Toronto Metropolitan University

Department of Aerospace Engineering
Faculty of Engineering & Architectural Science

Semester (Term, Year)	Winter, 2025
Course Code	AER627
Course Section	02
Course Title	Introduction to Space Robotics
Course Instructor	Dr. Anton de Ruiter
Submission	
Submission No.	2
Submission Due Date	2025-04-16
Title	Project 4 Report
Submission Date	2025-04-15

Submission by (Name):	Student ID (XXXX1234)	Signature
Octavio Guerra	XXXXX7207	O.G
Elliott Arpino	XXXXX5958	E.A.

By signing the above you attest that you have contributed to this submission and confirm that all work you contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, and "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Academic Integrity Policy 60 , which can be found at www.torontomu.ca/senate/policies /

1 Contents

2 Introduction

This objective of this project is to build and program a two-wheel differential rover. The rover design is obtained from the Lego website which provides simple steps to build the rover. The task of the rover is object avoidance; the rover is intended to start from its initial position and avoid obstacles until it reaches its ‘goal’ position. For the rover to complete a perfect trajectory, multiple steps are required in the programming phase. It is necessary to integrate rover kinematics and control system techniques to control the motion of the rover, it is also necessary to use AprilTags to obtain a scene which contains an origin, and the rovers and obstacles pose with respect to the origin. The method use for object avoidance is potential field planning technique; this method assigns an ‘attractive’ force to the objective position and repulsive forces to the obstacles; the rover is intended to follow the path with the least local potential field strength, which may not be the shortest distance – however this method is used for its robustness. The following sections outline the steps taken in this project from the rover design to the trajectory of the rover, emphasis is put in the programming section as it requires understanding of multiple areas of robotics.

3 Theory

This project combines theoretical knowledge of rover kinematics for a two-wheel differential rover, potential field path planner algorithms and knowledge of Matlab programming skills

For the rover, two large motors were used to control each wheel, the robot requires obstacles avoidance; hence it is required for the rover to turn. For the robot to be able to perform motion that requires turning, differential wheel kinematics needs to be implemented in the programming of the robot. For instance, if it is desired to turn left, then the right wheel must rotate quicker than the left wheel, producing a rotational motion to the left, the opposite is true. The governing equations for differential kinematics are found in Appendix A.

Once the rover scene is determined it is necessary to determine a path that the rover can take to reach the desired end location. There are several methods by which a path can be determined, those which particularly require the shortest distance to be taken are more difficult to implement as more variables ought to be considered. For this project, the potential field algorithm was used as the method to navigate through the obstacles. The potential field algorithm creates fictitious fields around the obstacles and desired end location. A repelling force is generated for obstacles, this produces the rover to avoid the obstacles while the desired location has the lowest possible potential field making the rover drive in towards it. The path is determined based on the one that has the lowest potential.

A control system was used for the rover. Figure 4.1 shows a block diagram for a PID controller, this controller uses potential field path planning and heading to control the rover as well as the kinematics of the rover. The controller assesses an error between the reference and actual heading of the rover, this is used to compute the angular velocity which is then used to generate rover commands through rover kinematics

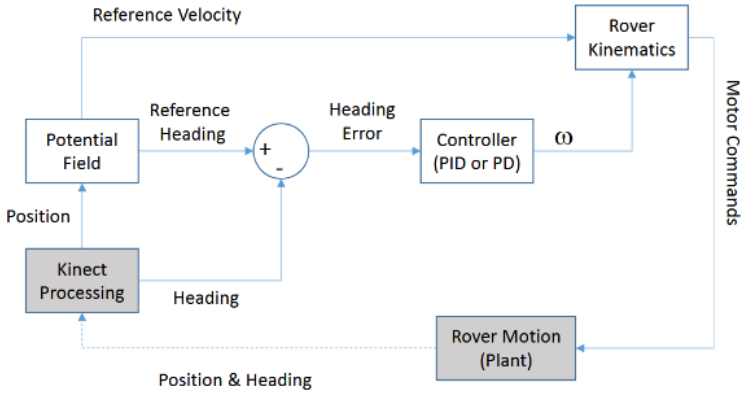

Figure 4.1: Controller

4 Package A

4.1 Code Design

This section outlines the design of the steering controller using in this project, which is stored in Appendix C Part 3. The steering controller is split into 2 parts: 3.a - Controller Settings and 3.b – Open Loop PD Control along Potential Path. This controller is integral to the rover's navigation system, enabling it to follow a predefined path derived from a potential field planner while ensuring stable and safe operation. Below, the key design elements of the steering controller are defined, focusing on its configuration and the implementation of an open-loop Proportional-Derivative (PD) control strategy.

4.1.1 Controller Settings

The controller settings define how the rover interacts with its hardware and establish the parameters for effective control. The controller connects to the LEGO EV3 brick, managing the left wheel with motor 'A' and the right wheel with motor 'D'. To ensure accurate motion calculations, the rover's physical characteristics are specified, including a wheel radius of 2.8 cm and a distance between wheels of 12 cm. Speed settings are carefully configured to balance performance and safety. The base forward speed is set at 50 cm/s, but a maximum forward speed cap of 30 cm/s and a velocity scale factor of 0.8 are applied to maintain smooth and controlled operation. Additionally, the maximum wheel speed is limited to 300 degrees per second to prevent motor overload.

The Proportional-Derivative (PD) controller is tuned with a proportional gain of 1.5 and a derivative gain of 1, which work together to minimize heading errors during navigation. A heading error threshold of 2 degrees (converted to radians) determines when the rover is sufficiently aligned with its target direction. To ensure an adequate system response, the controller updates every 0.01 seconds for precise adjustments, and the initial heading is set to 0 radians, aligning with the starting coordinate system.

4.1.2 Open-Loop PD Control along Potential Path

The open-loop PD control strategy directs the rover along a 'sparsified' potential field path, scaled by a factor of 0.1 and converted from meters to centimeters for compatibility with the rover's units. The rover navigates by following a series of waypoints extracted from this path. For each segment between consecutive waypoints, the controller executes a two-step process.

1. Heading Alignment:

   The desired heading is determined by calculating the arctangent of the coordinate differences between the current and next waypoints. The PD controller then computes an angular velocity command based on the heading error, which is normalized to stay within ±180 degrees. This command adjusts the wheel speeds differentially but overall turning the rover by driving the left and right wheels at the same rates, while ensuring the speeds do not exceed the maximum limit. The motors operate in brief 0.01 second bursts, repeating until the heading error falls below the threshold. In terms, of changing directions, this is done by rotating the wheels at the same speed, however in opposite directions.
2. Forward Motion:

   Once aligned, the rover moves forward at a controlled speed, starting from the base speed but adjusted by the velocity scale factor and capped at 30 cm/s. The controller calculates the time needed to cover the segment distance based on this speed. Both motors are then set to the same speed to drive the rover forward, with the speed again capped for safety. The motors run for the calculated duration, followed by a short pause to mimic real-time adjustments.

This sequence repeats for each waypoint until the rover reaches its final destination, at which point the motors stop.

4.2 Tuning and Performance Limitations

The steering controller's performance relies heavily on the precise tuning of its Proportional-Derivative (PD) controller, which ensures the rover aligns its heading with the desired direction efficiently. This section details the tuning process for the PD controller's gains, (K_p) and (K_d) and examines the performance limitations encountered during testing, along with the solutions implemented to address them.

4.2.1 Tuning the PD Controller

The PD controller is essential for minimizing heading errors as the rover navigates between waypoints. The proportional gain (K_p) controls the response to the current heading error, while the derivative gain (K_d) dampens the system to prevent overshooting or oscillations. Proper tuning of these parameters ensures quick and stable heading alignment.

Initial testing revealed two key challenges:

- **Slow Convergence:** Low (K_p) values caused the rover to take excessive time to reach the desired heading, resulting in inefficient navigation.
- **Oscillations:** High (K_p) values without adequate (K_d) damping led to oscillations around the target heading, preventing stable alignment.

An iterative tuning process was employed, adjusting K_p and K_d incrementally and observing the rover's behavior across multiple test runs. The objective was to achieve rapid convergence without instability. After extensive trials, the optimal values were:

- **Proportional Gain (K_p):** 1.5
- **Derivative Gain (K_d):** 1.0

These values enabled the rover to align its heading smoothly and efficiently, balancing speed and stability.

4.2.2 Performance Limitations and Solutions

Despite successful tuning, some limitations emerged during testing, primarily related to path planning and hardware constraints.

1. **Excessive Waypoints:**
    - The potential field path planner initially produced a dense set of waypoints, causing the rover to make frequent, minor adjustments. This inefficiency prevented the rover from reaching its goal in a timely manner, as it became trapped in a cycle of small corrections.
    - **Solution:** A segment threshold of 0.1 meters was introduced to sparsify the path, retaining only waypoints separated by this minimum distance. This reduced the waypoint count, allowing the rover to focus on significant path segments and improving navigation efficiency.
2. **Hardware and Scenario Constraints:**
    - **Wheel Friction:** The rover's rubber wheels generated significant friction on the cherry hardwood floor, impacting precise turns and consistent speed maintenance.
    - **Motor Inconsistency:** Variability in the motors' performance between trials introduced unpredictability in movement, especially during forward motion.
    - **Camera Calibration Errors:** Minor inaccuracies in the camera's calibration parameters led to small errors in the scene's coordinate system, affecting waypoint and obstacle positioning.

    To address these issues:
    - A velocity scale factor of 0.8 was applied to lower forward speed, enhancing control on the frictional surface.
    - The maximum wheel speed was capped at 300 degrees per second to prevent motor overload and mitigate speed inconsistencies.
    - Camera errors did occur, along with troubles from creating the occupancy map using the MATLAB function. To eradicate this, the coordinates for the obstacles on the occupancy map were retrieved from the scene renderings and hardcoded.

3. **Path Planning Trade-offs:**
    - The potential field planner prioritized obstacle avoidance over shortest-path efficiency, sometimes resulting in longer routes. This trade-off was accepted for its robustness.
    - Conversely, the A* planner offered shorter paths but risked bringing the rover too close to obstacles, increasing collision potential given hardware limitations.

    The potential field approach was ultimately favored for its safety margin, maintaining adequate distance from obstacles despite movement or perception uncertainties.

4.2.2.1 Conclusion

Tuning the PD controller with $K_p = 1.5$ and $K_d = 1.0$ achieved stable and efficient heading alignment through iterative testing. However, challenges such as excessive waypoints and hardware limitations necessitated adjustments like path sparsification and speed scaling. These measures enhanced the rover's navigation reliability, though they underscored the difficulties of precise control in a real-world setting with mechanical and perceptual constraints. Overall, the steering controller effectively guided the rover to its goal across multiple scenarios, validating the integrated planning and control approach.

5 Package B

This section of the report describes the algorithm implemented to the rover in order to avoid obstacles. The algorithm is based on the potential field path planning, the potential field method was previously mentioned and briefly described. For this project, the algorithm was tested for different scenes to understand its behavior under different scenarios, see where it could be improved, its limitations and its overall performance.

The develop the scene AprilTags were used, a total of four AprilTags were needed to generate a scense. As it can be seen in Figure 6.1, there are 4 AprilTags, the first AprilTag is on the bottom left of the image marked with an organe ‘x’, this AprilTag sets the origin of the scene. The second AprilTag sits on top of the rover, this AprilTag sets the location of the rover with respect to the origin. The other two AprilTags set the location of the two obstacles, these are arbitrarily placed in the scene. The scene is captured from the LabVision using an iphone camara which was connected to Matlab.

Once the scene was determined a rendered plot was generated as seen in Figure ii. This plot provides more insight into the scene, it shown the origin, the location of the rover and the location of the two obstacles. Furthermore, the plot shows the origin of each obstacle and rovers reference system. The render plot is useful to generate an occupancy map, which is needed to then generate the potential field path planning. The occupancy map which is shown in Figure iii for the first scene shows the two obstacles that the rover will have to avoid. In order to generate the occupancy map, the coordinates of the obstacles were hard coded as the software was not being receptive to importing the coordinates directly from the LabVision, this made the process slower and less automated.

The first method used to determine the path planning of the rover is potential field. This method assigns an attractive force to the goal and a repelling force to the obstacles. This method provides robustness as it ensures that rover doesn’t come close to the obstacles, however this planning method does not provide the shortest path to the goal. In Figure 6.4, the potential path planning for the first scene can be seen, the figure shows outward arrows from the obstacles indicating strong potential field, while at locations further from the obstacles the potential field is not strong, this drives the rover in those direction with small potential field.

The second path planning method used is the A* method. This method finds shortest path from the origin of the rover to the goal while avoiding the obstacles. However, this method is not

completely succesful at avoiding obstacles, it can be seen from Figure 6.5 that the path generated comes close to the obstacles, this is not adequate behavior as there is risk of collision with the obstacles. Lastly, Figure 6.7 shows the path the rover executed, it can be seen that it was successful at avoiding the obstacles and reaching the goal.

Two more scenes were developed and tested which were also succesfful showing that the method of potential field path planning works and is robust. The images for the other scenes can be seen in **Appendix A**.

5.1 Scene 1

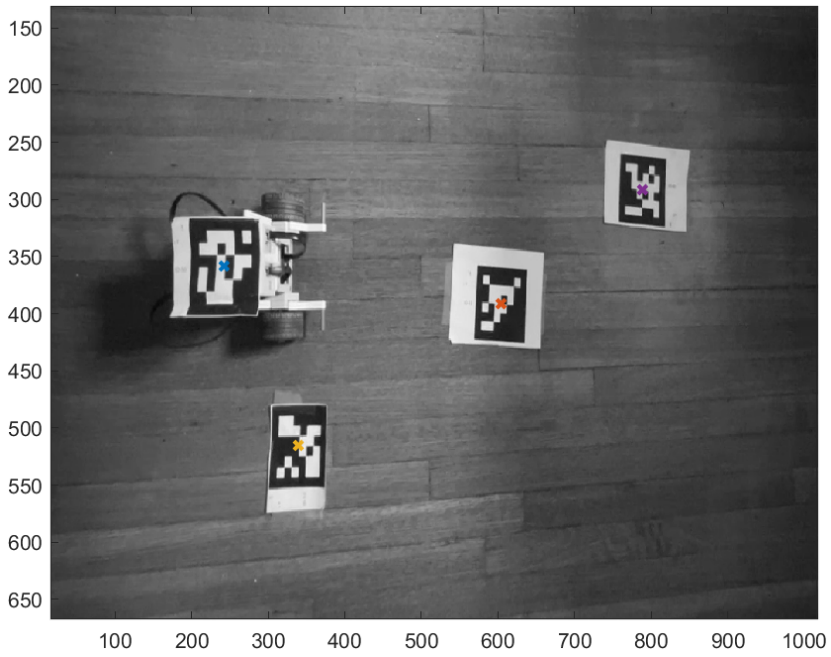

Figure 5.1: Scene 1

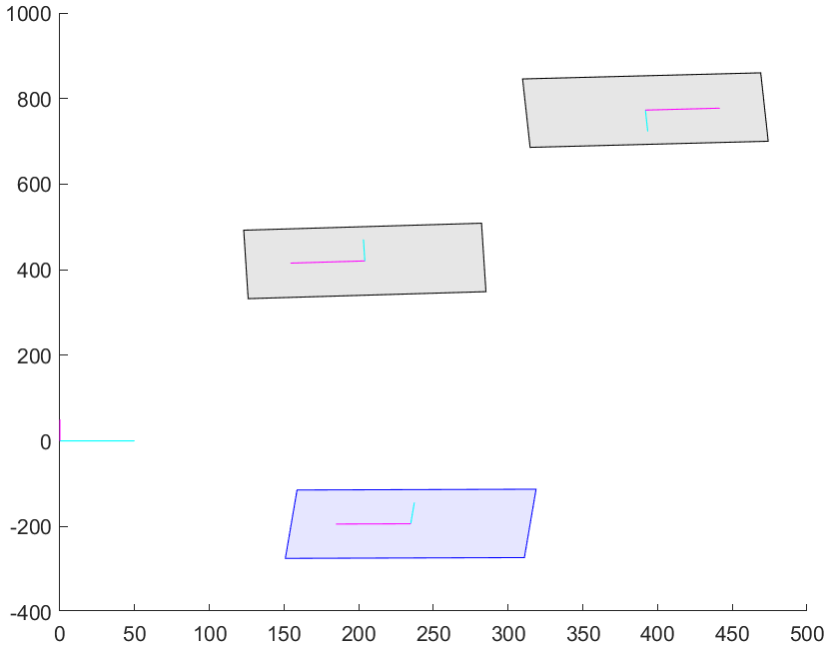

Figure 5.2: *Rendered Scene 1*

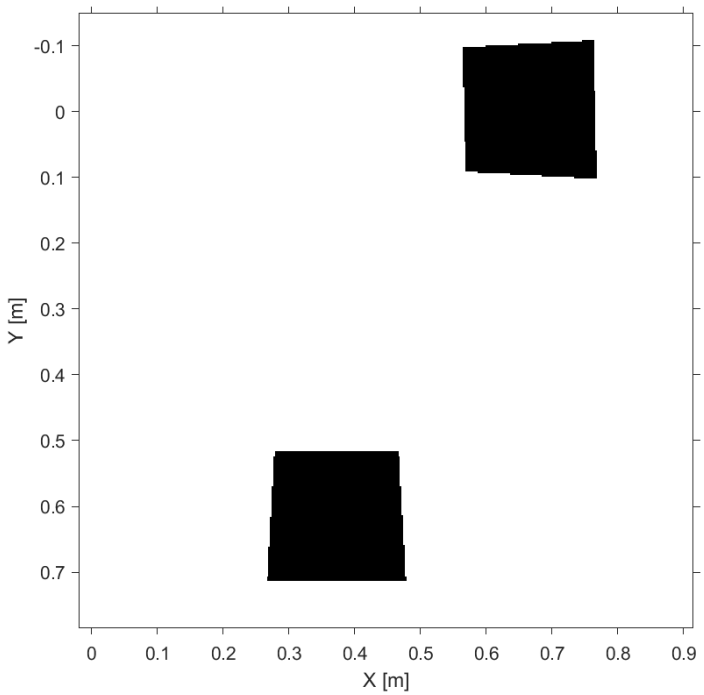

Figure 5.3: *Occupancy Map 1*

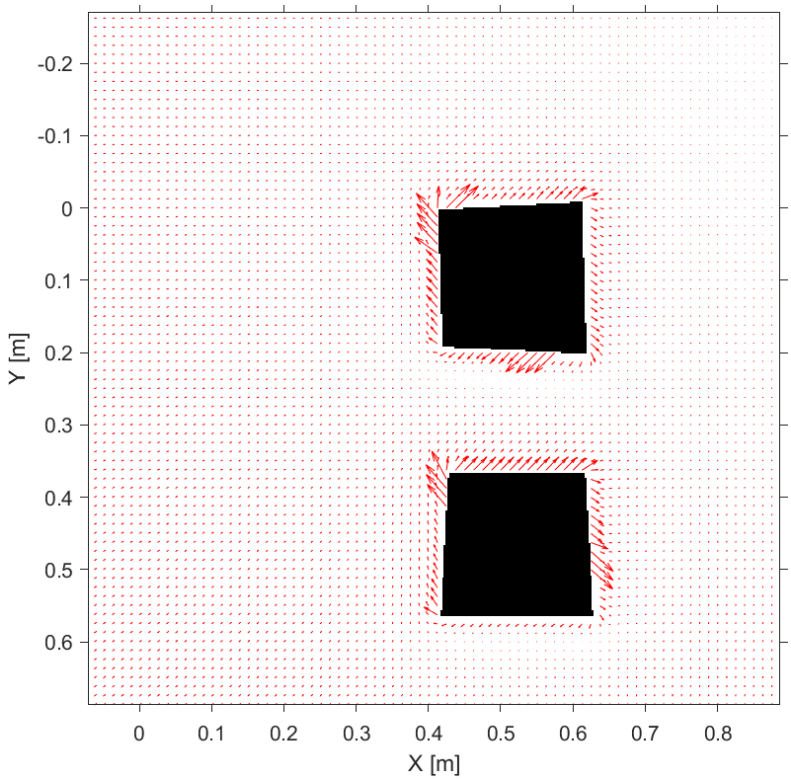

Figure 5.4: *Potential Field 1*

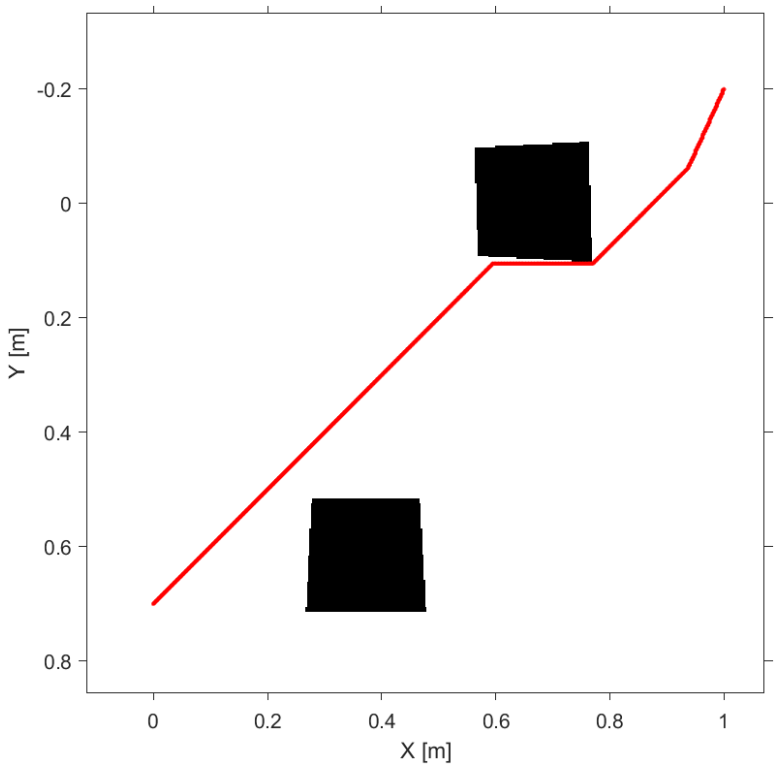

Figure 5.5*: A* Path Planning 1*

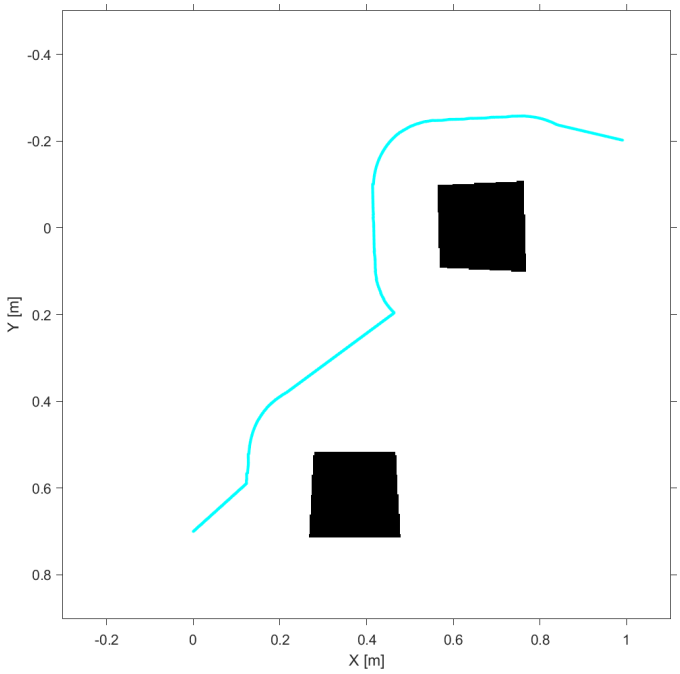

Figure 5.6: *Potential Field Path 1*

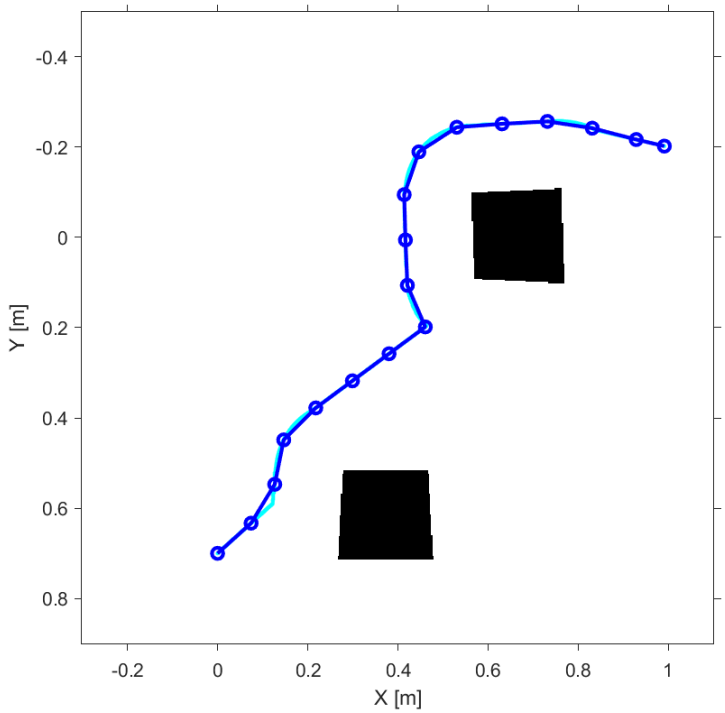

Figure 5.7: Rover Path 1

6 Project Operations

6.1 Part A

The potential field planner produces a gradient path using the *followGradient* function. This function considers all the forces using k_{att}, k_{rep}, and k_{rot}, along with the obstacles, to calculate the shortest path of the rover from start to finish. This path is then *sparsified* – creating waypoints for the rover to reach keeping it along the gradient path.

These waypoints are then used to compute motor commands for the rover. For each path segment, the desired heading is computed using the angle between the next waypoints. A Proportional-Derivative (PD) controller helps align the robots current heading with the desired heading before proceeding forward. This was done using the following formula.

$$\omega_{cmd} = K_p \epsilon_0 K_d \frac{d(\epsilon_0)}{dt}$$

Once the robot is aligned, it drives forward by a length equal to the distance of the next waypoint. A fixed forward velocity is used. Once the robot reaches the waypoint – this process is repeated until all the waypoints are reached.

The following modifications were made to achieve good performance:

- The robot was either taking too long to reach the desired heading or it would oscillate and never reach the desired heading. To solve this problem, the proportional gain (K_p) and (K_d) were adjusted many times until the robot smoothly and efficiently reached the desired heading. This occurred when K_p = 1 and K_d = 1.5.
- There were too many waypoints. The robot would never reach its goal attempting to hit all the waypoints. To solve this problem a segment threshold was introduced. A condition that would only retain waypoints separated by a minimum distance. Increasing this value decreased the amount of waypoints.

6.2 Part B

During the testing phase of this project there were several hardware issues encountered as well as limitations due to the equipment used and the environment. One of the major issues are the mechanical limitations that the rover itself presents. Some of these limitations, are wheel friction. The test was performed on a cherry hardwood floor, the wheels being rubber generate more friction on a rubber floor than on other surfaces such as plastic or metal. Another issue is the motors, the motors seem to not always run at a constant speed, it was seen that from one trial to another the motors would behave differently. Furthermore, another hardware issues that affects the overall performance is that the connections are by cable, this provides a lag which should be corrected using a correction factor.

An issue that affects the accuracy of the scene generation is the calibration of the camara. Even though the camara calibration was performed correctly, there are sources of error in the calibration which affects the location of the obstacles, origin of the scene and location of the rover.

6.3 Part C

To test the reliability and navigational capabilities of the rover, 10 different trials were performed under different scene scenarios.

Scene	Obstacle Avoidance 1	Obstacle Avoidance 2	Goal
1	Fail	Fail	Fail
2	Fail	Fail	Fail
3	Fail	Fail	Fail
4	Fail	Fail	Fail
5	Fail	Fail	Fail
6	Fail	Success	Fail
7	Success	Fail	Fail
8	Success	Success	Success
9	Success	Success	Success
10	Success	Success	Success

Many trials were conducted to achieve proper execution. The rover was originally unable to accomplish any of the goals. However, after introducing scale factors, a segment threshold for the sparse path, and adjusting the gain values, the robot was highly successful and able to maneuver smoothly around any type of scene.

7 Conclusion

This project demonstrated the successful integration of planning and control for a two-wheel differential rover tasked with obstacle avoidance. By combining accurate scene reconstruction with a potential field path planning and a tuned PD controller, the rover was able to navigate toward a goal while avoiding obstacles in various scenarios. Although initial trials faced challenges due to waypoint density, motor inconsistencies, and calibration errors, iterative adjustments—such as gain tuning, speed scaling, and conservative obstacle handling led to significant improvements in performance. Ultimately, the rover achieved reliable navigation, validating the robustness of the potential field method and the effectiveness of the overall control strategy in a real-world environment.

Appendix A – Scenes

Scene 2

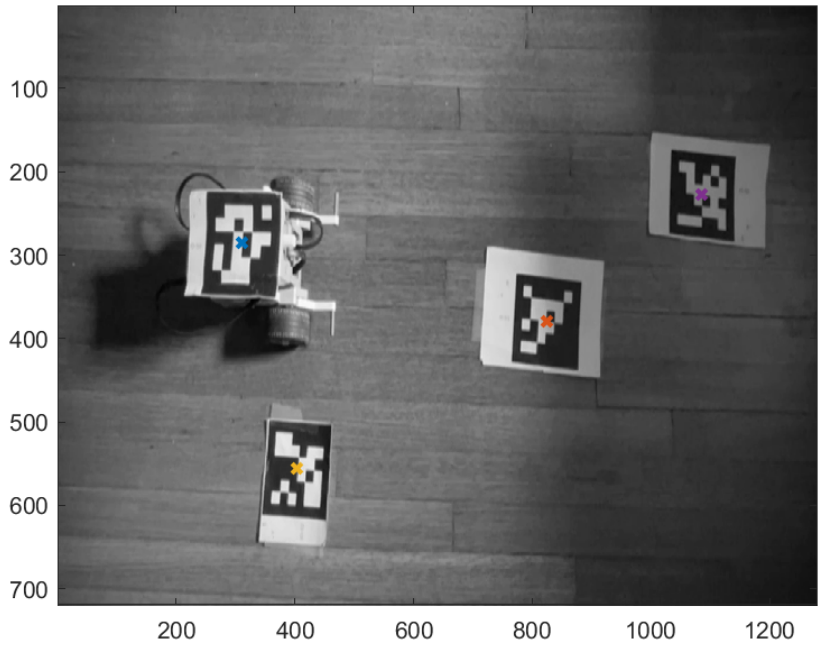

Figure 1: *Captured Scene 2*

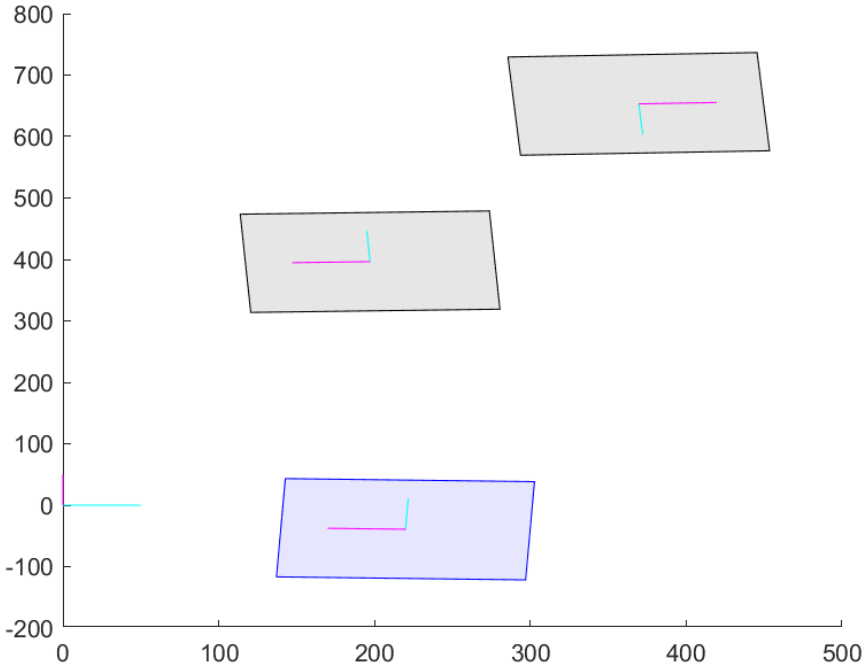

Figure 2: *Rendered Scene 2*

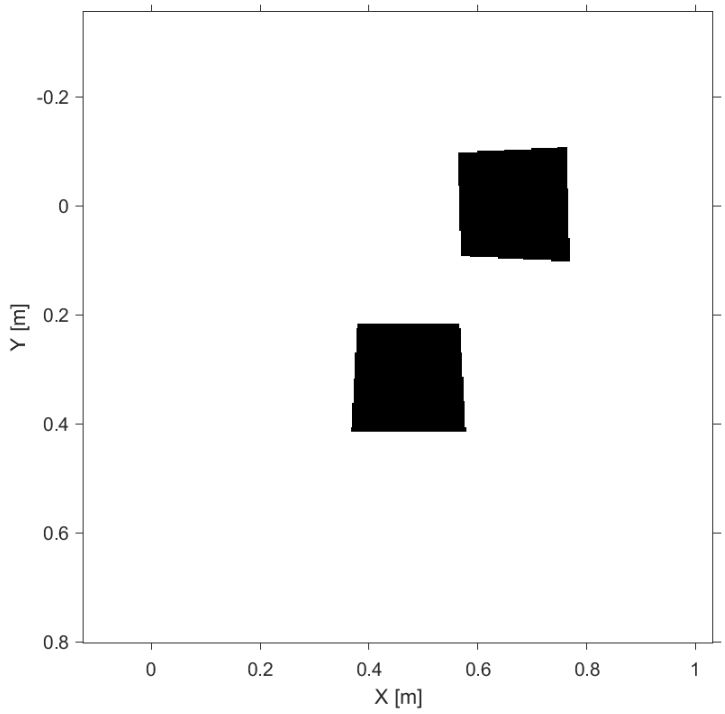

Figure 3: *Occupancy Map 2*

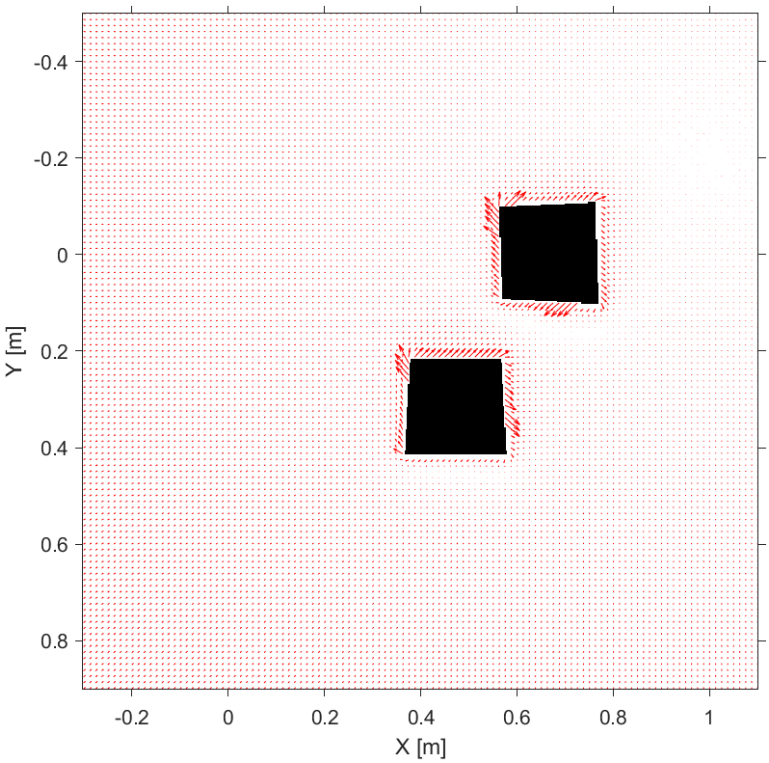

Figure 4: *Potential Field 2*

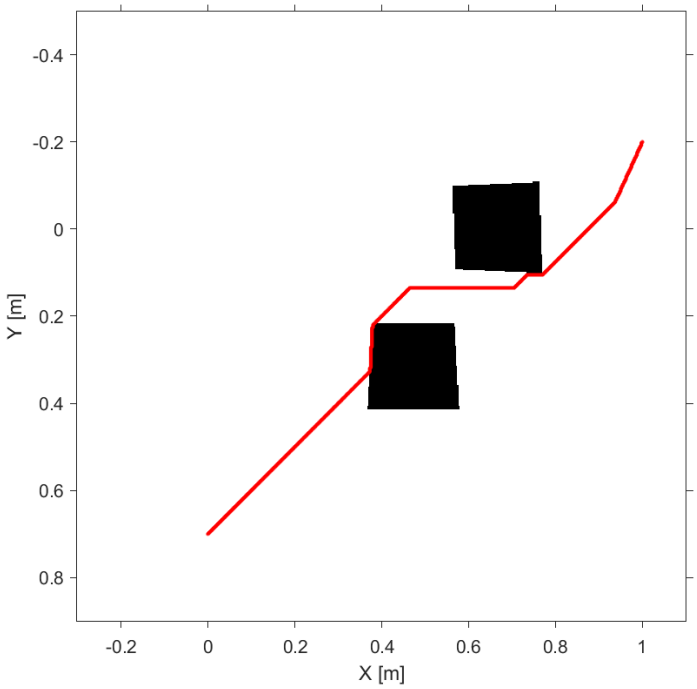

***Figure 1**: A* Path Planning 2*

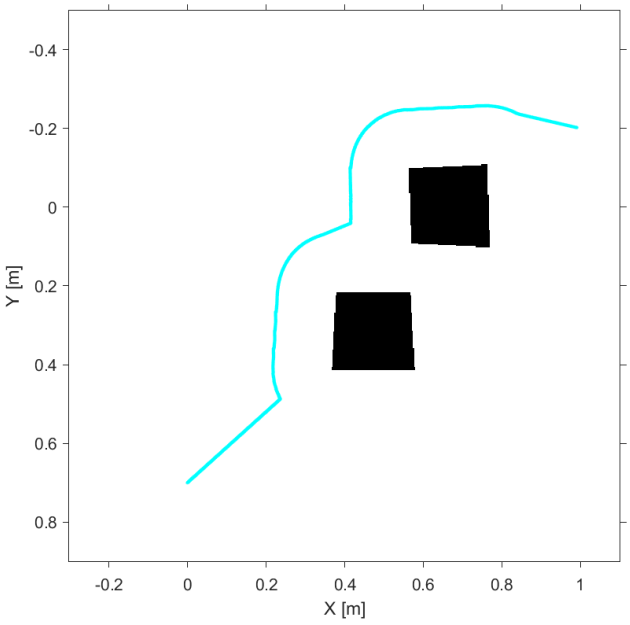

***Figure 2:** Potential Field Path 2*

Scene 3

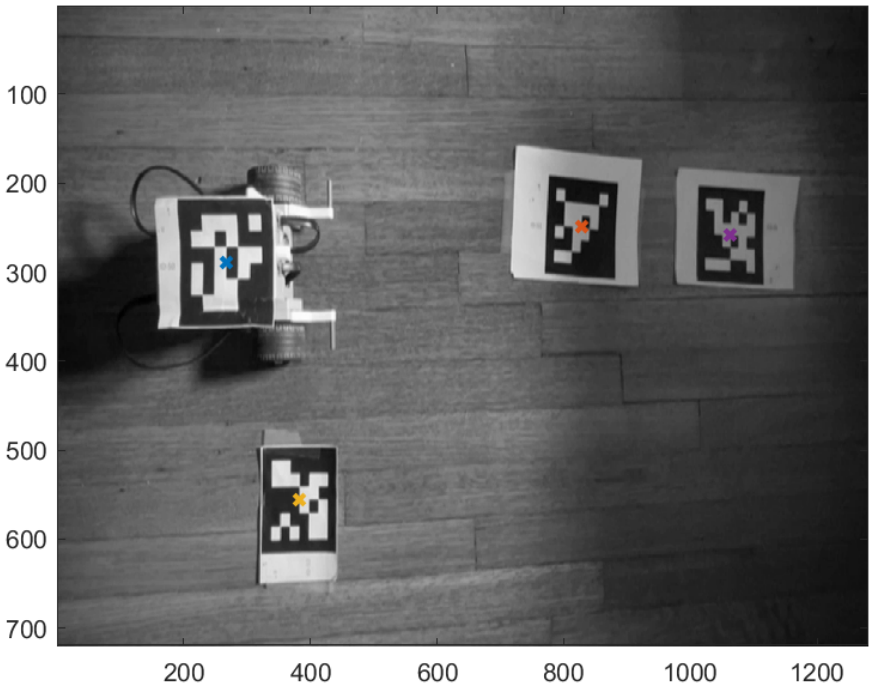

Figure 3: Scene 3

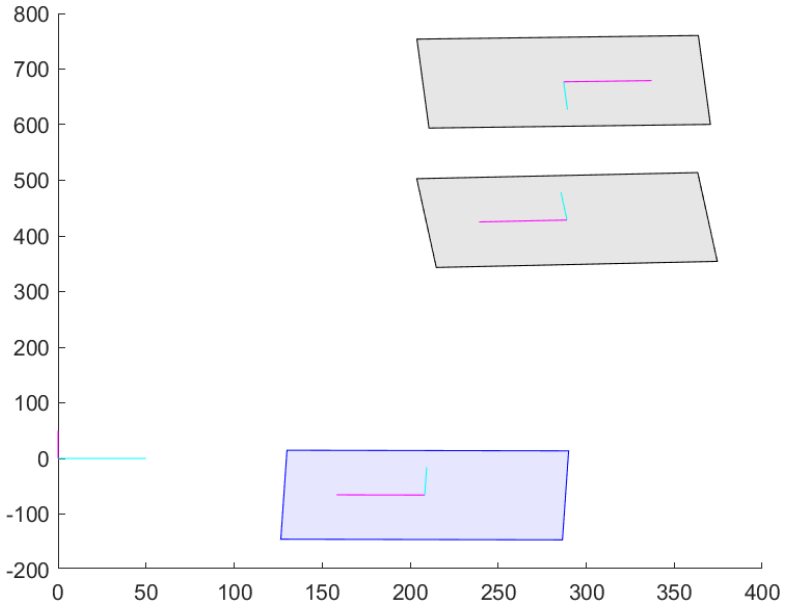

Figure 4: *Rendered Scene 3*

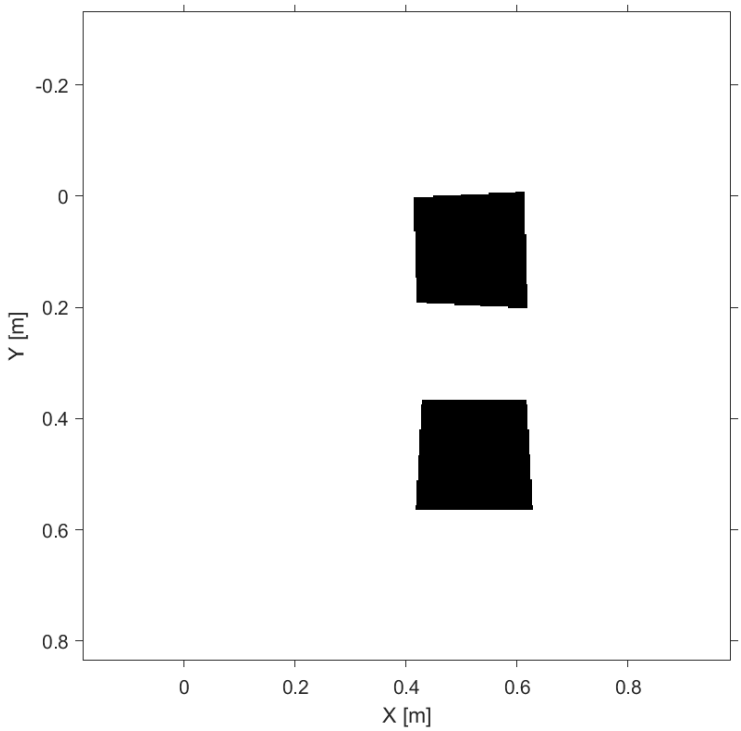

Figure 5: *Occupancy Map 3*

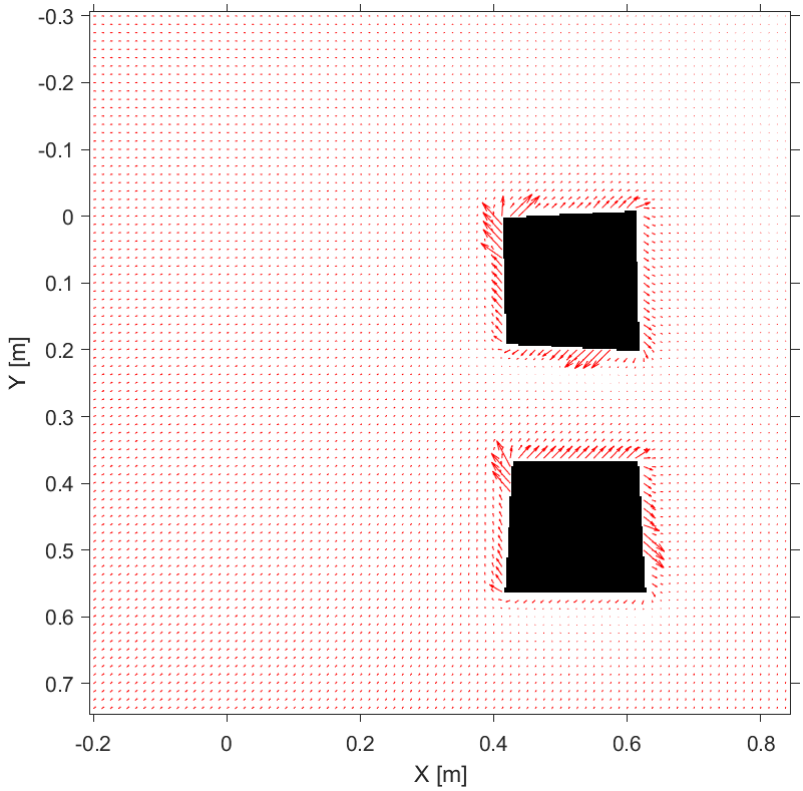

Figure 6: *Potential Field 3*

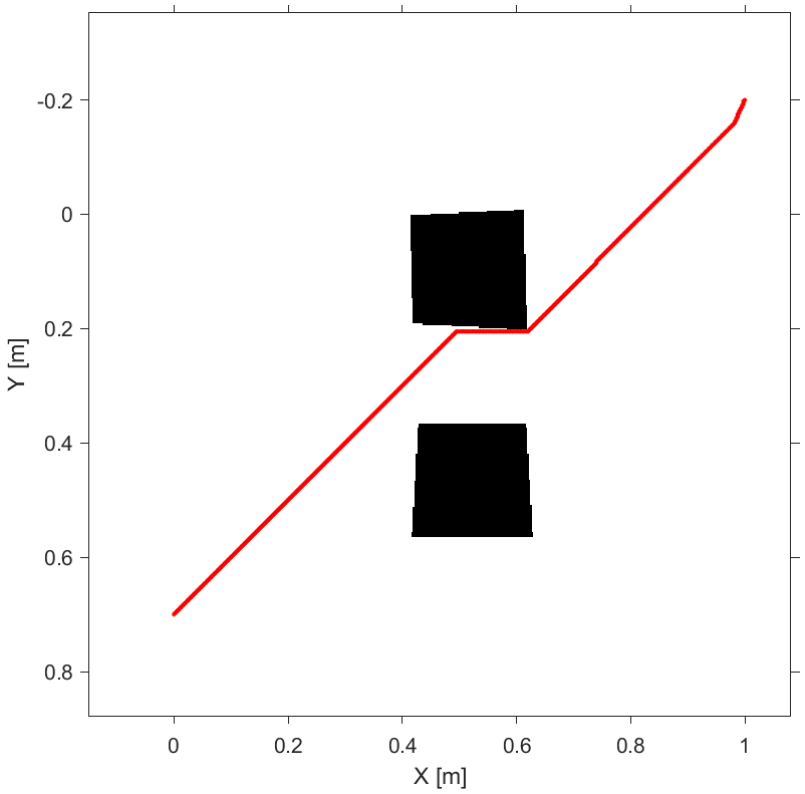

***Figure** 7: A* Path Planning 3*

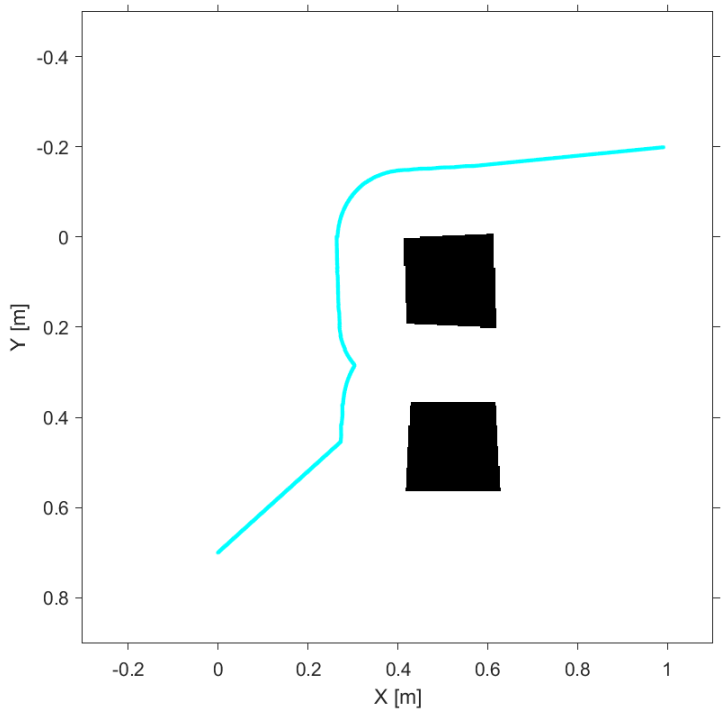

***Figure 8:** Potential Field Path 3*

Appendix B – Equations

$$\omega_R = \frac{v}{r} + \frac{L\omega}{2r} \tag{1}$$

$$\omega_L = \frac{v}{r} - \frac{L\omega}{2r} \tag{2}$$

$$v_R = v + \frac{L}{2}\omega \tag{3}$$

$$v_L = v - \frac{L}{2}\omega \tag{4}$$

Appendix C – MATLAB Code

7.1.1 Part 1: Apriltag Recognition

```
clear; close all; clc
vid_rec.vid_type = 'winvideo';
vid_rec.src_num = 2;
vid_rec.img_format = 'YUY2_1280x720';

calib = load('cameraParams.mat');
cam_calib = calib.cameraParams;
lv = labvision(vid_rec, cam_calib);

at_origin    = apriltag_obj(50, 'vehicle',   [-80 80 -80 80], 108);
at_vehicle   = apriltag_obj(51, 'obstacle',  [-80 80 -80 80], 108);
at_obstacle1 = apriltag_obj(52, 'origin',    [], 108);
at_obstacle2 = apriltag_obj(53, 'obstacle',  [-80 80 -80 80], 108);

lv.save_tag_obj(at_origin);
lv.save_tag_obj(at_vehicle);
lv.save_tag_obj(at_obstacle1);
lv.save_tag_obj(at_obstacle2);
lv.capture_scene;

scene = lv.scene;
K = lv.calib.Intrinsics.K;

T_CT = lv.origin_tag.T_CT;
XY = K * T_CT(1:3,4);
UV = [XY(1:2)/XY(3) scene(1).center]

T_CT = lv.vehicle_tag.T_CT;
XY = K * T_CT(1:3,4);
UV = [XY(1:2)/XY(3) scene([scene.id]==51).center]

for k = 1:numel(lv.obstacle_tags)
    T_CT = lv.obstacle_tags(k).T_CT;
    XY = K * T_CT(1:3,4);
    UV = [XY(1:2)/XY(3) scene([scene.id]==lv.obstacle_tags(k).id).center]
end

disp('Origin T_OT'); disp(lv.origin_tag.T_OT)
disp('Origin T_CT'); disp(lv.origin_tag.T_CT)
disp('Vehicle T_OT'); disp(lv.vehicle_tag.T_OT)
disp('Vehicle T_CT'); disp(lv.vehicle_tag.T_CT)
disp('Obstacle 1 T_OT'); disp(lv.obstacle_tags(1).T_OT)
disp('Obstacle 1 T_CT'); disp(lv.obstacle_tags(1).T_CT)
disp('Obstacle 2 T_OT'); disp(lv.obstacle_tags(2).T_OT)
disp('Obstacle 2 T_CT'); disp(lv.obstacle_tags(2).T_CT)

T = lv.vehicle_tag.T_OT;
position = T(1:3,4);
```

```
heading = atan2(T(2,1), T(1,1));
fprintf('Vehicle Position (x, y, z): [%.2f, %.2f, %.2f]\n', position(1), position(2),
position(3));
fprintf('Vehicle Heading (yaw): %.2f radians\n', heading);
```

7.1.2 Part 2: Occupancy Map

7.1.3 Obstacle Polygons – Scene 1 (Demo)

Scene 1

```
obs1X = ([463.08, 468.932, 668.672, 662.82] / 1000) + 0.1;
obs1Y = ([448.127, 258.415, 248.213, 458.329] / 1000) + 0.05;
obs2X = ([378.01, 567.065, 578.01, 367.065] / 1000) - 0.1;
obs2Y = ([-15.0181, -15.1024, -214.718, -214.803] / 1000) - 0.1;
```

7.1.4 Obstacle Polygons – Scene 2

Scene 2

```
obs1X = ([463.08, 468.932, 668.672, 662.82] / 1000) + 0.1;
obs1Y = ([448.127, 258.415, 248.213, 458.329] / 1000) + 0.05;
obs2X = ([378.01, 567.065, 578.01, 367.065] / 1000) - 0.0;
obs2Y = ([-15.0181, -15.1024, -214.718, -214.803] / 1000) + 0.2;
```

7.1.5 Obstacle Polygons – Scene 3

Scene 3

```
obs1X = ([463.08, 468.932, 668.672, 662.82] / 1000) - 0.05;
obs1Y = ([448.127, 258.415, 248.213, 458.329] / 1000) - 0.05;
obs2X = ([378.01, 567.065, 578.01, 367.065] / 1000) + 0.05;
obs2Y = ([-15.0181, -15.1024, -214.718, -214.803] / 1000) + 0.05;
```

```
resolution = 400;                    % Grid resolution: 400 cells per meter (~2.5 mm per cell)
xRange_m = -0.3:1/resolution:1.1;
yRange_m = -0.5:1/resolution:0.9;
[Xm, Ym] = meshgrid(xRange_m, yRange_m);
inObs1 = inpolygon(Xm, Ym, obs1X, obs1Y);
inObs2 = inpolygon(Xm, Ym, obs2X, obs2Y);
occupied = inObs1 | inObs2;
occupied_flip = flipud(occupied);
map = occupancyMap(occupied_flip, resolution);
cellSize = xRange_m(2) - xRange_m(1);
map.GridLocationInWorld = [min(xRange_m)+cellSize/2, min(yRange_m)+cellSize/2];

figure;
imshow(1 - occupied_flip, 'XData', xRange_m, 'YData', yRange_m);
set(gca, 'YDir', 'reverse');
```

```
axis equal; axis tight; axis on;
xlabel('X [m]'); ylabel('Y [m]');
```

7.1.6 Part 2a: Potential Field Computation

```
goalPos = [1, -0.2];          % IMPORTANT: goalPos defines the desired goal position [m]
K = [3e-2, 9e-7, 9e-7, 9e-2];       % IMPORTANT: K = [k_att, k_rep, k_rot, f_max]
r0_meters = 2;             % IMPORTANT: r0_meters is the obstacle influence radius in [m]
r0 = round(r0_meters * resolution);

[fx, fy] = gradient_path(occupied_flip, goalPos, K, r0, xRange_m, yRange_m, false);

figure;
imshow(1 - occupied_flip, 'XData', xRange_m, 'YData', yRange_m);
set(gca, 'YDir', 'reverse');
axis equal; axis tight; axis on;
hold on;
[Xg, Yg] = meshgrid(xRange_m, yRange_m);
ds = 5;
scaleFactor = 0.1;      % IMPORTANT: scaleFactor scales the force vectors for visualization
quiver(Xg(1:ds:end, 1:ds:end), Yg(1:ds:end, 1:ds:end), ...
       fx(1:ds:end, 1:ds:end)*scaleFactor, fy(1:ds:end, 1:ds:end)*scaleFactor, 0, 'r');
xlabel('X [m]'); ylabel('Y [m]');
hold off;
```

7.1.7 Part 2b: A* Path Planning

```
startPos = [0, 0.7];
gridIndStart = world2grid(map, startPos);
gridIndGoal  = world2grid(map, goalPos);
gridIndStart(1) = map.GridSize(1) - gridIndStart(1) + 1;
gridIndGoal(1)  = map.GridSize(1) - gridIndGoal(1) + 1;
planner_obj = plannerAStarGrid(map);
pathAStar = plan(planner_obj, [gridIndStart(1), gridIndStart(2)], [gridIndGoal(1),
gridIndGoal(2)]);
flippedPath = pathAStar;
flippedPath(:,1) = map.GridSize(1) - pathAStar(:,1) + 1;
worldCoordinates_AStar = grid2world(map, flippedPath);

figure;
imshow(1 - occupied_flip, 'XData', xRange_m, 'YData', yRange_m);
set(gca, 'YDir', 'reverse');
axis equal; axis tight; axis on;
hold on;
plot(worldCoordinates_AStar(:,1), worldCoordinates_AStar(:,2), 'r.-', 'LineWidth', 2);
xlabel('X [m]'); ylabel('Y [m]');
hold off;
```

7.1.8 Part 2c: Potential Field Path

```
stepSize = 0.001;    % Step size for gradient descent [m]
maxSteps = 10000;
potentialPath = followGradient(fx, fy, xRange_m, yRange_m, startPos, goalPos, stepSize,
maxSteps);

figure;
imshow(1 - occupied_flip, 'XData', xRange_m, 'YData', yRange_m);
set(gca, 'YDir', 'reverse');
axis equal; axis tight; axis on;
hold on;
plot(potentialPath(:,1), potentialPath(:,2), 'c.-', 'LineWidth', 2);
xlabel('X [m]'); ylabel('Y [m]');
hold off;
```

7.1.9 Part 2d: Sparsify Potential Path

```
segmentThreshold = 0.1;  % IMPORTANT: segmentThreshold is the minimum distance between path
points to keep [m]
sparsePath = potentialPath(1,:);
for i = 2:size(potentialPath,1)
    if norm(potentialPath(i,:) - sparsePath(end,:)) >= segmentThreshold
        sparsePath = [sparsePath; potentialPath(i,:)];
    end
end
if norm(sparsePath(end,:) - potentialPath(end,:)) > 1e-3
    sparsePath = [sparsePath; potentialPath(end,:)];
end

figure;
imshow(1 - occupied_flip, 'XData', xRange_m, 'YData', yRange_m);
set(gca, 'YDir', 'reverse');
axis equal; axis tight; axis on;
hold on;
plot(potentialPath(:,1), potentialPath(:,2), 'c.-', 'LineWidth', 1);
plot(sparsePath(:,1), sparsePath(:,2), 'bo-', 'LineWidth', 2, 'MarkerSize', 6);
xlabel('X [m]'); ylabel('Y [m]');
hold off;
```

7.1.10 Part 3: Robot Integration

7.1.11 Part 3a: Controller Settings

```
myev3 = legoev3('usb');
motorLeft  = motor(myev3, 'A');
motorRight = motor(myev3, 'D');

R = 2.8;        % Wheel radius in cm
L = 12.0;       % Distance between wheels in cm
```

```
v_forward_base = 50;          % Base forward speed [cm/s]
maxForwardSpeed_cmps = 30;    % Maximum allowed forward speed [cm/s]
velScaleFactor = 0.8;         % Additional speed scaling (0 < factor <= 1)
maxWheelSpeed_degPerSec = 300; % Maximum motor speed [deg/s] for safety

Kp_heading = 1.5;             % PD controller proportional gain
Kd_heading = 1;               % PD controller derivative gain
headingErrorThreshold = deg2rad(2);   % Threshold for heading alignment (2° in radians)
dt = 0.01;                    % Controller loop time [s]

initial_heading = 0;          % Set initial heading (simulation/starting condition)
current_heading = initial_heading;

fprintf('Starting open-loop execution with improved heading PD control along the scaled potential
path...\n');

% IMPORTANT: Scaling the sparse path and converting from meters to centimeters
pathScaleFactor = 0.1;        % Scale factor applied to the potential field path
path_cm = (sparsePath * pathScaleFactor) * 100;
```

7.1.12 Part 3b: Open-Loop PD control along Potential Path

```
numPoints = size(path_cm, 1);
for i = 1:numPoints-1
    current_wp = path_cm(i, :);
    next_wp = path_cm(i+1, :);
    delta = next_wp - current_wp;
    segmentDistance = norm(delta);
    thetaRef = atan2(delta(2), delta(1));
    prevError = 0;
    while true
        headingError = thetaRef - current_heading;
        headingError = atan2(sin(headingError), cos(headingError));
        if abs(headingError) < headingErrorThreshold
            break;
        end
        dError = (headingError - prevError) / dt;
        omegaCmd = Kp_heading * headingError + Kd_heading * dError;
        prevError = headingError;
        if omegaCmd > 0
            v_left = - (L/2)*abs(omegaCmd);
            v_right = (L/2)*abs(omegaCmd);
        else
            v_left = (L/2)*abs(omegaCmd);
            v_right = - (L/2)*abs(omegaCmd);
        end
        omega_left = v_left / R;
        omega_right = v_right / R;
        degSec_left = rad2deg(omega_left);
        degSec_right = rad2deg(omega_right);
        degSec_left = max(min(degSec_left, maxWheelSpeed_degPerSec), -maxWheelSpeed_degPerSec);
```

```
        degSec_right = max(min(degSec_right, maxWheelSpeed_degPerSec), -maxWheelSpeed_degPerSec);
        motorLeft.Speed = degSec_left;
        motorRight.Speed = degSec_right;
        start(motorLeft); start(motorRight);
        pause(dt);
        stop(motorLeft); stop(motorRight);
        current_heading = current_heading + omegaCmd * dt;
        current_heading = atan2(sin(current_heading), cos(current_heading));
        fprintf('Turning: Current Heading = %.1f°, Target = %.1f°, Error = %.1f°\n',...
            rad2deg(current_heading), rad2deg(thetaRef), rad2deg(headingError));
    end
    fprintf('Segment %d: Aligned to desired heading (%.1f°).\n', i, rad2deg(thetaRef));
    v_forward = v_forward_base;
    if v_forward > maxForwardSpeed_cmps
        v_forward = maxForwardSpeed_cmps;
    end
    v_forward = v_forward * velScaleFactor;
    t_forward = segmentDistance / v_forward;
    omega_forward = v_forward / R;
    degSec_forward = rad2deg(omega_forward);
    degSec_forward = max(min(degSec_forward, maxWheelSpeed_degPerSec), -maxWheelSpeed_degPerSec);
    fprintf('Segment %d: Driving forward %.2f cm over %.2f s (v_forward = %.2f cm/s)...\n',...
        i, segmentDistance, t_forward, v_forward);
    motorLeft.Speed = degSec_forward;
    motorRight.Speed = degSec_forward;
    start(motorLeft); start(motorRight);
    pause(t_forward);
    stop(motorLeft); stop(motorRight);
    pause(0.1);
end
stop(motorLeft); stop(motorRight);
fprintf('Open-loop execution with improved heading PD control complete.\n');
```

7.1.13 Part 4: Functions

```
function [fx, fy] = gradient_path(obs_map, goal, K, r0, x, y, plot_map)
    if nargin < 7, plot_map = 0; end
    k_att = K(1);
    k_rep = K(2);
    k_rot = K(3);
    f_max = K(4);

    [X, Y] = meshgrid(x, y);
    sz = size(obs_map);
    fx = -k_att * (X - goal(1));  % Attractive force in x-direction
    fy = -k_att * (Y - goal(2));  % Attractive force in y-direction
    F_mag = sqrt(fx.^2 + fy.^2);

    if f_max > 0
        exceed_idx = F_mag > f_max;
        fx(exceed_idx) = fx(exceed_idx) .* (f_max ./ F_mag(exceed_idx));
        fy(exceed_idx) = fy(exceed_idx) .* (f_max ./ F_mag(exceed_idx));
```

```
    end

    obs_perim = bwperim(obs_map);
    perim_idx = find(obs_perim(:));
    n_perim = length(perim_idx);
    [pyIdx, pxIdx] = ind2sub(sz, perim_idx);
    repulse_idx = -r0:r0;
    [rx, ry] = meshgrid(repulse_idx);
    cellSize = x(2) - x(1);
    rx = rx * cellSize;
    ry = ry * cellSize;
    r = sqrt(rx.^2 + ry.^2);
    mask = (r <= (r0 * cellSize)) & (r > 0);
    f_rep_x = zeros(size(r));
    f_rep_y = zeros(size(r));
    f_rot_x = zeros(size(r));
    f_rot_y = zeros(size(r));
    f_rep_x(mask) = k_rep * (1./(r(mask).^2)) .* (rx(mask)./r(mask));
    f_rep_y(mask) = k_rep * (1./(r(mask).^2)) .* (ry(mask)./r(mask));
    f_rot_x(mask) = k_rot * (-ry(mask)) ./ (r(mask).^3);
    f_rot_y(mask) = k_rot * (rx(mask)) ./ (r(mask).^3);
    repulse_map_x = f_rep_x + f_rot_x;
    repulse_map_y = f_rep_y + f_rot_y;
    Nx = length(x); Ny = length(y);
    for k = 1:n_perim
        r_center = pyIdx(k);
        c_center = pxIdx(k);
        r_dest_start = max(r_center - r0, 1);
        r_dest_end = min(r_center + r0, Ny);
        c_dest_start = max(c_center - r0, 1);
        c_dest_end = min(c_center + r0, Nx);
        src_r_start = 1 + (r_dest_start - (r_center - r0));
        src_r_end = (2*r0 + 1) - ((r_center + r0) - r_dest_end);
        src_c_start = 1 + (c_dest_start - (c_center - r0));
        src_c_end = (2*r0 + 1) - ((c_center + r0) - c_dest_end);
        fx(r_dest_start:r_dest_end, c_dest_start:c_dest_end) = ...
            fx(r_dest_start:r_dest_end, c_dest_start:c_dest_end) + ...
            repulse_map_x(src_r_start:src_r_end, src_c_start:src_c_end);
        fy(r_dest_start:r_dest_end, c_dest_start:c_dest_end) = ...
            fy(r_dest_start:r_dest_end, c_dest_start:c_dest_end) + ...
            repulse_map_y(src_r_start:src_r_end, src_c_start:src_c_end);
    end
    fx(obs_map > 0) = 0;
    fy(obs_map > 0) = 0;
    if plot_map
        figure;
        imshow(obs_map, 'XData', x, 'YData', y);
        set(gca, 'YDir', 'normal');
        axis equal; axis tight; axis on;
        hold on;
        ds = 5;
        scaleFactor = 0.005;
        quiver(X(1:ds:end,1:ds:end), Y(1:ds:end,1:ds:end),...
            fx(1:ds:end,1:ds:end)*scaleFactor, fy(1:ds:end,1:ds:end)*scaleFactor, 0, 'r');
```

```
        xlabel('X [m]'); ylabel('Y [m]');
        hold off;
    end
end

function path = followGradient(fx, fy, x, y, startPos, goalPos, stepSize, maxSteps)
    distThreshold = 0.01;
    currentPos = startPos;
    path = currentPos;
    for i = 1:maxSteps
        dx = interp2(x, y, fx, currentPos(1), currentPos(2), 'linear', 0);
        dy = interp2(x, y, fy, currentPos(1), currentPos(2), 'linear', 0);
        gradVec = [dx, dy];
        magGrad = norm(gradVec);
        if magGrad < 1e-9
            directVec = goalPos - currentPos;
            if norm(directVec) > 1e-9
                direction = directVec / norm(directVec);
            else
                warning('Goal reached or very close.');
                break;
            end
        else
            direction = gradVec / magGrad;
        end
        currentPos = currentPos + stepSize * direction;
        path = [path; currentPos];
        if norm(currentPos - goalPos) < distThreshold
            disp('Reached the goal region!');
            break;
        end
    end
    if norm(currentPos - goalPos) >= distThreshold
        disp('Potential flow path did not converge exactly; appending goal.');
        path = [path; goalPos];
    end
end
```

Published with MATLAB® R2023b

7.2