# ME 598

Introduction to Robotics

*Fall 2025*

*Final group project: Autonomous Object Sorting using a Mobile Robot*

## Group R2

## 12/16/2025

*Graduate students:*

*“I pledge that I have abided by the Graduate Student Code of Academic Integrity.”*

The report has been prepared by:

1. Lab Leader: Jefrey BullaInk 1, Shape
2. Reece Paz Ink 1, Shape
3. Aaron Reyes 
4. Pratham Waghela 

# Abstract

This report presents some methodological aspects for implementing autonomous sorting of objects using a mobile robot. Algorithm implementation is discussed as well as roadmap design. Results are tested in a simulation environment.

# Introduction

Mobile robotics are used in many contexts from toys, vacuums to factories. Mobile robots have many capabilities including sensing, navigating, moving objects etc.

This report focuses on a very specific application in mobile robotics: moving objects in a specific and useful way. We will demonstrate how to implement a control system for a mobile robot that can recognize objects of different colors and then be able to move those objects to specific locations. Command the robot to reposition the two targets to their respective colors while avoiding collision with the surrounding walls.

This report is organized as follows: part 1 presents an analysis of the robot’s environment and the robot’s sensors. Part 2 describes the main program of the proposed control implementation. Part 3 introduces the navigation algorithm. Part 4 details the roadmap design. Part 5 describes the supporting algorithms of the control system and finally Part 6 presents the results of the trials for all the possible system initial conditions.

# Part 1. Sensors and Arena analysis

## Procedure

The MATLAB robotics playground is used, and a set of robot sensors are provided. The robot operates in an Arena that has walls and some objects placed within these walls.

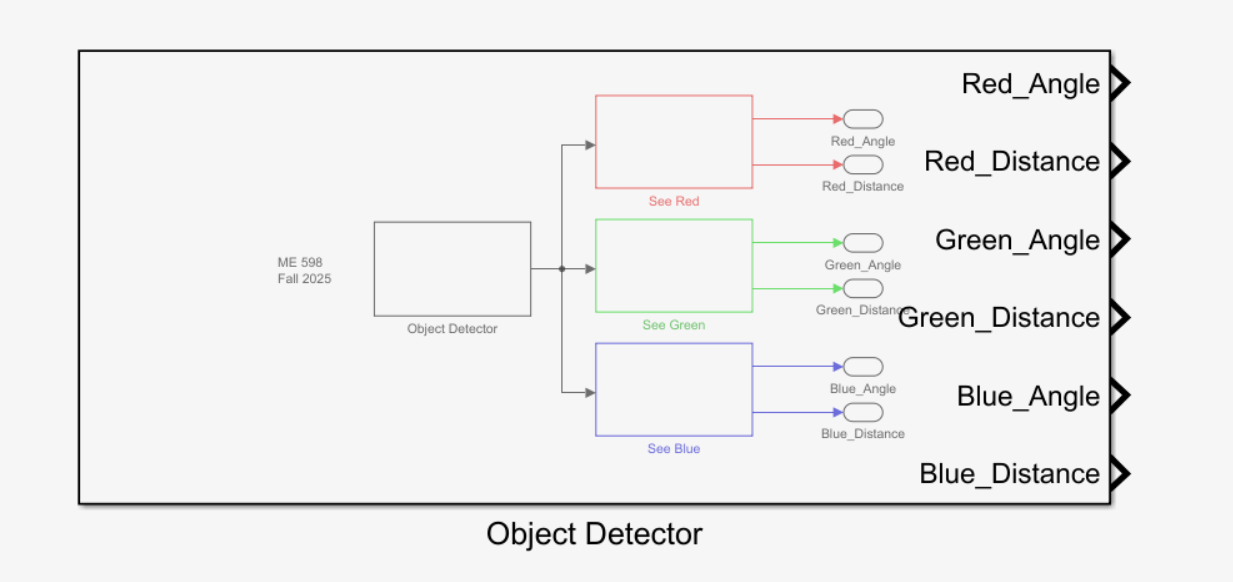
In this section, the robot sensors and the arena are analyzed.

## Results

### Object Sensors

The main robot is equipped with an object sensor that can detect up to three different colors, specifically red, green, and blue as shown in Figure 1.1. The sensor is able to determine the distance and angle for each detected color.

Based on the sensor settings and confirmed with observations, we know that the object detector can sense objects within a maximum of 2m with a minimum of approximately 0.38m and less if the robot collides with the wall. The angle range is within [-30 to 30] degrees

Figure 1.1. Object Detector

### Distance Sensors

The distance sensors are used to measure the distance from the robot to its surrounding walls. This sensor does not detect target objects. The distance sensor can detect its surroundings from the left, front, and right as shown in Figure 1.2.

Given the sensors settings and confirmed through experimentation this sensor detects walls within 2 meters. The minimum observed distance between the robot and the wall is about 0.2 meters. Any distance greater than 2 meters will be recorded as “NaN” or not a number.

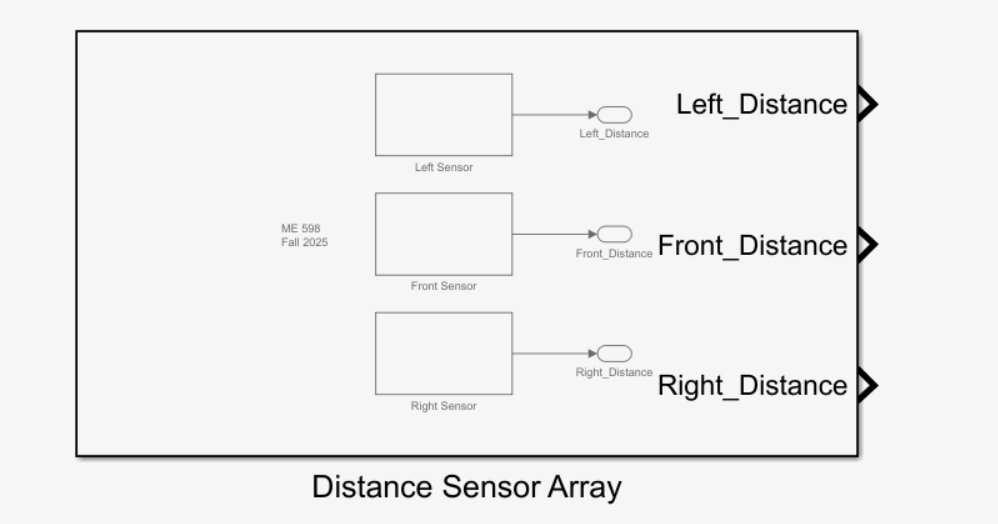
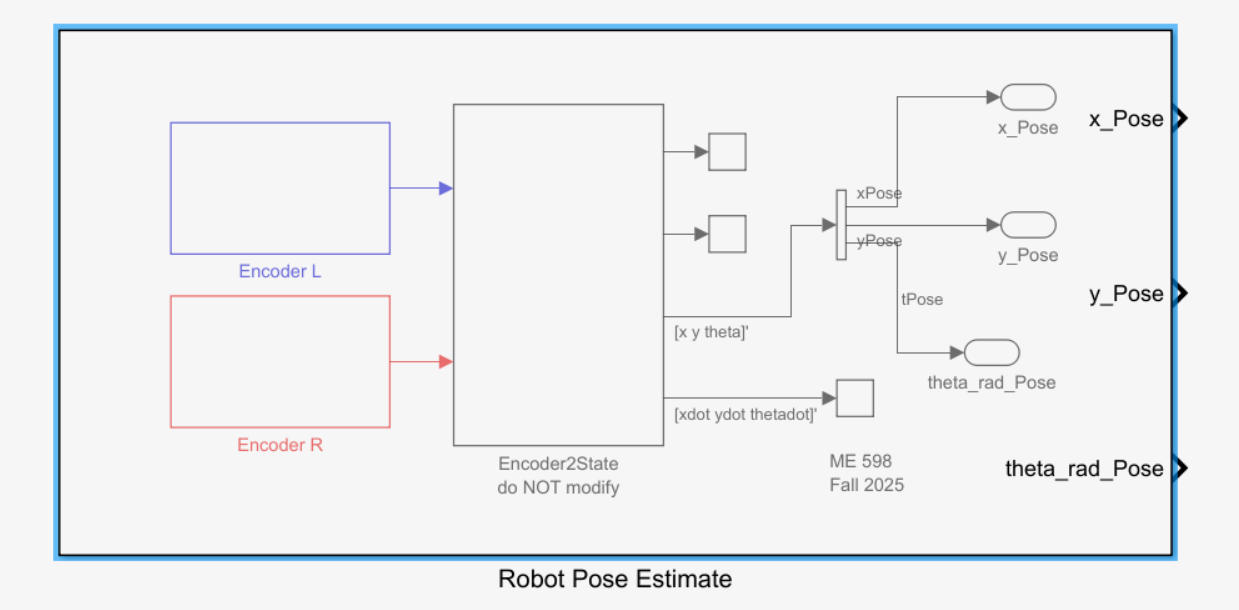


Figure 1.2 Distance Sensor Array in Project Library.

### Odometer

The Robot Pose Estimate Block determines the robot’s position based on its kinematics (X, Y, and theta) as shown in Figure 1.3. The two Encoders in the model represent the data for the left and right motors.

The odometer determines the x and y positions of the robot with respect to the center of the arena, position (0,0). Theta ranges between 0 and +180 (quadrants I and II) and 0 and -180 (quadrants III and IV).

Figure 1.3 Robot Pose Estimate block in Project Library.

### Arena Collection Zones

In the arena, there are 3 colored squares, referred in this document as the “the collection zones,” which serve as the final resting location for the two-colored objects that are initially located towards the center of the field. Figure 1.4 shows the arena with the three collection zones.

The arena’s dimension is 5m x 5m as shown in Figure 1.5. The size of each of the three collection zones is approximately 0.83m x 0.83m as shown in Figure 1.6.

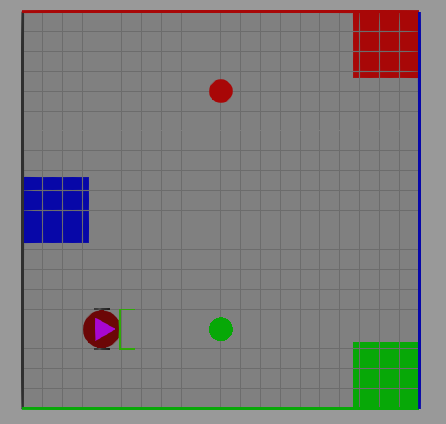


Figure 1.4. Object environment showing the bottom element green and top element red.

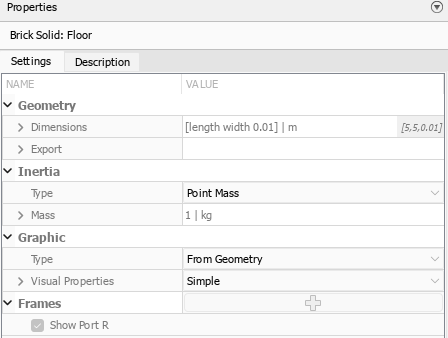


Figure 1.5 Arena Properties.

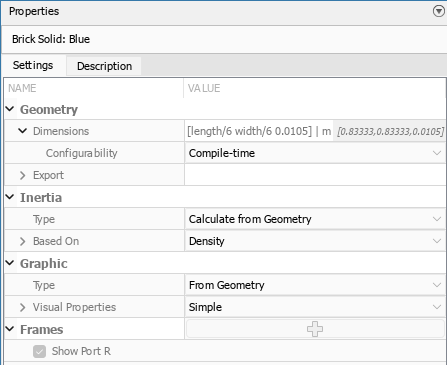


Figure 1.6: Collection Zones Properties (Blue zone).

## Discussion

We analyzed the Arena’s and sensor properties. This information is useful to design the algorithms that control the robot’s motion. If these properties were to change, a new algorithm design may be needed.

# Part 2. Main Program

## Procedure

Given our interest in embedded implementations, we decided to design the entire control system in a MATLAB function block within Simulink.

In this section we present the control states and main algorithm to accomplish the robot’s goal of relocating two objects to their correct resting locations.

The Arena has six possible initial conditions as follows:

* Arena(blue, red)
* Arena(green, red)
* Arena(red, blue)
* Arena(red, green)
* Arena(blue, green)
* Arena(green, blue)

where the first element indicates the bottom object and the second element indicates the top element. For example, the initial conditions shown in Figure 1.4 would be represented as Arena(green, red).

## Results

The objects located in position Object\_0 (bottom) and Object\_1(top) may change in color, however the two possible positions are known, and the collection zone positions remain the same.

Given these conditions we designed an algorithm that navigates in a pre-determined roadmap to find the object and relocate said object to the correct collection zone.

The control system keeps track of states such as determining object color, path planning, and navigating. Figure 2.1 details the main algorithm used.

Each of the tasks in Figure 2.1 is a state in the control system. The state value is persisted across execution loops by using MATLAB ‘persistent’ keyword. Appendix A shows how this is implemented in code.

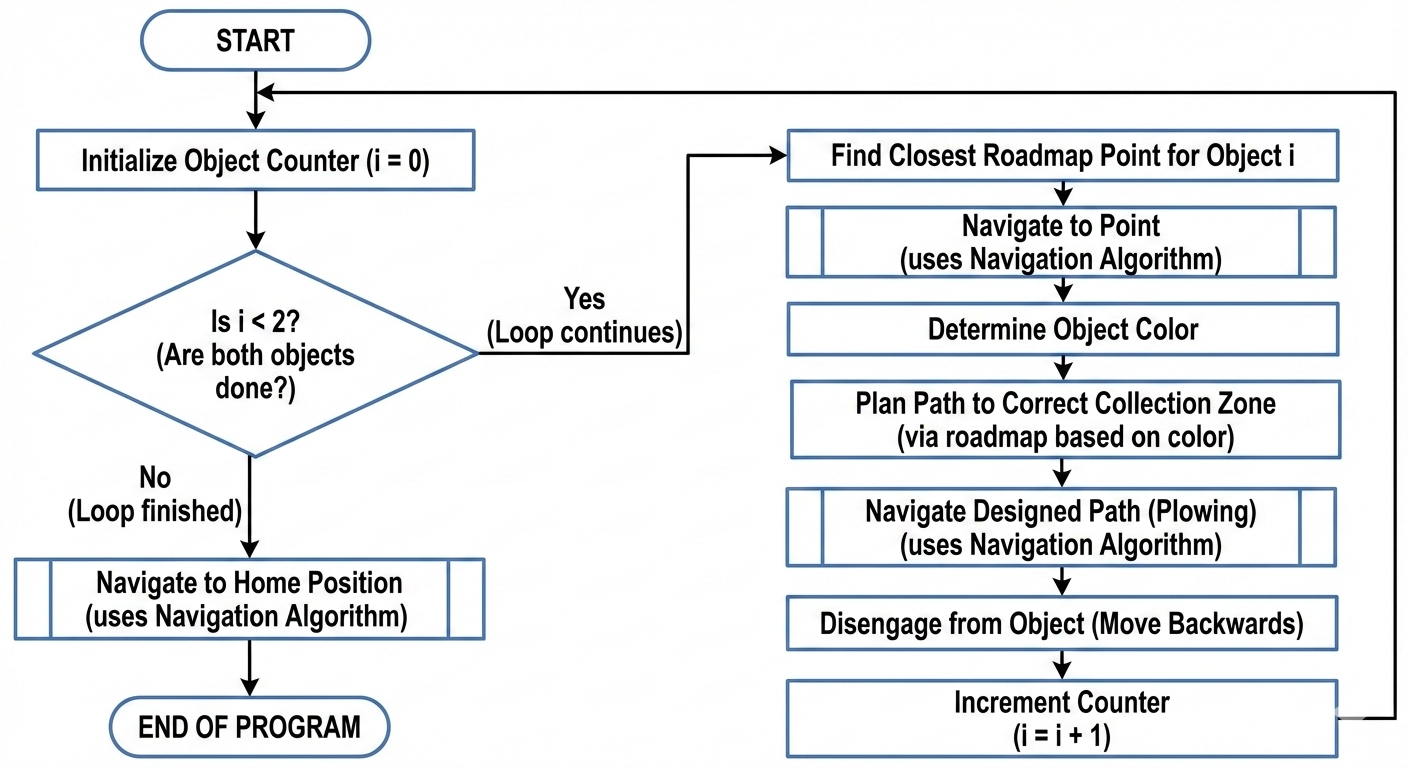


Figure 2.1. Main program that controls every state of execution.

## Discussion

We designed a main program that accomplishes the robot’s goal in a sequential manner. By dividing the program into multiple states and tasks, we can more easily design algorithms to accomplish each of said tasks.

Our main algorithm works for the initial given conditions of the robot, arena and objects. If these conditions change, a new algorithm may be required.

# Part 3. Navigation Algorithm

## Procedure

In this section we present the navigation algorithm that allows the robot to navigate a series of points in the Arena.

## Results

Based on our designed states, the main program commands the robot to navigate to different places such as collection zones or home position (0,0). To make our implementation simpler, we created a navigation routine that is reused in different states of the main program.

The navigation algorithm takes a set of one or more segments and commands the robot to move to those segments one by one until all segments have been visited as shown in detailed in Figure 3.1.

The algorithm starts by choosing the first segment in the navigation queue. Once a segment is selected, the robot determines the angle needed to get to the segment’s end. Then, the robot spins to reach said angle and finally the robot moves to the desired position. This sequence repeats n times if n is equal to the number of segments in the navigation queue.

To implement the “Move Forward until end of segment” step, we confirm that we have reached the end of the segment by checking whether robot x and y positions are within the vicinity of the destination. This vicinity check is implemented as a tolerance that can be tuned.

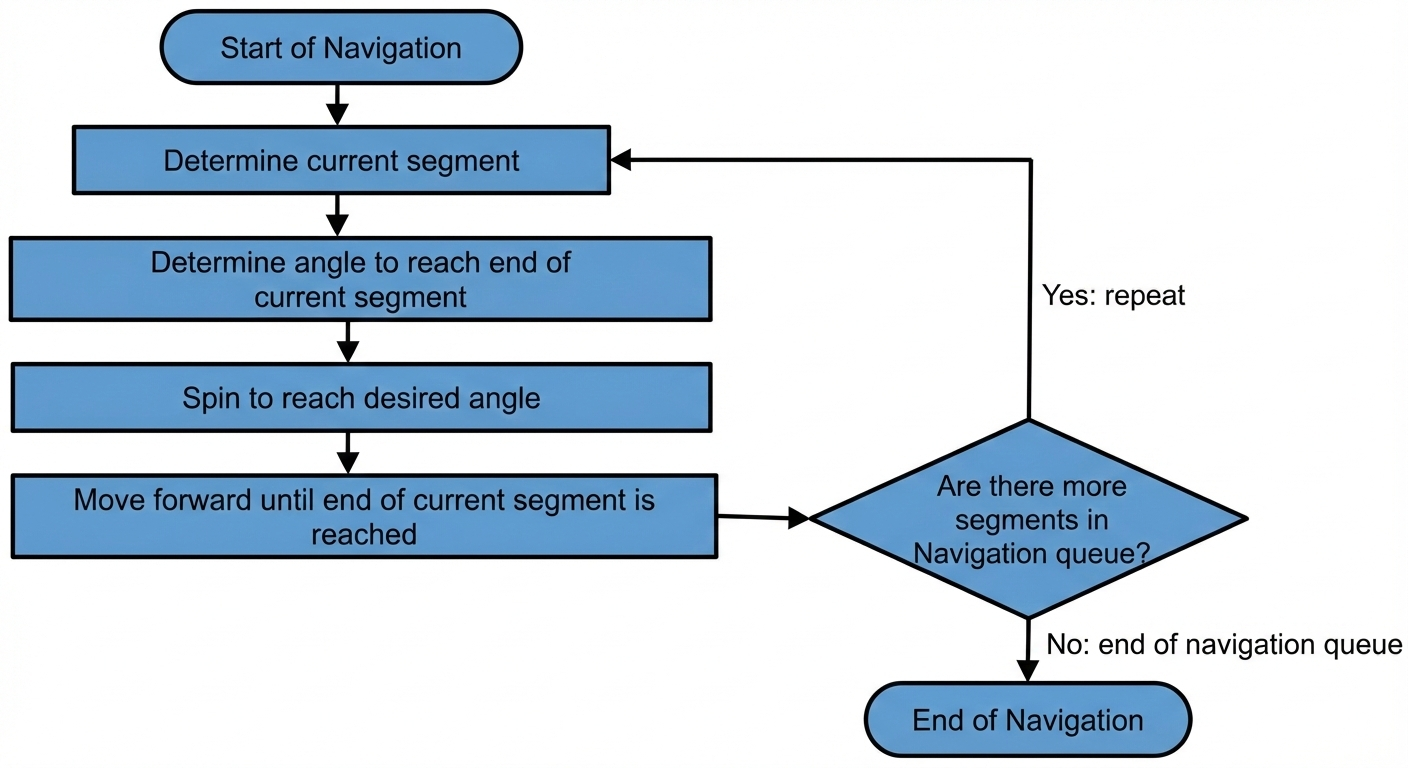


Figure 3.1. Navigation algorithm.

## Discussion

We implemented a navigation routine that is flexible and can be invoked from different states of the main program. The algorithm allows the robot to navigate to one or multiple points in sequence anywhere in the Arena. Given this flexibility, the navigation algorithm can be used even if the main program needs to be updated to account for new initial conditions for the robot, arena or objects.

# Part 4. Roadmap design

## Procedure

In this section, we show how we designed the roadmap. Our pre-determined roadmap considers each initial possible object's starting condition.

Let us define **PPP** or “**Pre-Plow Position**” as the point from which the robot can start a motion to plow an object and deliver it to its correct collection zone in one straight motion (no turns or spins)

There are 6 different object starting conditions for which we need to determine a pre-plow position. Table 4.1 shows all the 6 color configurations.

Table 4.1. Configurations that need a pre-plow position. RGB means ‘any color’.

|  |  |  |
| --- | --- | --- |
| Arena Configuration | Object 1 (Bottom obj) | Object 2 (Top obj) |
| Arena(red, \_) | Red | RGB |
| Arena(green, \_) | Green | RGB |
| Arena(blue,\_) | Blue | RGB |
| Arena(\_,red) | RGB | Red |
| Arena(\_,green) | RGB | Green |
| Arena(\_,blue) | RGB | Blue |

We want to find 6 pre-plow positions, one for each configuration in Table 4.1. On the results section we will discuss theorical and experimental approaches to find the pre-plow positions.

## Results

To find the PPP for each configuration, we first need to find the theoretical center points of the collection zones.

### Theoretical collection zone centers

Considering the initial conditions of the arena, it is a 5 m2 squared area with 3 squared collection zones. Figure 4.1 shows the points of interest in the Arena for this analysis.

Based on geometric approximations, these zones have a side with a length of l ≈ 0.84 m. Considering this length, we can calculate the position on the XY plane of each collection zone center.

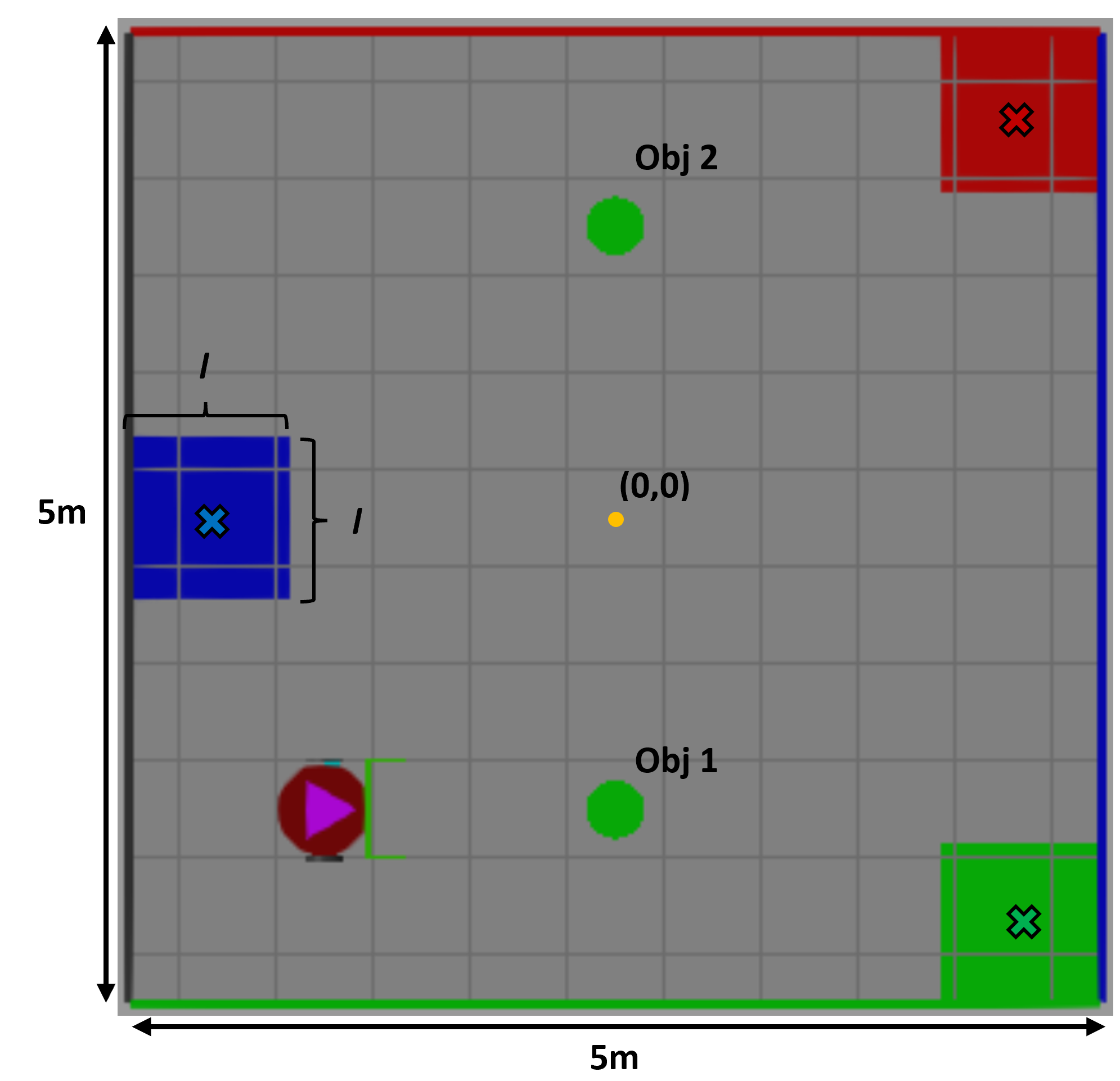


Figure 4.1. Collection zones centers.

Red zone center:

rzcx = 2.5 - 0.84/2 = 2.08  
rzcy = 2.5 - 0.84/2 = 2.08  
**rzc = [2.08 2.08]**

Blue zone center:

bzcx = -2.5 + 0.84/2 = -2.08  
bzcy =   
**bzc = [-2.08 0]**

Green zone center:

gzcx = 2.5 - 0.84/2 = 2.08  
gzcy = -2.5 + 0.84/2 = -2.08  
**gzc = [2.08 -2.08]**

### Pre-Plow position calculation

After finding the theoretical centers of the collection zones, now we’ll find the pre-plow positions. A pre-plow position will bring the object to its collection zone in one straight displacement. To achieve this, we must find a line that connects the respective object to its collection zone center. As an example, we provide mathematical calculations to obtain the pre-plow position of a blue object located in the bottom slot as shown in Figure 4.2.

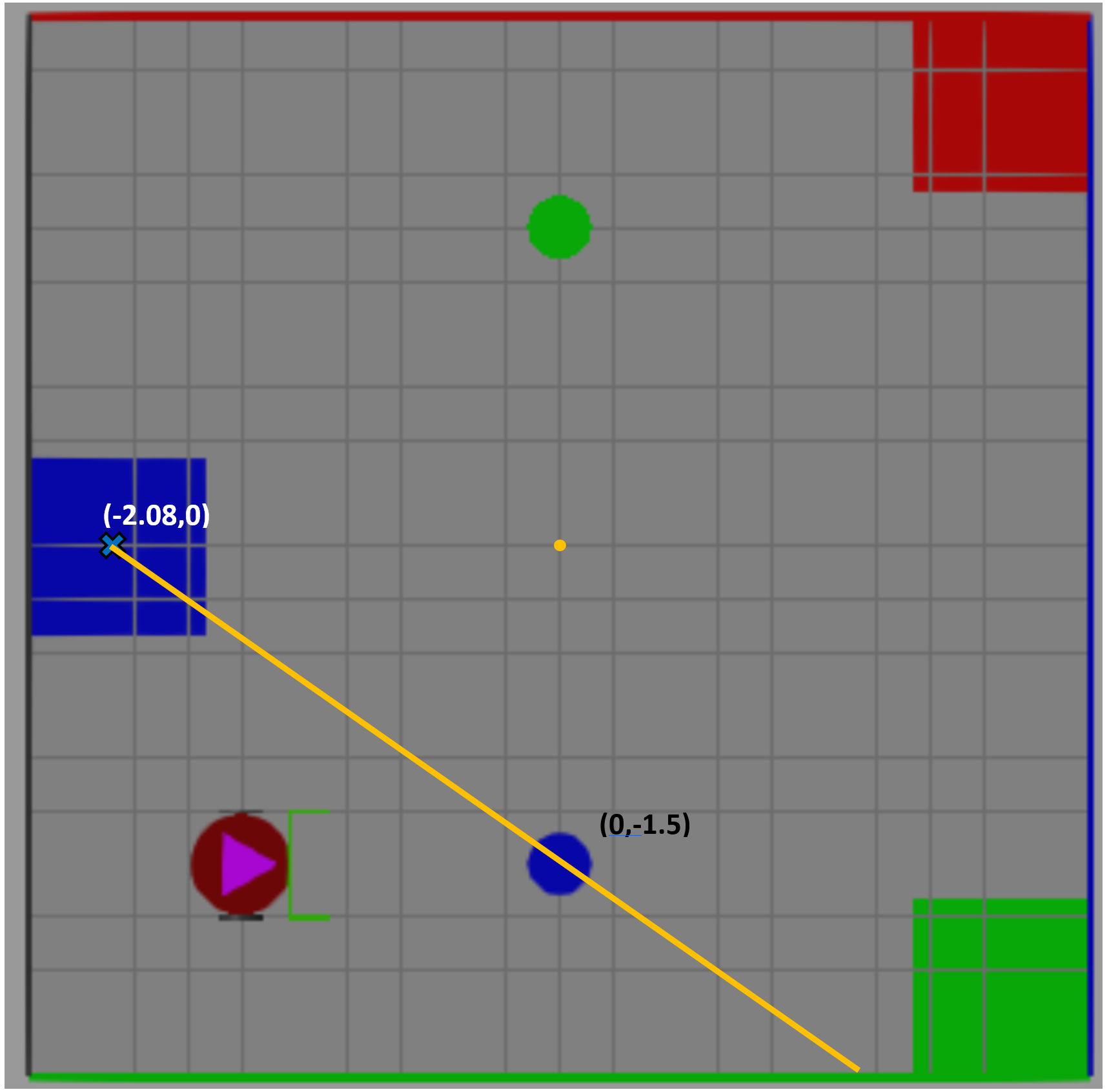


Figure 4.2 Determining Pre-Plow Position for Arena(blue, \_)

Considering the object position and the collection zone center, we can draw an imaginary straight line and calculate its formula:  
Object position:

F(x=0) = -1.5

Collection center zone position:

F(x=-2.08) = 0

Finding the linear equation:

F(x) = mx + c  
F(0) = -1.5 = c

F(x) = mx –1.5

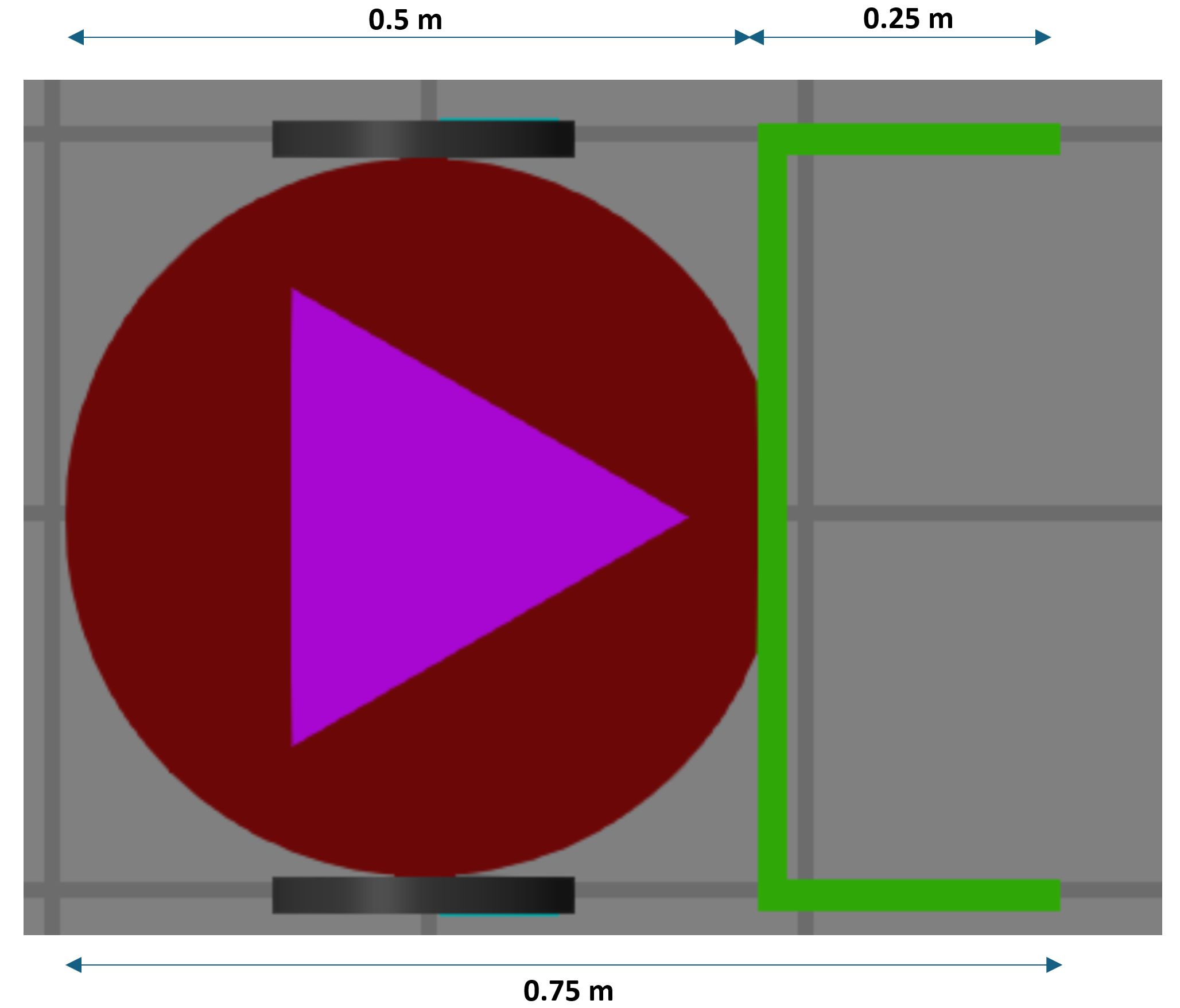
F(-2.08) = m(-2.08) -1.5 = 0  
m = -0.72

**F(x) = -0.72 x –1.5**

By finding the equation, we know that the pre-plow position must lie within the calculated line. However, we also need to verify that the robot will fit between the object and the limit walls by avoiding collision. To achieve this, we calculated the robot’s dimensions (Figure 4.3) and the space between the object and the wall.

Based on geometric approximations, we can define the robot’s dimensions as the following:

The total length of the robot is 0.75 m with its body with a diameter of 0.5m

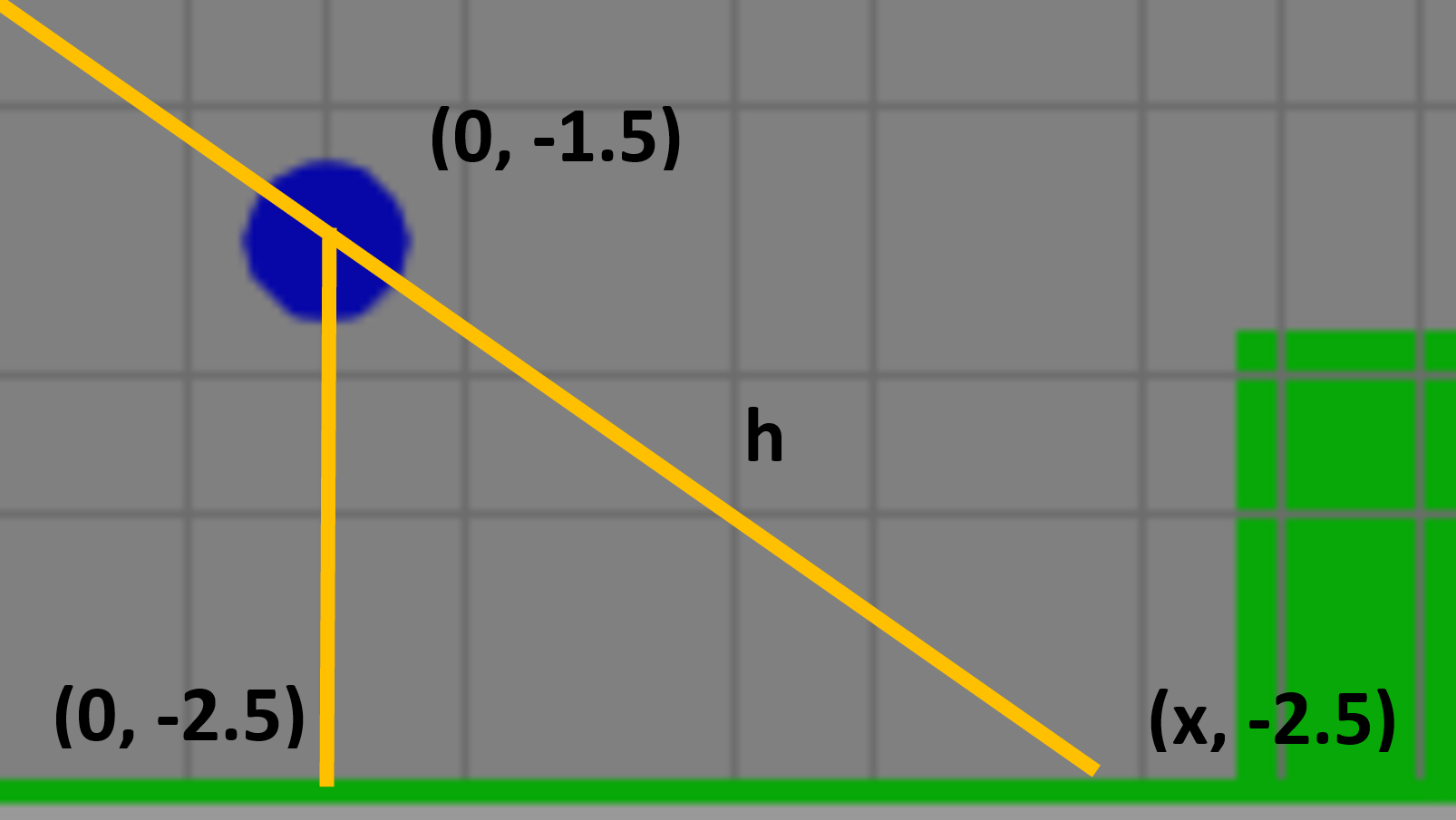
  
Figure 4.3. Robot physical dimensions.

To find if the robot can fit between the object and the wall, we found the respective distances:

F(x, -2.5) = -2.5 = -0.72x -1.5  
x = 1/0.72 -> x = 1.38

and the distance “h” from Figure 4.4:

h = sqrt(1^2 + 1.38^2) = 1.7

  
Figure 4.4 Distance between wall and object

Based on the calculated distance, the robot should be able to fit in this position. Nonetheless, there might be cases where there is not enough space for the robot, so we might need to use a different approach.

Let’s choose x as “1.38/2” as the half distance between the object and the wall as the pre-plow position.

F(x = 1.38/2) = F(0.69) = -0.72(0.69)-1.5 = -2

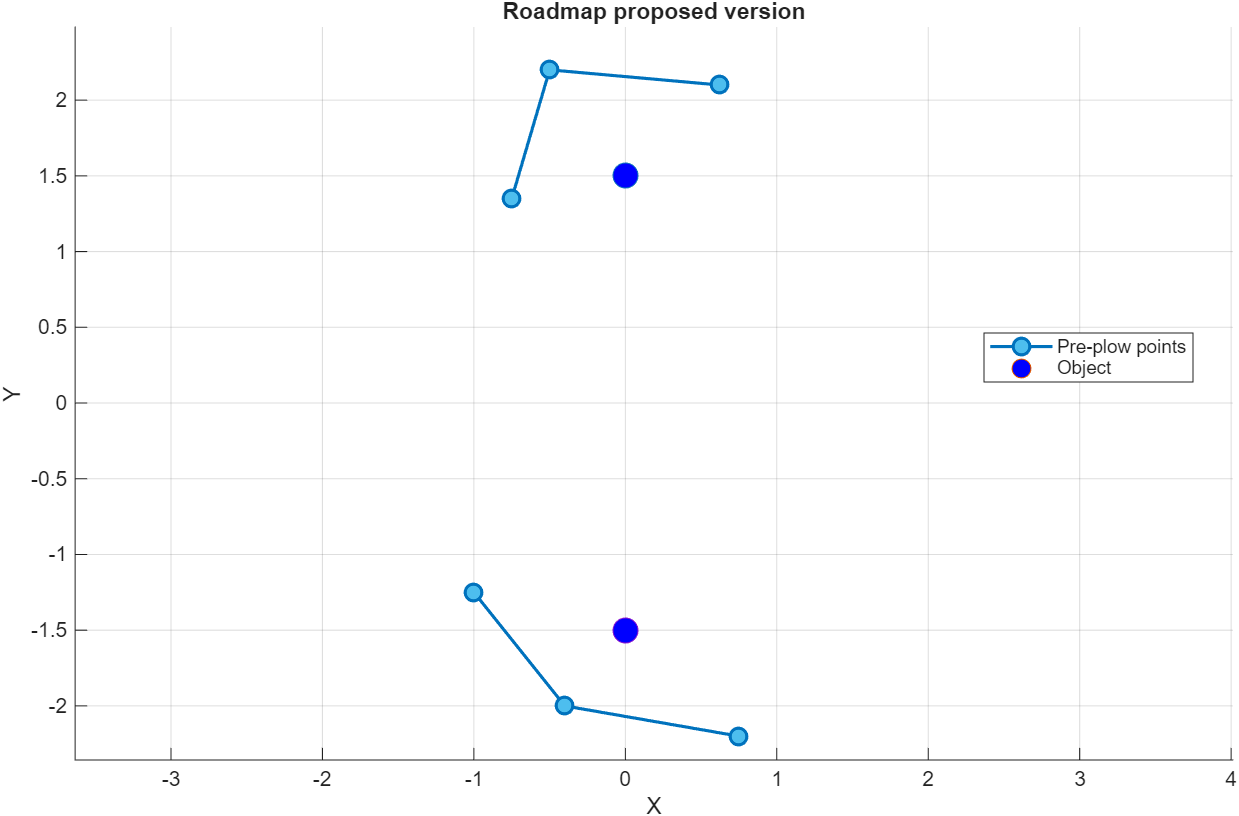
So, our pre-plow position will be[0.69, -2]. This represents the Arena configuration pre-plow position: Arena(blue,\_).

Finally, after applying the same analysis for all six configurations suggested in Table 4.1, we can find the rest of the theoretical pre-plow positions. During the implementation, we noticed that the system doesn’t fit exactly the theoretically calculated positions. We believe that there are some actuators and sensors errors that make the robot go slightly off the expected position. Therefore, we combined the analytical approach with experimentation, to find the pre-plow positions and zone targets shown in Table 4.2

Table 4.2. Pre-plow positions and zone targets found through analysis and experimentation

|  |  |  |
| --- | --- | --- |
| Arena Configuration | PPP | Collection Zone |
| Arena(red, -) | [-0.4 -2] | [2 2.2] |
| Arena(green, -) | [-1 -1.25] | [2.08 -1.9] |
| Arena(blue,-) | [0.75 -2.2] | [-2.25 0] |
| Arena(-,red) | [-0.75 1.35] | [2.2 2.2] |
| Arena(-,green) | [-0.5 2.2] | [2.2 -2.2] |
| Arena(-,blue) | [0.62 2.1] | [-2.2 0] |

We now use the six pre-plow positions to generate our roadmap as shown in Figure 4.5. These points are the minimum required to successfully complete each plow sequence because they are necessary to satisfy each arena configuration.

Figure 4.5. Proposed roadmap configuration. There are two roadmaps. Each roadmap has three points.

## Discussion

Given the roadmap, we made different trials by modifying the colors of the objects to evaluate if the robot succeeds by classifying appropriately the objects into their collection zones. We also recorded the completion time for each trial.

By defining the first object (bottom object) as blue, we can see on Figure 4.6 that the robot reaches the closest point from the roadmap, which is the [-1 –1.25] ; then passes through a second point, and then ends up on a third point [0.75 -2.2] that correspond to the pre-plow position for the blue object and is aligned with the blue collection zone. We can also see in Figure 4.7 that it takes around 44 seconds to bring the object to the desired position.

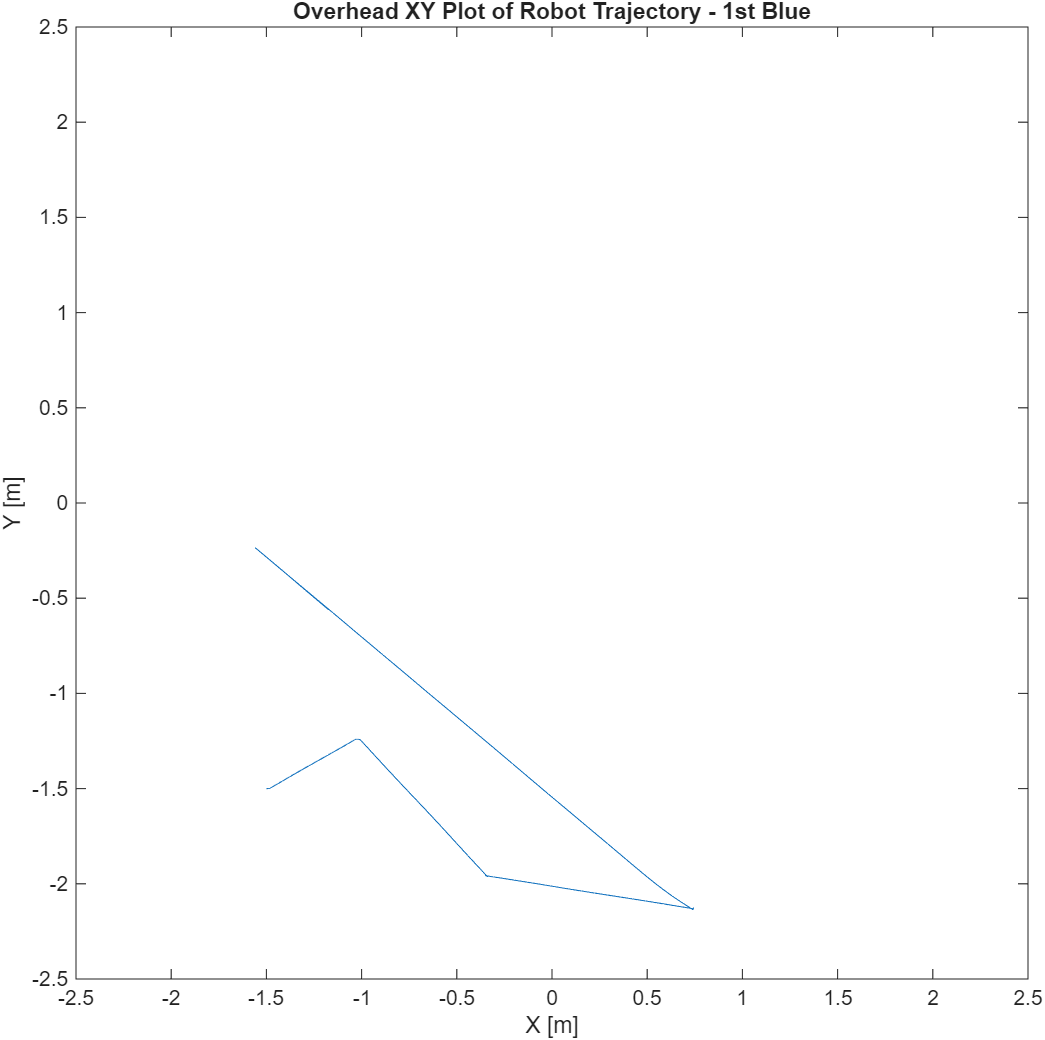


Figure 4.6 Trajectory of 1st object (as blue) plow sequence

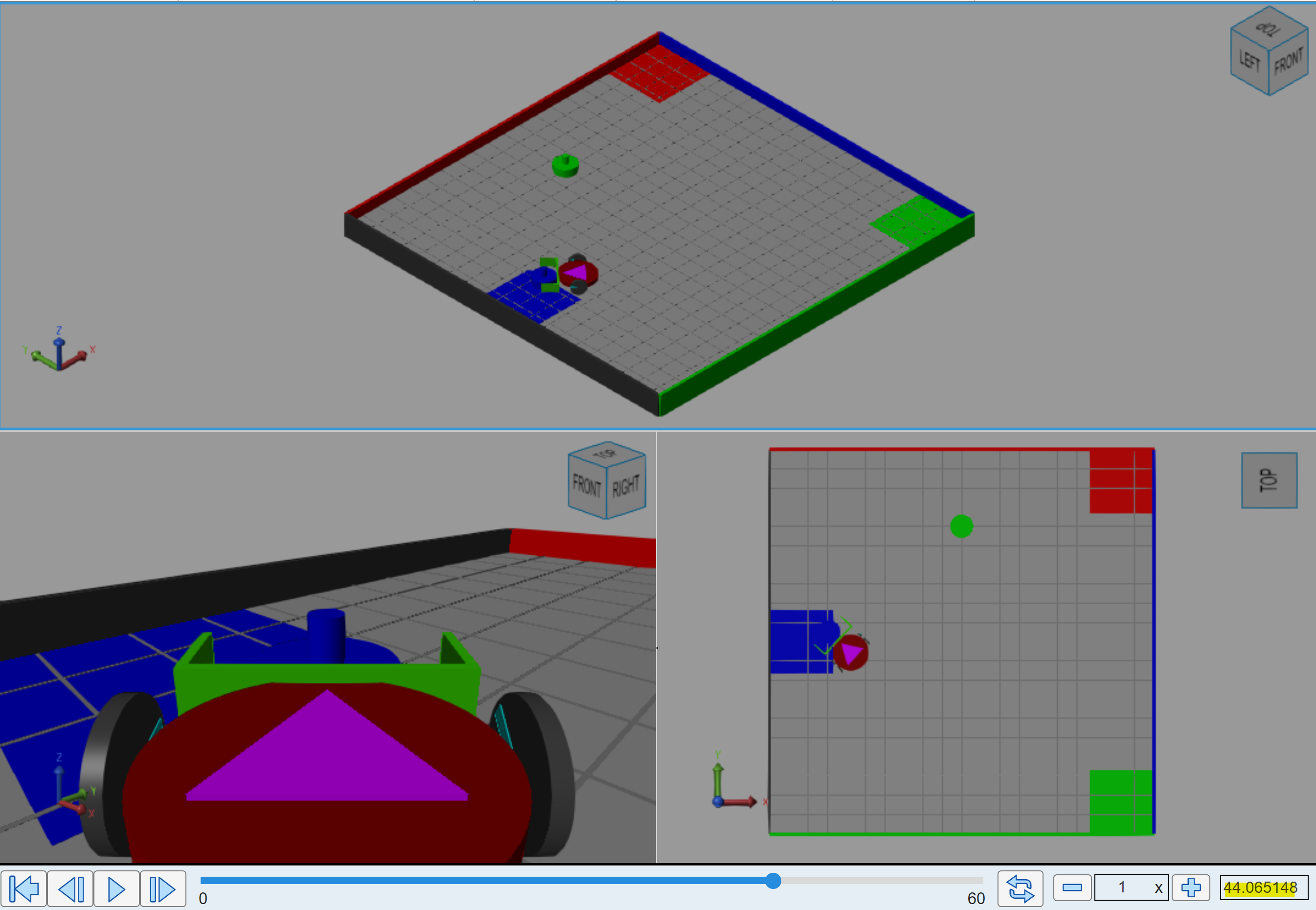


Figure 4.7 Trial duration of 1st object (as blue) plow sequence

By defining the first object (bottom object) as red, we can see on Figure 4.8 that the robot reaches the closest point from the roadmap, which is the [-1 –1.25] ; then passes through a second point [-0.4 -2] that correspond to the pre-plow position for the red object and is aligned with the red collection zone. We can also see in Figure 4.9 that it takes around 36.5 seconds to bring the object to the desired position. Despite the distance is greater than the blue object example, the robot can complete the task earlier because it accelerates in longer distances that doesn’t require too much maneuvering, and therefore, more precision. On the other hand, it only passes by two pre-plow position points, so it saves some time by avoiding going to the third point.

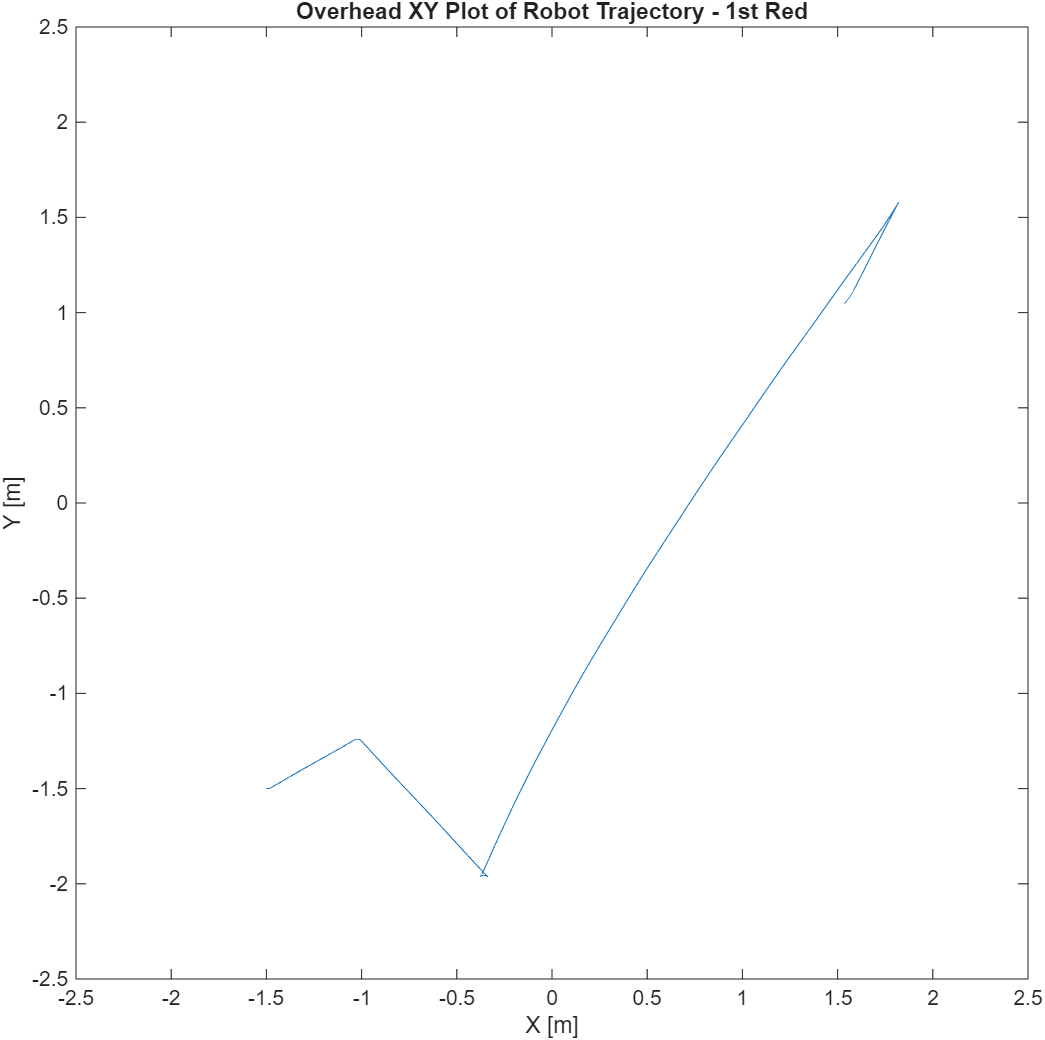


Figure 4.8 Trajectory of 1st object (as red) plow sequence

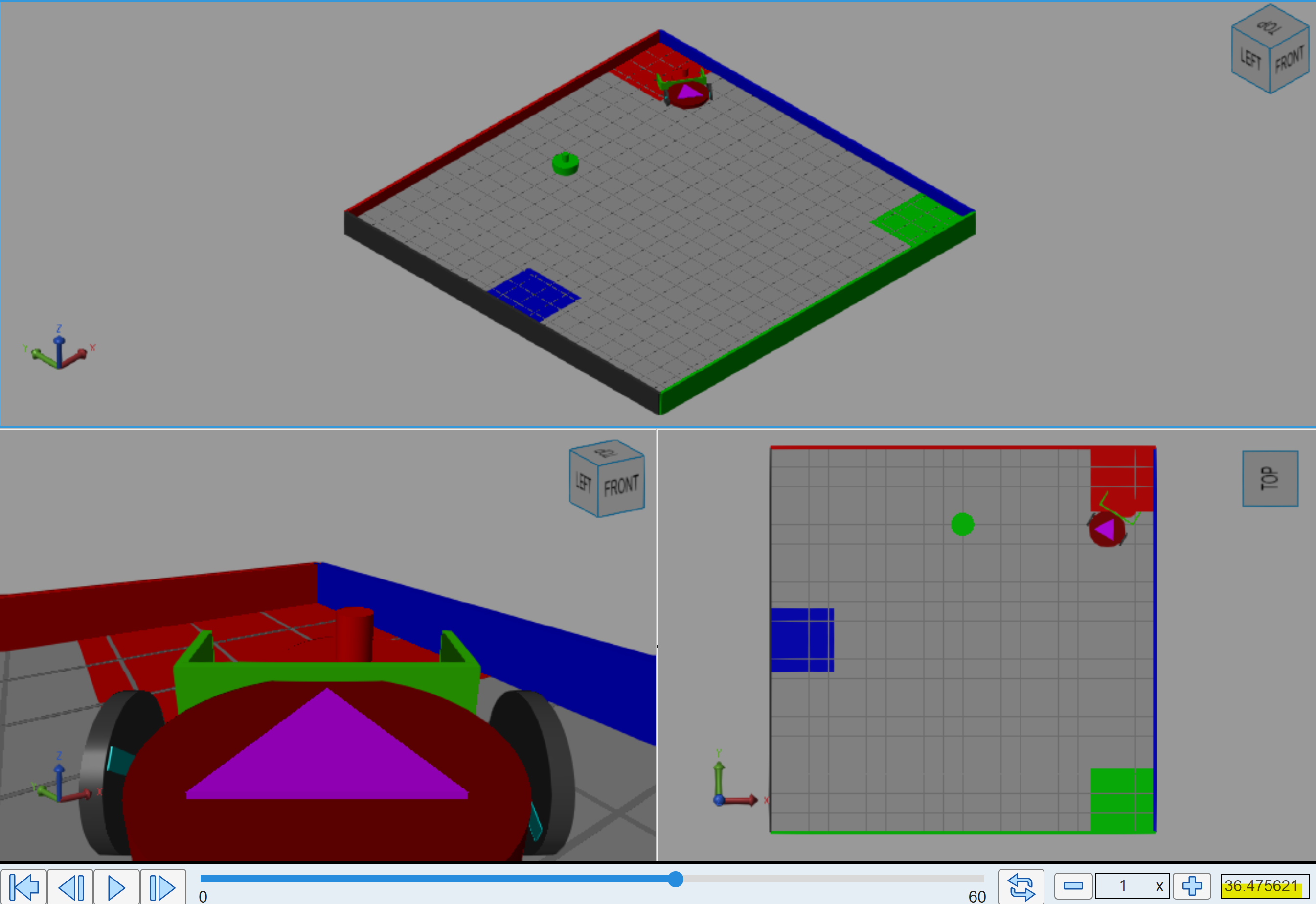


Figure 4.9 Trial duration of 1st object (as red) plow sequence

By defining the first object (bottom object) as green, we can see on Figure 4.10 that the robot reaches the closest point from the roadmap, which is the [-1 –1.25], and it happens to be the pre-plow position for the green object and is aligned with the green collection zone. We can also see in Figure 4.11 that it takes around 22.9 seconds to bring the object to the desired position. Because the object color requires to pass through fewer points in the roadmap and the distance to the collection zone is shorter, then it can navigate the path faster compared to the other configurations discussed earlier.

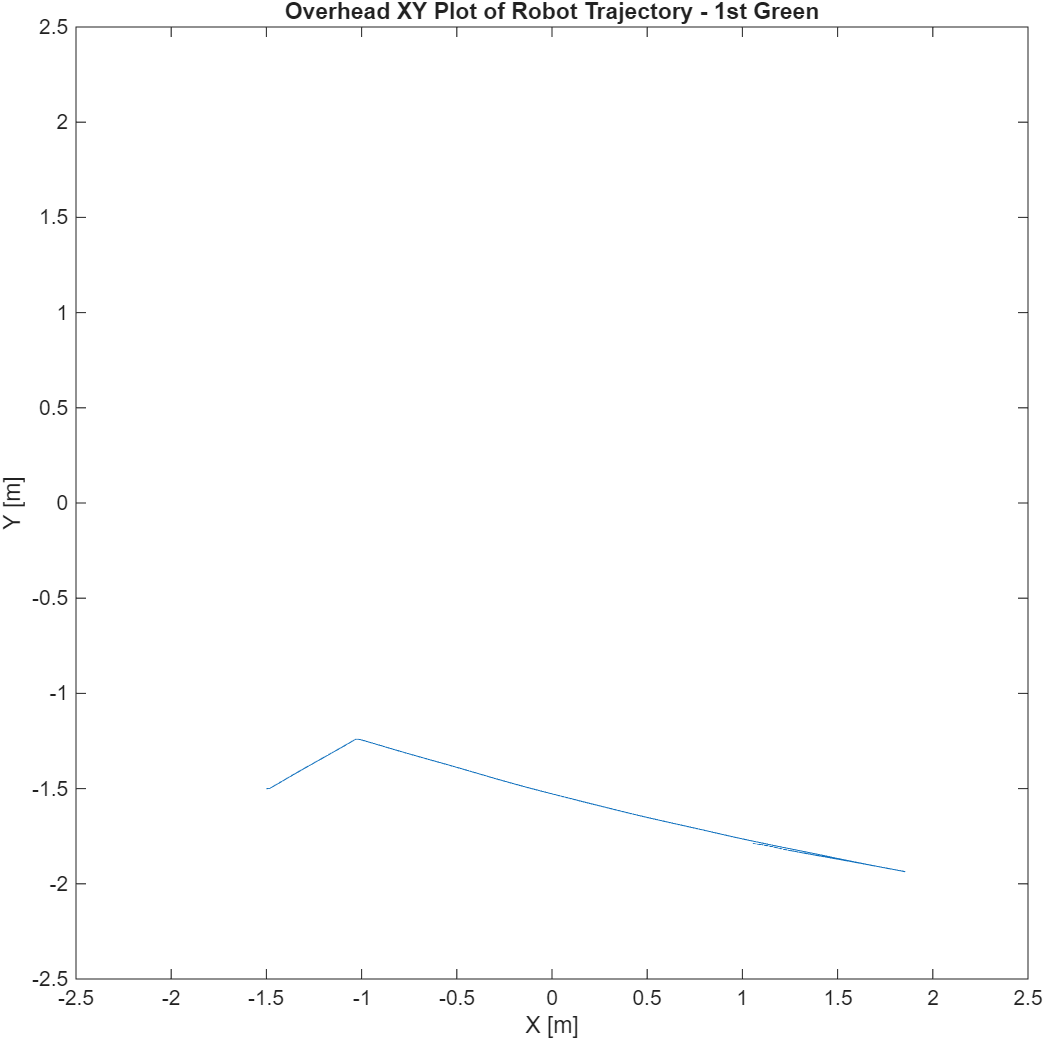


Figure 4.10 Trajectory of 1st object (as green) plow sequence

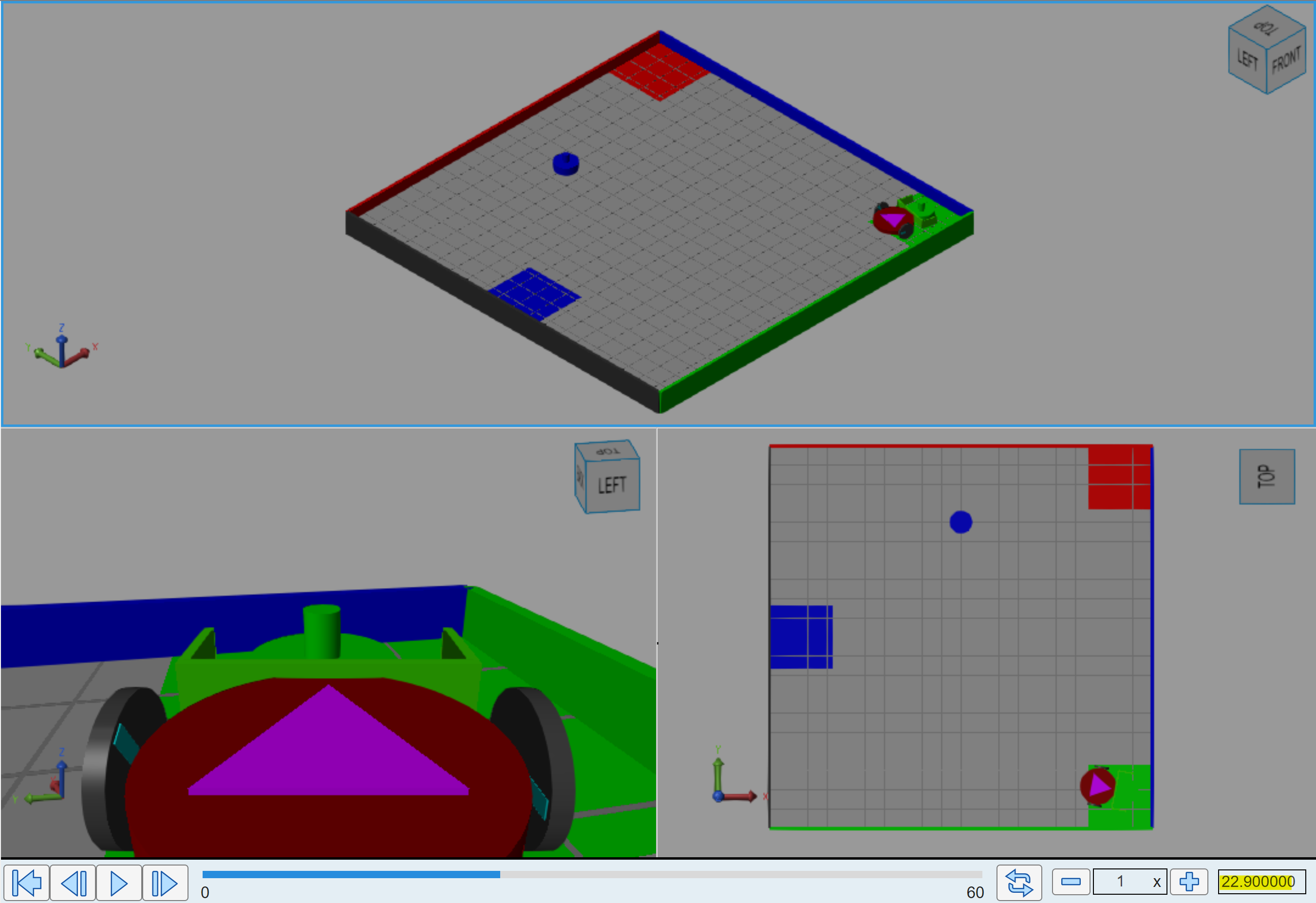


Figure 4.11 Trial duration of 1st object (as green) plow sequence

In summary, we considered the different initial conditions and parameters to verify that the system accomplishes the task of classifying accordingly the color of the objects.

## Movement in Roadmap

Let us combine the roadmap, object and movement views to see how they relate.

Figures 4.12, 4.13 and 4.15 show the navigation algorithm in action for each different color for the first object and in relation to its respective pre-plow position.

A graph with a dotted line

AI-generated content may be incorrect.

Figure 4.12. Navigation for 1st object as green color. Notice that the robot visits only one position on the roadmap.

A graph with a point and a dot

AI-generated content may be incorrect.

Figure 4.13 Navigation for 1st object as blue color. The robot visits three points on the roadmap.

A graph with a dotted line and a dotted line

AI-generated content may be incorrect.

Figure 4.14. Navigation for 1st object as red color. The robot visits two positions on the roadmap.

# Part 5. Algorithms

## Procedure

Besides the main program and the navigation algorithm, we designed other algorithms to accomplish subtasks to such as determining the closets point in a roadmap, planning a path and avoiding collisions. In this section we explain how those algorithms work

To enable autonomous sorting under dynamic conditions, the system relies on two core MATLAB functions: DETERMINE\_CLOSEST\_POINT and PATH\_PLANNER. These algorithms function hierarchically to first locate a safe entry onto the pre-calculated roadmap and then generate a specific sequence of waypoints (navigationMatrix) to guide the robot from its current position to the final collection zone.

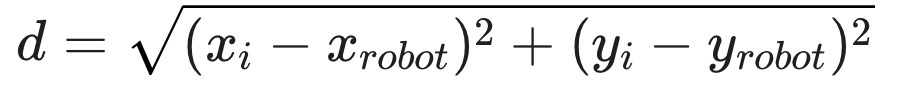
## Results

**Algorithm: DETERMINE\_CLOSEST\_POINT**

The robot's position is variable at the start of each sorting cycle (e.g., after returning from a collection zone). To safely rejoin the pre-defined roadmap without colliding with obstacles, the system must mathematically determine the optimal entry node.

**Function Logic:** The function DETERMINE\_CLOSEST\_POINT accepts the roadmap matrix (roadMap) and the robot’s current coordinates (robotX, robotY) as inputs. It employs a Euclidean distance minimization search:

1. **Initialization:** A minDistance variable is initialized to 5 meters (representing the maximum dimension of the arena) to ensure valid comparison.
2. **Iterative Search:** The algorithm iterates through every coordinate pair (xi ,yi ) in the provided roadmap.
3. **Distance Calculation:** For each point, it calculates the Euclidian distance d to the robot:



1. **Selection:** If the calculated distance d is less than the current minDistance, the algorithm updates the minimum value and records the index of that point.
2. **Output:** The function returns the (x,y) coordinates of the closest roadmap node. This allows the robot to navigate to the "highway" first, ensuring a collision-free path to the object's Pre-Plow Position (PPP).

**Algorithm: PATH\_PLANNER**

The PATH\_PLANNER is the primary decision engine. It constructs a trajectory queue (navigationMatrix) based on three inputs: the target object (Object 0 or Object 1), the detected color of the object, and the robot's proximity to the roadmap.

**Collection Zone Definitions:** The function hardcodes the exact center coordinates for the collection zones. Notably, the coordinates for the top object (Object 1) differ slightly from the bottom object (Object 0) to account for arena symmetry and safe approach angles:

* **Object 0 Zones:** Red [1.75,2.5], Green [1.9,−1.9], Blue [−2.25,0].
* **Object 1 Zones:** Red [2.2,2.2], Green [2.2,−2.2], Blue [−2.2,0].

**Navigation Logic Construction:** The algorithm builds a 3-row matrix where each row represents a sequential waypoint. The logic branches based on the object and color:

**A. Logic for Object 0 (Bottom Object)**

* **Ready-to-Plow Check:** The function first checks if the robot’s closest roadmap point is already the Pre-Plow Point (PPP). If true, it skips the approach phase and sets the navigationMatrix to point directly to the collection zone coordinates, signaling the robot to push immediately.
* **Red (Color 1):** The planner sets a 2-step path:
  + The Pre-Plow Point (PPP).
  + The Red Collection Zone [1.75,2.5].
* **Blue (Color 3):** To avoid the left wall, the planner inserts an intermediate waypoint. The path becomes:
  + The first point of the Object 0 roadmap roadMapObject0(1,:).
  + The Pre-Plow Point (PPP).
  + The Blue Collection Zone [−2.25,0].

**B. Logic for Object 1 (Top Object)** The logic for the top object is more complex due to the risk of colliding with the upper corners of the arena. The algorithm includes specific override coordinates ("Safety Waypoints") if the robot approaches from specific angles:

* **Red (Color 1):** Follows the standard sequence: Approach PPP → Push to Red Zone [2.2,2.2].
* **Blue (Color 3 - Left Approach Override):** If the DETERMINE\_CLOSEST\_POINT function indicates the robot is approaching from the far-left node of the roadmap, the algorithm forces a wide turn to avoid clipping the top-left wall.
  + **Waypoint 1:** Safety Point [−0.75,1.9].
  + **Waypoint 2:** PPP.
  + **Waypoint 3:** Blue Zone [−2.2,−0.25]. *(Note: A Y-offset of -0.25 is applied to the final Blue target to ensure the plow centers the object accurately).*
* **Green (Color 2 - Left Approach Override):** Similar to the Blue logic, if the robot approaches from the left, a safety point is inserted to prevent collision with the central obstacles or upper wall.
  + **Waypoint 1:** Closest Roadmap Point.
  + **Waypoint 2:** Safety Point [−0.5,1.97].
  + **Waypoint 3:** Green Zone [2.2,−2.2].

**Output:** The function returns the fully populated navigationMatrix and a navigationPointer (an integer index) that tells the main control loop which row of the matrix to execute first. This allows the system to dynamically shorten paths (e.g., starting at step 2 instead of step 1) if the robot is already in position.

**Algorithm: WALL AVOINDANCE**

The wall avoidance algorithm modifies a robot motor according to the distance measured by the distance sensors (left and right). If the robot is close to a wall in the right, the left motor speed decreases to command the robot to swerve left. If the robot is close to the left wall, the right motor speed decreases to command the robot to swerve right.

We designed the adjustments as a linear function that has a threshold. On Figure 5.1 we can see that after reaching a threshold of 0.8m, the robot will begin to decelerate. The closer the robot is from the wall, the lower the speed or the motor.

A graph with red lines and numbers

AI-generated content may be incorrect.

Figure 5.1. Wall avoidance design.

**Algorithm: FAST-SLOW OPERATION**

We designed two types of navigation operations: a fast-inaccurate operation and a slow-precise operation. Table 5.1 summarizes when each operation is used.

Table 5.1. Type of navigation operation usage

|  |  |  |
| --- | --- | --- |
| Use Case | Fast-Inaccurate | Slow-Precise |
| Long Distance trip | Yes | No |
| Approach roadmap | Yes | No |
| Navigate in roadmap | No | Yes |
| Approach home | Yes | No |
| Navigate to home | No | Yes |

Notice that we use fast-inaccurate operation when we need to get to a general area close to a point of interest and then we use slow-precise operation to get to the actual point of interest with increased precision.

In our implementation we defined long distance trip as a trip longer than 3.2m. This is a tunable parameter.

## Discussion

We designed algorithms to complete all subtasks required in the main program. The development of these algorithms required a lot of trial and error by testing all possible distinct configurations.

In particular, the path planning algorithm posed a challenge since the pre-plow positions needed adjustment when working on the second (top) object. For the top object, there are two approaches (from left and from right) and at times one approach would work while the other would not work. We had to override some of the waypoints in the roadmap for some of the configurations so we could be successful in all trials.

# Part 6. Trials

## Procedure

In this section we show the results of the trials for the six possible configurations.

## Results

### Arena (blue, red)

Figure 6.1 shows the robot motion when the bottom object is blue, and the top object is red.

A line graph with numbers

AI-generated content may be incorrect.

Figure 6.1. Robot motion for Arena (blue, red)

### Arena (green, red)

Figure 6.2 shows the robot motion when the bottom object is green, and the top object is red.

A diagram of a robot

AI-generated content may be incorrect.

Figure 6.2. Robot motion for Arena (green, red)

### Arena (red, blue)

Figure 6.3 shows the robot motion when the bottom object is red, and the top object is blue.

A line graph with numbers

AI-generated content may be incorrect.

Figure 6.3. Robot motion for Arena (red, blue)

### Arena (red, green)

Figure 6.4 shows the robot motion when the bottom object is red, and the top object is green.

A graph of a line graph

AI-generated content may be incorrect.

Figure 6.4. Robot motion for Arena (red, green)

### Arena (blue, green)

Figure 6.5 shows the robot motion when the bottom object is blue, and the top object is green.

A line graph with numbers

AI-generated content may be incorrect.

Figure 6.5. Robot motion for Arena (blue, green)

### Arena (green, blue)

Figure 6.6 shows the robot motion when the bottom object is green, and the top object is blue.

A graph of a line graph

AI-generated content may be incorrect.

Figure 6.6. Robot motion for Arena (green, blue)

## Discussion

### Accuracy and Repeatability

Our final implementation can successfully relocate the center of the objects to the correct collection zones in all six possible configurations. The process is repeatable and successful every time we run the simulation. That said, in the real world, the position of the objects could be slightly different and it’s possible that the current implementation needs to be further tuned.

### Time to completion

We measured the time that the robot took to complete the goal, and we observed different results depending of the initial object configuration. Table 6.1 shows the time to completion for all cases. We can see that the fastest completion time was achieved for Arena(red, blue) taking ~84.9 seconds while the slowest completion time was observed for Arena(green, blue) taking 117.9 seconds. We can see that traveled distance and roadmap points visited are the main factors affecting time to completion.

Table 6.1. Completion time for all possible configurations

|  |  |
| --- | --- |
| Initial condition | Completion time (seconds) |
| Arena (blue, red) | 97.2 |
| Arena (green, red) | 95.3 |
| Arena (red, blue) | 84.9 |
| Arena (red, green) | 107.3 |
| Arena (blue, green) | 113.8 |
| Arena (green, blue) | 117.9 |

### Level of performance

Given the arena, robot and object conditions and requirements our solution achieved all proposed goals. The robot can sort two objects in any of six required configurations.

## Videos

Best trial: <https://youtube.com/shorts/r1UMVo4w6Bs?feature=share>

All six conditions: <https://www.youtube.com/shorts/JweTuLRPeMM>

# Part 5: Student Feedback

We faced difficulties tuning the parameters of our control system. At times we would fix a trial but break another trial. Solving these issues required us to think differently and reassess our initial approaches.

We estimate that completing this project took us 80 men-hours.

We loved working on this project and learned a lot. We would recommend you keep it for future students.

# Conclusions

Robotic systems are inaccurate so relying on odometry is not enough. We can use a combination of sensor reading and algorithmic control to reduce these inaccuracies. We have proved these types of control systems allow us to correct errors and accomplish desired outcomes.

Our implementation works properly for sorting objects. However, it is constrained by the initial conditions of the environment, that is, initial positions remain the same and the collection zones remain in the same position. If these conditions were to change, additional sensors and algorithms may be necessary to meet the new requirements.

There is a trade-off between precision and speed. When speed is increased, precision decreases and vice versa. We were able to use this fact by using both types of operations (fast-inaccurate and slow-precise) at different stages of the main program execution.

# Appendices

## Appendix A: Main program – main\_robot\_slx

|  |
| --- |
| function [leftMotor, rightMotor, debugVal] = main\_robot( ...  redAngle, ...  redDistance, ...  greenAngle, ...  greenDistance, ...  blueAngle, ...  blueDistance, ...  robotX, ...  robotY, ...  robotAngle, ...  wallLeftDistance, ...  wallRightDistance ...  )  % Robot States  % 0 = Find closets roapmap point  % 1 = Navigate to a point  % 2 = Navigate over multiple points of a roadmap  % 3 = something else  % Object Colors  % 0 = Red  % 1 = Green  % 2 = Blue  function [leftMotor, rightMotor] = commandMotor(motion, distance)  if motion == "spin-left"  leftMotor = -30;  rightMotor = 30;  elseif motion == "spin-right"  leftMotor = 30;  rightMotor = -30;  elseif motion == "stop"  leftMotor = 0;  rightMotor = 0;  elseif motion == "forward"  leftMotor = 75;  rightMotor = 75;  elseif motion == "forward-fast"  leftMotor = 127;  rightMotor = 127;  elseif motion == "reverse"  leftMotor = -75;  rightMotor = -75;  elseif motion == "swerve-right"  leftMotor = 127;  rightMotor = 127\*distance/0.8;  elseif motion == "swerve-left"  leftMotor = 127\*distance/0.8;  rightMotor = 127;  end  end  persistent robotState  %persistent objectLocationX  %persistent objectLocationY  persistent objectColor  persistent currentObject  persistent targetPositionsMatrix  persistent targetPositionPointer  persistent jobVector  persistent jobVectorPointer  persistent delivery  persistent dropFlag  persistent longDistanceTrip  if isempty(robotState)  %robotState = 1.11;  robotState = 0;  end  %if isempty(objectLocationX)  % objectLocationX = -1.5;  %end  %if isempty(objectLocationY)  % objectLocationY = 0;  %end  if isempty(objectColor)  objectColor = -1;  end  if isempty(currentObject)  currentObject = 0;  end  if isempty(targetPositionsMatrix)  %targetPositionsMatrix = zeros(1,2);  %targetPositionsMatrix = zeros(2,2);  targetPositionsMatrix = zeros(3,2);  end  if isempty(targetPositionPointer)  targetPositionPointer = 1;  end  if isempty(jobVector)  jobVector = zeros(1,10);  end  if isempty(jobVectorPointer)  jobVectorPointer = 1;  end  if isempty(delivery)  delivery = 0;  end  if isempty(dropFlag)  dropFlag = 0;  end  if isempty(longDistanceTrip)  longDistanceTrip = -1;  end  % Default values for all paths  leftMotor = 0;  rightMotor = 0;  debugVal = robotState;  objectDistance = NaN;  objectAngle = NaN;  % Roadmap around object 0 (bottom)  roadMapObject0 = [-0.3 -1.9; -1 -1.25; 0.75 -2.2];  % Roadmap around object 1 (top)  %roadMapObject1 = [-0.75 1.35; -0.6 1.97; 0.52 2.01];  %roadMapObject1 = [-0.75 1.35; -0.5 1.97; 0.52 2.01]; %works for  %blue-green    % initial  %roadMapObject1 = [-0.75 1.35; -0.5 2.1; 0.62 2.1];  roadMapObject1 = [-0.75 1.35; -0.5 2.1; 0.52 2.01];  % robotState 0: MAIN CONTROL  % List of jobs:  % 1 -> Determine closets point in roadmap  % 2 -> Navigate to closest point  % 3 -> Plan a Path from current position to collection zone  % 4 -> Navigate to from current position to collection zone  % 5 -> Determine disengage position  % 6 -> Navigate to disengage position  % 7 -    if robotState == 0  % is current job complete, move to next job  if jobVector(1,jobVectorPointer)  jobVectorPointer = jobVectorPointer + 1  end  if jobVectorPointer == 1  % determine closets point in roadmap  robotState = 2  elseif jobVectorPointer == 2  % navigate to closest point  robotState = 1.11  elseif jobVectorPointer == 3  % determine path  robotState = 3  elseif jobVectorPointer == 4  % Navigate to collection zone via roadmap  delivery = 1  robotState = 1.11  elseif jobVectorPointer == 5  %Determine disengage position  robotState = 5  elseif jobVectorPointer == 6  % Navigate to disengage position  robotState = 6  elseif jobVectorPointer == 7  pr = 'in job 7 main control--->'  % Set a new system loop to operate on Object 1  if currentObject == 0  currentObject = 1  jobVector = zeros(1,10);  jobVectorPointer = 1  robotState = 0  longDistanceTrip = 1  elseif currentObject == 1  jobVector(1,jobVectorPointer) = 1;  else  error('currentObject must be 0 or 1')  end  elseif jobVectorPointer == 8  % Set up home navigation  robotState = 8  elseif jobVectorPointer == 9  % Navigate to home position  longDistanceTrip = 1  robotState = 1.11  elseif jobVectorPointer == 10  % Goal achieved. Do nothing.  robotState = 9  end    end  % robotState 1: Navigate to 1 or more points  if robotState >= 1 && robotState < 2  %targetPositionPointer  xTarget = targetPositionsMatrix(targetPositionPointer,1)  yTarget = targetPositionsMatrix(targetPositionPointer,2)  longDistanceTrip  % if longDistanceTrip has not been computed  if longDistanceTrip == -1  tripDistance = sqrt((robotX-xTarget)^2 + (robotY-yTarget)^2)  if tripDistance >= 3.21  longDistanceTrip = 1;  else  longDistanceTrip = 0;  end  end  % robotState 1.1: Navigate to a point  if robotState >= 1.1 && robotState < 1.2  if robotState == 1.11  % spin to find segment end  angleTarget = atan2d(yTarget - robotY, xTarget - robotX)    if robotAngle > angleTarget - 2 && robotAngle < angleTarget + 2  [leftMotor, rightMotor] = commandMotor("stop");  robotState = 1.12  else  % convert angles to a 0-360 range  ra = mod(robotAngle, 360);  ta = mod(angleTarget, 360);  angle\_difference = mod((ta - ra + 180), 360) - 180;    % Choose spin direction  if angle\_difference > 0  [leftMotor, rightMotor] = commandMotor("spin-left");  elseif angle\_difference < 0  [leftMotor, rightMotor] = commandMotor("spin-right");  else  % Already facing the target angle  [leftMotor, rightMotor] = commandMotor("stop");  end  end  end    if robotState == 1.12  % move to segment end  if dropFlag == 1 || longDistanceTrip == 1  tolerance = 0.8;  else  tolerance = 0.1;  end  if robotX > xTarget - tolerance && robotX < xTarget + tolerance && robotY > yTarget - tolerance && robotY < yTarget + tolerance  [leftMotor, rightMotor] = commandMotor("stop");  robotState = 1.2;  else  if dropFlag == 1 && longDistanceTrip == 1  if wallLeftDistance <= 0.8  [leftMotor, rightMotor] = commandMotor("swerve-right", wallLeftDistance);  elseif wallRightDistance <= 0.8  [leftMotor, rightMotor] = commandMotor("swerve-left", wallRightDistance);  else  [leftMotor, rightMotor] = commandMotor("forward-fast");  end  elseif dropFlag == 1  [leftMotor, rightMotor] = commandMotor("forward-fast");  else  [leftMotor, rightMotor] = commandMotor("forward");  end  end  end  end  % robotState 1.2: increase pointer or finish navigation  if robotState == 1.2  % check there are pending location to visit  if size(targetPositionsMatrix,1) > targetPositionPointer  targetPositionPointer = targetPositionPointer + 1  longDistanceTrip = -1;  % if this is the last segment of a delivery operation  if targetPositionPointer == size(targetPositionsMatrix,1) && delivery == 1  dropFlag = 1;  end  robotState = 1.11  else % all positions have ben visited  longDistanceTrip = -1;  delivery = 0;  dropFlag = 0;  % reset targetPositionPointer  targetPositionPointer = 1;  % Mark current job as completed  jobVector(1,jobVectorPointer) = 1;  robotState = 0;  end  end  end  % robotstate 2: Find closest point in roadmap  if robotState == 2  pr = 'determining closest point'  if currentObject == 0  [x, y] = DETERMINE\_CLOSEST\_POINT(roadMapObject0, robotX, robotY);  else  [x, y] = DETERMINE\_CLOSEST\_POINT(roadMapObject1, robotX, robotY);  end  % set pointer to 2 so only the last x y is used  targetPositionPointer = 3;  targetPositionsMatrix = [0 0;0 0;x y];  % Test moves for debugging  %targetPositionsMatrix = [0 0; 0 0];  %targetPositionsMatrix = [0 0; 2 -2];  %targetPositionsMatrix = [-0.4 -2; 2 2.2]; % Arena(red, \_)  % Mark current job as completed  jobVector(1, jobVectorPointer) = 1;  robotState = 0;  end  % Robot looks for object. Spin until object found  % Robots determines color of object. R: 1, G:2, B:3  % Robot determines pre-plow position for the given color  % Robot determines path to reach ppp and collection zone  if robotState == 3  if isnan(redDistance) && isnan(greenDistance) && isnan(blueDistance)  if robotX < 0 && robotY < 0 || robotX < 0 && robotY > 0  % second or third quadrant  [leftMotor, rightMotor] = commandMotor("spin-right");  elseif robotX > 0 && robotY > 0  % first quadrant  [leftMotor, rightMotor] = commandMotor("spin-left");  else  [leftMotor, rightMotor] = commandMotor("spin-right");  end    else % object detected  [leftMotor, rightMotor] = commandMotor("stop");  if ~isnan(redDistance)  objectColor = 1  elseif ~isnan(greenDistance)  objectColor = 2  elseif ~isnan(blueDistance)  objectColor = 3  else  debugVal = -1;  end  % Determine pre-plow position for given color  if currentObject == 0  pppX = roadMapObject0(objectColor,1)  pppY = roadMapObject0(objectColor,2)  %[x, y] = DETERMINE\_CLOSEST\_POINT(roadMapObject0, robotX, robotY);  else  pppX = roadMapObject1(objectColor,1)  pppY = roadMapObject1(objectColor,2)  %[x, y] = DETERMINE\_CLOSEST\_POINT(roadMapObject1, robotX, robotY);  end  [targetPositionsMatrix, targetPositionPointer] = PATH\_PLANNER(roadMapObject0, roadMapObject1, currentObject, objectColor, robotX, robotY, pppX, pppY)  targetPositionsMatrix  targetPositionPointer  % Mark current job as completed  pr = "Path planning"  jobVector(1,jobVectorPointer) = 1;  robotState = 0;  return  end  end  if robotState == 5 % WIP  % calculate position that will create separation with object  desired\_separation = 0.5;  x = robotX - desired\_separation \* cosd(robotAngle);  y = robotY - desired\_separation \* sind(robotAngle);    targetPositionsMatrix(3,1) = x;  targetPositionsMatrix(3,2) = y;  targetPositionPointer = 3;  % Mark current job as completed  jobVector(1,jobVectorPointer) = 1;  robotState = 0;  end    % Navigate in reverse to disengage object  if robotState == 6  xTarget = targetPositionsMatrix(targetPositionPointer,1)  yTarget = targetPositionsMatrix(targetPositionPointer,2)  tolerance = 0.2;  if robotX > xTarget - tolerance && robotX < xTarget + tolerance && robotY > yTarget - tolerance && robotY < yTarget + tolerance  [leftMotor, rightMotor] = commandMotor("stop");  pr = 'job 6 DONE'  jobVector(1,jobVectorPointer) = 1;  robotState = 0;  else  [leftMotor, rightMotor] = commandMotor("reverse")  end  end  if robotState == 8    targetPositionsMatrix = [0 0;0 0;0 0]  targetPositionPointer = 2;  % Mark current job as completed  jobVector(1,jobVectorPointer) = 1;  robotState = 0;  end  end |

## Appendix B: Main program – DETERMINE\_CLOSEST\_POINT

|  |
| --- |
| function [x, y] = DETERMINE\_CLOSEST\_POINT(roadMap, robotX, robotY)  % This function determines the closest point in the roadmap with respect to  % robot's current position.  % Returns x, y of the closest position.  minDistance = 5;  resultIndex = 0;  for i= 1:1:size(roadMap, 1)  distance = sqrt((roadMap(i,1) - robotX)^2 +(roadMap(i,2) - robotY)^2);  if distance < minDistance  minDistance = distance;  resultIndex = i;  end  end  x = roadMap(resultIndex, 1);  y = roadMap(resultIndex,2);  end |

## Appendix C: Main program – PATH\_PLANNER

|  |
| --- |
| function [navigationMatrix, navigationPointer] = PATH\_PLANNER(roadMapObject0, roadMapObject1, currentObject, objectColor, robotX, robotY, pppX, pppY)  % Collection zone positions [r; g; b]  collectionZonePositionObject0 = [1.75 2.5; 1.9 -1.9; -2.25 0];  collectionZonePositionObject1 = [2.2 2.2; 2.2 -2.2; -2.2 0];  % initial values  %collectionZonePositionObject0 = [2 2.3; 1.9 -1.9; -2.25 0];  %collectionZonePositionObject1 = [2.2 2.2; 2.2 -2.2; -2.2 0];  % default values  navigationPointer = 3;  navigationMatrix = [0 0;0 0;0 0];  %currentObject = 0;  if currentObject == 0  [x, y] = DETERMINE\_CLOSEST\_POINT(roadMapObject0, robotX, robotY);  if [x, y] == [pppX, pppY]  % Add collection point to navigation. Ready to plow!  x = collectionZonePositionObject0(objectColor,1);  y = collectionZonePositionObject0(objectColor,2);  % add x, y to navigation  navigationMatrix = [0 0;0 0;x y];  return  end  % Plan a path to get to ppp  if objectColor == 1  navigationPointer = navigationPointer - 1;  navigationMatrix(navigationPointer,1) = pppX;  navigationMatrix(navigationPointer,2) = pppY;  elseif objectColor == 3  % add ppp to navigation  navigationPointer = navigationPointer - 1;  navigationMatrix(navigationPointer,1) = pppX;  navigationMatrix(navigationPointer,2) = pppY;  % add intermediate point to navigation  navigationPointer = navigationPointer - 1;  x = roadMapObject0(1,1);  y = roadMapObject0(1,2);  navigationMatrix(navigationPointer,1) = x;  navigationMatrix(navigationPointer,2) = y;  else  error("error in plan planning. Object color invalid")  end  x = collectionZonePositionObject0(objectColor,1);  y = collectionZonePositionObject0(objectColor,2);  navigationMatrix(3,1) = x;  navigationMatrix(3,2) = y;  return  else  % Object 1 - WIP: similar pattern to Object0    if objectColor == 1 % Red  % Add ppX, ppY and collection point to navigation.  x = collectionZonePositionObject1(objectColor,1);  y = collectionZonePositionObject1(objectColor,2);  navigationPointer = 2;  navigationMatrix = [0 0;pppX pppY;x y];  return  elseif objectColor == 3  [x, y] = DETERMINE\_CLOSEST\_POINT(roadMapObject1, robotX, robotY);  if x == roadMapObject1(1,1) && y == roadMapObject1(1,2)  % if approaching from the left  navigationMatrix(1,1) = -0.75;  navigationMatrix(1,2) = 1.9;  navigationMatrix(2,1) = pppX;  navigationMatrix(2,2) = pppY;  x = collectionZonePositionObject1(objectColor,1);  y = collectionZonePositionObject1(objectColor,2);  navigationMatrix(3,1) = x;  navigationMatrix(3,2) = y-0.25;  navigationPointer = 1;  else  % Add ppX, ppY and collection point to navigation.  x = collectionZonePositionObject1(objectColor,1);  y = collectionZonePositionObject1(objectColor,2);  navigationPointer = 2;  navigationMatrix = [0 0;pppX pppY;x y];  end  elseif objectColor == 2 % green  % Add closest point in roadmap to navigation  [x, y] = DETERMINE\_CLOSEST\_POINT(roadMapObject1, robotX, robotY);  navigationMatrix(1,1) = x;  navigationMatrix(1,2) = y;  % Add pre\_plow position to navigation  if x == roadMapObject1(1,1) && y == roadMapObject1(1,2)  % if approaching from the left, use another ppt to avoid  % collision  navigationMatrix(2,1) = -0.5;  navigationMatrix(2,2) = 1.97;  else  navigationMatrix(2,1) = pppX;  navigationMatrix(2,2) = pppY;  end  % Add collection zone position to navigation  x = collectionZonePositionObject1(objectColor,1);  y = collectionZonePositionObject1(objectColor,2);  navigationMatrix(3,1) = x;  navigationMatrix(3,2) = y;  navigationPointer = 1;  else  error('PLATH\_PLANNER: objectcolor invalid')  end  end  end |