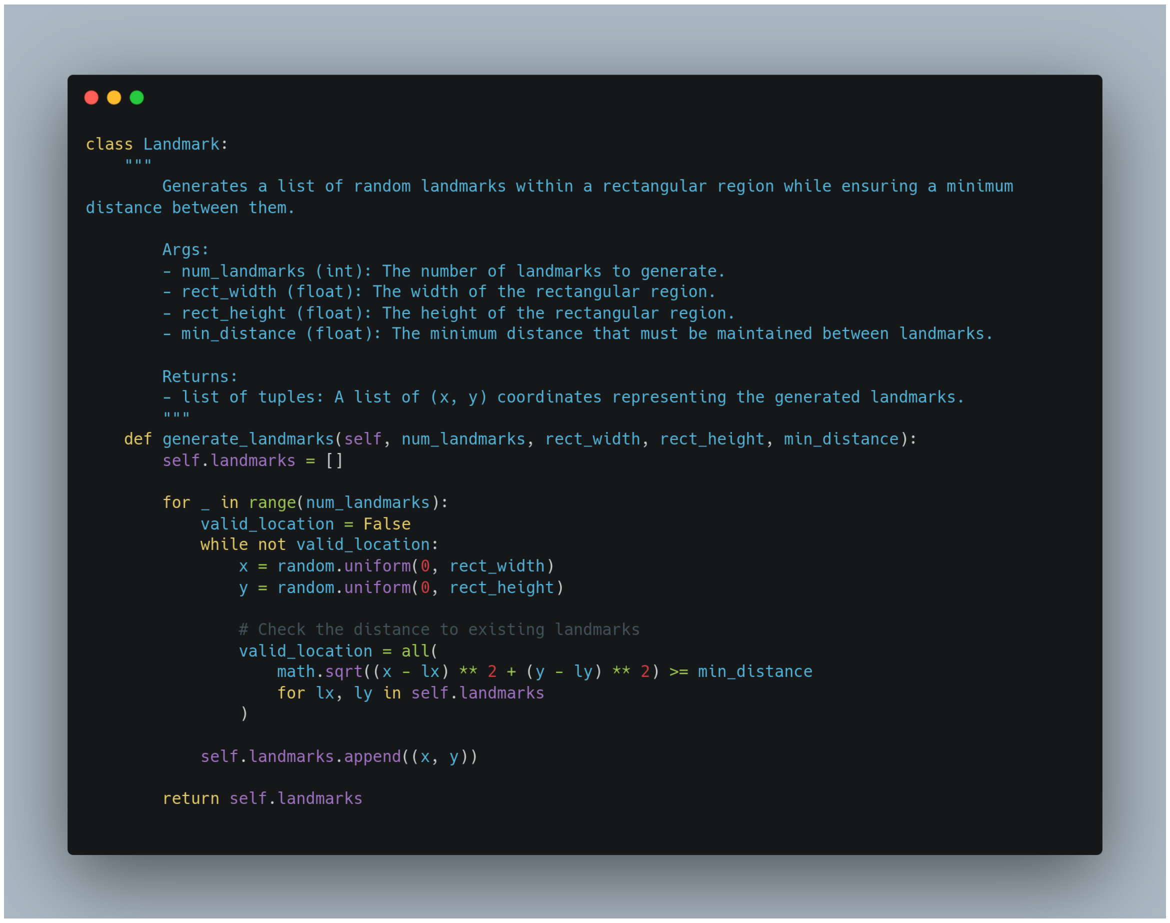
- To Model the problem and create a simulation environment to generate random goal locations consecutively for the robot's movement.
- Move from the current location to the generated goal location in a straight line and generate odometry according to the model given. Use a time interval of 0.1 seconds and simulate for 100 seconds.

***Approach -***

- First, we generate landmarks such that consecutive landmarks are 1m away from each other.
- Secondly, We store the landmark's locations and Draw them on the screen
- We pass these landmarks to move to goal function for the robot to be able to move form one landmark to another
- In move to Goal function we spawn the robot and calculate the odom of it w.r.t World frame.
- We give bot the specified angular and linear velocity.
- The robot first waits on a landmark and orients itself to the next landmark direction.
- Once that's complete the bot gains linear velocity to reach the goal in a staight line.
- Once the landmark is reached it gets it's next landmark location and repeats the above steps, all while constantly updating it's Odom frame.
- Once the last landmark is reached the code automatically exit's.
- Simulation time can be decreased and increased by manipulating the number of landmarks and speed.

***Code/functionalities -***

- **Class Landmark -**
  This code defines a class Landmark with a method generate_landmarks for generating random landmarks within a specified rectangular region while ensuring a minimum distance between them. The documentation briefly describes the purpose of the method, its arguments, and the returned result.



```
class Landmark:
    """
        Generates a list of random landmarks within a rectangular region while ensuring a minimum
distance between them.

        Args:
        - num_landmarks (int): The number of landmarks to generate.
        - rect_width (float): The width of the rectangular region.
        - rect_height (float): The height of the rectangular region.
        - min_distance (float): The minimum distance that must be maintained between landmarks.

        Returns:
        - list of tuples: A list of (x, y) coordinates representing the generated landmarks.
        """
    def generate_landmarks(self, num_landmarks, rect_width, rect_height, min_distance):
        self.landmarks = []

        for _ in range(num_landmarks):
            valid_location = False
            while not valid_location:
                x = random.uniform(0, rect_width)
                y = random.uniform(0, rect_height)

                # Check the distance to existing landmarks
                valid_location = all(
                    math.sqrt((x - lx) ** 2 + (y - ly) ** 2) >= min_distance
                    for lx, ly in self.landmarks
                )

            self.landmarks.append((x, y))

        return self.landmarks
```

- **Class Robot -**
  The Robot class has methods for generating odometry data, drawing the robot's position, and calculating the vector to a specified point. Below is a summary of the methods and their functionality:

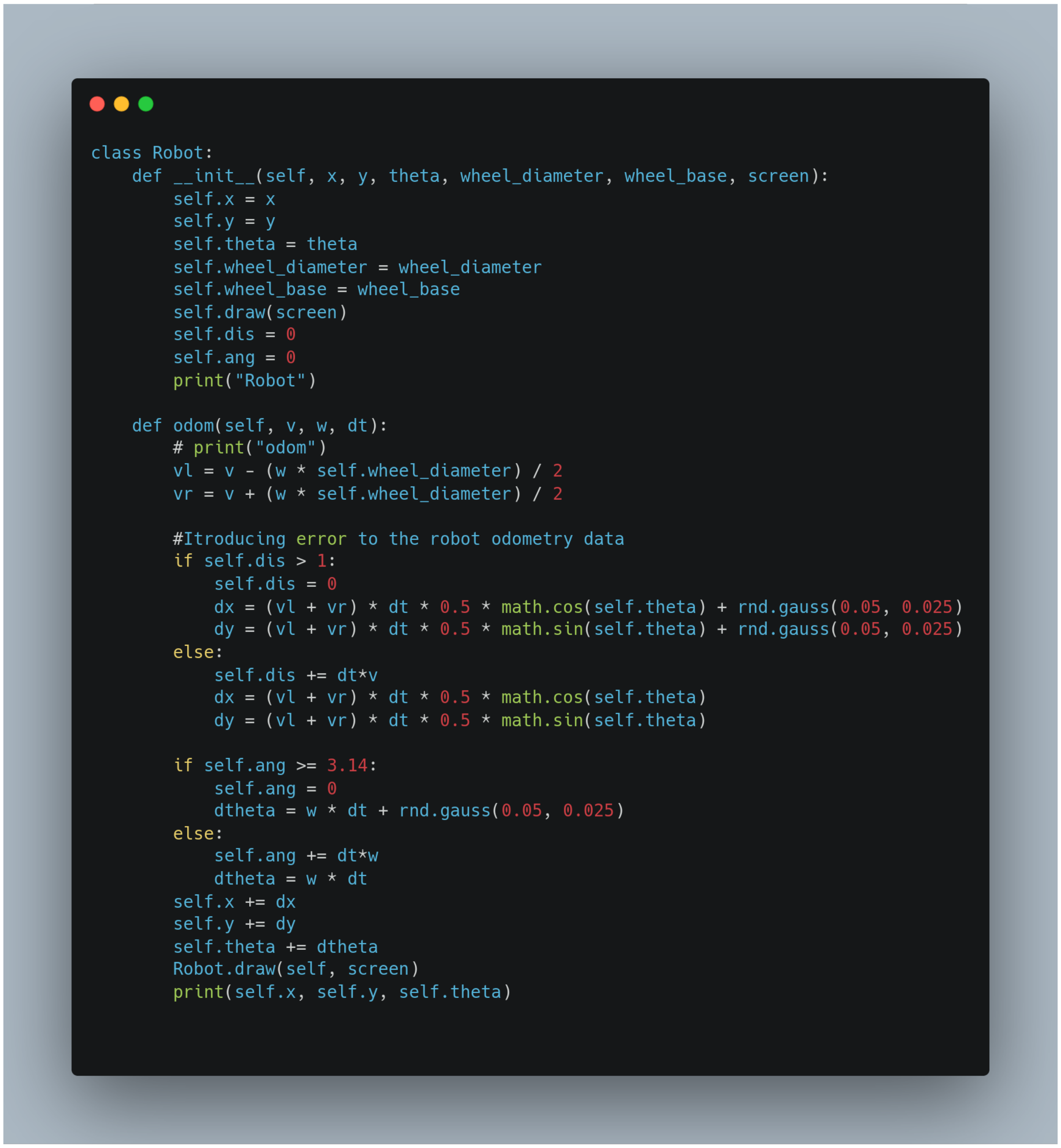
  **__init__:** The constructor initializes the robot's position (x, y), orientation (theta), wheel diameter, wheel base, and a Pygame screen. It also initializes variables for odometry data and prints "Robot."

  **odom:** This method simulates the robot's movement based on linear velocity (v), angular velocity (w), and the time step (dt). It introduces error in odometry data based on distance and angle traveled. It updates the robot's position and calls the draw method to visualize the new position on the screen.

  **draw:** This method updates the Pygame screen to display the robot's current position and orientation. It also draws landmarks as red circles.

**calc_vect:** This method calculates the vector (linear and angular velocity) required for the robot to reach a specified point (xf, yf). It takes into account the robot's current orientation and returns the necessary velocities.

*Please note that the code contains commented-out parts related to drawing the robot's body and rotating the image. You can uncomment and adapt these parts based on your visualization needs.*

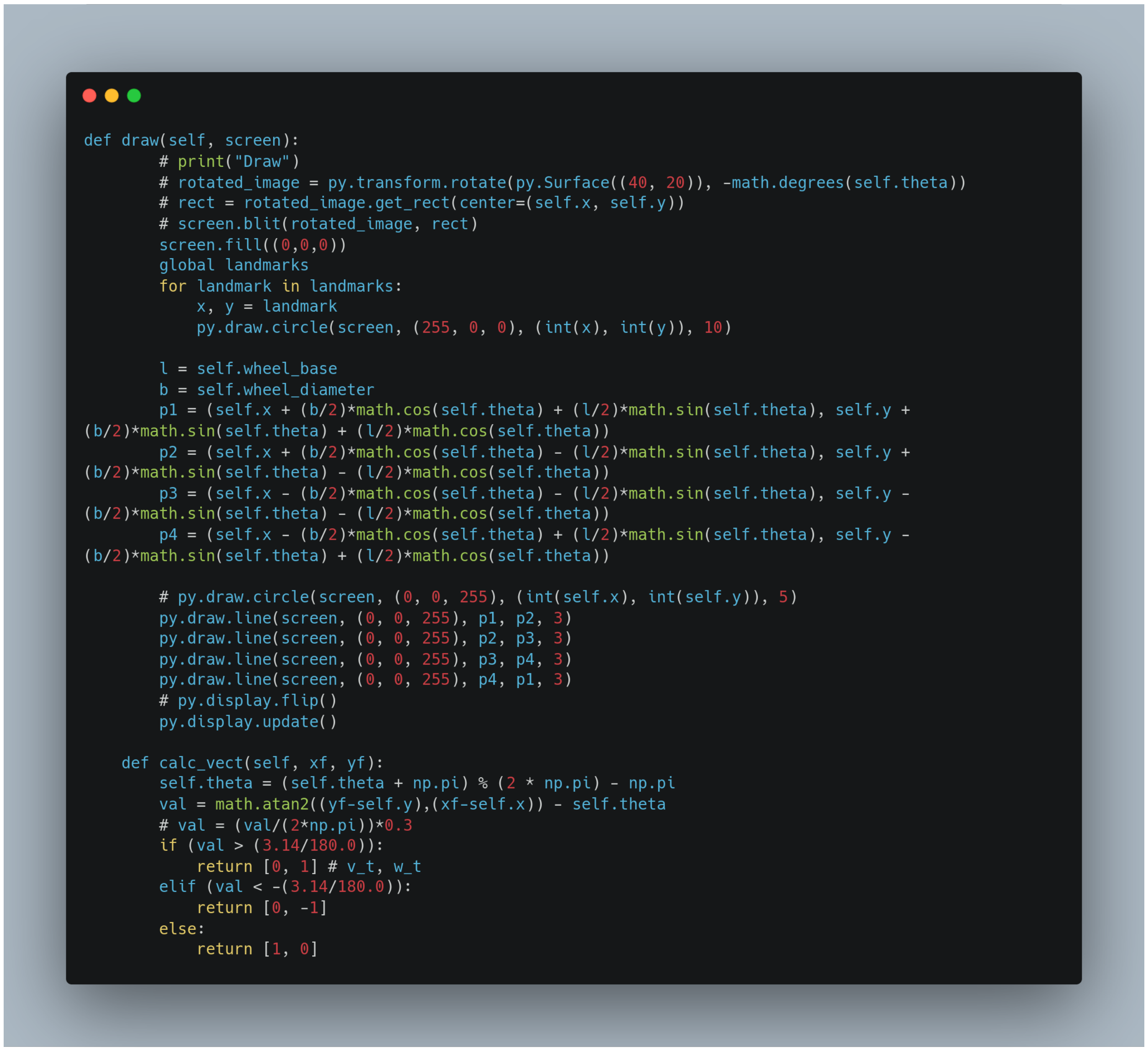


```python
class Robot:
    def __init__(self, x, y, theta, wheel_diameter, wheel_base, screen):
        self.x = x
        self.y = y
        self.theta = theta
        self.wheel_diameter = wheel_diameter
        self.wheel_base = wheel_base
        self.draw(screen)
        self.dis = 0
        self.ang = 0
        print("Robot")

    def odom(self, v, w, dt):
        # print("odom")
        vl = v - (w * self.wheel_diameter) / 2
        vr = v + (w * self.wheel_diameter) / 2

        #Itroducing error to the robot odometry data
        if self.dis > 1:
            self.dis = 0
            dx = (vl + vr) * dt * 0.5 * math.cos(self.theta) + rnd.gauss(0.05, 0.025)
            dy = (vl + vr) * dt * 0.5 * math.sin(self.theta) + rnd.gauss(0.05, 0.025)
        else:
            self.dis += dt*v
            dx = (vl + vr) * dt * 0.5 * math.cos(self.theta)
            dy = (vl + vr) * dt * 0.5 * math.sin(self.theta)

        if self.ang >= 3.14:
            self.ang = 0
            dtheta = w * dt + rnd.gauss(0.05, 0.025)
        else:
            self.ang += dt*w
            dtheta = w * dt
        self.x += dx
        self.y += dy
        self.theta += dtheta
        Robot.draw(self, screen)
        print(self.x, self.y, self.theta)
```



```
def draw(self, screen):
        # print("Draw")
        # rotated_image = py.transform.rotate(py.Surface((40, 20)), -math.degrees(self.theta))
        # rect = rotated_image.get_rect(center=(self.x, self.y))
        # screen.blit(rotated_image, rect)
        screen.fill((0,0,0))
        global landmarks
        for landmark in landmarks:
            x, y = landmark
            py.draw.circle(screen, (255, 0, 0), (int(x), int(y)), 10)

        l = self.wheel_base
        b = self.wheel_diameter
        p1 = (self.x + (b/2)*math.cos(self.theta) + (l/2)*math.sin(self.theta), self.y +
(b/2)*math.sin(self.theta) + (l/2)*math.cos(self.theta))
        p2 = (self.x + (b/2)*math.cos(self.theta) - (l/2)*math.sin(self.theta), self.y +
(b/2)*math.sin(self.theta) - (l/2)*math.cos(self.theta))
        p3 = (self.x - (b/2)*math.cos(self.theta) - (l/2)*math.sin(self.theta), self.y -
(b/2)*math.sin(self.theta) - (l/2)*math.cos(self.theta))
        p4 = (self.x - (b/2)*math.cos(self.theta) + (l/2)*math.sin(self.theta), self.y -
(b/2)*math.sin(self.theta) + (l/2)*math.cos(self.theta))

        # py.draw.circle(screen, (0, 0, 255), (int(self.x), int(self.y)), 5)
        py.draw.line(screen, (0, 0, 255), p1, p2, 3)
        py.draw.line(screen, (0, 0, 255), p2, p3, 3)
        py.draw.line(screen, (0, 0, 255), p3, p4, 3)
        py.draw.line(screen, (0, 0, 255), p4, p1, 3)
        # py.display.flip()
        py.display.update()

    def calc_vect(self, xf, yf):
        self.theta = (self.theta + np.pi) % (2 * np.pi) - np.pi
        val = math.atan2((yf-self.y),(xf-self.x)) - self.theta
        # val = (val/(2*np.pi))*0.3
        if (val > (3.14/180.0)):
            return [0, 1] # v_t, w_t
        elif (val < -(3.14/180.0)):
            return [0, -1]
        else:
            return [1, 0]
```

- **Function Move_to_goal:**
  **move_to_goal** function *moves a robot to a series of goal locations (landmarks) using a simple control strategy based on linear and angular velocity commands*. Here's a summary of how the function works:

  The function takes two arguments: landmarks, a list of goal locations, and screen, a Pygame screen where the robot's motion is visualized.

  It initializes a Robot object with an initial position at the first landmark (m, n), zero orientation, and specified wheel parameters.

The linear velocity (v) and angular velocity (w) are set to control the robot's motion. dt is the time step used for simulation.

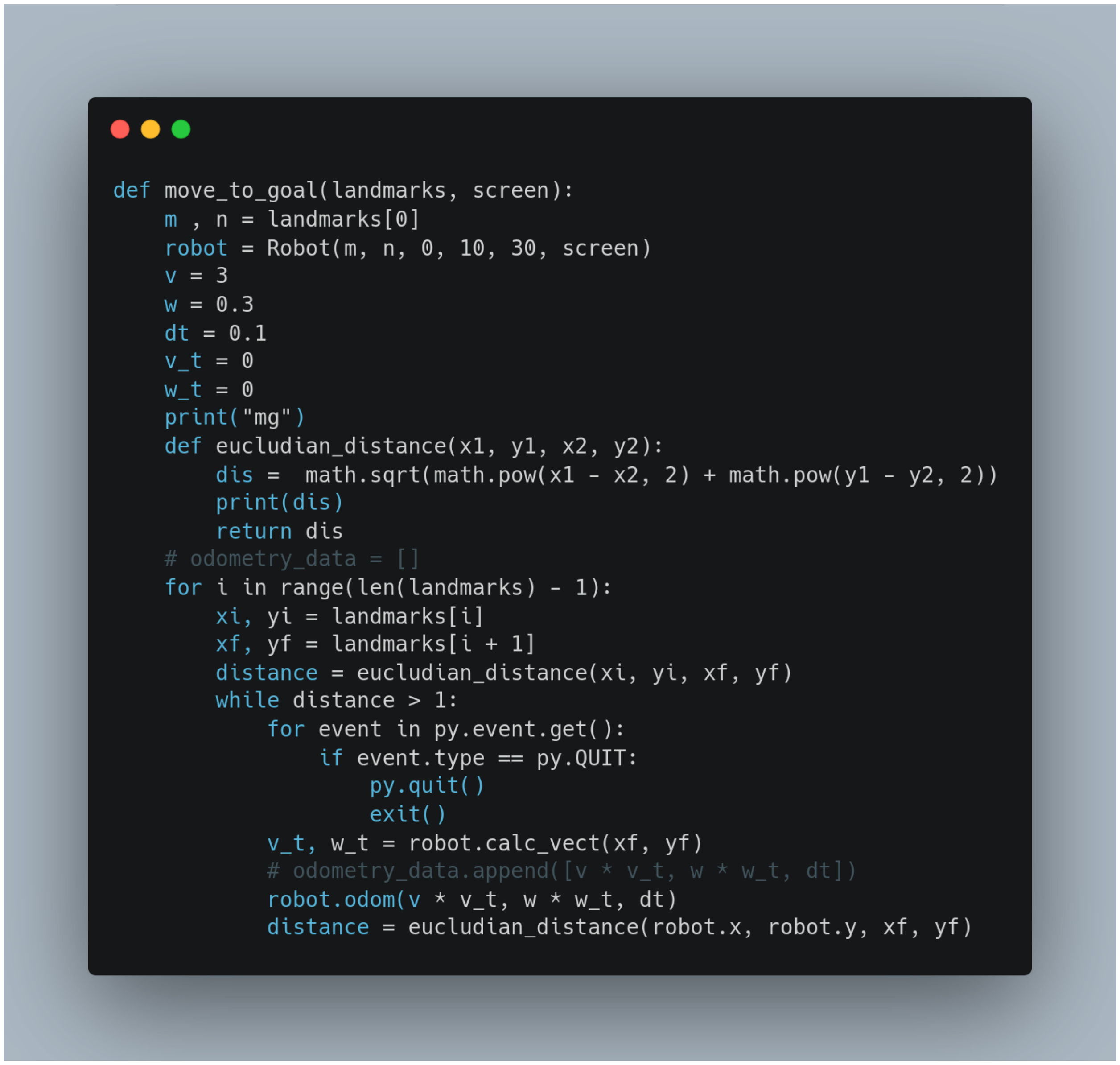
A nested function, euclidean_distance, calculates the Euclidean distance between two points (x1, y1) and (x2, y2). This function is used to check if the robot has reached the goal location.

The main loop iterates through the landmarks. For each pair of adjacent landmarks, it calculates the distance between them using euclidean_distance.

While the robot's distance to the next goal (xf, yf) is greater than 1 (indicating it hasn't reached the goal), it does the following:

Checks for any Pygame quit events.
Calculates linear and angular velocity commands (v_t and w_t) using the calc_vect method of the Robot class. This method computes the velocity needed to reach the goal while considering the robot's orientation.
Calls the odom method of the Robot class to update the robot's position and orientation based on the calculated velocities.
The loop continues until the robot reaches the last landmark, at which point it has moved through the entire list of landmarks.

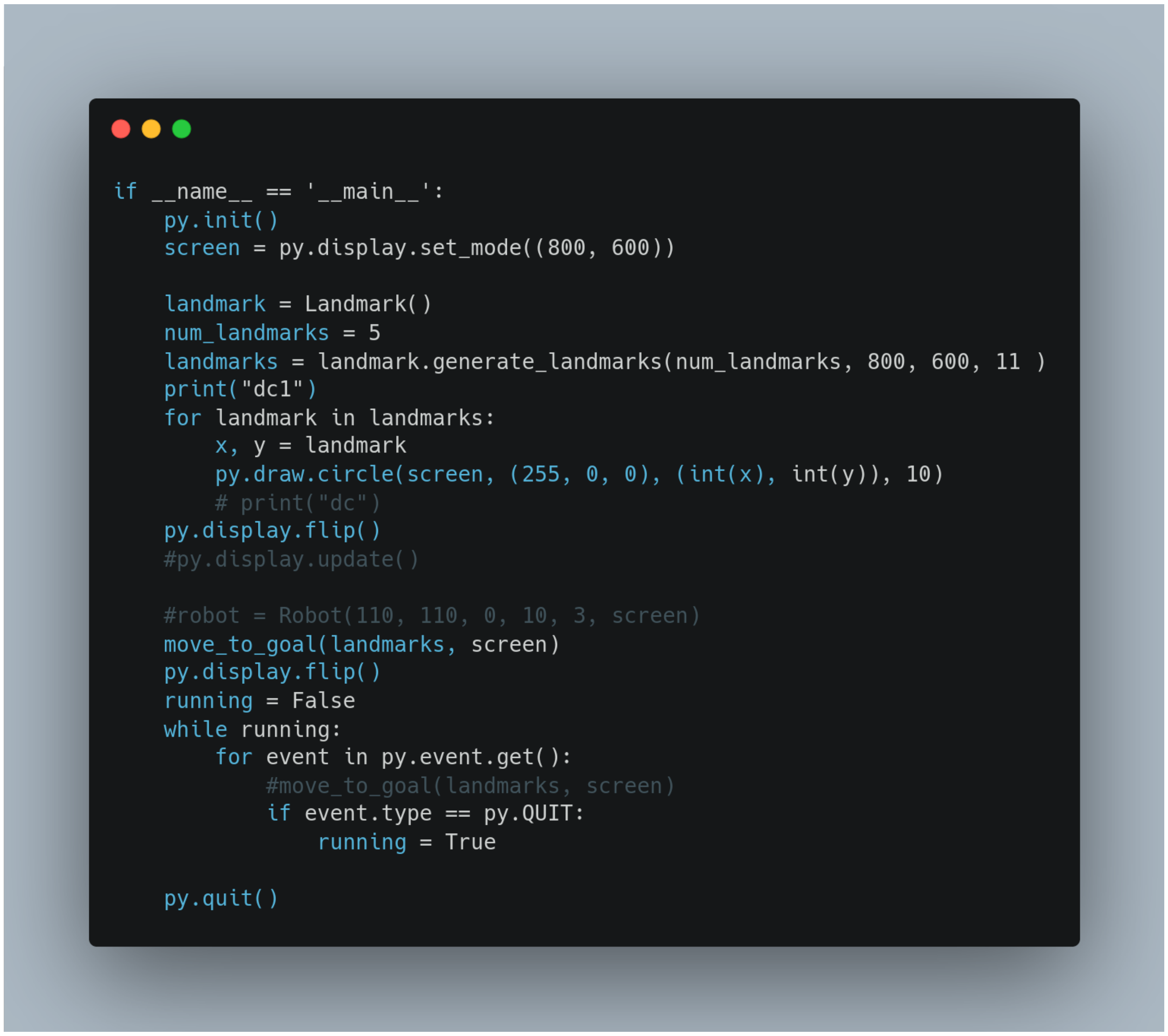
The function is designed to simulate the robot's movement from one landmark to another based on the calculated velocity commands. The actual motion and visualization occur within the Robot class and the Pygame environment.



```python
def move_to_goal(landmarks, screen):
    m , n = landmarks[0]
    robot = Robot(m, n, 0, 10, 30, screen)
    v = 3
    w = 0.3
    dt = 0.1
    v_t = 0
    w_t = 0
    print("mg")
    def eucludian_distance(x1, y1, x2, y2):
        dis =  math.sqrt(math.pow(x1 - x2, 2) + math.pow(y1 - y2, 2))
        print(dis)
        return dis
    # odometry_data = []
    for i in range(len(landmarks) - 1):
        xi, yi = landmarks[i]
        xf, yf = landmarks[i + 1]
        distance = eucludian_distance(xi, yi, xf, yf)
        while distance > 1:
            for event in py.event.get():
                if event.type == py.QUIT:
                    py.quit()
                    exit()
            v_t, w_t = robot.calc_vect(xf, yf)
            # odometry_data.append([v * v_t, w * w_t, dt])
            robot.odom(v * v_t, w * w_t, dt)
            distance = eucludian_distance(robot.x, robot.y, xf, yf)
```

- **The __name__ == ‘__main__’**
  Pygame to create a simulation of a robot moving between landmarks generated by a class called Landmark. The program initializes Pygame, generates landmarks, and then calls the move_to_goal function to move the robot between these landmarks. However, there are a few issues with the code:

  The while running loop is set to False at the beginning, which means that the loop will not execute. You should set running = True initially to ensure that the loop runs.
  The loop for handling Pygame events is missing the running = False line to exit the loop when the window is closed. It should be running = False instead of running = True to correctly exit the loop.



```
if __name__ == '__main__':
    py.init()
    screen = py.display.set_mode((800, 600))

    landmark = Landmark()
    num_landmarks = 5
    landmarks = landmark.generate_landmarks(num_landmarks, 800, 600, 11 )
    print("dc1")
    for landmark in landmarks:
        x, y = landmark
        py.draw.circle(screen, (255, 0, 0), (int(x), int(y)), 10)
        # print("dc")
    py.display.flip()
    #py.display.update()

    #robot = Robot(110, 110, 0, 10, 3, screen)
    move_to_goal(landmarks, screen)
    py.display.flip()
    running = False
    while running:
        for event in py.event.get():
            #move_to_goal(landmarks, screen)
            if event.type == py.QUIT:
                running = True

    py.quit()
```

***Limitations and scope of future work-***

- In the draw function the robot can be modeled to look more real.
- We have introduced the errors in odom but haven't filtered it in this code atleast.

**This Document Henceforth provides the full documentation of the Task-1 code, assumptions made, available functionalities, limitations, and scope of future work.**

**Thanks for reading you can find the simulation code in the code folder, and the simulation recording in the ss/recodings folder named according to their respective task name.**