RBE 550 - Motion Planning Project Final Report

Shreyas Chigurupati
schigurupati@wpi.edu

Howard Ho
hho2@wpi.edu

Philip Ni
pcni@wpi.edu

***Abstract*—Livestock monitoring represents a challenge for the agriculture industry, especially for those managing large pastures. In this paper, we propose using a an unmanned aerial vehicle (UAV) quadrotor to complete menial or repetitive tasks. We will focus on how to find optimal paths to specific checkpoints to complete prioritized tasks, while avoiding obstacles.**

I. Introduction

The integration of technology into agriculture has revolutionized several aspects of the industry, from crop monitoring to irrigation systems. However, a unique challenge exists with monitoring and herding livestock. Traditional methods are often labor-intensive and may result in inaccuracies due to human error [1]. GPS collar-based monitoring is another option, but it requires manual data analysis and may not provide timely updates [2]. Unmanned Aerial Vehicles (UAVs), commonly known as drones, have the potential to offer a more efficient method of monitoring cattle [3].

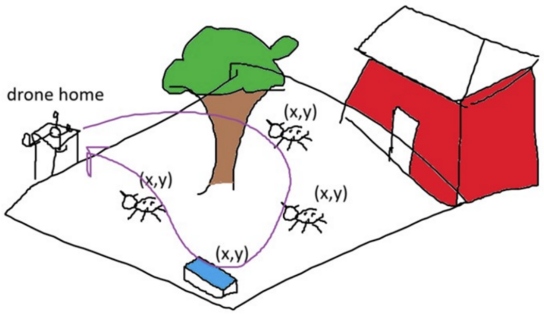

Fig. 1. Demonstrates the quadrotor making a lap from home base, to inspect each livestock, and back to home base. Notice the the quadrotor is expected to avoid obstacles - like the tree - while also performing side tasks like visiting the watering trough.

II. Objectives

A. Reliable Livestock Coverage

- Goal: The Drone will fly to each individual livestock in a given area.
  - *Assumption*: Livestock locations will be predetermined and fixed.
  - *Assumption*: The Drone will position itself to get sensor data for the entire circumference of the livestock while it is at its designated location for a specified time.
  - *Reach Goal*: Livestock change position over time.
- Goal: The drone will travel above the clearance heights. For example, the Drone will fly the perimeter of the fences to inspect for breaks, or fly above the livestock without colliding. Upon arrival to an animal the drone will capture the tag data of the livestock to store the perceived status with the appropriate livestock tag.
  - *Assumption*: Sensor data will be able to perceive all tag readings and livestock statuses from its regular clearance height.
  - *Assumption*: Livestock will stay in open areas not hindered by trees.

B. Waypoint Priority

- Goal: The drone will fly to each fixed checkpoint location.
- Goal: Prioritize emergency deviations. The drone will return to where it left off on its task priority after the deviation has been dealt with.

C. Optimal Path to Checkpoints

- Goal: The drone will assess a set of waypoints to determine the shortest route to the target or fastest route to the target, whichever is preferred for the task.
- Goal: Determine if the focus is efficiency (shortest path) or speed (path with fastest speed).

D. Reach Objective - Path correction

- Goal: If the drone is knocked off its path by wind, etc., it will find a way back to the original path, or determine a new one. Recover the path planning when controls are unable to compensate.
  - Assumption: Compliance and/or compensation for wind is assumed to be handled by the Control layer.
  - Assumption: Absolute position data will be retained for where the drone is.

III. Methodology

A. Development and Simulation Environment

We used Cyberbotics' Webots [15] to create a 3D simulated environment that mimics the real-world layout of a cattle field. The Webots interface and world visualizer lets us place objects into the environment including the cattle and various obstacles such a barn, trees, fences, and so on. Beyond the visuals and 3D environment, Webots allows for the created environment to be analyzed and manipulated via scripts in various languages. We used this feature to control the simulated quad copter, and execute back-end utility functions in Python 3 [13].

B. Algorithms

1) Configuration Space Analyzer: For any motion planning or path finding algorithm, it needs a known free space to path through. The ”nodes” in our path finding tree are simply all the (x,y,z) coordinates of the farm where the quad copter does not hit an obstacle. Our implementation of finding these points was done in the Cartesian space utilizing Webots' built-in collision detection between objects. The 3D environment of the farm was divided into a 3D grid of points 0.1m apart. Via script, a quad copter stand-in model was then placed sequentially at each point on the 3D grid to check if it collided with anything at that position. The coordinates where the stand-in model did not collide with anything was saved to a list of non-colliding nodes that the path finding algorithm could path through. An added benefit of analyzing the free space this way is that the stand-in model can be made bigger or smaller to adjust how close to obstacles we want to allow the quad copter to get.

2) Terrain Following and Obstacle Avoidance: We will model the topography of the terrain and obstacles as a 3D point cloud. From that, we can use a b-spline to generate a set of altitude waypoints that allow the quadrotor to follow the contour of the terrain.

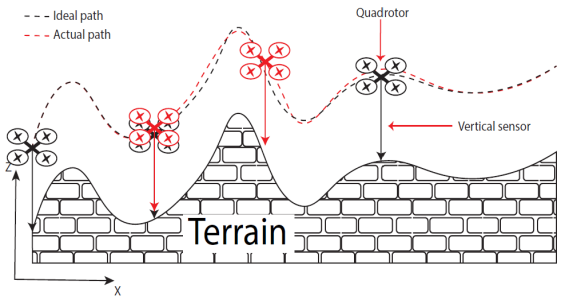

Fig. 2. Demonstrates terrain following altitude pathfinding using curve-fitting b-spline. Image provided by AlQahtani, et. al. [9]

Furthermore, we can add weights to those waypoints to indicate a checkpoint (or task goal) to prioritize or obstacle to avoid. This will help us determine what pathing behavior is required to navigate through that waypoint.

3) Motion Planning Algorithms: Based on the task and priority, we will need to decide what type of path profile we want. The path plan will be effected by the task's priority, energy profile, and any obstacles or hazards. The energy profile describes whether the fastest path or shortest path is preferred.

We will choose the optimal path from the waypoints generated from the 3d point cloud using one or more of the following algorithms:

- A*
- D*
- Rapidly-exploring Random Trees (RRT)
- Dijkstra

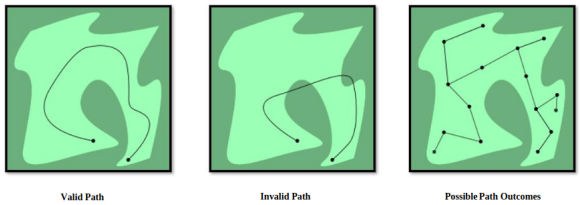

Fig. 3. Simple demonstration of path finding.

IV. Path Planning - Algorithm Implementations

A. *Rapid Random Tree (RRT* variant)*

1) Overview: RRT is an optimized version of the RRT algorithm designed for efficient path planning in high-dimensional spaces. It is particularly useful for non-holonomic systems and systems with differential constraints.

2) Logic:

- Initialization: Start with an initial point as the root of the tree.
- Random Sampling: At each iteration, a point is randomly sampled in the search space.
- Nearest Neighbor: The algorithm then finds the nearest node in the tree to this random point.
- Steering: A new node is then created by steering the tree from the nearest node towards the random point.
- Optimization: The tree is rewired to ensure that the paths are optimal.

```python
class Node:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.parent = None

    def sample_point():
        return Node(random.uniform(0, 10), random.uniform(0, 10))

    def find_nearest_node(random_point, tree):
        return min(tree, key=lambda node: distance(node, random_point))

    def extend_tree(nearest_node, random_point):
        new_node = Node(nearest_node.x + (random_point.x - nearest_node.x) / 2, nearest_node.y + (random_point.y - nearest_node.y) / 2)
        new_node.parent = nearest_node
        return new_node

    def rewire_tree(new_node, tree):
        for node in tree:
            if distance(node, new_node) < distance(new_node.parent, new_node):
                new_node.parent = node
```

Listing 1. Python Implementation - RRT*

B. *A* Algorithm*

1) Overview: A* is a pathfinding algorithm that uses heuristics to efficiently find the shortest path between a start and a goal node. It is complete and optimal, given that the heuristic is admissible.

2) Logic:

- Initialization: Both the open and closed lists are initialized. The start node is added to the open list.
- Main Loop: The node with the lowest f value is moved from the open to the closed list.
- Successor Nodes: Successor nodes are generated for the current node and evaluated.
- Path Reconstruction: Once the goal node is reached, the path is reconstructed.

```
class Node:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.f = 0
        self.g = 0
        self.h = 0
        self.parent = None

def a_star(start, goal):
    open_list = [start]
    closed_list = []
    while open_list:
        current_node = min(open_list, key=lambda x:
    x.f)
        open_list.remove(current_node)
        closed_list.append(current_node)
        if current_node == goal:
            return reconstruct_path(current_node)
        for neighbor in get_neighbors(current_node):
            if neighbor in closed_list:
                continue
            tentative_g = current_node.g + distance(
    current_node, neighbor)
            if neighbor not in open_list:
                open_list.append(neighbor)
            elif tentative_g >= neighbor.g:
                continue
            neighbor.g = tentative_g
            neighbor.h = distance(neighbor, goal)
            neighbor.f = neighbor.g + neighbor.h
            neighbor.parent = current_node
```

Listing 2. Python Implementation - A*

C. *D* Algorithm*

1) Overview: D* is an extension of the A* algorithm designed for path planning in dynamic environments. It can efficiently update paths when changes in the environment are detected.

2) Logic:

- Initialization: All nodes are initialized with infinite g values except for the goal node.
- Cost Propagation: Costs are propagated from the goal node to the start node.
- Dynamic Changes: If changes in the environment are detected, the affected nodes are updated.
- Replanning: The path is replanned.

```
class Node:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.g = float('infinity')
        self.rhs = float('infinity')
        self.parent = None

class DStar:
    def __init__(self):
        self.open_list = []
        self.closed_list = []

def propagate_costs(self):
    while self.open_list:
        current_node = min(self.open_list, key=
    lambda x: x.g + x.rhs)
        self.open_list.remove(current_node)
        self.closed_list.append(current_node)
        for neighbor in get_neighbors(current_node):
            if neighbor.rhs > current_node.g + cost(
    current_node, neighbor):
                neighbor.rhs = current_node.g + cost
    (current_node, neighbor)
                neighbor.parent = current_node
                if neighbor not in self.open_list:
                    self.open_list.append(neighbor)

def dynamic_changes_detected(self):
    # Example: Obstacle detected
    return True

def update_affected_nodes(self, obstacle):
    for node in self.closed_list:
        if is_affected(node, obstacle):
            self.open_list.append(node)
            self.closed_list.remove(node)

def replan(self):
    self.propagate_costs()
    if self.dynamic_changes_detected():
        obstacle = get_new_obstacle()  # Example:
    New obstacle coordinates
        self.update_affected_nodes(obstacle)
        self.replan()
```

Listing 3. Python Implementation - D*

D. *Dijkstra Algorithm*

1) Overview: Dijkstra's Algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge weights, producing a shortest path tree. This algorithm exists in various variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the 'source' node and finds shortest paths from the source node to all other nodes in the graph.

2) Logic:

- Initialization: Start with an initial node and set its distance to zero. All other nodes have infinite distance.
- Current Node: Select the node with the smallest tentative distance as the current node.
- Update Neighbors: Update the tentative distance of each neighbor of the current node.

- Mark Visited: Mark the current node as visited.
- Termination: Stop if the destination node is visited or if the smallest tentative distance is infinity.

```python
import heapq

def dijkstra(graph, start):
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
```

Listing 4. Python Implementation - Dijkstra

V. Simulation Environment

1) Object Placement: Webots offers the ability to create a simulation environment by placing 3D models into a world. Simple bounding geometries can also be added around the models to use in obstacle collision, path finding, and collision of the drone.

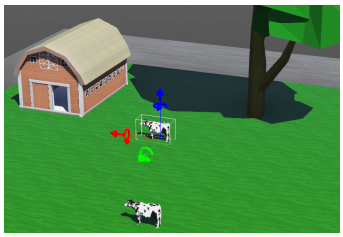

Fig. 4. A barn, tree, and two cows placed in the Webots simulation environment. The selected cow shows its collision box

The size, position, and orientation of objects are easily adjustable by clicking and dragging the arrows. In this way, the positions of the obstacles can also be shuffled to test the robustness of the path finding algorithm in multiple scenarios.

2) Position Randomization: Using Webots' supervisor controller, coding languages can be used to manipulate the components of the simulation. This allows for automatic shuffling of the cows, trees, and other objects around the simulation each time it is run. In this case, Python is used to randomly place 5 cows around the farm. The number of objects can also be randomly chosen, which would further test the capabilities of our algorithms under various circumstances.

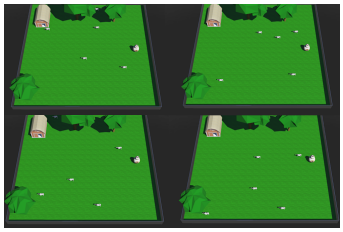

Fig. 5. Four examples of randomized initial cow positions by script

3) Object Collision: Webots allows placing simple geometries, like boxes, cylinders, and spheres, around the models to act as collision boundaries. These are important to make sure the drone is actually penalized for hitting or getting too close to obstacles in the simulation, as it would be in the real world. As discussed in the Methodology section, these collision boxes are also used in the free space analyzer. When these bounding boxes overlap with the stand-in model during analysis, it eliminates that respective coordinate from the free space. Multiple geometries can be combined to outline the entire model, like in the case of the tree. One odd limitation of Webots is that these bounding boxes are not scaled automatically with the model so they have to be adjusted individually. With object

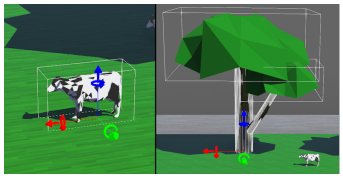

Fig. 6. Collision boundaries around a cow and tree

collision and environment generation handled by Webots, the simulation is ready to test the quad copter navigating through the world to test the performance of different algorithms. The

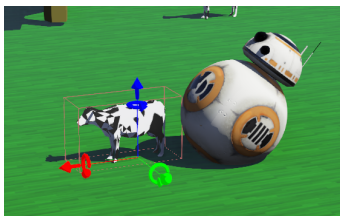

Fig. 7. A rolling robot collides with a cow

(x,y,z) positions of the objects to visit will be fed into the path finding algorithm. The resulting calculated path would then be fed back into the simulation for the drone to follow. We can then watch the results to see how successfully the drone navigates the farm, avoids the obstacles, and visits the points of interest.

VI. Quadcopter Control in Webots Simulation

We imported quadcopter body from the Webots sample library. There are four motors to control X, Y, and Z axis movement and rotations.

Fig. 8. Image of the simple quadcopter body provided from the Webots sample library.

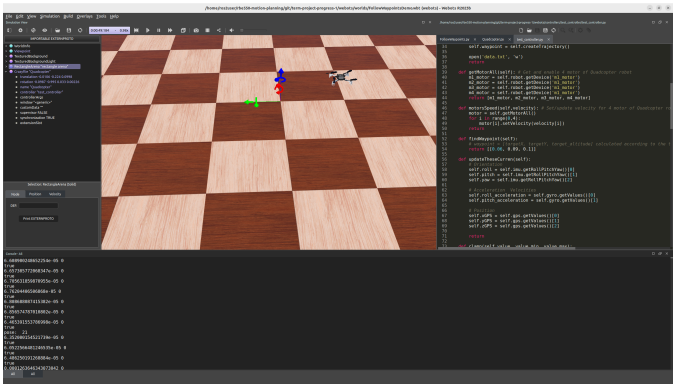

Fig. 9. Image of the simple quadcopter demo using Python PID library for controls.

A simple PID controller was implemented using the Python simple_pid library. We retrieved the craft's position, velocity, and acceleration data from the GPS, intertial measurement unit, gyro, and compass libraries of the Python control systems package. The simulation demonstrates that the controller could take a pre-planned trajectory as input, and control the motors to make the craft follow that path.

The screenshot below demonstrates the Webots environment. The robot and environment objects are established on the left-hand panel. The right-hand panel has the Python/source code editor. The center panel is used to display the graphics, where the robot is simulated. This center panel allow the user to manipulate the scene, or view the scene as it's rendered. The panel on the bottom is the terminal output for the code.

A recording of the demonstration is provided in the report's presentation.

In our controller demonstration, we provided a circular path for the quadcopter to follow. The quadcopter starts at the center of the circle, and must rise and translate to reach its first waypoint at the 12 o'clock position. Then, it makes a complete lap around the circumference of the circular path. The craft finally reaches an end when it reaches the 12 o'clock position again.

Below is a graph of the overall performance of the quadcopter controller. Looking at the red "Yaw" line, we can see that the initial kick to launch causes a rotation of the quadcopter as it gets into positions. The Z-value is the altitude, and can be seen to level out after 2 seconds. This diagram represents the first ten seconds of the simulation. More data was collected, but could not be parsed.

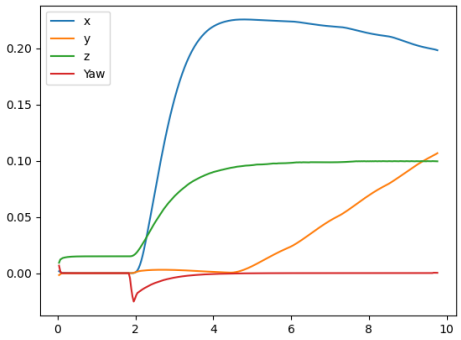

Fig. 10. PID controller data. Relative time-scale shown on the graph's X-axis. The Y-axis shows change in position. Yaw plot used to indicate relative stability

VII. Results

The execution of various pathfinding algorithms within the Webots simulation environment has culminated in the autonomous navigation of a quadrotor drone, modeled after Bitcraze's Crazyflie, through a complex virtual environment peppered with static obstacles. Among the algorithms tested—Dijkstra's, RRT*, D*, and A*—the A* algorithm emerged as the superior approach for this application, offering the most efficient pathfinding capabilities both in terms of computational time and path efficiency. The other algorithms did not resolve a trajectory in a reasonable amount of time.

The drone's task was to traverse from a start point to multiple goal points while skillfully avoiding obstacles, such as trees and barriers, within a defined arena. The simulation results revealed that the drone was able to accurately map its environment and maintain adherence to the calculated paths. A* algorithm's heuristic approach enabled the quadrotor to make informed decisions, leading to smoother and more direct routes compared to the other algorithms tested.

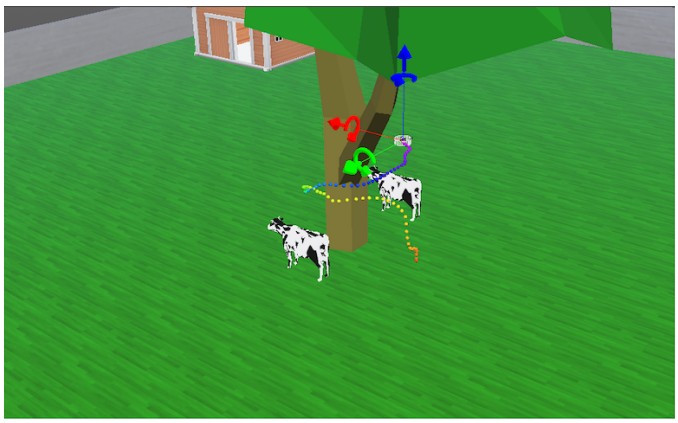

Fig. 11. Image of Crazyflie performing A* algorithm.

Throughout the trials, the quadrotor consistently maintained the predefined flying altitude of 1.0 meter. This altitude control

can be attributed to the finely tuned Proportional-Integral-Derivative (PID) controllers, which governed the drone's attitude and altitude. These controllers, through precise adjustments, ensured the drone's stability even when confronted with sudden environmental changes or when navigating through tight spaces.

The quadrotor's flight path, delineated by a sequence of colored dots in the provided images, illustrates a clear trajectory from the starting position to each goal. The effectiveness of the motion planning system is highlighted by the drone's ability to reach the end points without any collisions or deviations from the path, a testament to the robustness of the A* algorithm within this simulated context.

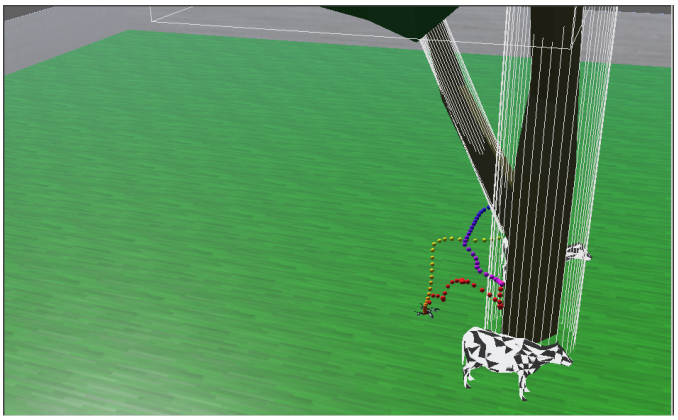

Fig. 12. Collision avoidance with the obstacles.

A point of note was the initial orientation discrepancy observed at the start of the simulation, where the quadrotor's rotation deviated from the expected values specified in the PROTO file. Despite this initial state anomaly, the control system's compensatory measures ensured the drone's performance remained unaffected, adhering to the intended flight plan.

VIII. Challenges

We addressed the avoidance in our implementation of the A* algorithm. We also discover all free spaces using a supervisor node in the Webots simulation. This created a list of all available navigable nodes, and excluded any obstacle nodes.

1) Integration of Code: We successfully addressed our integration challenges from the progress report. By using the same version of Webots, along with Git on BitBucket for version control, we were able to successfully integrate each of our parts into the final project.

Each member of our team has successfully installed and set up the development environment. So, we have been implementing our parts as separate demonstrations. Each member is using Webots on different operating systems.

- Shreyas Chigurupati: Webots on masOS
- Howard Ho: Webots on Windows 10/11
- Philip Ni: Webots on GNU/Linux Ubuntu

2) PID Control: Our PID controller is very unstable at higher velocities. Moving more than 0.3 meters per time step generated unstable motor velocities for our PID tuning. In the future, we would like to explore more adaptive controllers.

3) A Algorithm Efficiency:* We had to restrict our search area to within 5 meters of the quadrotor's initial position. Otherwise, it took an unreasonable amount of time to generate the trajectories with our A* implementation. In the future, we would like to optimize our free space exploration to improve the A* performance. We found A* to work best for our purposes out of all the other algorithms.

IX. Conclusion

The culmination of this project marks a significant advancement in the domain of autonomous navigation for quadrotor drones, leveraging the computational robustness of the A* pathfinding algorithm. The simulations carried out within a meticulously crafted virtual environment (Webots) have unequivocally demonstrated the algorithm's prowess in charting efficient trajectories amidst an array of obstacles. By meticulously comparing the A* algorithm against other navigational strategies such as Dijkstra, RRT*, and D*, it was evidenced that A* not only ensures optimality in pathfinding but does so with remarkable speed, making it a superior choice for real-time applications.

A critical aspect of this project's success hinged upon the precision tuning of the PID controllers, which are instrumental in maintaining the drone's stability during flight. The fine-tuning of the proportional, integral, and derivative gains was performed iteratively, striking a delicate balance between the drone's responsiveness and aerodynamic steadiness. This meticulous calibration process underscored the nuanced interplay between algorithmic path planning and the physical control of drones, enlightening us on the multifaceted nature of autonomous systems.

The project, while meeting its core objectives, unfolds a spectrum of possibilities for future enhancement. The adaptability of the A* algorithm could be tested further in dynamic environments where obstacles are not static, and conditions are unpredictable. Additionally, the integration of more sophisticated sensors and real-world testing could bridge the gap between simulation and practical deployment.

The implications of this study extend beyond the academic exploration of an algorithm's efficiency; they touch the realms of practical applications that could revolutionize industries. From search and rescue missions in treacherous terrains to precision agriculture and rapid delivery services, the applications are boundless. The competency of A* in handling complex navigational challenges promises a future where drones can autonomously and reliably be deployed in critical scenarios.

In reflection, this project has not only contributed a substantial body of knowledge for all of us but has also been a conduit for learning and understanding the intricacies involved in the orchestration of algorithms and machinery. It serves as a testament to the synergy between theoretical algorithms and their

practical implementation, a synergy that is the cornerstone of advancements in robotics and autonomous systems.

X. Division of Labor

We propose the following responsibilities for the group members:

- **Literature Review** - [Philip; Shreyas] Research existing solutions and algorithms.
- **Simulation Environment** - [Philip; Howard] Design and set up the simulation field with obstacles and livestock.
- **Basic Drone Navigation** - [Philip; Howard] Establish rudimentary drone controls within the simulation.
- **Object Recognition** - [Shreyas; Howard] Implement and test path search and priority search algorithms.
- **Motion Planning** - [Philip; Shreyas; Howard] Develop, implement, and integrate motion planning algorithms.
- **Testing and Debugging** - [Howard; Shreyas; Philip] Verify the functionalities and fine-tune the system.
- **Final Report and Presentation** - [Shreyas; Philip; Howard] Compile findings and present the project.

XI. Appendices

A. Appendix - Contributions of Shreyas Chigurupati

Shreyas Chigurupati conducted a comprehensive analysis of four advanced path planning algorithms- Dijkstra, A*, D* and RRT* -to lay groundwork for their integration into on our drone-based livestock monitoring system.

B. Appendix - Contributions of Howard Ho

Howard Ho set up the Webots environment in preparation for simulating the drone navigating the farm. Elements include scripts using the Webots supervisor node to handle randomized object placement and free space analysis, and collision geometries placed around the 3D models so the robots and obstacles will interact during the simulation.

C. Appendix - Contributions of Philip Ni

Philip Ni is responsible for the Webots simulation of the quadrotor controller. He implemented the PID controller for a quadrotor following a circular trajectory, and ran the simluation using Webots. The Python code is presented below.

```
from controller import Robot, Motor, GPS, LED,
    InertialUnit, Gyro, Compass
from simple_pid import PID
import numpy as np

class Quadcopter:
    def __init__(self):
        self.robot = Robot()
        # Setup MOTOR for Quadcopter robot
        self.motor_names = ['m1_motor','m2_motor','
    m3_motor','m4_motor']
        self.motors = []
        for motor_names in self.motor_names:
            motor = self.robot.getDevice(motor_names
    )
            motor.setPosition(float('inf'))
            self.motors.append(motor)
        # Setup GPS for Quadcopter robot
        self.gps = GPS("gps")
        self.gps.enable(8)
        # Setup IMU for Quadcopter robot
        self.imu = InertialUnit("inertial unit")
        self.imu.enable(8)
        # Setup GYRO for Quadcopter robot
        self.gyro = Gyro("gyro")
        self.gyro.enable(8)
        # Setup function PID (parameters Kp,Ki,Kd
    not fully optimized yet)
        self.throttlePID = PID(19, 5.3 ,15, setpoint
    =0.1)
        self.pitchPID = PID(4.97, 0.013, 4.57,
    setpoint=0.1)
        self.rollPID = PID(4.92, 0.013, 4.57,
    setpoint=0.1)
        self.yawPID = PID(3.2, 0.013, 2.1, 0.7)
        # create current pose(Start point)
        self.current_pose = [0,0,0]

        self.pointer = 0
        self.target = [0,0,0]
        self.waypoint = self.createTrajectory()

        open('data.txt', 'w')
        return

    def getMotorAll(self): # Get and enable 4 motor
    of Quadcopter robot
        m1_motor = self.robot.getDevice('m1_motor')
        m2_motor = self.robot.getDevice('m1_motor')
        m3_motor = self.robot.getDevice('m1_motor')
        m4_motor = self.robot.getDevice('m1_motor')
        return [m1_motor, m2_motor, m3_motor,
    m4_motor]

    def motorsSpeed(self,velocity): # Set/update
    velocity for 4 motor of Quadcopter robot
        motor = self.getMotorAll()
        for i in range(0,4):
            motor[i].setVelocity(velocity[i])
        return

    def findWaypoint(self):
        # waypoint = [targetX, targetY,
    target_altitude] calculated according to the
    trajectory generation function
        return [[0.06, 0.09, 0.1]]

    def updateTheseCurren(self):
        # Orientation
        self.roll = self.imu.getRollPitchYaw()[0]
        self.pitch = self.imu.getRollPitchYaw()[1]
        self.yaw = self.imu.getRollPitchYaw()[2]

        # Acceleration  Velocities
        self.roll_acceleration = self.gyro.getValues
    ()[0]
        self.pitch_acceleration = self.gyro.
    getValues()[1]

        # Position
        self.xGPS = self.gps.getValues()[0]
        self.yGPS = self.gps.getValues()[1]
        self.zGPS = self.gps.getValues()[2]

        return

    def clamp(self,value, value_min, value_max):
        return min(max(value, value_min), value_max)

    def error(self,des,now_):
        return abs(des-now_)

    def find_angle(self,target_position,current_pose
    ):
```

```
        # a = 1 if target_position[0]>=0 else -1
        # b = 1 if target_position[0]>=0 else -1
        angle_left = (np.arctan2(target_position[0], target_position[1]) )
        angle_left = (angle_left+np.pi)%(np.pi)
        return [0,0]
        return [angle_left,angle_left/abs(angle_left)]

    def createTrajectory(self):
        trajectory = []
        number_of_reference_points = 50
        theta = np.linspace(0, 2*np.pi, number_of_reference_points)
        # the radius of the circle
        r = np.sqrt(0.05)
        # compute x1 and x2
        x1 = r*np.cos(theta)
        x2 = r*np.sin(theta)

        for i in range(0,number_of_reference_points):
            trajectory.append([x1[i],x2[i],0.1])
        return trajectory

    def getTarget(self):
        self.updateTheseCurren()
        target_yaw = self.find_angle(self.target, self.current_pose)
        print(self.yaw,target_yaw[0])
        if(self.error(target_yaw[0],self.yaw)<0.01):
            print("true")
        if(self.error(self.target[0],self.xGPS) <0.007 and self.error(self.target[1],self.yGPS) <0.007 and self.error(self.target[2],self.zGPS) <0.007 and self.error(target_yaw[0],self.yaw) <0.007):
            print("pose: ",self.pointer)
            self.current_pose = self.waypoint[self.pointer-1]
            self.pointer += 1
            if self.pointer >= len(self.waypoint):
                self.pointer = len(self.waypoint) - 1
        self.target = self.waypoint[self.pointer]
        return

    def findSpeedMotor(self):# target is a waypoint
        # Update target for PID function
        Quadcopter.getTarget()
        self.rollPID.setpoint = self.target[1]
        self.pitchPID.setpoint = self.target[0]
        self.throttlePID.setpoint = self.target[2]
        target_yaw = self.find_angle(self.target, self.current_pose)
        self.yawPID.setpoint = target_yaw[0]

        # Get these curren
        self.updateTheseCurren()

        vertical_input = round(self.throttlePID(round(self.zGPS,3)),2)
        roll_input = 30*self.clamp(self.roll, -1, 1) + self.roll_acceleration  + self.rollPID(self.yGPS)
        pitch_input =  -15*self.clamp(self.pitch, -1, 1) - self.pitch_acceleration + self.pitchPID(self.xGPS)
        yaw_input = -self.yawPID((self.yaw))


        # Calc velocity for 4 motor of robot
        m1_input = round(54 + vertical_input + roll_input - pitch_input - yaw_input,3)
        m2_input = round(54 + vertical_input + roll_input + pitch_input + yaw_input,3)
        m3_input = round(54 + vertical_input - roll_input + pitch_input - yaw_input,3)
        m4_input = round(54 + vertical_input - roll_input - pitch_input + yaw_input,3)

        # Check min/max of velocity
        m1_input = self.clamp(m1_input,-600,600)
        m2_input = self.clamp(m2_input,-600,600)
        m3_input = self.clamp(m3_input,-600,600)
        m4_input = self.clamp(m4_input,-600,600)

        # Update Current pose
        return [-m1_input,m2_input,-m3_input,m4_input]

    def motorsSpeed(self,velocity):
        for i in range(0,4):
            self.motors[i].setVelocity(velocity[i])
        return

    def run(self):
        motor_speed = self.findSpeedMotor()
        self.motorsSpeed(motor_speed)
        with open('data.txt', 'a+') as f:
            f.write(str(self.xGPS) + ',' + str(self.yGPS) + ',' +  str(self.zGPS) + ',' +  str(self.yaw) + ',' + str(self.robot.getTime()) +'\n')

        return


if __name__ == "__main__":
    Quadcopter = Quadcopter() # Create robot

    Quadcopter.createTrajectory() # Create Trajectory for robot

    timestep = int(Quadcopter.robot.getBasicTimeStep())

    while Quadcopter.robot.step(timestep) != -1: # Run the script
        Quadcopter.run()
```

Listing 5. Quadrotor Controller

REFERENCES

[1] Smith, J., Brown, T., & Jones, R. (2020). "Automated Livestock Monitoring: A Review," IEEE Transactions on Agriculture, 15(2), 790-799.

[2] Johnson, K., & Roberts, P. (2018). "Challenges in GPS Collar Designs: A Case Study," Journal of Livestock Technology, 7(3), 67-77.

[3] Li, X., Wang, M., & Zhao, Q. (2019). "UAV-based Monitoring Systems for Agricultural Applications: A Review," Journal of Unmanned Systems Technology, 11(1), 15-27.

[4] Martin, T., & Smith, L. (2020). "Drones in Agriculture: A Comprehensive Review," Journal of Smart Agriculture, 4(2), 210-233.

[5] Lee, S., & Kim, H. (2021). "Comparative Analysis of Motion Planning Algorithms in Autonomous Vehicles," IEEE Transactions on Intelligent Transportation Systems, 22(1), 139-149.

[6] Gupta, A., & Sharma, S. (2017). "A Comparative Study of Pathfinding Algorithms for Robotic Applications," International Journal of Advanced Robotic Systems, 14(4), 1729881417716016.

[7] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). "You Only Look Once: Unified, Real-Time Object Detection," arXiv preprint arXiv:1506.02640.

[8] Ayers, Bradley. "Livestock Drone Applications." Interview by Joseph Gibson. Worcester Polytechnic Institute, 03 September 2023.

[9] N. A. AlQahtani, B. J. Emran and H. Najjaran, "Adaptive motion planning for terrain following quadrotors," 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Banff, AB, Canada, 2017, pp. 2625-2630, doi: 10.1109/SMC.2017.8123021.

[10] Flickinger, D. M. (2023, August 28). Course Introduction [PowerPoint Slides]. Canvas. https://canvas.wpi.edu/courses/52697/pages/course-introduction
[11] Flickinger, D. M. (2023, August 28). Graph Search Algorithms [PowerPoint Slides]. Canvas. https://canvas.wpi.edu/courses/52697/pages/graph-search-algorithms
[12] Steven Macenski, et al., Robot Operating System 2: Design, architecture, and uses in the wild.Sci. Robot.7,eabm6074(2022).DOI:10.1126/scirobotics.abm6074
[13] Van Rossum, G., & Drake, F. L. (2009). Python 3 Reference Manual. Scotts Valley, CA: CreateSpace.
[14] "Gazebo". Gazebo Simulator. Archived from the original on 2018-01-16. Retrieved 2019-03-24. https://web.archive.org/web/20180116081153/http://www.gazebosim.org/
[15] Webots Reference Manual. Release R2023b. Cyberbotics. Switzerland. https:www.cyberbotics.comdocreferenceindex