DEVELOPMENT OF NAVIGATION METHODS FOR MOBILE ROBOTS IN CLUTTERED ENVIRONMENTS

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY MEGHALAYA
SAITSOHPEN SOHRA, EAST KHASI HILLS, MEGHALAYA, INDIA- 793108

Presented by:
Shivesh Kumar (B22CS038)
Gurijala Meghana (B22CS019)
Karipireddy Surya Teja Gopal Reddy (B22CS022)

Under the Supervision of
Dr. Ngangbam Herojit Singh

CONTENTS

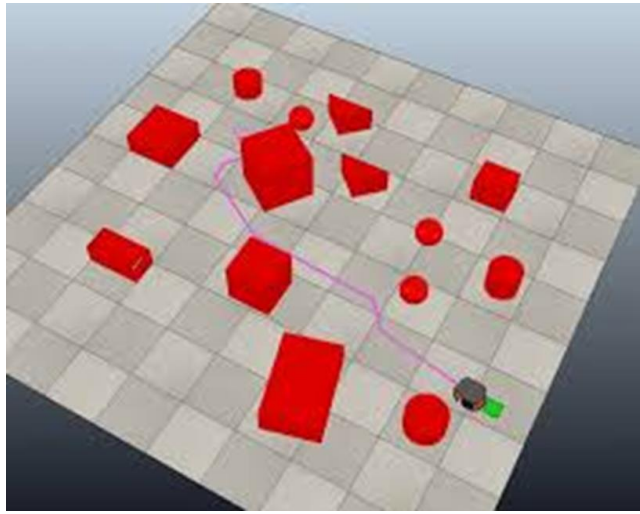
INTRODUCTION

- Importance of mobile robots in industry, rescue, and home
- Challenge: navigation in cluttered/dynamic environments
- Objective: Implement and test basic navigation methods in simulation

TOOLS & PLATFORM

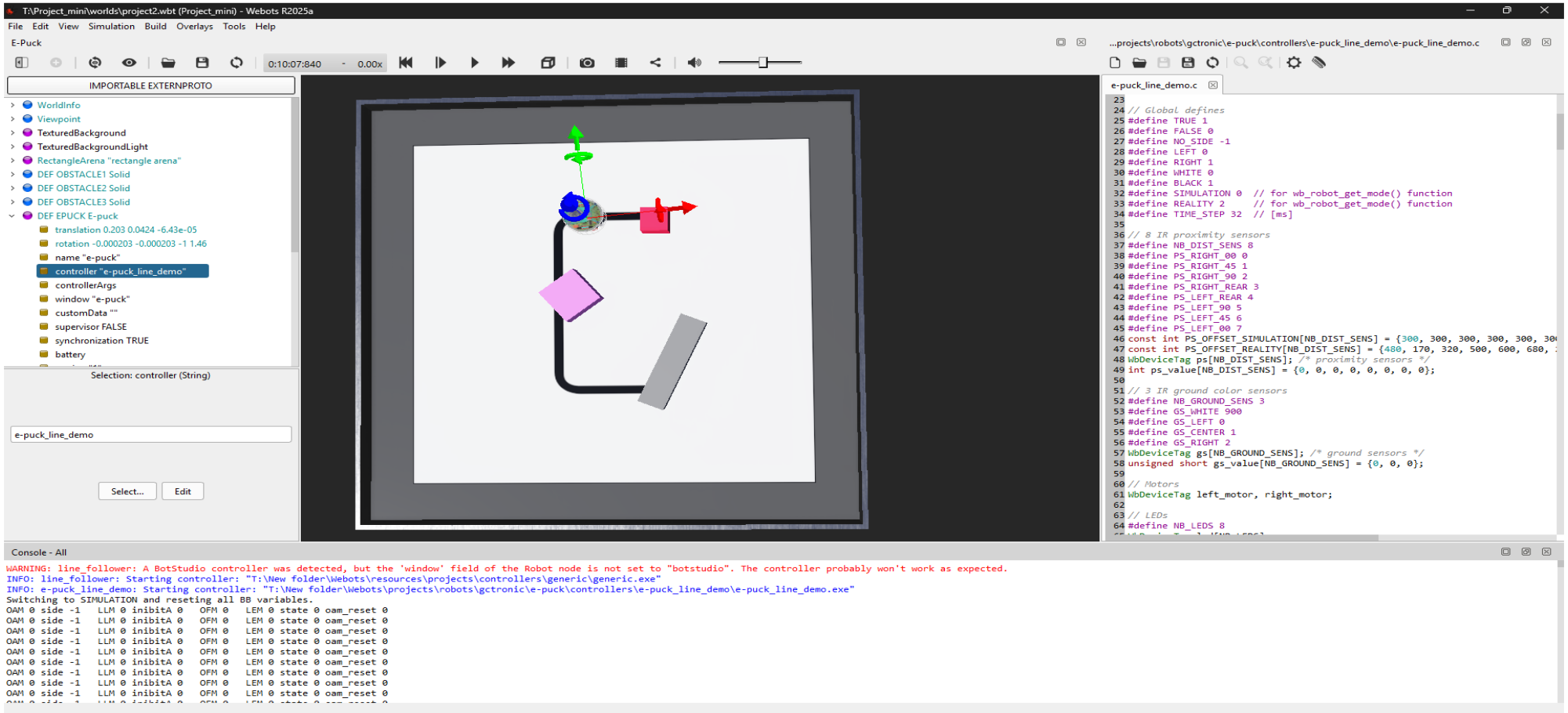
Simulation Environment: Webots

- Open-source 3D robot simulator developed by Cyberbotics.
- Provides realistic physics, sensor simulation, and robot-environment interaction.
- **Key Features:**
  - Built-in support for various robot models (e.g., e-puck, drones, humanoids).
  - Rich sensor simulation: IR sensors, LIDAR, cameras, GPS, etc.
  - Programmable using multiple languages: **Python**, **C/C++**, **Java**, **MATLAB**.
  - Integrated 3D environment editor for custom scenarios.
  - Physics engine for realistic motion, collisions, and friction.



TOOLS & PLATFORM

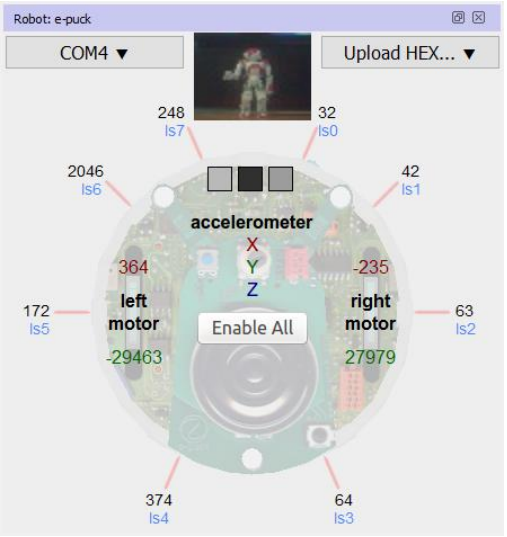
Webots Workspace



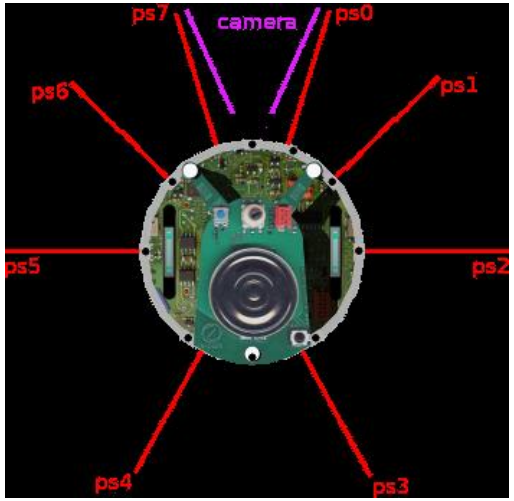
TOOLS & PLATFORM

Robot Used: e-puck (Simulated in Webots)

- Compact differential-drive robot designed for education and research.
- **Specifications:**
  - Diameter: 71 mm | Height: 50 mm | Weight: 160 g
  - Max speed: 0.25 m/s | Rotation speed: 6.28 rad/s
- **Sensors:**
  - 8 Infrared proximity sensors
  - Ground sensors (for line detection)
  - Camera, accelerometer, gyroscope, LEDs
- Fully programmable and easy to integrate into simulated environments.



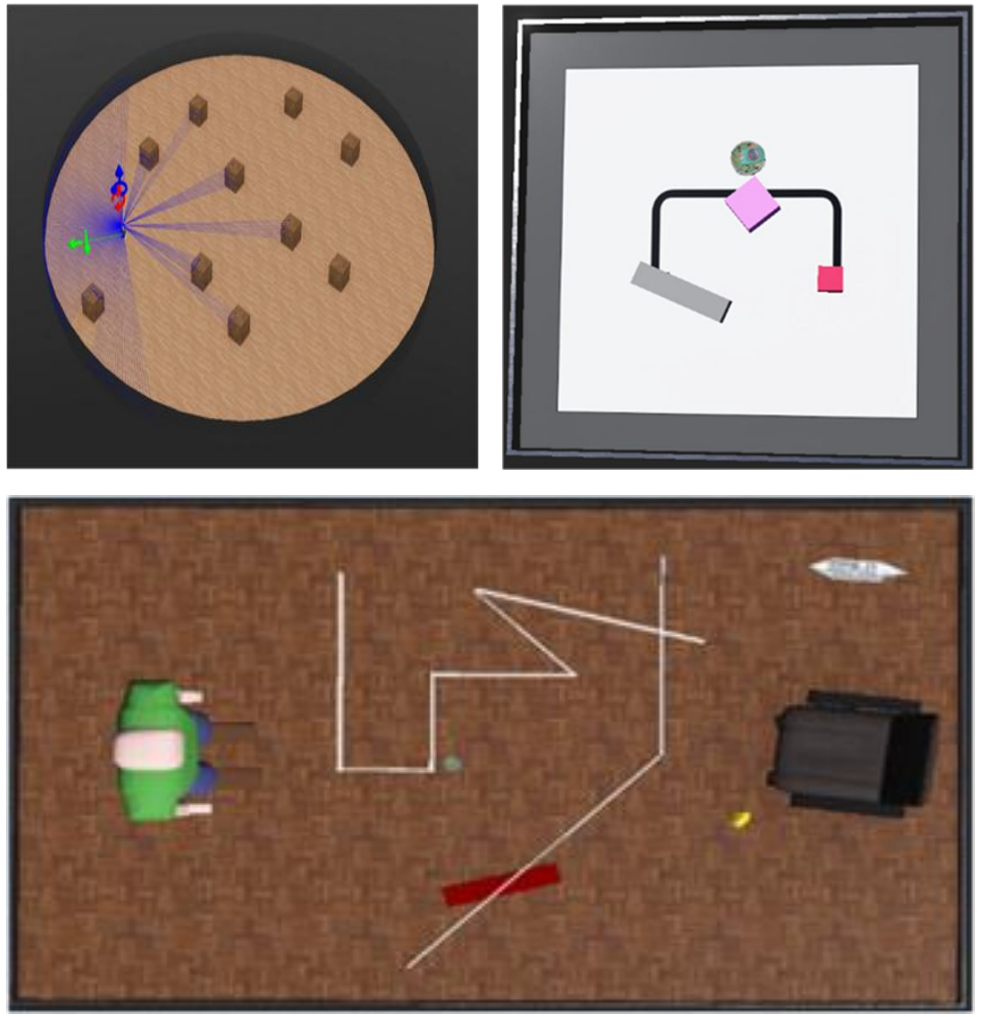
e-puck robot window



Sensors, LEDs and Camera

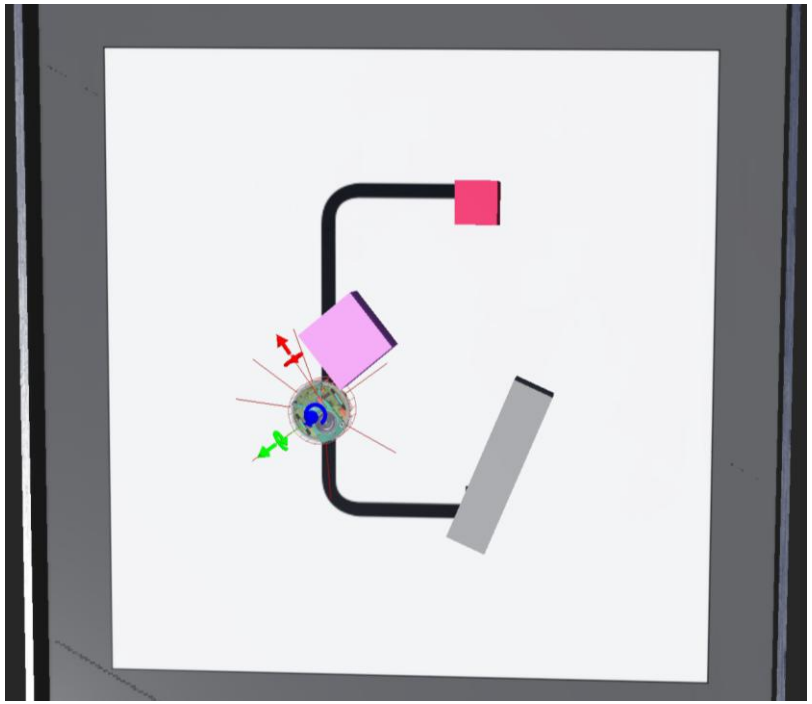
NAVIGATION TECHNIQUES OVERVIEW

- **Line Following**
- **Wall Following**
- **Obstacle Avoidance**
- **Robot Vacuum Navigation**
- **Target Finding & Obstacle Avoidance**



LINE FOLLOWING

Technical Details:



- **Sensors Used:**
  - **3 IR ground sensors** → detect the position of the black line.
  - **8 proximity sensors** → detect nearby obstacles.
- **Main Modules:**
  - **Line Following Module (LFM):**
    - Uses sensor values to steer the robot along the line.
    - A proportional control system adjusts wheel speeds based on line deviation.
  - **Obstacle Avoidance Module (OAM):**
    - Activates when obstacles are detected.
    - Uses proximity data to steer the robot away from the object.

LINE FOLLOWING

- **Line Leaving Module (LLM):**
  - Detects deviation or loss of the line.
  - Prepares for detour or transition.
- **Obstacle Following Module (OFM):**
  - Allows the robot to trace alongside an obstacle until it can return to the path.
- **Line Entering Module (LEM):**
  - Helps the robot re-detect and align with the line after avoiding an obstacle.

```
### Code Logic
if center_sensor detects line:
    left_speed = max_speed
    right_speed = max_speed
    # print("Move straight")
elif left_sensor detects line:
    left_speed = max_speed * 0.5
    right_speed = max_speed
    # print("Veer slightly right to center")
elif right_sensor detects line:
    left_speed = max_speed
    right_speed = max_speed * 0.5
    # print("Veer slightly left to center")
else:
    left_speed = -max_speed
    right_speed = -max_speed
    # print("Line lost – reverse to search")
```

Behavior Control:

- A **finite state machine** manages transitions between these modules.
- Ensures smooth, coordinated behavior between following and avoiding.

WALL FOLLOWER

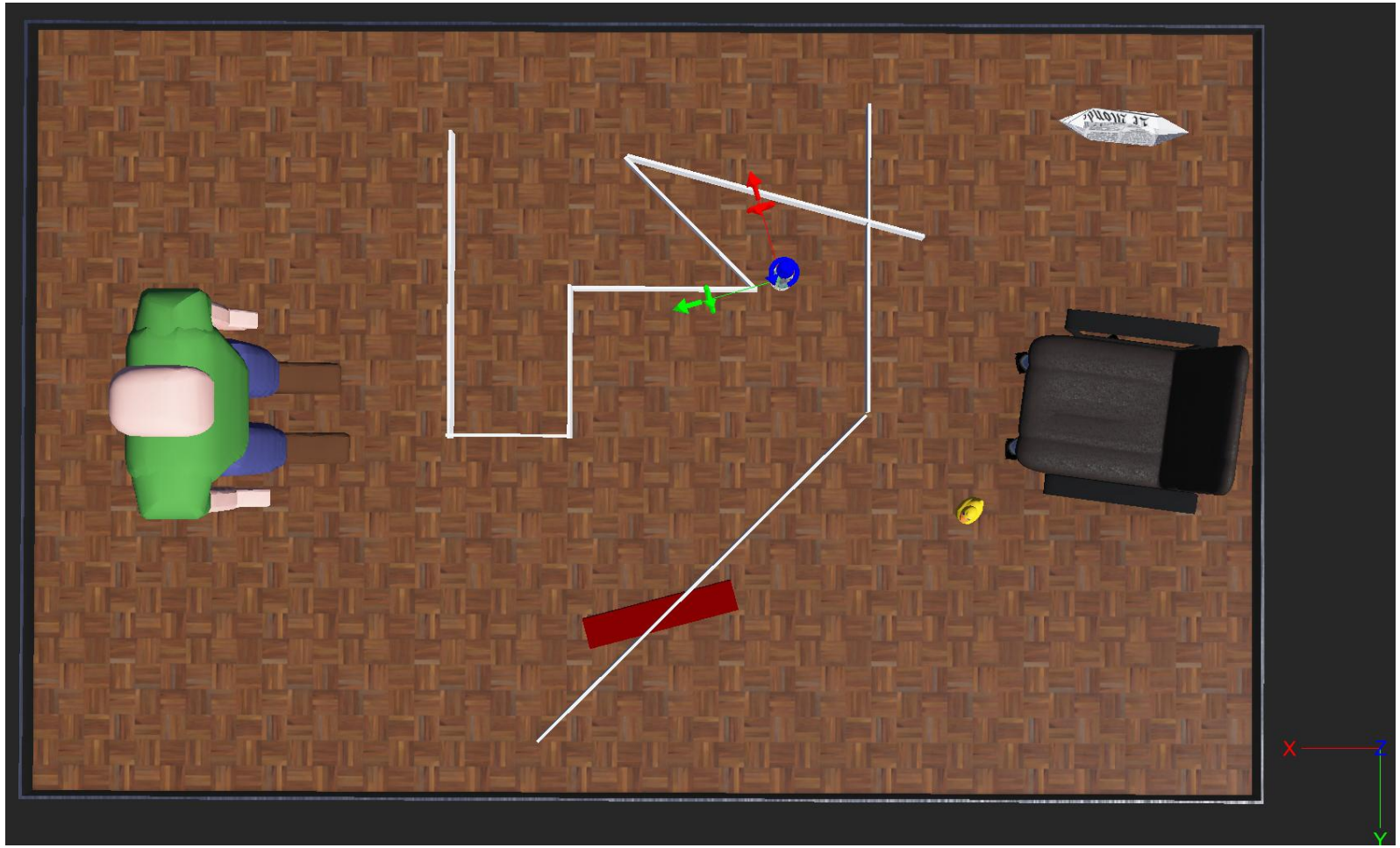
Demonstrates a simple yet effective wall-following robot using proximity sensors and reactive behaviour.

- **Sensors Used:**
  - **Proximity sensors** (typically on the sides and front) detect nearby surfaces or walls.
- **Logic Implemented:**

The robot constantly reads its proximity sensors and makes navigation decisions based on detected obstacles. The **left-hand wall-following algorithm** operates with the following logic:

- **Front wall detected** → Robot turns right.
- **No wall on the left** → Robot turns left to search for the wall.
- **Wall on the left** → Robot moves straight forward to maintain distance.

WALL FOLLOWER



```
### Code Logic
if front_wall:
    left_speed = max_speed
    right_speed = -max_speed
    #print("Front blocked – turning right")
elif not left_wall:
    left_speed = max_speed * 0.25
    right_speed = max_speed
    #print("No wall on left – turning left")
else:
    left_speed = max_speed
    right_speed = max_speed
    #print("Wall on left – moving forward")
```

AVOID OBSTACLES

- To develop a **reactive obstacle avoidance system** using **LIDAR-based sensing** and **Braitenberg-inspired behavior** to enable smooth navigation in dense, cluttered environments.
- **Working Principle:**

Inspired by Braitenberg vehicles, this approach uses sensor-motor coupling to generate smooth, real-time reactions to obstacles.

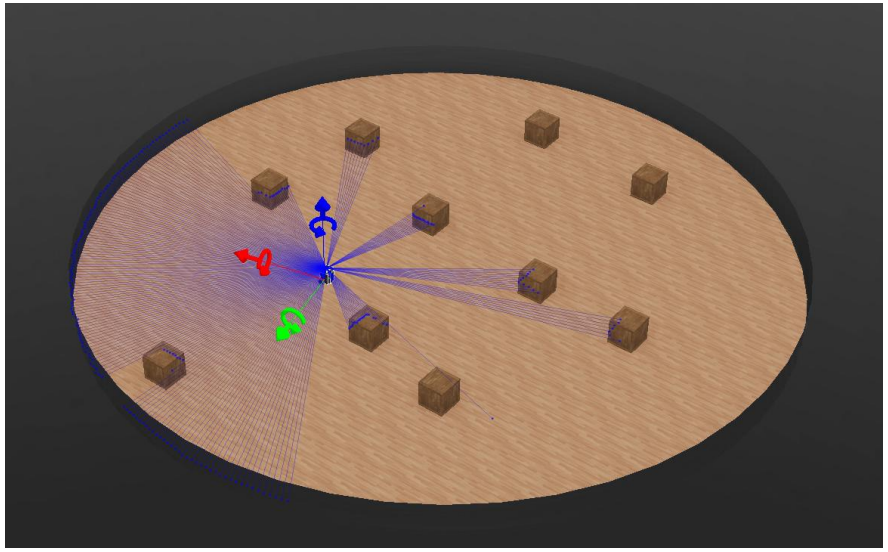
Sensor Input :

- A 2D LIDAR scan returns distances to nearby objects in all directions.
- A selective angular window (25% to 50% of scan width) is processed for detecting relevant frontal obstacles.

AVOID OBSTACLES

Control Strategy:

- For each pair of readings (left and right):
  - The **closer** an obstacle is, the **stronger** the repulsion on that side.
  - Robot **turns away** from the side where the obstacle is closer.
- The system uses **differential wheel speed control**:
  - Increase speed on the side **opposite** to the obstacle.
  - This creates a smooth, curved turn.



AVOID OBSTACLES

Mathematical Model (Braitenberg Logic):

Let:

- Di = Distance on the **left** (angle *i*)
- Dj = Distance on the **right** (mirror of *i*)
- R = Maximum LIDAR range
- Gk = Gaussian coefficient (more weight to front sensors)

Then:

- left_speed += Gk * ((1 - Di/R) - (1 - Dj/R)) = Gk * (Dj - Di)/R
- right_speed += Gk * ((1 - Dj/R) - (1 - Di/R)) = Gk * (Di - Dj)/R
- D i <D j → obstacle is closer on the left → turn right
- D j <D i → obstacle is closer on the right → turn left
- D i ≈D j → move forward

```
### wheel velocity calculation
left_speed += coefficient *
  (left_obstacle_strength -
   right_obstacle_strength);
right_speed += coefficient *
  (right_obstacle_strength -
   left_obstacle_strength);
```

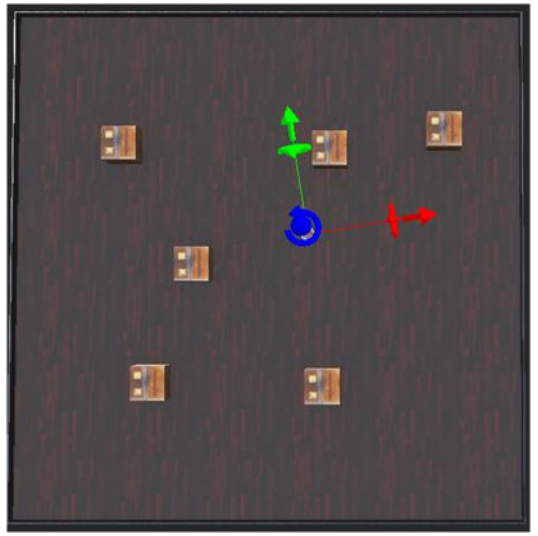
```
### Braitenberg Logic
left_speed += braitenberg_coefficients[k]
  * ((1.0 - lidar_values[i] / max_range) -
  (1.0 - lidar_values[j] / max_range));
right_speed += braitenberg_coefficients[k] *
  ((1.0 - lidar_values[j] / max_range) -
   (1.0 - lidar_values[i] / max_range));
```

ROBOT VACUUM NAVIGATION

Objective:

Optimizing robot vacuum navigation involves improving how efficiently and effectively the robot covers an area while avoiding obstacles and returning to the dock. Several algorithms for the given navigation method are as follow:

1. Random Algorithm
2. Zigzag Algorithm
3. Spiral Algorithm
4. Snake Algorithm

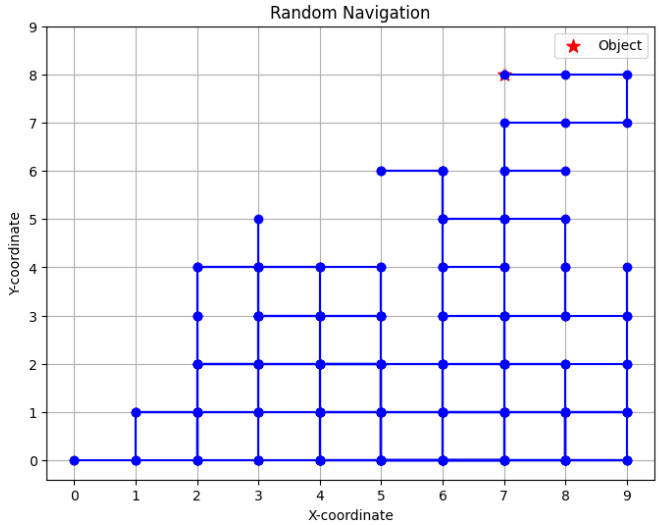


ROBOT VACUUM NAVIGATION

1. Random Navigation

- The robot moves in random directions.
- Changes direction when it hits or approaches an obstacle.
- **Simple to implement**, but **inefficient** and prone to repeating paths.

```
def random_navigation(self):
    if self.read_front_obstacle():
        turn = random.choice([-1, 1])
        self.set_velocity(SPEED * turn, -SPEED * turn)
    else:
        if random.random() < 0.05:
            turn = random.choice([-1, 1])
            self.set_velocity(SPEED * turn, -SPEED * turn)
        else:
            self.set_velocity(SPEED, SPEED)
```

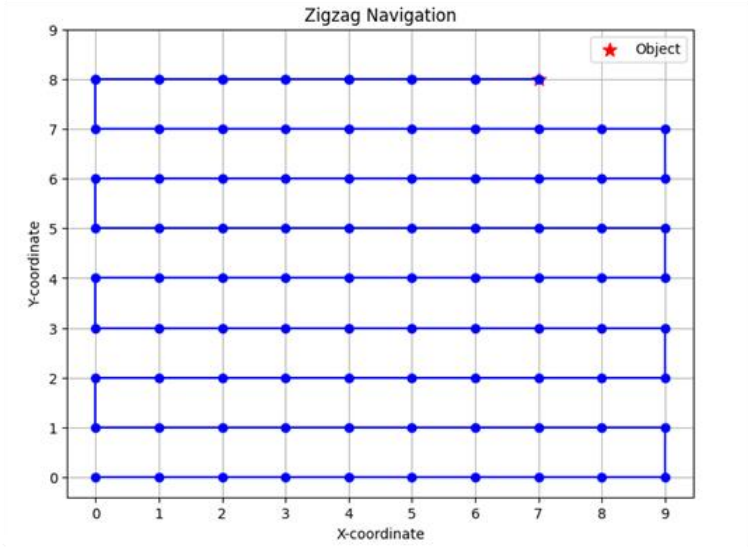


ROBOT VACUUM NAVIGATION

2. Zigzag Navigation

- Robot moves in alternating straight lines at a slight angle.
- Turns about 180° with a small offset when hitting a wall.
- Helps ensure **gradual forward movement** and better area coverage.

```
def zigzag_navigation(self):
    if self.read_front_obstacle():
        self.set_velocity(TURN_SPEED, -TURN_SPEED)
        self.zigzag_dir *= -1
    else:
        if self.zigzag_dir == 1:
            self.set_velocity(SPEED, SPEED * 0.8)
        else:
            self.set_velocity(SPEED * 0.8, SPEED)
```

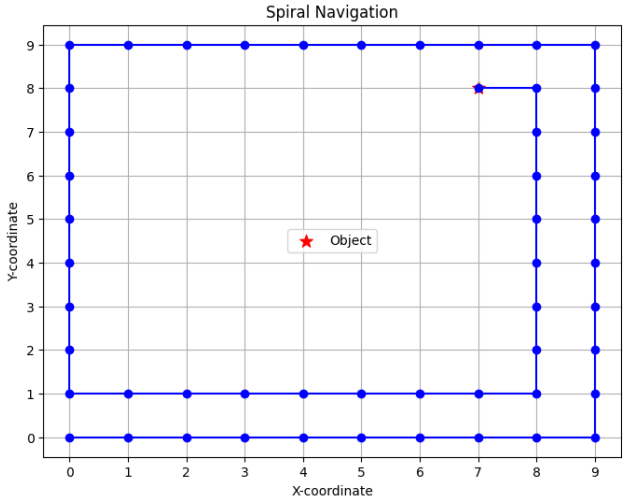


ROBOT VACUUM NAVIGATION

3. Spiral Navigation

- Starts from the center, expands outward in a rectangular spiral.
- Increases path length after every two turns.
- **Systematic and efficient** in empty areas.
- May struggle near **edges** or with **obstacles**.

```
def spiral_navigation(self):
    if self.read_front_obstacle():
        self.set_velocity(-SPEED, SPEED)
    else:
        self.spiral_angle += 0.01
        left = SPEED * (1.0 - math.sin(self.spiral_angle))
        right = SPEED
        self.set_velocity(left, right)
```

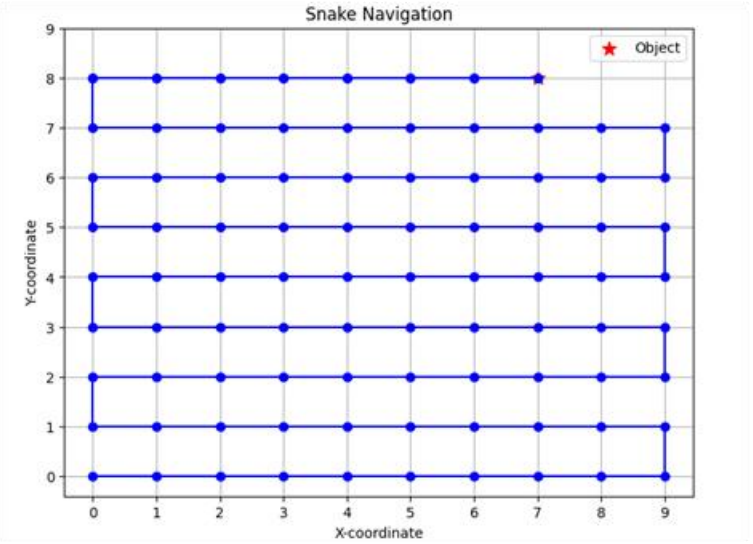


ROBOT VACUUM NAVIGATION

4. Snake Navigation

- Moves in **parallel straight lines**, like a typewriter.
- After each row, it shifts slightly and reverses direction.
- Guarantees **full area coverage** with **minimal overlap**.

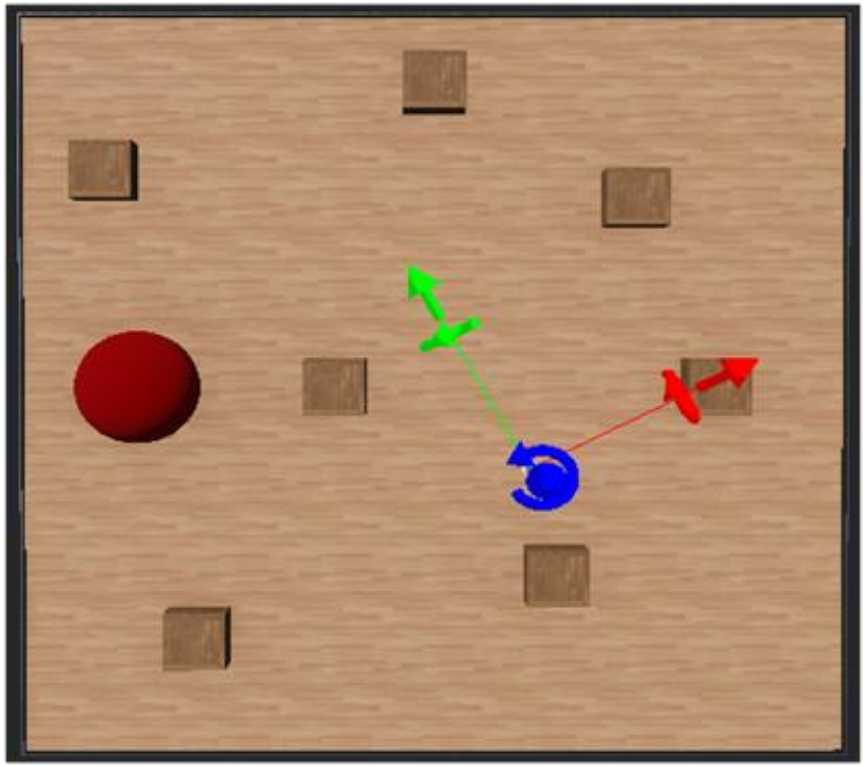
```
def snake_navigation(self):
    if self.read_front_obstacle():
        self.snake_toggle = not self.snake_toggle
        if self.snake_toggle:
            self.set_velocity(SPEED, -SPEED)
        else:
            self.set_velocity(-SPEED, SPEED)
    else:
        self.set_velocity(SPEED, SPEED)
```



TARGET FINDING & OBSTACLE AVOIDANCE

To design and evaluate a **lightweight, deterministic navigation system** that allows the **e-puck robot** to:

- Locate a **specified target** (e.g. colored object),
- Avoid static obstacles using **infrared (IR) proximity sensors**,
- Reach the goal with **low computational cost**.



TARGET-FINDING

Bug2 Algorithm – Principle:

Bug2 is a **minimal-resource path planning** method ideal for small robots in unknown or partially known environments.

1.Follow M-Line:

- Move directly toward the goal along a virtual straight line (called the **M-line**).

2.Obstacle Encounter:

- If the robot detects an obstacle (via IR), it switches to **wall-following mode**.

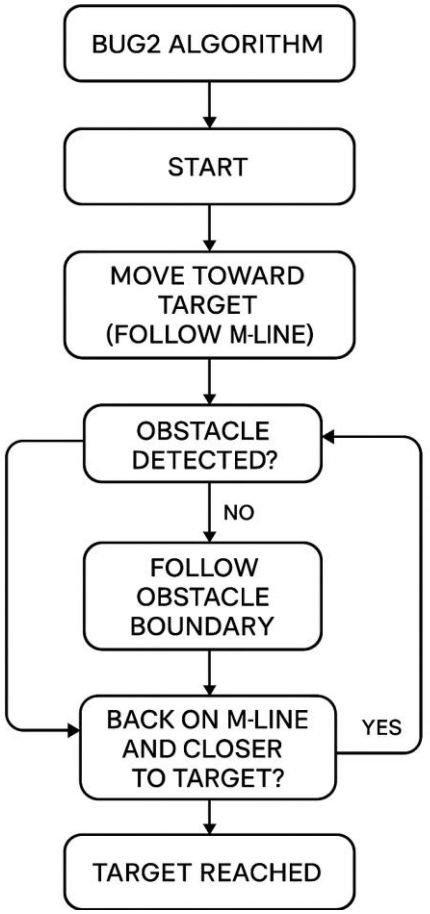
3.Wall-Following:

- It follows the obstacle's boundary using the **left-hand or right-hand rule**.

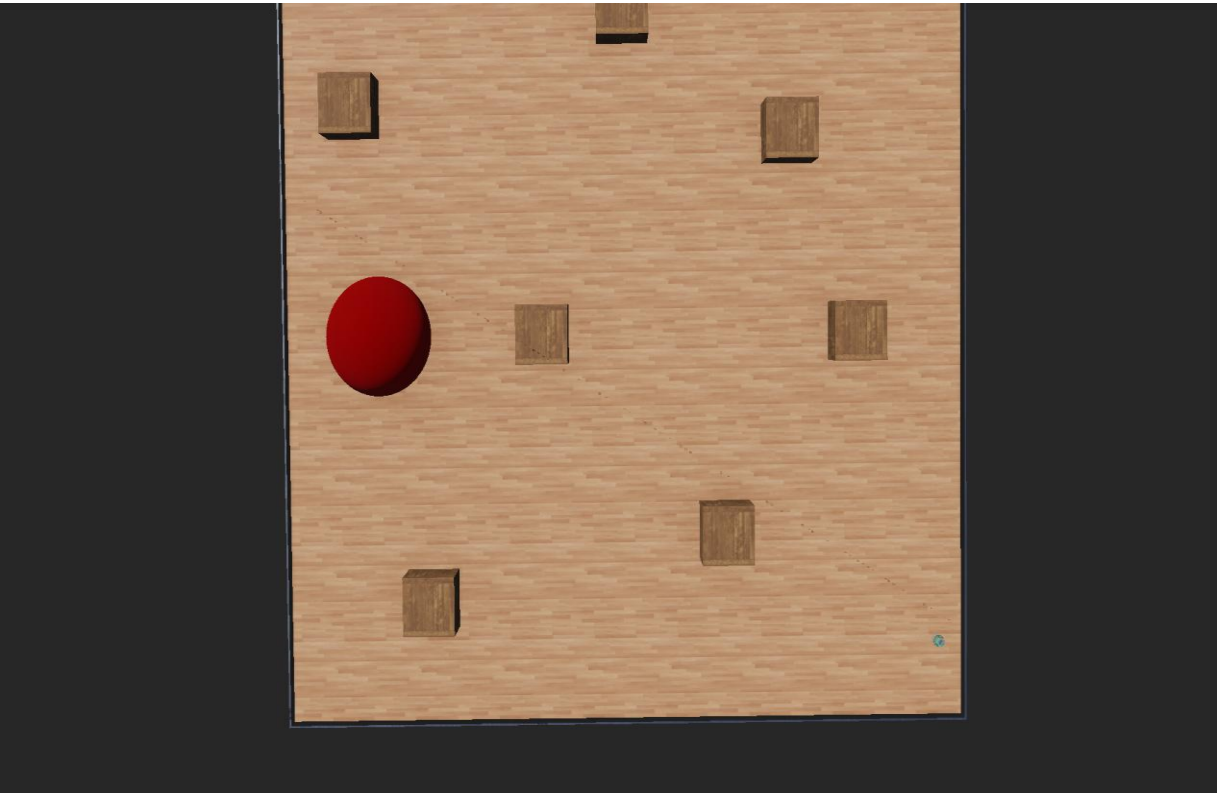
4.Rejoining M-Line:

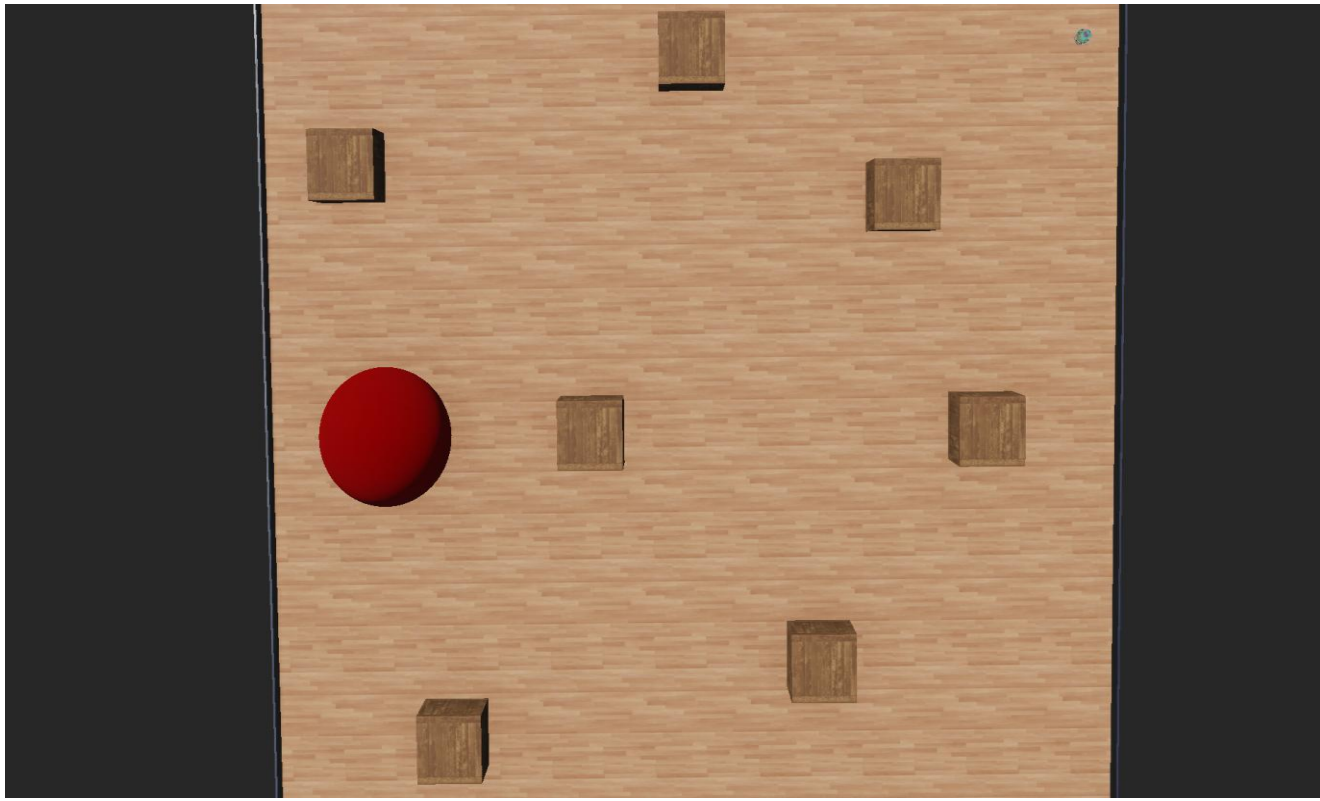
- When it re-encounters the M-line at a point **closer to the goal**, it resumes **straight-line movement**.

FLOWCHART



TARGET-FINDING





RESULTS & DISCUSSION

Key Observations by Method:

- **Bug2 Algorithm:**
  - **Most successful** in goal-oriented tasks.
  - Required **shortest path and time** to reach a target.
  - Handled cluttered paths **more robustly** than other methods.
- **Spiral & Snake Algorithms:**
  - Best suited for **coverage tasks** like vacuum cleaning.
  - Achieved **high area coverage** with **minimal overlap**.
  - Performance dropped when **obstacles were introduced**.

RESULTS & DISCUSSION

- LIDAR + Braitenberg Obstacle Avoidance:
  - Enabled **smooth and reactive navigation** in dense obstacle fields.
  - Excelled in avoiding collisions without stopping.
  - Lacks goal direction — not ideal for target-seeking.
- Wall Following & Line Following:
  - **Simple, effective** strategies for bounded or structured environments.
  - Easy to implement but **limited flexibility** in open or dynamic scenarios.


Thank You