Project Title

Autonomous Drone Navigation

Submitted by
Group B

Submitted in
Course (MOBILE ROBOTICS) CS-666

Center for Intelligent Cyber Physical Systems
Indian Institute of Technology Guwahati

Executive Summary

Objective:

The project aims to develop an autonomous drone navigation system capable of maneuvering through a simulated environment with obstacles. The primary focus is on achieving stable flight and efficient path planning in a complex arena.

Key Components and Methodology:

Arena Creation: A simulated environment is designed with buildings serving as obstacles, creating a realistic setting for drone navigation.

Drone Selection: The Crazyflie drone is utilized for its agility and suitability in a simulated environment.

Stabilization: Implementation of a PID (Proportional, Integral, Derivative) controller to ensure the drone's stable flight amidst environmental variables.

Point-to-Point Navigation: The drone is programmed to fly autonomously from one specified point to another within the arena.

Node Class Development: A specialized Node class is created for managing the drone's navigation. This class is responsible for storing neighboring nodes, calculating GPS coordinates, and identifying obstacle nodes.

Path Planning: The A* algorithm is employed for efficient path planning. This algorithm helps in determining the optimal route from the start point to the destination while avoiding obstacles.

Path Following: The drone follows the path charted by the A* algorithm, moving from the start node to the goal node.

Obstacle Sensing: The drone is equipped with sensors to detect obstacles in its path. Upon detection, it hovers, and the obstacle node is marked. Subsequently, the A* algorithm is re-engaged to re-plan the path.

Re-planned Path Navigation: After path re-planning, the drone resumes its journey, following the new route to the goal.

Landing Upon Goal Achievement: Once the drone reaches the goal node, it is programmed to execute a safe landing.

Conclusion:

This project showcases the integration of advanced control systems, path planning algorithms, and real-time obstacle detection and avoidance in drone technology. The successful implementation of these elements in a simulated environment paves the way for real-world applications in various fields such as surveillance, delivery services, and disaster management.

Team Members

Project Team Members from Left to Right, Rohit Sharma (234156029), Parth Sanil (234156026), Rahul Kumar (234156027), Suchit Sarkar (234156017), Jitender Singh (234156007)

Contents

Introduction to Crazyflie

The Crazyflie 2.1 is a lightweight, versatile open-source flying development platform designed for a wide range of applications including education, research, and swarming projects. Here's a detailed documentation of the Crazyflie 2.1:

Overview

- **Weight & Size**: Weighs 27g and has dimensions of 92x92x29mm (motor-to-motor, including motor mount feet).
- **Design**: Durable and easy to assemble without soldering. It supports automatic detection of expansion decks.

Capabilities

- **Control Options**: Can be controlled using a mobile device (iOS and Android) via Bluetooth LE or a computer using the Crazyradio or Crazyradio PA.
- **Radio Performance**: Features low-latency, long-range radio, tested beyond 1 km range line-of-sight with the Crazyradio PA.
- **Firmware**: Supports wireless firmware updates and on-board charging via standard USB.

Technical Specifications

- **Microcontrollers**: Equipped with an STM32F405 main application MCU and an nRF51822 radio/power management MCU.
- **Interfaces**: Includes a micro-USB connector, LiPo charger, USB device interface, and partial USB OTG capability.
- **Sensors**: Incorporates a 3-axis accelerometer/gyroscope (BMI088) and a high-precision pressure sensor (BMP388).
- **Flight Characteristics**: Offers a flight time of 7 minutes with the stock battery and a maximum recommended payload weight of 15g.

Expansion & Customization

- **Expansion Interface**: Features a flexible expansion interface for attaching various decks, accessible via buses like UART, I2C, SPI, and interfaces like PWM, analog in/out, and GPIO .

Application

- **Flexibility**: Designed for flexibility, the firmware of the Crazyflie 2.1 is modifiable, catering to custom application needs and creative experimentation.

This compact and highly adaptable platform is ideal for hobbyists, researchers, and educators looking to explore drone technology and develop innovative applications.

Reference

[Link to bitcraze crazyflie product]

Flight Mechanics of drone

The Figure 1 provides a visual guide for understanding which propellers (and thus motors) need to be accelerated to achieve specific movements of a quadcopter drone. Here's a breakdown of each movement and the propeller actions required to accomplish it:

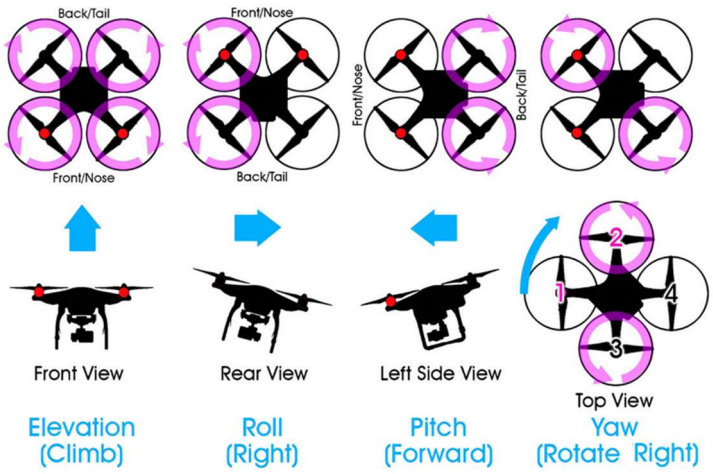

Figure 1 Flight Mechanics of Drone

- **Elevation (Climb):** To climb or increase altitude, all four propellers increase their speed uniformly. This creates more lift and causes the drone to rise.
- **Roll (Right):** To roll the drone to the right, the motors on the left side (propellers 1 and 3 in the top view) must spin faster than the motors on the right side (propellers 2 and 4). This creates more lift on the left, tilting the drone to the right.
- **Pitch (Forward):** To pitch the drone forward, the motors at the back (propellers 3 and 4) must spin faster than the motors at the front (propellers 1 and 2). This creates more lift at the back, tilting the drone's nose down and causing it to move forward.
- **Yaw (Rotate Right):** To yaw or rotate the drone to the right, the motors need to create differential rotational forces. This is achieved by increasing the speed of the motors spinning counter-clockwise (propellers 1 and 4) and decreasing the speed of the motors spinning clockwise (propellers 2 and 3). This creates a torque that rotates the drone to the right around its vertical axis.

These movements are controlled by varying the speed of the propellers, which in turn is achieved by changing the power delivered to the motors. The interaction of these forces with the design of the drone results in the various movements it can perform. Drone controllers typically have sticks or knobs that allow the pilot to increase or decrease the speed of the motors, thereby controlling the drone's movement in the air. The precise control of motor speed is often managed by an onboard computer or flight controller, which responds to the pilot's inputs to maintain stability and perform the desired maneuvers.

Drone Stabilization

The VelocityHeightPIDController class in the provided code is designed to control the velocity and altitude of an aerial vehicle, such as a drone, using a PID (Proportional-Integral-Derivative) control strategy. Here's an overview of how it works:

PID Control Concept

PID control is a widely used control loop feedback mechanism. It continuously calculates an error value as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms.

Components of the Class

Initialization (__init__ method):

- Initializes error variables for velocity (in x and y directions), altitude, pitch, and roll to zero. These variables store the previous cycle's errors for use in the current cycle.
- Initializes an integral term for altitude control(altitude_integral), which accumulates the error over time.
- Sets up the PID control gains for different aspects of the control, like attitude (yaw, roll, pitch), velocity (x and y directions), and altitude.

PID Computation (compute_pid method):

- **Velocity Control**: Calculates the error in x and y velocities (difference between target and current velocities). It then computes the derivative of these errors (change in error over time). The desired pitch and roll are calculated using the proportional and derivative terms of the velocity errors.
- **Altitude Control**: Similar to velocity control, it calculates the altitude error and its derivative. The integral of the altitude error is also updated. The altitude control output is then computed using the PID formula, which includes proportional, integral, and derivative terms.
- **Attitude Control**: Computes the errors in pitch and roll (difference between desired and current values) and their derivatives. The outputs for roll and pitch adjustments are calculated using their respective PID terms.
- The previous error values are updated for use in the next control cycle.

Motor Command Calculation:

Based on the outputs of the altitude, roll, pitch, and yaw controls, the method calculates the commands for four motors. These commands determine how much power each motor should receive.
The motor commands are constrained within a specified range (0 to 600 in this case) to ensure they are within valid operational limits.

How It Controls the Vehicle

- The PID controller adjusts the motor speeds to achieve the desired movement and stability.
- By manipulating the motor speeds based on the PID outputs, the controller can make the vehicle move in a specific direction (controlled by pitch and roll), rotate at a certain rate (yaw control), and maintain or change altitude (altitude control).

Dynamics and control logic of Controller

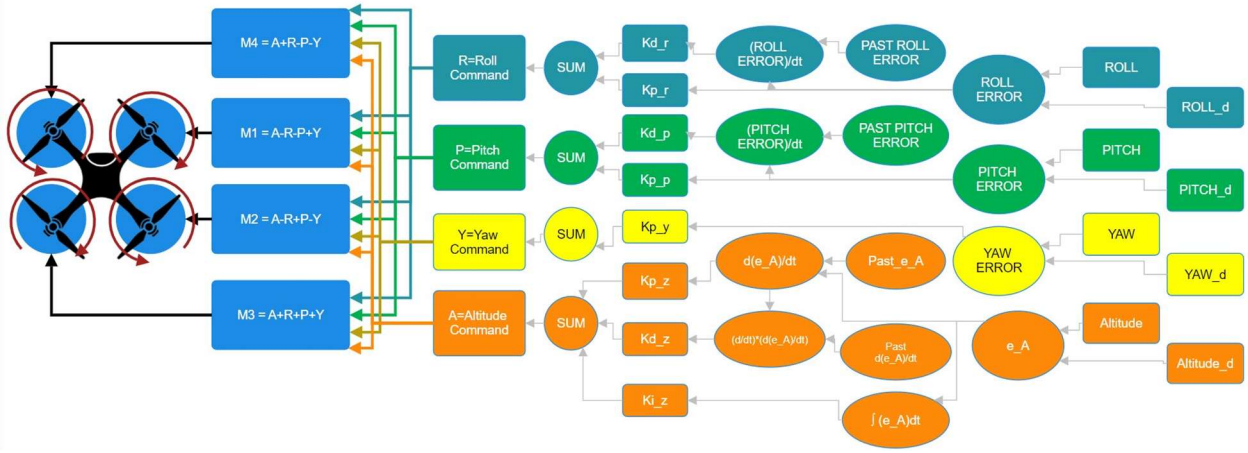

Figure 2 Controller Logic

In figure 1 on the left side, there's a representation of a quadcopter with four motors (M1 to M4), each receiving a different control signal (A±R±P±Y), where A stands for Altitude, R for Roll, P for Pitch, and Y for Yaw. These signals determine the behavior of the quadcopter's movement in 3D space.

The central and right part of the diagram shows a series of control loops for Roll, Pitch, Yaw, and Altitude. These loops represent how the quadcopter's controller adjusts these four variables. The controller uses a PID (Proportional-Integral-Derivative) control strategy for each axis, which is a common approach in control systems to maintain a desired setpoint or to follow a desired trajectory. The PID controller adjusts the motor speeds based on the error between the desired and the actual state, which is determined by sensor readings.

- **PID Control**: This is characterized by three constants: Kp (proportional gain), Ki (integral gain), and Kd (derivative gain). Each of these parameters is tuned to get the desired response from the system.
- **Roll, Pitch, Yaw**: These are the rotational axes of the quadcopter. Roll is rotation about the front-to-back axis, pitch is about the side-to-side axis, and yaw is about the vertical axis.
- **Altitude**: This is controlled by the collective thrust of the motors.
- **Error signals**: These are the differences between the desired position or angle and the actual one. For each of the Roll, Pitch, Yaw, and Altitude, there's an error calculation that feeds into the PID controllers.
- **Sum Blocks**: These typically sum up the desired command and the error to determine the necessary adjustment.
- **d(e)/dt**: Represents the derivative of the error, a part of the PID controller that responds to the rate of change of the error.
- **∫(e)dt**: Represents the integral of the error over time, another part of the PID controller that responds to the accumulated offset over time.

Reference of Code:

[link to github repository]

Code Explanation:

Required import

1. Numpy libraries

Code

```
import numpy as np

class VelocityHeightPIDController:
    def __init__(self):
        # Initialize error variables for velocity, altitude, pitch, and roll
        self.previous_velocity_x_error = 0.0
        self.previous_velocity_y_error = 0.0
        self.previous_altitude_error = 0.0
        self.previous_pitch_error = 0.0
        self.previous_roll_error = 0.0

        # Initialize the integral component for altitude control
        self.altitude_integral = 0.0

        # Define PID control gains for various control aspects
        self.control_gains = {
            "attitude_yaw_kp": 1, "attitude_yaw_kd": 0.5,
            "attitude_rp_kp": 0.5, "attitude_rp_kd": 0.1,
            "velocity_xy_kp": 2, "velocity_xy_kd": 0.5,
            "altitude_kp": 10, "altitude_ki": 5, "altitude_kd": 5
        }

    def compute_pid(self, time_delta, target_vx, target_vy, target_yaw_rate, target_altitude,
                    current_roll, current_pitch, current_yaw_rate, current_altitude, current_vx,
current_vy):
        """
        Compute PID outputs for controlling velocity and maintaining fixed height.
        """

        # Calculate the error in x-direction velocity
        error_vx = target_vx - current_vx
        # Calculate the derivative of the x-direction velocity error
        derivative_vx = (error_vx - self.previous_velocity_x_error) / time_delta

        # Calculate the error in y-direction velocity
        error_vy = target_vy - current_vy
        # Calculate the derivative of the y-direction velocity error
        derivative_vy = (error_vy - self.previous_velocity_y_error) / time_delta

        # Calculate the desired pitch using velocity error and derivative in x-direction
          pitch_target = self.control_gains["velocity_xy_kp"] * np.clip(error_vx, -1, 1) +
self.control_gains["velocity_xy_kd"] * derivative_vx
        # Calculate the desired roll using velocity error and derivative in y-direction
          roll_target = -self.control_gains["velocity_xy_kp"] * np.clip(error_vy, -1, 1) -
self.control_gains["velocity_xy_kd"] * derivative_vy

        # Update the previous velocity errors for the next iteration
        self.previous_velocity_x_error = error_vx
        self.previous_velocity_y_error = error_vy

        # Calculate the error in altitude
        altitude_error = target_altitude - current_altitude
        # Calculate the derivative of the altitude error
        altitude_derivative = (altitude_error - self.previous_altitude_error) / time_delta
```

```
        # Update the integral of the altitude error
        self.altitude_integral += altitude_error * time_delta
        # Calculate the altitude control output
        altitude_output = (self.control_gains["altitude_kp"] * altitude_error +
                           self.control_gains["altitude_kd"] * altitude_derivative +
                            self.control_gains["altitude_ki"] * np.clip(self.altitude_integral,
-2, 2) + 48)

        # Update the previous altitude error for the next iteration
        self.previous_altitude_error = altitude_error

        # Calculate the error in pitch
        error_pitch = pitch_target - current_pitch
        # Calculate the derivative of the pitch error
        derivative_pitch = (error_pitch - self.previous_pitch_error) / time_delta
        # Calculate the error in roll
        error_roll = roll_target - current_roll
        # Calculate the derivative of the roll error
        derivative_roll = (error_roll - self.previous_roll_error) / time_delta

        # Calculate the roll control output
          roll_output  =  self.control_gains["attitude_rp_kp"]  *  np.clip(error_roll,  -1,  1)  +
self.control_gains["attitude_rp_kd"] * derivative_roll
        # Calculate the pitch control output
         pitch_output  =  self.control_gains["attitude_rp_kp"]  *  np.clip(error_pitch,  -1,  1)  +
self.control_gains["attitude_rp_kd"] * derivative_pitch
        # Calculate the yaw control output
            yaw_output  =  self.control_gains["attitude_yaw_kp"]  *  np.clip(target_yaw_rate  -
current_yaw_rate, -1, 1)

        # Update the previous pitch and roll errors for the next iteration
        self.previous_pitch_error = error_pitch
        self.previous_roll_error = error_roll

        # Calculate motor commands based on the control outputs
        motor1 = np.clip(altitude_output - roll_output - pitch_output + yaw_output, 0, 600)
        motor2 = np.clip(altitude_output - roll_output + pitch_output - yaw_output, 0, 600)
        motor3 = np.clip(altitude_output + roll_output + pitch_output + yaw_output, 0, 600)
        motor4 = np.clip(altitude_output + roll_output - pitch_output - yaw_output, 0, 600)

        # Return the calculated motor commands
        return [motor1, motor2, motor3, motor4]
```

Fly Function

Introduction to mechanism

Below figure 3 is a simplified control diagram explaining the basic functions of a drone's movement, specifically altitude, yaw, and pitch. Here's what each part of the image represents:

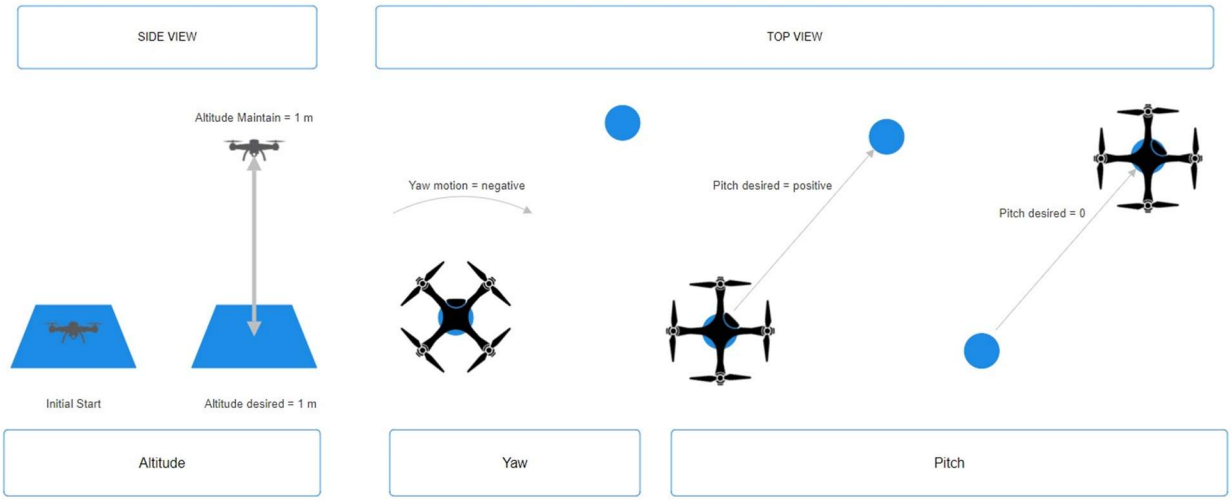

Figure 3 Flying mechanism

Side View (Altitude):

The diagram shows a drone at an initial start position at ground level.

The upward arrow with "Altitude Maintain = 1 m" indicates the drone should ascend and maintain a height of 1 meter above the ground.

The second part of the side view shows a downward arrow, suggesting that the drone is desired to reach or maintain an altitude of 1 meter. This implies a controlled ascent or descent to a specific altitude level.

Top View (Yaw and Pitch):

The top left part of the diagram with "Yaw motion = negative" represents the drone rotating to the left around its vertical axis, which is also known as counter-clockwise yaw. This motion is typically achieved by varying the speeds of the propellers to create torque.

The middle top view shows a stationary drone with no pitch ("Pitch desired = 0"), meaning it is level and not tilting forwards or backwards.

The top right view with the line pointing towards a blue dot labeled "Pitch desired = positive" represents the drone tilting forward. This forward pitch is usually achieved by increasing the speed of the rear propellers and/or decreasing the speed of the front propellers, causing the drone to move towards the blue dot.

Each section of the diagram corresponds to a different control input and desired drone movement. To achieve these movements, a drone pilot would adjust the throttle (for altitude), yaw (rotation), and pitch (tilt) controls on the drone's remote control or flight system.

The function is a simple control algorithm for a drone to navigate from its current position to a specified destination while maintaining a certain altitude and adjusting its yaw angle. The control logic involves adjusting the drone's height, yaw angle, and forward motion based on its current state and position relative to the destination.

Initialization and Parameters:

- maintain_altitude: The target altitude to be maintained by the drone.
- yd2: The desired yaw angle.
- fl: A flag indicating whether the drone has reached its destination.
- stop: A flag indicating whether the drone should stop.
- altitude: The current altitude of the drone.
- x_global, y_global: Current coordinates of the drone.
- x, y: coordinates of the destination.
- height_desired: Desired height for the drone.
- sideways_desired, yaw_desired, forward_desired: Desired values for sideways motion, yaw angle, and forward motion, respectively.

Algorithm :-

Calculating Distance to Destination:

- Euclidean distance between the current (x_global, y_global) and the destination (x,y) is calculated

Yaw Angle Conversion:

- The yaw angle is converted to a format suitable for the control logic. The converted angle (yaw_m) is used for further calculations.

Altitude Control:

- If the drone is below the desired altitude (maintain_altitude) and has not yet reached the destination (fl == 0), it increases its altitude
- If the drone is above the desired altitude, it sets the yaw_desired to 0 and starts changing its orientation to desired angle

Rotating to Desired Angle:

- If the drone has not reached the desired angle it adjusts the yaw_desired .
- The compute_yaw_change function is used ,To take into consideration any errors in yaw calculations or any changes in yaw required to to change in oreintation of Drone because of external force.

Final Approach and Stopping:

- The Forward Desired is increased to initiate a forward motion of the drone towards the target coordinate
- Once the drone is close to the destination (considering an error of 0.5) it sets the stop flag to indicate that it should stop (stop = 1).
- If the drone has moved too far from the desired angle or is not close to the destination, it resets the flag fl to 0.

Return Values:

- The function returns the updated values of height_desired, sideways_desired, yaw_desired, forward_desired, fl, and stop which will be given as an input to the PID Controller.

Required import

1. math library

Code with Comments:-

```
def adjust_for_boundary_wrap(angle_diff):
    """
    Adjust the angle difference for the 0°/360° boundary wrap-around.
    This function ensures that the angle difference is represented in the range [-180°,
180°].
    """
    if angle_diff > 180:
        return angle_diff - 360
    elif angle_diff < -180:
        return angle_diff + 360
    return angle_diff

def compute_yaw_change(yaw_m, yd2, yaw_change):
    """
    Compute the change in yaw (rotation) needed, considering the boundary wrap-around.
    """
    raw_diff = yd2 - yaw_m
    adjusted_diff = adjust_for_boundary_wrap(raw_diff)
    return yaw_change if adjusted_diff > 0 else -yaw_change

# Importing the WallFollowing class from the wall_following module
from wall_following import WallFollowing

def  fly(maintain_altitude,  yd2,  fl,  stop,  altitude,  x_global,  y_global,  x,  y,
height_desired, sideways_desired, yaw_desired, forward_desired):
    """
    Function to control the flight of a drone.

    Parameters:
    - maintain_altitude: The altitude that the drone should maintain.
    - yd2: The desired yaw.
```

```
    - fl: Flag to indicate a certain state in the flight process.
    - stop: Flag to indicate whether the drone should stop.
    - altitude: The current altitude of the drone.
    - x_global, y_global: The global coordinates of the drone.
    - x, y: The local coordinates of the drone.
     - height_desired, sideways_desired, yaw_desired, forward_desired: Control parameters
for the drone's movement.
    """
    sideways_desired = 0
    ee = math.sqrt((x_global - x) ** 2 + (y_global - y) ** 2)

    # Convert yaw to a value between 0 and 360 degrees
    if yaw >= 0:
        yaw_m = yaw * 60
    elif yaw < 0:
        yaw_m = 180 + (180 + yaw * 60)
    if yaw_m < 2 and yaw > 355:
        yaw_m = 0

    # Control logic for altitude, yaw, and position
    if altitude < maintain_altitude and fl == 0:  # First, reach the desired altitude
        height_desired += 0.5 * dt
    elif (abs(abs(yd2) - abs(yaw_m)) > 2) and fl == 0:  # Then, rotate to the desired angle
        yaw_change = 0.3
        yaw_desired += compute_yaw_change(yaw_m, yd2, yaw_change)
        forward_desired = 0
    elif altitude > maintain_altitude and fl == 0:
        yaw_desired = 0
        if ee > 0.2:  # Move to the specified x,y coordinate
            forward_desired += 0.2
        elif ee < 0.2:
            fl = 1
    elif ee < 0.5 and fl == 1:
        stop = 1
        fl = 0
    elif abs(abs(yd2) - abs(yaw_m)) > 5 or ee > 0.5:
        fl = 0

    return height_desired, sideways_desired, yaw_desired, forward_desired, fl, stop
```

ADD OBSTACLE FUNCTION

Purpose:

The function is designed to update a node map by adding obstacles at specified coordinates on the grid.

Parameters:

- **node_map**: This likely represents a map/grid composed of nodes. Each node probably contains information about the coordinates and potentially other attributes.
- **obstacle_coordinates**: It's a list of coordinates (x, y) where obstacles need to be placed on the grid.
- **obstacle_nodes**: This appears to be a list that keeps track of nodes that are marked as obstacles.

Functionality:

- Iteration through Obstacle Coordinates: The function iterates through each coordinate in obstacle_coordinates.
- Conversion of Coordinates:
- It extracts the x and y values from the coordinate.
- Node Search:
- It uses a function search_node(x, y) to find the corresponding grid cell for the given coordinates.
- This function is not explicitly defined within the code provided, but it's assumed to convert the (x, y) coordinates into a grid cell or node representation.
- It retrieves the row and col values, which likely correspond to the row and column indices of the grid for that specific (x, y) coordinate.
- Obstacle Addition:
- It checks if the obtained (row, col) pair isn't already in the obstacle_nodes list. If not, it appends it.
- Then, it marks the corresponding node in the node_map as an obstacle by setting the is_obstacle attribute of the node at node_map[row][col] to True.

CODE

```
def add_obstacle(node_map, obstacle_coordinates, obstacle_nodes):
    for obstacle_coordinate in obstacle_coordinates:
        x, y = obstacle_coordinate
        # Find the corresponding grid cell using search_node
        [row, col] = search_node(x, y)
        # Add the grid cell to the obstacle_nodes list
        if (row, col) not in obstacle_nodes:
            obstacle_nodes.append((row, col))
        # Mark the node as an obstacle in the node_map (optional)
        node_map[row][col].is_obstacle = True
```

Path Planning function

Auxiliary function for Path Planning function

Def heuristic(node1, node2):

Explanation:-

- This is the heuristic function used in the A* algorithm for estimating the distance from one node to another. Here's a breakdown:

Purpose:

The heuristic function provides an estimation of the distance between two nodes. In A* search, this function guides the algorithm towards the goal by providing an approximate cost to reach the target from the current node.

Method:

Euclidean Distance Calculation:

- It calculates the distance between two nodes based on their GPS coordinates.
- The function assumes the existence of a method get_gps_coordinates within each node, which likely returns the GPS coordinates, with parameters such as a scaling factor (1.0) and the size of the map.

Coordinates and Distance Calculation:

- For two nodes, node1 and node2, it retrieves their GPS coordinates using the assumed get_gps_coordinates method.
- It then calculates the differences in the x and y coordinates (dx and dy, respectively).
- The Euclidean distance formula sqrt((x2 - x1)^2 + (y2 - y1)^2) is applied to the differences to estimate the distance between the nodes.

CODE

```
    def heuristic(node1, node2):
        """
        Heuristic function to estimate the cost from the current node to the goal.
        Uses Euclidean distance as the heuristic.
        """
        x1, y1 = node1.get_gps_coordinates(1.0, len(node_map))
        x2, y2 = node2.get_gps_coordinates(1.0, len(node_map))
        dx = x2 - x1
        dy = y2 - y1
        return math.sqrt(dx**2 + dy**2)
```

def reconstruct_path(came_from, current_node):

Explanation:-

This function, reconstruct_path, is a crucial part of the A* pathfinding algorithm. It's responsible for reconstructing the path from the start node to the goal node based on the information stored in the came_from dictionary.

Objective:

The goal of reconstruct_path is to generate the sequence of nodes representing the path from the starting node to the goal node.

Process:

Initialization:

- It starts by initializing the path with the current_node, which is initially the goal node.

Backtracking the Path:

- It iterates through the came_from dictionary to backtrack the optimal path.
- The while loop continues until it reaches the starting node. It uses the came_from information to trace the path backward.

Updating the Path:

- Within each iteration, it updates the current_node to its predecessor, as defined by the came_from dictionary.
- It inserts this previous node at the beginning of the path list, ensuring the correct order of nodes from the start to the goal.

Return:

- Finally, it returns the complete path representing the optimal route from the start node to the goal node.

CODE

```
    def reconstruct_path(came_from, current_node):
        """
        Reconstructs the path from the start node to the current node.
        """
        path = [current_node]
        while current_node in came_from:
            current_node = came_from[current_node]
            path.insert(0, current_node)
        return path
```

def get_neighbors(current_node, node_map):

Initialization:

- `neighbors`: An empty list to store neighboring nodes.

- `n`: The size of the node map used to check boundary conditions.

Exploring Neighbors:

- Nested loops exploring a 5x5 grid centered at the `current_node`, excluding the node itself (`dx=0`, `dy=0`).
- Calculates new row and column positions based on the offset from the current node.

Boundary Checks:

- Verifies whether the new row and column indices fall within the boundaries of the node map.

Obstacle Check:

- Retrieves the potential neighbor node from the `node_map`.
- Checks if the potential neighbor is not an obstacle using the `is_obstacle` function.

List of Neighbors:

- If the neighbor is not an obstacle and falls within the boundaries, it's considered a valid neighbor and added to the `neighbors` list.

Return:

- Returns the list of valid neighboring nodes found in the 5x5 grid centered at the `current_node`.
- This function systematically examines a grid around the `current_node`, excluding the node itself and checking for obstacles to generate a list of valid neighboring nodes.

CODE

```
def get_neighbors(current_node, node_map):
    """
    Get all valid neighbors of a given node in a grid.

    Parameters:
    current_node: The node for which neighbors are to be found.
    node_map: A 2D grid representing the map.

    Returns:
    A list of neighbor nodes.
    """

    neighbors = []  # List to store valid neighbors
    n = len(node_map)  # Size of the grid

    # Loop through a 5x5 grid centered on the current node
```

```
    for dx in [-2, -1, 0, 1, 2]:
        for dy in [-2, -1, 0, 1, 2]:
            # Skip the current node itself
            if dx == 0 and dy == 0:
                continue

            # Calculate the row and column of the potential neighbor
            new_row, new_col = current_node.row + dx, current_node.col + dy

            # Check if the new row and column are within the grid boundaries
            if 0 <= new_row < n and 0 <= new_col < n:
                neighbor = node_map[new_row][new_col]

                # Check if the neighbor is not an obstacle
                if not is_obstacle(neighbor):
                    # If it's a valid neighbor, add it to the list
                    neighbors.append(neighbor)

    return neighbors
```

def astar_path_planning(node_map, start_node, goal_node):

This code implements the A* algorithm, a popular pathfinding algorithm used to find the shortest path between two nodes in a graph. Here's a detailed breakdown:

Objective:

The function `astar_path_planning` seeks to find the most efficient path from a `start_node` to a `goal_node` within a provided `node_map` using the A* algorithm.

Components:

Heuristic Function (`heuristic`):

- *Purpose*: Estimates the cost from a node to the goal.
- *Method*: Uses Euclidean distance based on GPS coordinates of nodes to provide an approximate cost from a node to the goal node.

Path Reconstruction Function (`reconstruct_path`):

- *Objective*: Reconstructs the path from the `start_node` to the `goal_node` using the information stored in the `came_from` dictionary.
- *Method*: Backtracks through the `came_from` dictionary, which records the most efficient path, and constructs the sequence of nodes forming the path.

Summary:

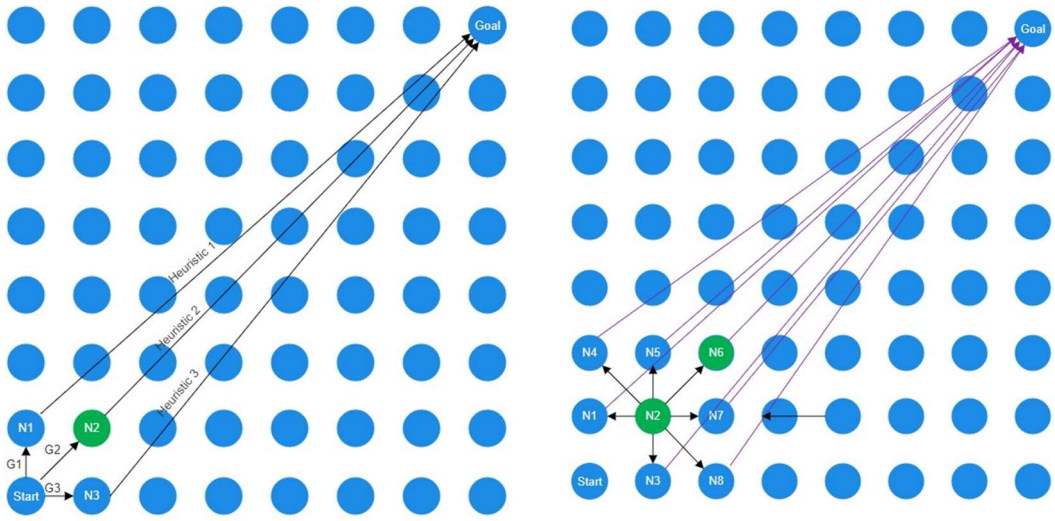

Figure 4
Left figure = Searching for neighbour node from start node
Right figure = Now conitnue from previous best node,search for neighbour node

The provided A* pathfinding algorithm is designed to find the shortest path between two nodes on a grid. The algorithm uses a heuristic function, specifically the Euclidean distance, to estimate the cost from the current node to the goal. The process includes:

1. Initialization: It starts with an open set containing the starting node with a cost of 0. It also sets up a **`g_score`** dictionary, with a default value of infinity for all nodes except the starting node, and a **`came_from`** dictionary to track the most efficient previous step.
2. Pathfinding Loop: The algorithm repeatedly takes the node with the lowest f-score (g-score + heuristic) from the open set, evaluating its neighbors.
3. Neighbors Evaluation: For each neighbor, it calculates a tentative g-score (distance from start to this neighbor through the current node). If this tentative score is better (lower) than the previously recorded g-score, it updates the **`came_from`** and **`g_score`** for this neighbor and adds the neighbor to the open set with its new f-score.
4. Goal Check: If the current node is the goal node, the algorithm reconstructs the path from the goal to the start by tracing back through the **`came_from`** dictionary.
5. Termination: The algorithm ends when the goal is found, returning the path as a sequence of GPS coordinates. If the open set is empty without finding the goal, it returns **`None`**, indicating that there is no path.

The **`get_neighbors`** function is used to find all valid neighboring nodes around the current node that are not obstacles and within grid boundaries.

The figure 4 a and b represent visualizations of the A* algorithm in action, showing different heuristic paths (labeled as "Heuristic 1", "Heuristic 2", etc.) towards a goal from a starting point. The nodes are visual representations of the points on the grid, with some marked as part of the

path (in green) and others just as possible steps (in blue). The different paths shown may represent the algorithm evaluating different potential routes to the goal.

Required import

1. math library
2. heapq library
3. itertools library

CODE

```
import math
import heapq
import itertools

def astar_path_planning(node_map, start_node, goal_node):
    """
    A* pathfinding algorithm to find the shortest path between two nodes in a grid.

    Parameters:
    node_map: A 2D grid representing the map.
    start_node: The starting node for the path.
    goal_node: The goal node for the path.
    """

    def heuristic(node1, node2):
        """
        Heuristic function to estimate the cost from the current node to the goal.
        Uses Euclidean distance as the heuristic.
        """
        x1, y1 = node1.get_gps_coordinates(1.0, len(node_map))
        x2, y2 = node2.get_gps_coordinates(1.0, len(node_map))
        dx = x2 - x1
        dy = y2 - y1
        return math.sqrt(dx**2 + dy**2)

    def reconstruct_path(came_from, current_node):
        """
        Reconstructs the path from the start node to the current node.
        """
        path = [current_node]
        while current_node in came_from:
            current_node = came_from[current_node]
            path.insert(0, current_node)
        return path
```

```
    open_set = []  # Priority queue for nodes to be evaluated
    counter = itertools.count()  # Tie-breaking counter for nodes with equal f-
scores

    # Add the start node to the open set with a cost of 0
    heapq.heappush(open_set, (0, next(counter), start_node))

    # Initialize g_score for each node: distance from start to the node
    g_score = {node: float("inf") for row in node_map for node in row}
    g_score[start_node] = 0

    came_from = {}  # For each node, which node it can most efficiently be reached
from

    while open_set:
        # Pop the node with the lowest f-score from the open set
        _, _, current_node = heapq.heappop(open_set)

        # If the goal node is reached, reconstruct and return the path
        if current_node == goal_node:
            return [
                node.get_gps_coordinates(1.0, len(node_map))
                for node in reconstruct_path(came_from, goal_node)
            ]

        # For each neighbor of the current node
        for neighbor in get_neighbors(current_node, node_map):
            # Calculate tentative g_score for the neighbor
                            tentative_g_score    =    g_score[current_node]    +
current_node.calculate_distance(neighbor)

            # If the tentative g_score is less than the g_score for the neighbor
            if tentative_g_score < g_score[neighbor]:
                 # This path to neighbor is better than any previous one. Record
it!
                came_from[neighbor] = current_node
                g_score[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic(neighbor, goal_node)
                heapq.heappush(open_set, (f_score, next(counter), neighbor))

    return None  # If no path is found, return None
```

Search Node and Landing

Purpose of the Code:

The code appears to be part of a control system for guiding a flying object, possibly a drone or aircraft. The fly function adjusts the desired height, sideways movement, yaw angle, and forward movement based on various conditions such as maintaining altitude, reaching a specific yaw angle, and proximity to a target location. The code utilizes functions like adjust_for_boundary_wrap to handle angle differences and compute_yaw_change to determine the change in yaw.

Inputs and Outputs:

Inputs:

- maintain_altitude: The desired altitude to be maintained.
- yd2: The desired yaw angle.
- fl: Flag indicating a altitude is maintenance and Orientation condition.
- stop: Flag indicating that whether we reached the nearest planned node or not.
- altitude: Current altitude of the flying object.
- x_global, y_global: Global coordinates.
- x, y: Current coordinates of the flying object.
- height_desired, sideways_desired, yaw_desired, forward_desired: Current desired parameters.

Outputs:

- height_desired, sideways_desired, yaw_desired, forward_desired: Updated desired parameters.
- fl: Updated flag.
- stop: Updated flag.

How it Works?

- Coordinate Calculation:
    - Calculates the Euclidean distance (ee) between the global coordinates and the current coordinates of the flying object.
- Yaw Angle Adjustment:
    - Converts the current yaw angle (yaw) to a range of 0 to 360 degrees.
    - If the converted yaw angle is close to 0 or 360, it sets it to 0.
- Altitude Adjustment:
    - If the current altitude is below the desired altitude (maintain_altitude) and a specific condition (fl) is not met, it incrementally increases the desired height.
- Yaw Angle Adjustment (Part 2):
    - If the absolute difference between the desired yaw angle (yd2) and the current converted yaw angle is greater than 2 degrees and a specific condition (fl) is not met, it adjusts the desired yaw angle.
- Altitude and Forward Movement:
    - If the current altitude is above the desired altitude and a specific condition (fl) is not met: Sets the desired yaw angle to 0.
    - If the Euclidean distance (ee) is greater than 0.2, incrementally increases the desired forward movement.

    - If the Euclidean distance (ee) is less than 0.2, sets a flag (fl) to 1.
- Stop Conditions:
    - If the Euclidean distance (ee) is less than 0.5 and a specific condition (fl) is 1, sets a stop flag to 1 and resets the flag (fl) to 0.
    - If the absolute difference between the desired yaw angle (yd2) and the current converted yaw angle is greater than 5 degrees or the Euclidean distance (ee) is greater than 0.5, resets the flag (fl) to 0.
- Outputs:
    - Returns the updated desired height, sideways movement, yaw angle, and forward movement, along with the updated flags.

Code Explanation

- The land function is designed to adjust parameters for landing based on specified conditions. It takes several parameters related to the flying object's state and desired parameters, such as altitude, yaw angle, and global coordinates. The function begins by initializing the sideways_desired parameter to zero. It then converts the raw yaw angle to a range of 0 to 360 degrees and adjusts it for boundary wrap-around, ensuring a smooth transition between 0 and 360 degrees. The function proceeds to handle altitude adjustments: if the current altitude is below the desired altitude and a specific condition (`fl`) is not met, it incrementally increases the desired height; similarly, if the altitude is above the desired altitude, it decreases the desired height. Finally, the function resets the yaw and forward desired parameters to zero and returns the updated parameters and flags.
- The search_node function aims to locate a specific node in a node map based on specified starting coordinates (x_required_to_start and y_required_to_start). It initializes iterator variables `i` and `j` to zero and enters a loop that continues until the desired node is found. Within the loop, the function retrieves the GPS coordinates of the current node in the node map using the iterator variables. It then checks if the x and y coordinates are within a tolerance of 0.71 meters from the required starting values. If not, it increments the iterator variables accordingly. Once a node with coordinates satisfying the conditions is found, the loop breaks, and the function returns the integer coordinates of the located node in the node map.

The code thus provides a structured approach to landing parameter adjustments and a systematic search for a specific node in a node map. The land function ensures controlled descent based on altitude, while the search_node function iteratively refines the search for a specific node within the node map.

Code

```
def land(maintain_altitude, yd2, fl, stop, altitude, x_global, y_global, x, y,
height_desired, sideways_desired, yaw_desired, forward_desired):
    """
```

```
    Function to control the landing of a vehicle (e.g., drone).

    Parameters:
    maintain_altitude: The altitude to maintain during the landing process.
    yd2, fl, stop: Control parameters
    altitude: Current altitude of the vehicle.
    x_global, y_global: Global position coordinates of the vehicle.
    x, y: Local position coordinates of the vehicle.
    height_desired: Desired height to achieve.
    sideways_desired: Desired sideways movement.
    yaw_desired: Desired yaw angle.
    forward_desired: Desired forward movement.
    """

    # Reset sideways movement to zero as it's a landing procedure
    sideways_desired = 0

    # Calculate yaw movement
    if yaw >= 0:
        yaw_m = yaw * 60
    elif yaw < 0:
        yaw_m = 180 + (180 + yaw * 60)

    # Normalize yaw movement
    if yaw_m < 2 and yaw > 355:
        yaw_m = 0

    # Adjust height to reach the desired maintain_altitude
      if altitude < maintain_altitude and fl == 0:  # Ascend if below
maintain_altitude
        height_desired += 0.1 * dt
      if altitude > maintain_altitude and fl == 0:  # Descend if above
maintain_altitude
        height_desired -= 0.1 * dt

    # Set yaw and forward movement to zero as it's a landing procedure
    yaw_desired = 0
    forward_desired = 0

     return height_desired, sideways_desired, yaw_desired, forward_desired, fl,
stop

################################################

def search_node(x_required_to_start, y_required_to_start):
```

```
    """
    Function to find the closest node in a grid to a given set of coordinates.

    Parameters:
    x_required_to_start, y_required_to_start: The coordinates for which the closest
node is to be found.
    """

    iii = 0.0
    jjj = 0.0
    while True:
        # Get the GPS coordinates of the current node
        loc = node_map[int(iii)][int(jjj)].get_gps_coordinates(1, n)

        # Check if the current node's x-coordinate is close enough to the required
x-coordinate
        if not (abs(loc[0] - (x_required_to_start)) < 0.71):
            jjj += 1.0
            continue

        # Check if the current node's y-coordinate is close enough to the required
y-coordinate
        if not (abs(loc[1] - (y_required_to_start)) < 0.71):
            iii += 1.0
            continue

        # If both coordinates are close enough, break the loop
        else:
            break

    return [int(iii), int(jjj)]
```

CLASS NODE

Purpose

The provided code defines a Python class called NODE that represents a node in a 2D grid. This class is designed for applications related to grid-based navigation, such as pathfinding algorithms or robotic motion planning.

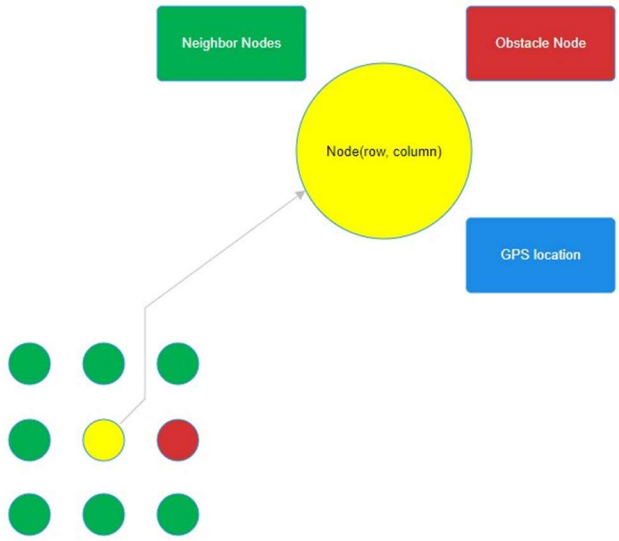

Figure 5 2D Node Matrix

The NODE class encapsulates functionality related to grid-based navigation, providing methods for adjacency checks, distance calculations, and GPS coordinate conversions. This modular approach enhances code readability and promotes reusability in applications involving 2D grid manipulation as shown in figure 5.

Class Overview

Constructor (__init__):

- **Inputs**:
  - **row**: The row index of the node in the 2D grid.
  - **col**: The column index of the node in the 2D grid.
- **Purpose**: Initializes a Node object with the specified row and column coordinates.

__str__ Method:

- **Outputs:** Returns a string representation of the node in the format "Node(row, col)".
- **Purpose:** Provides a human-readable string representation of the node.

is_diagonal_neighbor Method:

- **Inputs**:
  - **other_node**: Another Node object to check for diagonal adjacency.
- **Outputs**: Returns a boolean indicating whether the given node is a diagonal neighbor.
- **Purpose**: Checks if the provided node is a diagonal neighbor of the current node.

calculate_distance Method:

- **Inputs**:
  - **other_node:** Another Node object to calculate the distance to.
- **Outputs**: Returns the Euclidean distance between the current node and the provided node.
- **Purpose**: Calculates the distance between nodes, considering diagonal adjacency using the Euclidean distance formula.

get_gps_coordinates Method:

- **Inputs**:
  - **meter_scale:** A scaling factor for converting grid coordinates to GPS coordinates.
  - **n:** The size of the 2D grid (number of rows or columns).
- **Outputs**: Returns a tuple containing the GPS coordinates (x, y) of the node.
- **Purpose**: Converts grid coordinates to GPS coordinates, considering the center of the grid as the reference point.

How It Works:

- **Initialization**: Creates a Node object with specified row and column coordinates.
- **String** Representation: Provides a readable string representation of the node.
- **Diagonal** Neighbor Check: Determines whether another node is a diagonal neighbor.
- **Distance** Calculation: Computes the distance between two nodes, considering diagonal adjacency.
- **GPS Coordinate Calculation:** Converts grid coordinates to GPS coordinates based on the specified scaling factor and grid size.

Code Explanation

The provided code defines a Python class Node, representing a node in a 2D matrix with row and column coordinates. The class includes methods for checking if another node is a diagonal neighbor, calculating the distance between nodes (considering Euclidean distance for diagonal neighbors and specific distances for horizontal/vertical neighbors), and obtaining GPS coordinates for a node based on a given meter scale and the size of the matrix. The __str__ method provides a human-readable string representation of a node. Overall, the class encapsulates functionality related to node properties and relationships in a 2D grid, offering methods for common operations used in algorithms or simulations involving grid-based structures. The code utilizes Python's object-oriented features to organize and encapsulate these functionalities within the Node class.

Required import

1. math libraries

CODE:

```
import math

class Node:
    def __init__(self, row, col):
        """
```

```
        Initialize a new Node with specified row and column coordinates.
        """
        self.row = row
        self.col = col

    def __str__(self):
        """
        String representation of the Node.
        """
        return f"Node({self.row}, {self.col})"

    def is_diagonal_neighbor(self, other_node):
        """
        Check if another node is a diagonal neighbor.
         Diagonal neighbors are those where both the row and column differ by
exactly 1.
        """
        dx = abs(self.row - other_node.row)
        dy = abs(self.col - other_node.col)
        return dx == 1 and dy == 1

    def calculate_distance(self, other_node):
        """
        Calculate the distance to another node.
        If the nodes are diagonal neighbors, use Euclidean distance.
         Otherwise, use a distance of 1 for horizontal/vertical neighbors and
sqrt(2) for other diagonals.
        """
        if self.is_diagonal_neighbor(other_node):
            dx = abs(self.row - other_node.row)
            dy = abs(self.col - other_node.col)
            return math.sqrt(dx**2 + dy**2)
        else:
            return (
                1
                if (self.row == other_node.row or self.col == other_node.col)
                else math.sqrt(2)
            )

    def get_gps_coordinates(self, meter_scale, n):
        """
        Convert the node's grid position to GPS-like coordinates.
        The center of the grid is treated as (0,0), and each unit distance in the
grid is scaled by 'meter_scale'.
        The grid size 'n' determines the offset for calculating coordinates.
```

```
        """
        if n % 2 != 0:
            even = 0
            x = (self.col - int(n / 2)) * meter_scale - even
            y = (-self.row + int(n / 2)) * meter_scale - even
            return (x, y)
        else:
            even = 0.5
            x = (self.col - int(n / 2)) * meter_scale - even
            y = (self.row - int(n / 2)) * meter_scale - even
            return (x, y)
```

WallFollowing Class Documentation

The WallFollowing class is a sophisticated implementation designed for autonomous navigation of a drone, specifically a Crazyflie, in an environment where it follows along walls. This class is part of a larger system that enables the drone to autonomously navigate by sensing and reacting to its surroundings, particularly walls. Below is a detailed breakdown of the class, its nested enumerations, methods, and the logic behind its operation.

Class Overview

- Class Name: WallFollowing
- Purpose: To enable a drone (Crazyflie) to autonomously navigate by following walls in its environment.

Nested Enumerations

StateWallFollowing

Defines the various states the drone can be in during wall following.

- FORWARD: Moving forward.
- HOVER: Hovering in place.
- TURN_TO_FIND_WALL: Turning to locate a wall.
- TURN_TO_ALIGN_TO_WALL: Turning to align with a wall.
- FORWARD_ALONG_WALL: Moving forward along a wall.
- ROTATE_AROUND_WALL: Rotating around a wall.
- ROTATE_IN_CORNER: Rotating in a corner.
- FIND_CORNER: Finding a corner.

WallFollowingDirection

Defines the direction in which the drone should follow the wall.

- LEFT: Follow the wall on the left side.
- RIGHT: Follow the wall on the right side.

Constructor

- __init__(self, ...)

Initializes a new instance of the WallFollowing class.

Parameters

- reference_distance_from_wall: The desired distance from the wall.
- max_forward_speed: Maximum forward speed of the drone.
- max_turn_rate: Maximum turn rate of the drone.
- wall_following_direction: Direction to follow the wall (WallFollowingDirection enum).
- first_run: Indicates if this is the first run of the wall-following algorithm.

- prev_heading: The previous heading of the drone.
- wall_angle: The angle of the wall relative to the drone.
- around_corner_back_track: Indicates if the drone is backtracking around a corner.
- state_start_time: The start time of the current state.
- ranger_value_buffer: Buffer value for ranger measurements.
- angle_value_buffer: Buffer value for angle measurements.
- range_lost_threshold: Threshold for when the wall is considered lost.
- in_corner_angle: The angle to turn when in a corner.
- wait_for_measurement_seconds: Time to wait for a measurement before starting.
- init_state: Initial state of the drone (StateWallFollowing enum).

Methods

Helper Methods

- value_is_close_to(self, real_value, checked_value, margin)
- Checks if a value is within a specified margin of another value.
- wrap_to_pi(self, number)
- Wraps a number to the range [−π,π].

Command Functions

- command_turn(self, reference_rate)
  - Commands the drone to turn around its yaw axis.
- command_align_corner(self, reference_rate, side_range, wanted_distance_from_corner)
  - Commands the drone to align with a corner at a specific distance.
- command_hover(self)
  - Commands the drone to hover in place.
- command_forward_along_wall(self, side_range)
  - Commands the drone to move forward along a wall.
- command_turn_around_corner_and_adjust(self, radius, side_range)
  - Commands the drone to turn around a corner and adjust its distance.

State Machine Helper Functions

- state_transition(self, new_state)
  - Transitions to a new state and resets the state timer.
- adjust_reference_distance_wall(self, reference_distance_wall_new)
  - Adjusts the reference distance from the wall.

Main State Machine Method

- wall_follower(self, front_range, side_range, current_heading, wall_following_direction, time_outer_loop)

- The main function of the wall-following state machine. It handles state transitions and commands based on sensor inputs and current state.

Parameters

- front_range: Distance to an obstacle in front of the drone.
- side_range: Distance to the wall on the side of the drone.
- current_heading: Current heading of the drone.
- wall_following_direction: Direction to follow the wall.
- time_outer_loop: Current time of the outer loop.

Returns

- command_velocity_x: Commanded velocity in the x-direction.
- command_velocity_y: Commanded velocity in the y-direction.
- command_yaw_rate: Commanded yaw rate.
- self.state: Current state of the drone.

Usage

This class is intended to be used as part of a larger system controlling a drone, specifically a Crazyflie. It requires sensor inputs (like distance to walls) and integrates these inputs to navigate autonomously by following walls. The class is designed to be adaptable to different environments and can be fine-tuned through its parameters.

Code Explanation

a step-by-step explanation of the code:

1. Class Definition and Enums

- Class WallFollowing: This is the main class that encapsulates the wall-following logic.
- Nested Enums:
    - StateWallFollowing: Defines various states of the wall-following process, like moving forward, hovering, turning, etc.
    - WallFollowingDirection: Indicates the direction to follow the wall (left or right).

2. Constructor __init__

- Initializes the class with various parameters like reference_distance_from_wall, max_forward_speed, max_turn_rate, etc.
- These parameters control how close the robot stays to the wall, its speed, and how quickly it can turn.
- Initializes state variables like first_run, prev_heading, wall_angle, and others that are used to track the robot's progress and state.

3. Helper Functions

- value_is_close_to: Checks if a value is within a certain margin of another value.
- wrap_to_pi: Normalizes an angle to the range [−,][−π,π].

4. Command Functions

- Functions to command the robot's movements based on the current state:
    - command_turn: Commands the robot to turn.
    - command_align_corner: Aligns the robot at a corner.
    - command_hover: Makes the robot hover in place.
    - command_forward_along_wall: Commands the robot to move forward along the wall.
    - command_turn_around_corner_and_adjust: Commands the robot to turn around a corner and adjust its position.

5. State Machine Helper Functions

- state_transition: Handles transitions between different states.
- adjust_reference_distance_wall: Adjusts the reference distance from the wall.

6. Main State Machine: wall_follower

- The core function that implements the wall-following logic.
- Takes sensor readings (front_range, side_range), current heading, wall-following direction, and time as inputs.
- Based on these inputs and the current state, it decides what action the robot should take next.
- The function goes through various states, each representing a different phase of wall following (e.g., moving forward, turning, aligning with the wall).
- In each state, it uses the command functions to determine the robot's movement.

7. Command Execution

- At the end of the wall_follower function, it calculates the final command velocities (command_velocity_x, command_velocity_y, command_yaw_rate) based on the current state.
- These velocities are then used to control the robot's movement.

Summary

The code is a well-structured implementation of a state machine for a wall-following robot. It uses sensor inputs to navigate along walls, handling corners and aligning itself as needed. The state machine approach allows for clear and manageable handling of the different phases of wall following.

State Actions:

The state actions in the WallFollowing class are the specific behaviors the robot will exhibit in each state of the wall-following algorithm. Let's break down these actions based on the states defined in the StateWallFollowing enum:

1. FORWARD

- Action: The robot moves straight forward at its maximum forward speed.
- Implementation: Sets command_velocity_x to max_forward_speed and keeps command_velocity_y and command_yaw_rate at 0, indicating straight forward movement.

2. HOVER

- Action: The robot stays in place, hovering without moving.
- Implementation: All command velocities (command_velocity_x, command_velocity_y, command_yaw_rate) are set to 0.

3. TURN_TO_FIND_WALL

- Action: The robot turns in place to find the wall. This is typically used when the robot has lost track of the wall and needs to reorient itself.
- Implementation: Calls command_turn with max_turn_rate, setting command_velocity_x to 0 and adjusting command_yaw_rate for turning.

4. TURN_TO_ALIGN_TO_WALL

- Action: Once the wall is located, the robot turns to align itself parallel to the wall.
- Implementation: Similar to TURN_TO_FIND_WALL, but the turn rate and direction are adjusted to align with the wall based on the wall_angle.

5. FORWARD_ALONG_WALL

- Action: The robot moves forward while maintaining a constant distance from the wall.
- Implementation: Calls command_forward_along_wall to adjust command_velocity_x for forward movement and command_velocity_y to maintain the distance from the wall.

6. ROTATE_AROUND_WALL

- Action: Used when navigating around corners or when the wall ends. The robot rotates around the wall to reorient itself.
- Implementation: Calls command_turn_around_corner_and_adjust to adjust both command_velocity_x, command_velocity_y, and command_yaw_rate for smooth corner navigation.

7. ROTATE_IN_CORNER

- Action: Specifically for tight corner navigation, where the robot needs to make a sharper turn.
- Implementation: Similar to ROTATE_AROUND_WALL, but with potentially tighter turn rates and adjustments.

8. FIND_CORNER

- Action: The robot actively searches for a corner or an end of the wall to make decisions about its next movement.

- Implementation: Calls command_align_corner to adjust command_velocity_y and command_yaw_rate to find and align with the corner.

Summary

Each state represents a specific scenario in wall-following, like moving along the wall, turning at corners, or reorienting when the wall is lost. The actions in each state are designed to handle these scenarios, using sensor inputs to make decisions and adjust the robot's movement accordingly. The state machine approach ensures that the robot can smoothly transition between these different actions based on its environment and objectives.

Reference of Code:

[link to github repository]

Required import

1. math library
2. enum library

Code

```
import math
from enum import Enum


class WallFollowing():
    class StateWallFollowing(Enum):
        FORWARD = 1
        HOVER = 2
        TURN_TO_FIND_WALL = 3
        TURN_TO_ALIGN_TO_WALL = 4
        FORWARD_ALONG_WALL = 5
        ROTATE_AROUND_WALL = 6
        ROTATE_IN_CORNER = 7
        FIND_CORNER = 8

    class WallFollowingDirection(Enum):
        LEFT = 1
        RIGHT = -1

    def __init__(self, reference_distance_from_wall=0.0,
                 max_forward_speed=0.2,
                 max_turn_rate=0.5,
                 wall_following_direction=WallFollowingDirection.LEFT,
                 first_run=False,
                 prev_heading=0.0,
                 wall_angle=0.0,
                 around_corner_back_track=False,
```

```
                state_start_time=0.0,
                ranger_value_buffer=0.2,
                angle_value_buffer=0.1,
                range_lost_threshold=0.3,
                in_corner_angle=0.8,
                wait_for_measurement_seconds=1.0,
                init_state=StateWallFollowing.FORWARD):
        """
        __init__ function for the WallFollowing class

          reference_distance_from_wall is the distance from the wall that the
Crazyflie
            should try to keep
        max_forward_speed is the maximum speed the Crazyflie should fly forward
        max_turn_rate is the maximum turn rate the Crazyflie should turn with
        wall_following_direction is the direction the Crazyflie should follow the
wall
            (WallFollowingDirection Enum)
         first_run is a boolean that is True if the Crazyflie is in the first run
of the
            wall following demo
         prev_heading is the heading of the Crazyflie in the previous state (in
rad)
        wall_angle is the angle of the wall in the previous state (in rad)
        around_corner_back_track is a boolean that is True if the Crazyflie is in
the
            around corner state and should back track
        state_start_time is the time when the Crazyflie entered the current state
(in s)
        ranger_value_buffer is the buffer value for the ranger measurements (in m)
        angle_value_buffer is the buffer value for the angle measurements (in rad)
        range_lost_threshold is the threshold for when the Crazyflie should stop
            following the wall (in m)
         in_corner_angle is the angle the Crazyflie should turn when it is in the
corner (in rad)
        wait_for_measurement_seconds is the time the Crazyflie should wait for a
            measurement before it starts the wall following demo (in s)
        init_state is the initial state of the Crazyflie (StateWallFollowing Enum)
         self.state is a shared state variable that is used to keep track of the
current
            state of the Crazyflie's wall following
        self.time_now is a shared state variable that is used to keep track of the
current (in s)
        """
```

```
        self.reference_distance_from_wall = reference_distance_from_wall
        self.max_forward_speed = max_forward_speed
        self.max_turn_rate = max_turn_rate
                                    self.wall_following_direction_value        =
float(wall_following_direction.value)
        self.first_run = first_run
        self.prev_heading = prev_heading
        self.wall_angle = wall_angle
        self.around_corner_back_track = around_corner_back_track
        self.state_start_time = state_start_time
        self.ranger_value_buffer = ranger_value_buffer
        self.angle_value_buffer = angle_value_buffer
        self.range_threshold_lost = range_lost_threshold
        self.in_corner_angle = in_corner_angle
        self.wait_for_measurement_seconds = wait_for_measurement_seconds

        self.first_run = True
        self.state = init_state
        self.time_now = 0.0
        self.speed_redux_corner = 3.0
        self.speed_redux_straight = 2.0

    # Helper function
    def value_is_close_to(self, real_value, checked_value, margin):
         if real_value > checked_value - margin and real_value < checked_value +
margin:
            return True
        else:
            return False

    def wrap_to_pi(self, number):
        if number > math.pi:
            return number - 2 * math.pi
        elif number < -math.pi:
            return number + 2 * math.pi
        else:
            return number

    # Command functions
    def command_turn(self, reference_rate):
        """
        Command the Crazyflie to turn around its yaw axis

        reference_rate and rate_yaw is defined in rad/s
        velocity_x is defined in m/s
```

```
        """
        velocity_x = 0.0
        rate_yaw = self.wall_following_direction_value * reference_rate
        return velocity_x, rate_yaw

            def     command_align_corner(self,     reference_rate,     side_range,
wanted_distance_from_corner):
        """
        Command the Crazyflie to align itself to the outer corner
            and make sure it's at a certain distance from it

        side_range and wanted_distance_from_corner is defined in m
        reference_rate and rate_yaw is defined in rad/s
        velocity_x is defined in m/s
        """
        if side_range > wanted_distance_from_corner + self.range_threshold_lost:
            rate_yaw = self.wall_following_direction_value * reference_rate
            velocity_y = 0.0
        else:
            if side_range > wanted_distance_from_corner:
                velocity_y = self.wall_following_direction_value * \
                    (-1.0 * self.max_forward_speed / self.speed_redux_corner)
            else:
                            velocity_y   =   self.wall_following_direction_value   *
(self.max_forward_speed / self.speed_redux_corner)
            rate_yaw = 0.0
        return velocity_y, rate_yaw

    def command_hover(self):
        """
        Command the Crazyflie to hover in place
        """
        velocity_x = 0.0
        velocity_y = 0.0
        rate_yaw = 0.0
        return velocity_x, velocity_y, rate_yaw

    def command_forward_along_wall(self, side_range):
        """
        Command the Crazyflie to fly forward along the wall
            while controlling it's distance to it

        side_range is defined in m
        velocity_x and velocity_y is defined in m/s
        """
```

```
        velocity_x = self.max_forward_speed
        velocity_y = 0.0
        check_distance_wall = self.value_is_close_to(
                                self.reference_distance_from_wall,    side_range,
self.ranger_value_buffer)
        if not check_distance_wall:
            if side_range > self.reference_distance_from_wall:
                velocity_y = self.wall_following_direction_value * \
                    (-1.0 * self.max_forward_speed / self.speed_redux_straight)
            else:
                            velocity_y   =   self.wall_following_direction_value   *
(self.max_forward_speed / self.speed_redux_straight)
        return velocity_x, velocity_y

    def command_turn_around_corner_and_adjust(self, radius, side_range):
        """
        Command the Crazyflie to turn around the corner
            and adjust it's distance to the corner

        radius is defined in m
        side_range is defined in m
        velocity_x and velocity_y is defined in m/s
        """
        velocity_x = self.max_forward_speed
       rate_yaw = self.wall_following_direction_value * (-1 * velocity_x / radius)
        velocity_y = 0.0
        check_distance_wall = self.value_is_close_to(
                                self.reference_distance_from_wall,    side_range,
self.ranger_value_buffer)
        if not check_distance_wall:
            if side_range > self.reference_distance_from_wall:
                velocity_y = self.wall_following_direction_value * \
                    (-1.0 * self.max_forward_speed / self.speed_redux_corner)
            else:
                            velocity_y   =   self.wall_following_direction_value   *
(self.max_forward_speed / self.speed_redux_corner)
        return velocity_x, velocity_y, rate_yaw

    # state machine helper functions
    def state_transition(self, new_state):
        """
        Transition to a new state and reset the state timer

        new_state is defined in the StateWallFollowing enum
        """
```

```
        self.state_start_time = self.time_now
        return new_state

    def adjust_reference_distance_wall(self, reference_distance_wall_new):
        """
        Adjust the reference distance to the wall
        """
        self.reference_distance_from_wall = reference_distance_wall_new

    # Wall following State machine
    def wall_follower(self, front_range, side_range, current_heading,
                      wall_following_direction, time_outer_loop):
        """
        wall_follower is the main function of the wall following state machine.
        It takes the current range measurements of the front range and side range
       sensors, the current heading of the Crazyflie, the wall following direction
        and the current time of the outer loop (the real time or the simulation
time)
        as input, and handles the state transitions and commands the Crazyflie to
        to do the wall following.

        front_range and side_range is defined in m
        current_heading is defined in rad
        wall_following_direction is defined as WallFollowingDirection enum
        time_outer_loop is defined in seconds (double)
        command_velocity_x, command_velocity_ y is defined in m/s
        command_rate_yaw is defined in rad/s
        self.state is defined as StateWallFollowing enum
        """

                                    self.wall_following_direction_value        =
float(wall_following_direction.value)
        self.time_now = time_outer_loop

        if self.first_run:
            self.prev_heading = current_heading
            self.around_corner_back_track = False
            self.first_run = False

        # -------------- Handle state transitions ---------------- #
        if self.state == self.StateWallFollowing.FORWARD:
                        if   front_range   <   self.reference_distance_from_wall   +
self.ranger_value_buffer:
                                                                    self.state       =
self.state_transition(self.StateWallFollowing.TURN_TO_FIND_WALL)
```

```
        elif self.state == self.StateWallFollowing.HOVER:
            print('hover')
        elif self.state == self.StateWallFollowing.TURN_TO_FIND_WALL:
            # Turn until 45 degrees from wall such that the front and side range
sensors
            #   can detect the wall
            side_range_check = side_range < (self.reference_distance_from_wall /
                                                    math.cos(math.pi/4)  +
self.ranger_value_buffer)
             front_range_check = front_range < (self.reference_distance_from_wall
/
                                                    math.cos(math.pi/4)  +
self.ranger_value_buffer)
            if side_range_check and front_range_check:
                self.prev_heading = current_heading
                # Calculate the angle to the wall
                self.wall_angle = self.wall_following_direction_value * \
                            (math.pi/2  -  math.atan(front_range  /  side_range)  +
self.angle_value_buffer)
                                                                    self.state       =
self.state_transition(self.StateWallFollowing.TURN_TO_ALIGN_TO_WALL)
            # If went too far in heading and lost the wall, go to find corner.
                        if   side_range   <   self.reference_distance_from_wall   +
self.ranger_value_buffer and \
                             front_range  >  self.reference_distance_from_wall  +
self.range_threshold_lost:
                self.around_corner_back_track = False
                self.prev_heading = current_heading
                                                                    self.state       =
self.state_transition(self.StateWallFollowing.FIND_CORNER)
        elif self.state == self.StateWallFollowing.TURN_TO_ALIGN_TO_WALL:
            align_wall_check = self.value_is_close_to(
                            self.wrap_to_pi(current_heading   -   self.prev_heading),
self.wall_angle, self.angle_value_buffer)
            if align_wall_check:
                                                                    self.state       =
self.state_transition(self.StateWallFollowing.FORWARD_ALONG_WALL)
        elif self.state == self.StateWallFollowing.FORWARD_ALONG_WALL:
            # If side range is out of reach,
            #    end of the wall is reached
                        if   side_range   >   self.reference_distance_from_wall   +
self.range_threshold_lost:
                                                                    self.state       =
self.state_transition(self.StateWallFollowing.FIND_CORNER)
            # If front range is small
```

```
            #    then corner is reached
                    if   front_range   <   self.reference_distance_from_wall   +
self.ranger_value_buffer:
                self.prev_heading = current_heading
                                                                    self.state       =
self.state_transition(self.StateWallFollowing.ROTATE_IN_CORNER)
        elif self.state == self.StateWallFollowing.ROTATE_AROUND_WALL:
                    if   front_range   <   self.reference_distance_from_wall   +
self.ranger_value_buffer:
                                                                    self.state       =
self.state_transition(self.StateWallFollowing.TURN_TO_FIND_WALL)
        elif self.state == self.StateWallFollowing.ROTATE_IN_CORNER:
            check_heading_corner = self.value_is_close_to(
                math.fabs(self.wrap_to_pi(current_heading-self.prev_heading)),
                self.in_corner_angle, self.angle_value_buffer)
            if check_heading_corner:
                                                                    self.state       =
self.state_transition(self.StateWallFollowing.TURN_TO_FIND_WALL)
        elif self.state == self.StateWallFollowing.FIND_CORNER:
            if side_range <= self.reference_distance_from_wall:
                                                                    self.state       =
self.state_transition(self.StateWallFollowing.ROTATE_AROUND_WALL)
        else:
            self.state = self.state_transition(self.StateWallFollowing.HOVER)

        # -------------- Handle state actions ---------------- #
        command_velocity_x_temp = 0.0
        command_velocity_y_temp = 0.0
        command_angle_rate_temp = 0.0

        if self.state == self.StateWallFollowing.FORWARD:
            command_velocity_x_temp = self.max_forward_speed
            command_velocity_y_temp = 0.0
            command_angle_rate_temp = 0.0
        elif self.state == self.StateWallFollowing.HOVER:
                            command_velocity_x_temp,    command_velocity_y_temp,
command_angle_rate_temp = self.command_hover()
        elif self.state == self.StateWallFollowing.TURN_TO_FIND_WALL:
                           command_velocity_x_temp,    command_angle_rate_temp    =
self.command_turn(self.max_turn_rate)
            command_velocity_y_temp = 0.0
        elif self.state == self.StateWallFollowing.TURN_TO_ALIGN_TO_WALL:
                             if    self.time_now    -    self.state_start_time    <
self.wait_for_measurement_seconds:
```

```
                            command_velocity_x_temp,   command_velocity_y_temp,
command_angle_rate_temp = self.command_hover()
            else:
                            command_velocity_x_temp,   command_angle_rate_temp   =
self.command_turn(self.max_turn_rate)
                command_velocity_y_temp = 0.0
        elif self.state == self.StateWallFollowing.FORWARD_ALONG_WALL:
                           command_velocity_x_temp,    command_velocity_y_temp    =
self.command_forward_along_wall(side_range)
            command_angle_rate_temp = 0.0
        elif self.state == self.StateWallFollowing.ROTATE_AROUND_WALL:
            # If first time around corner
            #   first try to find the wall again
            # if side range is larger than preffered distance from wall
                      if   side_range   >   self.reference_distance_from_wall   +
self.range_threshold_lost:
                # check if scanning already occured
                if self.wrap_to_pi(math.fabs(current_heading - self.prev_heading))
> \
                        self.in_corner_angle:
                    self.around_corner_back_track = True
                # turn and adjust distance to corner from that point
                if self.around_corner_back_track:
                    # rotate back if it already went into one direction
                             command_velocity_y_temp,  command_angle_rate_temp  =
self.command_turn(
                        -1 * self.max_turn_rate)
                    command_velocity_x_temp = 0.0
                else:
                             command_velocity_y_temp,  command_angle_rate_temp  =
self.command_turn(
                        self.max_turn_rate)
                    command_velocity_x_temp = 0.0
            else:
                # continue to turn around corner
                self.prev_heading = current_heading
                self.around_corner_back_track = False
                             command_velocity_x_temp,   command_velocity_y_temp,
command_angle_rate_temp = \
                    self.command_turn_around_corner_and_adjust(
                        self.reference_distance_from_wall, side_range)
        elif self.state == self.StateWallFollowing.ROTATE_IN_CORNER:
                           command_velocity_x_temp,    command_angle_rate_temp    =
self.command_turn(self.max_turn_rate)
            command_velocity_y_temp = 0.0
```

```
        elif self.state == self.StateWallFollowing.FIND_CORNER:
                            command_velocity_y_temp,    command_angle_rate_temp    =
self.command_align_corner(
                                        -1    *    self.max_turn_rate,    side_range,
self.reference_distance_from_wall)
            command_velocity_x_temp = 0.0
        else:
            # state does not exist, so hover!
                                command_velocity_x_temp,    command_velocity_y_temp,
command_angle_rate_temp = self.command_hover()

        command_velocity_x = command_velocity_x_temp*0.5
        command_velocity_y = command_velocity_y_temp*0.5
        command_yaw_rate = command_angle_rate_temp*0.5

       return command_velocity_x, command_velocity_y, command_yaw_rate, self.state
```

Main function

Overview

The system is meticulously designed to navigate a drone from a designated start point to a predetermined goal point within a digitally mapped environment. The process begins with the initialization phase, where the robot's core functionalities are activated, sensors such as GPS, IMU, and gyroscopes are calibrated, and the drone's motors are tested for operational readiness. Following this, a digital map of the drone's operating environment is created, with specific start and goal points defined. Known obstacles are integrated into this map to facilitate initial path planning.

The operational phase of the system involves executing the planned path from the start to the goal point. The drone navigates through the map, moving sequentially from one waypoint to another. A critical aspect of this system is its ability to dynamically adapt to new obstacles. As the drone's sensors detect new obstacles during its flight, these are recorded and added to the obstacle list, prompting an immediate update of the map. If the new obstacle requires a change in the drone's path, a re-planning process is initiated. The new starting point for this re-planning is the nearest navigable node to the drone's current location, while the goal point remains the same. The drone is then directed to follow this newly planned path to reach its destination.

Components

Robot Initialization: An instance of a Robot class is created, and the basic timestep for the simulation is set. This timestep dictates how often the robot's sensors update and control commands are sent.

```
# Create an instance of the Robot class
robot = Robot()
# Set the basic time step of the world's simulation
timestep = int(robot.getBasicTimeStep())
```

Motor Setup: Four motors (m1_motor, m2_motor, m3_motor, m4_motor) are initialized. Their positions are set to infinity, indicating velocity control mode, and initial velocities are assigned.

```
# Initialize and configure motors
# Motor 1 setup
m1_motor = robot.getDevice("m1_motor")
m1_motor.setPosition(float('inf'))  # Set position to infinity for velocity control
m1_motor.setVelocity(-1)            # Set initial velocity

# Motor 2 setup
m2_motor = robot.getDevice("m2_motor")
m2_motor.setPosition(float('inf'))
m2_motor.setVelocity(1)

# Motor 3 setup
m3_motor = robot.getDevice("m3_motor")
m3_motor.setPosition(float('inf'))
m3_motor.setVelocity(-1)

# Motor 4 setup
```

```
m4_motor = robot.getDevice("m4_motor")
m4_motor.setPosition(float('inf'))
m4_motor.setVelocity(1)
```

Sensor Initialization: Various sensors like an inertial unit (IMU), GPS, gyroscope, camera, and range finders are initialized and enabled with the timestep set earlier. These sensors would provide the robot with environmental awareness and self-positioning capabilities.

```
# Initialize and enable sensors
# Inertial Unit
imu = robot.getDevice("inertial_unit")
imu.enable(timestep)

# GPS
gps = robot.getDevice("gps")
gps.enable(timestep)

# Gyroscope
gyro = robot.getDevice("gyro")
gyro.enable(timestep)

# Camera
camera = robot.getDevice("camera")
camera.enable(timestep)

# Range finders for obstacle detection
range_front = robot.getDevice("range_front")
range_front.enable(timestep)
range_left = robot.getDevice("range_left")
range_left.enable(timestep)
range_back = robot.getDevice("range_back")
range_back.enable(timestep)
range_right = robot.getDevice("range_right")
range_right.enable(timestep)
```

Control and State Variables: Variables for tracking position, time, and other states are initialized. A PID controller for velocity control is also set up.

```
# Initialize control and state variables
# Variables for tracking position and time
past_x_global = 0
past_y_global = 0
past_time = 0
first_time = True
stop = 0

# Initialize PID controller for velocity control
PID_crazyflie = VelocityHeightPIDController()
PID_update_last_time = robot.getTime()
sensor_read_last_time = robot.getTime()
```

Wall following initialization: The script includes initialization for wall-following behavior and path planning using A* algorithm. It also includes a utility function for range conversion and hardcoded obstacle coordinates.

```
# Set desired height and yaw for flight control
height_desired = FLYING_ATTITUDE
yaw_desired = 0

# Initialize wall following behavior with specified parameters
```

```
                    wall_following          =          WallFollowing(angle_value_buffer=0.01,
reference_distance_from_wall=0.5,
                                                                                max_forward_speed=0.3,
init_state=WallFollowing.StateWallFollowing.FORWARD)

    # Set the mode of operation (autonomous or manual)
    autonomous_mode = True  # Change to False for manual control

    # Utility function for range conversion
    def num_to_range(num, inMin, inMax, outMin, outMax):
        # Convert a number from one range to another
        return outMin + (float(num - inMin) / float(inMax - inMin) * (outMax - outMin))
```

Node Class Definition: The Node class used to create node_map is not defined. This class should contain properties like row, col, and possibly others like g, h, f values for A* algorithm.

```
    # Define the grid size for path planning
    n = 61
    rows, cols = n, n

    # Create a 2D grid of nodes for path planning
    node_map = [[Node(i, j) for j in range(cols)] for i in range(rows)]
```

Obstacle Handling: The function add_obstacle is used but not defined. This function should take the node_map, obstacle_coordinates, and obstacle_nodes list, and mark the corresponding nodes in node_map as obstacles.

```
    # List to store nodes that are obstacles
    obstacle_nodes = []

    # Hardcoded obstacle coordinates
    obstacle_coordinates = [
        (20.755617763765958, -20.530367055839346), (20.0, -20.0),
        (20.310253990881947, -20.54893299676575), (18.0, -18.0),
        (20.101197626780323, -20.5909426585509), (19.0, -18.0),
        (20.08065910975286, -20.528085134744405), (18.0, -19.0),
        (19.804392395333274, -20.607163001012946), (20.0, -18.0),
        (19.80843750433482, -20.552655553159475), (18.0, -20.0)
    ]
```

Obstacle Coordinates Conversion: The obstacle_coordinates are given in what seems to be a coordinate system different from the grid coordinates. You need a method to convert these coordinates to grid indices.

```
    # Add obstacles to the node map
    add_obstacle(node_map, obstacle_coordinates, obstacle_nodes)

    # Function to check if a node is an obstacle
    def is_obstacle(node):
        return (node.row, node.col) in obstacle_nodes
```

Search Node Function: The function search_node is used to find the grid indices of the start and goal nodes but is not defined. This function should convert real-world coordinates (start_x, start_y, goal_x, goal_y) to grid indices.

```
# Goal and start coordinates
goal_x = -26
goal_y = 26
start_x = 26
start_y = -26
maintain_altitude = 1

# Find the start node
[i, j] = search_node(start_x, start_y)
print(f"row ={i} col = {j}")
start_node = node_map[i][j]

# Find the goal node
[i, j] = search_node(goal_x, goal_y)
print(f"row ={i} col = {j}")
goal_node = node_map[i][j]
```

A Path Planning Function: The A *_path_planning function is called but not defined. This function should implement the A* algorithm to find the shortest path from the start node to the goal node, considering obstacles.

```
# Perform A* path planning
path = astar_path_planning(node_map, start_node, goal_node)
```

Path Existence Check: The check if path: might not work as expected if path is an empty list. It's better to check if path is not None and len(path) > 0:.

```
# Check if a path is found
if path:
    print("Planned Path (GPS Coordinates):")
    # Uncomment below to print each node in the path
    # for node in path:
    #     print(node)
else:
    # Default path if no path is found
    path = [(x_global, y_global), (goal_x, goal_y)]
    print("No path found.")
```

Path Conversion: The section where you extract dd and ff from path assumes that path contains tuples of (x, y) coordinates. Ensure that your astar_path_planning function returns the path in this format.

```
# Initialize arrays to store x and y coordinates of the path
dd = []
ff = []

# Extract coordinates from the path if it exists
if path is not None:
    for i in range(len(path)):
        dd.append(path[i][0])  # Extract and store the x-coordinate
```

```
            ff.append(path[i][1])  # Extract and store the y-coordinate
    else:
        stop = 0
    # Initialize variables for the first coordinates in the path
    o = 0
    x = dd[0]
    y = ff[0]
```

Execution Loop

This script is designed for an autonomous robot (likely a drone) to navigate through an environment. It uses various sensors for orientation, position, and obstacle detection, and implements control algorithms for obstacle avoidance, path planning, and goal-oriented navigation.

Key Components

1. Initialization

flag_obs and flag_goal: Flags to indicate the presence of obstacles and whether the goal has been reached.

```
    # Flags to indicate the presence of obstacles and the goal
    flag_obs = 0
    flag_goal = 0
```

- first_time: A flag to initialize certain variables during the first iteration of the loop.
- past_time, past_x_global, past_y_global: Variables to store past time and global position for calculating velocities.

```
        # Initialize the global position and time during the first iteration
        if first_time:
            past_x_global = gps.getValues()[0]
            past_y_global = gps.getValues()[1]
            past_time = robot.getTime()
            first_time = False
```

2. Main Control Loop

- while robot.step(timestep) != -1: The main loop that runs continuously during the robot's operation.

```
    # Main control loop of the robot
    while robot.step(timestep) != -1:

        # Calculate the time difference since the last loop iteration
        dt = robot.getTime() - past_time
        # Dictionary to store the current state of the robot
        actual_state = {}
```

3. Sensor Data Acquisition

- IMU: Retrieves roll, pitch, and yaw.
- Gyroscope: Provides the yaw rate.

- GPS: Used for global positioning and velocity calculations.
- Range Sensors: Measure distances to detect obstacles.

```
        # Get sensor data
        # Roll, pitch, and yaw from the Inertial Measurement Unit (IMU)
        roll = imu.getRollPitchYaw()[0]
        pitch = imu.getRollPitchYaw()[1]
        yaw = imu.getRollPitchYaw()[2]

        # Yaw rate from the gyroscope
        yaw_rate = gyro.getValues()[2]

        # Global position from GPS and calculate global velocities
        x_global = gps.getValues()[0]
        v_x_global = (x_global - past_x_global) / dt
        y_global = gps.getValues()[1]
        v_y_global = (y_global - past_y_global) / dt
        altitude = gps.getValues()[2]

        # Get range sensor values in meters
        range_front_value = range_front.getValue() / 1000
        range_right_value = range_right.getValue() / 1000
        range_left_value = range_left.getValue() / 1000
        range_back_value = range_back.getValue() / 1000
```

- **Conversion:** Convert to global velocities to body fixed velocities

```
        # Convert global velocities to body-fixed velocities
        cos_yaw = cos(yaw)
        sin_yaw = sin(yaw)
        v_x = v_x_global * cos_yaw + v_y_global * sin_yaw
        v_y = -v_x_global * sin_yaw + v_y_global * cos_yaw
```

CASE 1: Obstacle Detection and Avoidance

- Checks if obstacles are within a predefined distance (hit_point).
- Updates obstacle coordinates and recalculates the path if an obstacle is detected.
- Implements a wall-following algorithm for obstacle avoidance.

```
        # Threshold distance for obstacle detection
        hit_point = 0.3

        # Check if any of the range sensors detect an obstacle within the hit point distance
        if (range_front_value < hit_point or range_right_value < hit_point or range_left_value
< hit_point):

            # Check if this is the first time an obstacle is detected
```

```
            if flag_obs == 0:
                 # Add the current GPS coordinates to the list of obstacle coordinates if not
already present
                if (x_global, y_global) not in obstacle_coordinates:
                    obstacle_coordinates.append((x_global, y_global))
                    add_obstacle(node_map, [(x_global, y_global)], obstacle_nodes)

             # Add the current target coordinates to the list of obstacle coordinates if not
already present
            if (x, y) not in obstacle_coordinates:
                obstacle_coordinates.append((x, y))
                add_obstacle(node_map, [(x, y)], obstacle_nodes)

            # Uncomment below to print each node in obstacle_nodes
            # for node in obstacle_nodes:
            #     print(node)

            # Set the wall following direction and get the side range value
            direction = WallFollowing.WallFollowingDirection.LEFT
            range_side_value = range_right_value

            # Get the velocity commands from the wall following state machine
                                    cmd_vel_x,     cmd_vel_y,     cmd_ang_w,     state_wf     =
wall_following.wall_follower(range_front_value,      range_side_value,      yaw,      direction,
robot.getTime())
```

CASE 2: RE-PLANNING AFTER OBSTACLE AVOIDED

5. Path Planning

- Utilizes A* algorithm for path planning when an obstacle is detected.
- Updates the robot's path and target waypoints accordingly.

```
        # Check if an obstacle was previously detected
        if (flag_obs == 1):
            # Reset the obstacle detection flag
            flag_obs = 0

             # Find the node in the node map corresponding to the robot's current
global position
            [i, j] = search_node(x_global, y_global)
            print(f"row ={i} col = {j}")

            # Set this node as the new start node for path planning
            start_node = node_map[i][j]

            # Find the node in the node map corresponding to the goal position
            [i, j] = search_node(goal_x, goal_y)
            print(f"row ={i} col = {j}")

            # Set this node as the goal node for path planning
            goal_node = node_map[i][j]
```

```
            # Perform A* path planning from the new start node to the goal node
            path = astar_path_planning(node_map, start_node, goal_node)

            # Initialize arrays to store x and y coordinates of the new path
            dd = []
            ff = []

            # If a path is found, extract and store the x and y coordinates
            if path is not None:
                for i in range(len(path)):
                    dd.append(path[i][0])  # Extract and store the x-coordinate
                    ff.append(path[i][1])  # Extract and store the y-coordinate
            else:
                # If no path is found, set the stop flag
                stop = 0

            # Set the first waypoint from the new path as the next target
            o = 1
            x = dd[1]
            y = ff[1]
```

CASE 3: GOAL POINT REACHED

Goal Detection and Landing

- Checks if the goal is reached and initiates a landing sequence.
- Gradually decreases altitude for a controlled landing.

```
        if flag_goal == 1:
            print("landing")
            # Uncomment below to print the obstacle coordinates
            # print(obstacle_coordinates)

            # Gradually decrease the altitude for landing
            if maintain_altitude > 0:
                maintain_altitude -= 0.001

            # Set the current position as the target for landing
            x = x_global
            y = y_global

            # Call the landing function to get the desired states for landing
             (height_desired, sideways_desired, yaw_desired, forward_desired, fl,
stop,) = land(
```

```
                maintain_altitude, yd2, fl, stop, altitude, x_global, y_global, x,
y,
                height_desired, sideways_desired, yaw_desired, forward_desired)

            # Use the PID controller to calculate motor power for landing
            motor_power = PID_crazyflie.pid(dt, forward_desired, sideways_desired,
                                            yaw_desired, height_desired,
                                            roll, pitch, yaw_rate,
                                            altitude, v_x, v_y)

            # Set motor velocities based on the calculated motor power
            m1_motor.setVelocity(-motor_power[0])
            m2_motor.setVelocity(motor_power[1])
            m3_motor.setVelocity(-motor_power[2])
            m4_motor.setVelocity(motor_power[3])

            # Update the past time and global position for the next iteration
            past_time = robot.getTime()
            past_x_global = x_global
            past_y_global = y_global

            continue  # Continue to the next iteration of the loop
```

CASE 4: NO OBSTACLE AND NO GOAL POINT REACHED

```
        # Check if the robot didnt detected any obstacle or also didnt reached the goal
        if stop == 0:

            # Call the fly function to get the desired states for flying
            (height_desired, sideways_desired, yaw_desired, forward_desired, fl, stop,) =
fly(
                maintain_altitude, yd2, fl, stop, altitude, x_global, y_global, x, y,
                height_desired, sideways_desired, yaw_desired, forward_desired)

        # Use the PID controller to calculate motor power for flying
        motor_power = PID_crazyflie.compute_pid(dt, forward_desired, sideways_desired,
                                                yaw_desired, height_desired,
                                                roll, pitch, yaw_rate,
                                                altitude, v_x, v_y)

        # Set motor velocities based on the calculated motor power
        m1_motor.setVelocity(-motor_power[0])
        m2_motor.setVelocity(motor_power[1])
        m3_motor.setVelocity(-motor_power[2])
        m4_motor.setVelocity(motor_power[3])

        # Update the past time and global position for the next iteration
        past_time = robot.getTime()
```

```
        past_x_global = x_global
        past_y_global = y_global
```

6. Control and Movement

- Calculates desired states (position, velocity, yaw) for navigation.
- Uses a PID controller to compute motor power for achieving these states.
- Adjusts motor velocities based on the PID controller's output.

```
            # Use the PID controller to calculate motor power for landing
            motor_power = PID_crazyflie.pid(dt, forward_desired, sideways_desired,
                                            yaw_desired, height_desired,
                                            roll, pitch, yaw_rate,
                                            altitude, v_x, v_y)

            # Set motor velocities based on the calculated motor power
            m1_motor.setVelocity(-motor_power[0])
            m2_motor.setVelocity(motor_power[1])
            m3_motor.setVelocity(-motor_power[2])
            m4_motor.setVelocity(motor_power[3])

            # Update the past time and global position for the next iteration
            past_time = robot.getTime()
            past_x_global = x_global
            past_y_global = y_global
```

Key Variables and Functions

- dt: Time difference since the last loop iteration.
- actual_state: Dictionary to store the current state of the robot.
- desired_state: Array to store the desired state of the robot.
- motor_power: Array to store motor power calculated by the PID controller.
- obstacle_coordinates: List to store coordinates of detected obstacles.
- path, dd, ff: Variables used for storing and handling the planned path.
- PID_crazyflie: Instance of a PID controller class.
- fly, land: Functions for handling flying and landing states.

Control Algorithms

- Wall Following: Used for navigating around obstacles.
- PID Control: For maintaining desired velocities and altitude.
- A Path Planning*: For recalculating the path when encountering obstacles.

Usage

- It requires a real-time environment where sensor data is continuously fed and motor commands are executed.
- The robot is expected to start in an autonomous mode, navigate towards a goal, avoid obstacles, and land upon reaching the destination.

Required import

1. math library
2. from controller import Robot
3. from math import cos, sin
4. from pid_controller import VelocityHeightPIDController
5. heapq library
6. itertools library
7. from wall_following import WallFollowing

Code

```
import math
from controller import Robot
from math import cos, sin
from pid_controller import VelocityHeightPIDController
import heapq
import itertools
from wall_following import WallFollowing

FLYING_ATTITUDE = 0
fl=0
# Main entry point of the script
if __name__ == '__main__':

    # Create an instance of the Robot class
    robot = Robot()
    # Set the basic time step of the world's simulation
    timestep = int(robot.getBasicTimeStep())

    # Initialize and configure motors
    # Motor 1 setup
    m1_motor = robot.getDevice("m1_motor")
    m1_motor.setPosition(float('inf'))  # Set position to infinity for velocity control
    m1_motor.setVelocity(-1)            # Set initial velocity

    # Motor 2 setup
    m2_motor = robot.getDevice("m2_motor")
    m2_motor.setPosition(float('inf'))
    m2_motor.setVelocity(1)

    # Motor 3 setup
    m3_motor = robot.getDevice("m3_motor")
    m3_motor.setPosition(float('inf'))
    m3_motor.setVelocity(-1)

    # Motor 4 setup
    m4_motor = robot.getDevice("m4_motor")
    m4_motor.setPosition(float('inf'))
    m4_motor.setVelocity(1)
```

```
    # Initialize and enable sensors
    # Inertial Unit
    imu = robot.getDevice("inertial_unit")
    imu.enable(timestep)

    # GPS
    gps = robot.getDevice("gps")
    gps.enable(timestep)

    # Gyroscope
    gyro = robot.getDevice("gyro")
    gyro.enable(timestep)

    # Camera
    camera = robot.getDevice("camera")
    camera.enable(timestep)

    # Range finders for obstacle detection
    range_front = robot.getDevice("range_front")
    range_front.enable(timestep)
    range_left = robot.getDevice("range_left")
    range_left.enable(timestep)
    range_back = robot.getDevice("range_back")
    range_back.enable(timestep)
    range_right = robot.getDevice("range_right")
    range_right.enable(timestep)

    # Initialize control and state variables
    # Variables for tracking position and time
    past_x_global = 0
    past_y_global = 0
    past_time = 0
    first_time = True
    stop = 0

    # Initialize PID controller for velocity control
    PID_crazyflie = VelocityHeightPIDController()
    PID_update_last_time = robot.getTime()
    sensor_read_last_time = robot.getTime()

    # Set desired height and yaw for flight control
    height_desired = FLYING_ATTITUDE
    yaw_desired = 0

    # Initialize wall following behavior with specified parameters
                    wall_following         =         WallFollowing(angle_value_buffer=0.01,
reference_distance_from_wall=0.5,
                                                                   max_forward_speed=0.3,
init_state=WallFollowing.StateWallFollowing.FORWARD)
```

```python
# Set the mode of operation (autonomous or manual)
autonomous_mode = True  # Change to False for manual control

# Utility function for range conversion
def num_to_range(num, inMin, inMax, outMin, outMax):
    # Convert a number from one range to another
    return outMin + (float(num - inMin) / float(inMax - inMin) * (outMax - outMin))

# Define the grid size for path planning
n = 61
rows, cols = n, n

# Create a 2D grid of nodes for path planning
node_map = [[Node(i, j) for j in range(cols)] for i in range(rows)]

# List to store nodes that are obstacles
obstacle_nodes = []

# Hardcoded obstacle coordinates
obstacle_coordinates = [
    (20.755617763765958, -20.530367055839346), (20.0, -20.0),
    (20.310253990881947, -20.54893299676575), (18.0, -18.0),
    (20.101197626780323, -20.5909426585509), (19.0, -18.0),
    (20.08065910975286, -20.528085134744405), (18.0, -19.0),
    (19.804392395333274, -20.607163001012946), (20.0, -18.0),
    (19.80843750433482, -20.552655553159475), (18.0, -20.0)
]

# Add obstacles to the node map
add_obstacle(node_map, obstacle_coordinates, obstacle_nodes)

# Function to check if a node is an obstacle
def is_obstacle(node):
    return (node.row, node.col) in obstacle_nodes

# Goal and start coordinates
goal_x = -26
goal_y = 26
start_x = 26
start_y = -26
maintain_altitude = 1

# Find the start node
[i, j] = search_node(start_x, start_y)
print(f"row ={i} col = {j}")
start_node = node_map[i][j]

# Find the goal node
[i, j] = search_node(goal_x, goal_y)
print(f"row ={i} col = {j}")
```

```
    goal_node = node_map[i][j]

    # Perform A* path planning
    path = astar_path_planning(node_map, start_node, goal_node)

    # Check if a path is found
    if path:
        print("Planned Path (GPS Coordinates):")
        # Uncomment below to print each node in the path
        # for node in path:
        #     print(node)
    else:
        # Default path if no path is found
        path = [(x_global, y_global), (goal_x, goal_y)]
        print("No path found.")

    # Initialize arrays to store x and y coordinates of the path
    dd = []
    ff = []

    # Extract coordinates from the path if it exists
    if path is not None:
        for i in range(len(path)):
            dd.append(path[i][0])  # Extract and store the x-coordinate
            ff.append(path[i][1])  # Extract and store the y-coordinate
    else:
        stop = 0

    # Initialize variables for the first coordinates in the path
    o = 0
    x = dd[0]
    y = ff[0]
    # Flags to indicate the presence of obstacles and the goal
    flag_obs = 0
    flag_goal = 0

    # Main control loop of the robot
    while robot.step(timestep) != -1:

        # Calculate the time difference since the last loop iteration
        dt = robot.getTime() - past_time
        # Dictionary to store the current state of the robot
        actual_state = {}

        # Initialize the global position and time during the first iteration
        if first_time:
            past_x_global = gps.getValues()[0]
            past_y_global = gps.getValues()[1]
            past_time = robot.getTime()
            first_time = False
```

```
        # Get sensor data
        # Roll, pitch, and yaw from the Inertial Measurement Unit (IMU)
        roll = imu.getRollPitchYaw()[0]
        pitch = imu.getRollPitchYaw()[1]
        yaw = imu.getRollPitchYaw()[2]

        # Yaw rate from the gyroscope
        yaw_rate = gyro.getValues()[2]

        # Global position from GPS and calculate global velocities
        x_global = gps.getValues()[0]
        v_x_global = (x_global - past_x_global) / dt
        y_global = gps.getValues()[1]
        v_y_global = (y_global - past_y_global) / dt
        altitude = gps.getValues()[2]

        # Get range sensor values in meters
        range_front_value = range_front.getValue() / 1000
        range_right_value = range_right.getValue() / 1000
        range_left_value = range_left.getValue() / 1000
        range_back_value = range_back.getValue() / 1000

        # Convert global velocities to body-fixed velocities
        cos_yaw = cos(yaw)
        sin_yaw = sin(yaw)
        v_x = v_x_global * cos_yaw + v_y_global * sin_yaw
        v_y = -v_x_global * sin_yaw + v_y_global * cos_yaw

        # Initialize desired state variables
        desired_state = [0, 0, 0, 0]
        forward_desired = 0
        sideways_desired = 0
        yaw_desired = 0
        height_diff_desired = 0

        # Threshold distance for obstacle detection
        hit_point = 0.3

       # Check if any of the range sensors detect an obstacle within the hit point distance
            if (range_front_value < hit_point or range_right_value < hit_point or
range_left_value < hit_point):

            # Check if this is the first time an obstacle is detected
            if flag_obs == 0:
                 # Add the current GPS coordinates to the list of obstacle coordinates if
not already present
                if (x_global, y_global) not in obstacle_coordinates:
                    obstacle_coordinates.append((x_global, y_global))
                    add_obstacle(node_map, [(x_global, y_global)], obstacle_nodes)
```

```
            # Add the current target coordinates to the list of obstacle coordinates if
not already present
            if (x, y) not in obstacle_coordinates:
                obstacle_coordinates.append((x, y))
                add_obstacle(node_map, [(x, y)], obstacle_nodes)

            # Uncomment below to print each node in obstacle_nodes
            # for node in obstacle_nodes:
            #     print(node)

            # Set the wall following direction and get the side range value
            direction = WallFollowing.WallFollowingDirection.LEFT
            range_side_value = range_right_value

            # Get the velocity commands from the wall following state machine
                                cmd_vel_x,     cmd_vel_y,     cmd_ang_w,     state_wf     =
wall_following.wall_follower(range_front_value,    range_side_value,    yaw,    direction,
robot.getTime())

            # If in autonomous mode, set the desired velocities and angular velocity
            if autonomous_mode:
                sideways_desired = cmd_vel_y
                forward_desired = cmd_vel_x
                yaw_desired = cmd_ang_w

              # Use the PID controller to calculate motor power for maintaining desired
velocities and height
            motor_power = PID_crazyflie.pid(dt, forward_desired, sideways_desired,
                                            yaw_desired, height_desired,
                                            roll, pitch, yaw_rate,
                                            altitude, v_x, v_y)

            # Set motor velocities based on the calculated motor power
            m1_motor.setVelocity(-motor_power[0])
            m2_motor.setVelocity(motor_power[1])
            m3_motor.setVelocity(-motor_power[2])
            m4_motor.setVelocity(motor_power[3])

            # Update the past time and global position for the next iteration
            past_time = robot.getTime()
            past_x_global = x_global
            past_y_global = y_global
            # Set the flag to indicate an obstacle has been detected
            flag_obs = 1
            continue  # Continue to the next iteration of the loop


        # Check if an obstacle was previously detected
        if (flag_obs == 1):
```

```
            # Reset the obstacle detection flag
            flag_obs = 0

             # Find the node in the node map corresponding to the robot's current global
position
            [i, j] = search_node(x_global, y_global)
            print(f"row ={i} col = {j}")

            # Set this node as the new start node for path planning
            start_node = node_map[i][j]

            # Find the node in the node map corresponding to the goal position
            [i, j] = search_node(goal_x, goal_y)
            print(f"row ={i} col = {j}")

            # Set this node as the goal node for path planning
            goal_node = node_map[i][j]

            # Perform A* path planning from the new start node to the goal node
            path = astar_path_planning(node_map, start_node, goal_node)

            # Initialize arrays to store x and y coordinates of the new path
            dd = []
            ff = []

            # If a path is found, extract and store the x and y coordinates
            if path is not None:
                for i in range(len(path)):
                    dd.append(path[i][0])  # Extract and store the x-coordinate
                    ff.append(path[i][1])  # Extract and store the y-coordinate
            else:
                # If no path is found, set the stop flag
                stop = 0

            # Set the first waypoint from the new path as the next target
            o = 1
            x = dd[1]
            y = ff[1]
            # Uncomment below to print the next target coordinates
            # print(f"{x} x y {y}")

        # Initialize desired state and movement variables
        desired_state = [0, 0, 0, 0]
        forward_desired = 0
        sideways_desired = 0
        yaw_desired = 0
        height_diff_desired = 0

        # Check if the robot has reached the last waypoint
        if o >= len(dd) - 1:
```

```
            stop = 0
            flag_goal = 1  # Set the flag indicating the goal has been reached

        # If the robot has not stopped, update the next waypoint
        if stop != 0:
            o = o + 1
            stop = 0
            x = dd[o]  # Update the x-coordinate of the next waypoint
            y = ff[o]  # Update the y-coordinate of the next waypoint

        # Update the desired height
        height_desired += height_diff_desired * dt

        # Calculate the angle to the next waypoint
        angle = math.degrees(math.atan2((y - y_global), (x - x_global)))
        yd2 = angle if angle > 0 else 180 + (180 + angle)

        # Convert the angle to a range suitable for the robot's control system
        if angle > 0:
            yd = num_to_range(angle, 0, 180, 0, 3)
        else:
            yd = num_to_range(angle, 0, -180, 0, -3)

        # Check if the goal has been reached to initiate landing
        if flag_goal == 1:
            print("landing")
            # Uncomment below to print the obstacle coordinates
            # print(obstacle_coordinates)

            # Gradually decrease the altitude for landing
            if maintain_altitude > 0:
                maintain_altitude -= 0.001

            # Set the current position as the target for landing
            x = x_global
            y = y_global

            # Call the landing function to get the desired states for landing
            (height_desired, sideways_desired, yaw_desired, forward_desired, fl, stop,) =
land(
                maintain_altitude, yd2, fl, stop, altitude, x_global, y_global, x, y,
                height_desired, sideways_desired, yaw_desired, forward_desired)

            # Use the PID controller to calculate motor power for landing
            motor_power = PID_crazyflie.pid(dt, forward_desired, sideways_desired,
                                            yaw_desired, height_desired,
                                            roll, pitch, yaw_rate,
                                            altitude, v_x, v_y)

            # Set motor velocities based on the calculated motor power
```

```
            m1_motor.setVelocity(-motor_power[0])
            m2_motor.setVelocity(motor_power[1])
            m3_motor.setVelocity(-motor_power[2])
            m4_motor.setVelocity(motor_power[3])

            # Update the past time and global position for the next iteration
            past_time = robot.getTime()
            past_x_global = x_global
            past_y_global = y_global

            continue  # Continue to the next iteration of the loop

        # Check if the robot didnt detected any obstacle or also didnt reached the goal
        if stop == 0:

            # Call the fly function to get the desired states for flying
            (height_desired, sideways_desired, yaw_desired, forward_desired, fl, stop,) =
fly(
                maintain_altitude, yd2, fl, stop, altitude, x_global, y_global, x, y,
                height_desired, sideways_desired, yaw_desired, forward_desired)

        # Use the PID controller to calculate motor power for flying
        motor_power = PID_crazyflie.compute_pid(dt, forward_desired, sideways_desired,
                                                yaw_desired, height_desired,
                                                roll, pitch, yaw_rate,
                                                altitude, v_x, v_y)

        # Set motor velocities based on the calculated motor power
        m1_motor.setVelocity(-motor_power[0])
        m2_motor.setVelocity(motor_power[1])
        m3_motor.setVelocity(-motor_power[2])
        m4_motor.setVelocity(motor_power[3])

        # Update the past time and global position for the next iteration
        past_time = robot.getTime()
        past_x_global = x_global
        past_y_global = y_global
```

Simulated Path example:

The figure 6 is a screenshot from a simulation from webots. It shows two different scenarios or environments along with their corresponding path plots below each one.

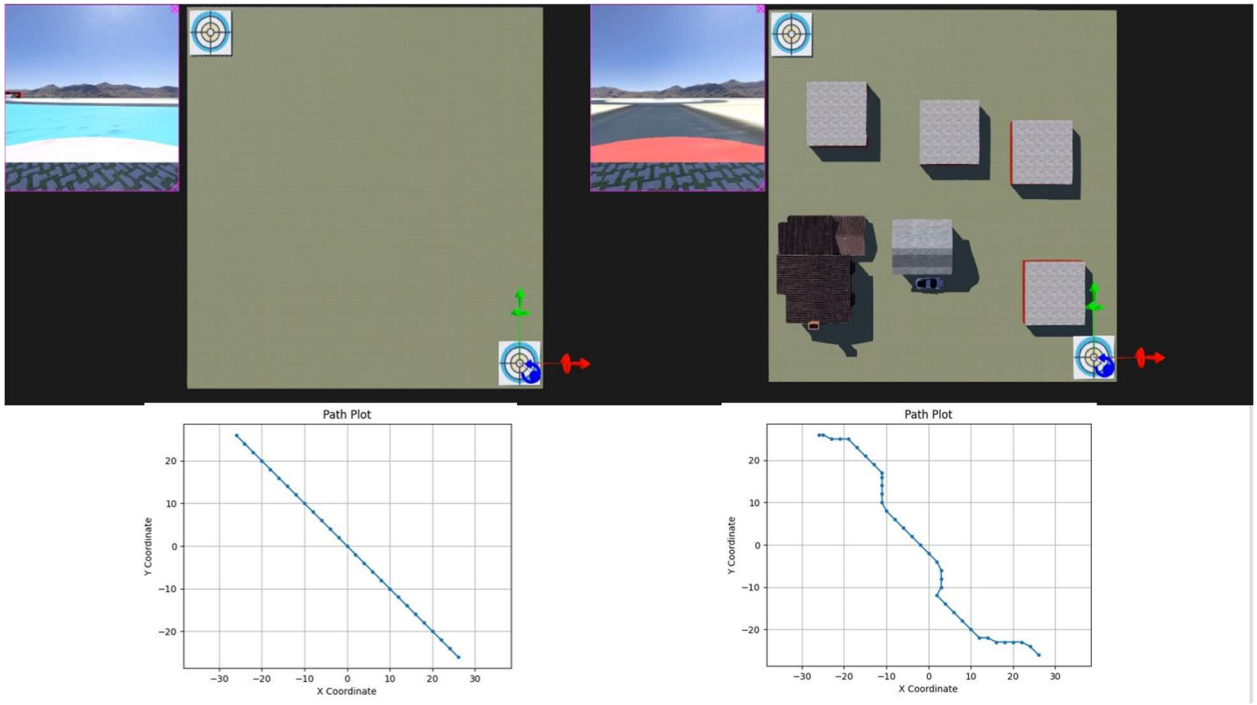

Figure 6 Planned Path in different scenario

1. Left Scenario:
    - Environment Visualization: This shows an open, relatively obstacle-free area with a starting point bottom right (26, -26) and a target destination top left (-26,26). The blue line could represent the water or a different terrain type, suggesting that the environment may be a simulation for an amphibious vehicle or a scenario with varied terrain.
    - Path Plot: The corresponding plot shows a simple, direct path from the start to the target, indicated by a straight line on the graph. The axes labeled 'X Coordinate' and 'Y Coordinate' plot the position of the vehicle over time, suggesting that the vehicle is expected to move in a straight line from the start to the goal without any deviations.
2. Right Scenario:
    - Environment Visualization: This one is more complex, featuring several obstacles that the vehicle must navigate around. These obstacles are represented by variously sized and textured cubes. The starting and target points are marked similarly to the first scenario.

- Path Plot: Below, the graph shows a more complex path with turns and changes in direction, which corresponds to the need to navigate around obstacles. The plotted line indicates the vehicle's path planning algorithm has computed a route that successfully avoids these obstacles and reaches the goal.

In both scenarios, the path plots serve as a graphical representation of the planned path the vehicle will take in two dimensions (X and Y). The starting point is typically at the origin of the plot, and the end goal is where the line terminates. The path is likely calculated using an algorithm similar to the A* pathfinding algorithm you described earlier, taking into account the drone current position, the target location, and any potential obstacles in the environment to determine the most efficient path.

Conclusion

In concluding the exploration of mobile robotics, it is evident that the field has made significant strides, particularly in the realm of path planning and obstacle navigation. The document highlights the sophisticated algorithms that enable mobile robots to maneuver through complex environments, showcasing their advanced computational and navigational capabilities.

The inclusion of a path plot, as discussed in the final sections, illustrates a practical example of these advancements. This plot, detailing a route with various turns and directional changes, exemplifies the robot's ability to adapt to its surroundings and overcome physical barriers. Such capability is not just a feat of engineering; it represents a culmination of interdisciplinary efforts encompassing robotics, computer science, and artificial intelligence.

Central to this achievement is the path planning algorithm, likely akin to the A* pathfinding algorithm. This algorithm represents the core of the robot's decision-making process, balancing the need to reach a target location efficiently while avoiding obstacles. By considering the robot's current position, the destination, and potential hindrances, the algorithm plots a course that optimizes for both safety and speed. This is indicative of how mobile robotics has evolved from basic navigational tasks to handling more intricate scenarios.

Furthermore, these advancements in mobile robotics have broader implications. They pave the way for more autonomous systems capable of performing complex tasks in varied environments. From industrial applications to search and rescue operations, the potential for these robots is vast and continually expanding.

In essence, the field of mobile robotics stands at a pivotal juncture. As technology advances, so too do the possibilities for these robots. They are no longer confined to controlled settings; instead, they are increasingly capable of operating in dynamic, unpredictable environments. This marks a significant leap forward, not just for robotics, but for the myriad industries that stand to benefit from these technological marvels. The future of mobile robotics, therefore, is not just about technological advancement, but about the integration of these machines into the very fabric of daily life, enhancing efficiency, safety, and capabilities across various sectors.