Stack Inspection

Team Members

Chandler Beiler: cbeiler@ncsu.edu
Anupam Mijar: aamijar@ncsu.edu
Scott Burnett: srburne2@ncsu.edu
Roshan Raju: rrraju@ncsu.edu
Sreedhar Ramesh: sramesh6@ncsu.edu

Table of Contents

Abstract

This document presents the outcomes of a project aimed at developing an effective object detection approach for the autonomous inspection of vertical surfaces, particularly power stacks, in various buildings. We explore related studies that employ a similar control structure, as well as businesses that develop commercial products for similar applications. To achieve autonomous object tracking for curved walls, we utilized a 360-degree Lidar sensor that detects the closest point to the drone. The drone controller then employs this point to track while moving horizontally/vertically along any surface. Our final results demonstrate a physical SAM4 drone executing our python controller script that implements the closest point algorithm utilizing the Stamtec S2Lidar sensor. The findings from our real-world demonstration reveal that our program successfully maintained a user-specified distance with a perpendicular yaw from different angled walls while taking user-input to alter horizontal and vertical movement.

Introduction

Our project aimed to develop a drone payload that would leverage range detection sensors to map and detect nearby surfaces of vertical walls, using a drone-control algorithm to maintain a user-specified distance from the wall. We opted to use a Lidar-based solution to achieve range detection, feeding this data to a closest point algorithm to determine the drone's closest point of approach to the wall. The closest point algorithm computed the velocity of the drone in the x, y, and z directions, as well as the drone's yaw, which directed the front of the drone towards the front of the wall.

This document provides an outline of the specific approaches we undertook to achieve our objectives, as well as a detailed discussion of the results of our experiments and tests. We will present the technical details of our payload design, including the sensors we utilized, as well as the software and algorithms that enabled the drone to operate autonomously. Furthermore, we will conduct an in-depth analysis of our experimental results, including a comprehensive discussion of any challenges we encountered and our methods for overcoming them.

Related Work

Drone technology has been rapidly expanding and has found several business applications. One of the key benefits of drones is their ability to access hard-to-reach areas and minimize safety risks for inspectors. They can also perform inspections faster than people, making them an attractive option for businesses. Some of the popular applications of drone inspections include buildings, smokestacks and chimneys, roofs, solar and windmill farms, pipelines and gas lines, cell towers, antennas, and nuclear power plants, as well as confined spaces like tanks. Businesses such as DSLRPros, Dronegenuity, and Intertek provide drone inspection services using DJI technology stack, and many of these businesses also offer drone surveying and mapping services.

Other 592 & 492 Teams have also explored various applications of drone technology. One such team developed a drone on a leash, where the drone hovers on a string above the user like a flying dog. It uses the drone set attitude function with a PID controller. Another team explored the first-person shooter (FPS) - first-person view (FPV) drone, which allows the user to fly the drone around like a plane in a video game, with a video feed and WASD inputs.

Overall, drone technology has a wide range of potential applications in various industries, including but not limited to inspection services. As the technology continues to improve, we can expect to see even more innovative use cases in the future utilizing object detection.

Approach

Real Life Lidar Module

The RPLIDAR C++ SDK is a set of C++ libraries that allow developers to interface with the RPLIDAR series of sensors manufactured by RoboPeak. These libraries provide an easy-to-use interface for controlling and receiving data from the RPLIDAR sensors.

Using the RPLIDAR C++ SDK, developers can easily access the functionality of the RPLIDAR sensors without having to worry about low-level details of the sensor communication protocol. The SDK provides a set of high-level functions for controlling the sensor and a set of data structures for representing the sensor data.
To use the RPLIDAR C++ SDK in Python, we used the Boost.Python C++ library to create Python bindings for the SDK. This allowed us to use the RPLIDAR C++ functions directly in Python code, without having to use C++ for our main application. Boost.Python is a library that allows Python code to call C++ functions and vice versa, making it easy to integrate C++ libraries into Python projects.

To create the Python bindings, we first defined a set of wrapper functions in C++ that called the RPLIDAR C++ functions. Specifically we created functions to start(), getScan(), and stop() the RPLIDAR. We then used Boost.Python to create Python functions that called these wrapper functions. This allowed us to expose the RPLIDAR functionality to Python in a way that was easy to use and understand.

Using Boost.Python to create Python bindings for the RPLIDAR C++ SDK allowed us to easily integrate the RPLIDAR sensors into Python projects, making it easier to develop robotics applications that use these sensors. This also enabled real-time communication between our Raspberry Pi and Lidar.
By leveraging the power of both C++ and Python, we were able to create a robust and flexible system for controlling and receiving data from RPLIDAR sensors.

User Input
The code uses the sshkeyboard library to listen for keypress events. When a key is pressed, the key_on_press function is called, which modifies global variables representing the roll, pitch, yaw, and throttle values of the drone.

The run_simulation function is the main loop of the program, and it is responsible for updating the drone's position and displaying LIDAR data. Inside this loop, the get_target_drone_roll_pitch_yaw_thrust_pid function is called from the Drone_Controller class to calculate the desired roll, pitch, yaw, and throttle values based on the input from the LIDAR sensor.

The key_press_thread function runs in a separate thread and listens for keyboard inputs. When a key is pressed, the corresponding function in the key_on_press function is called to update the global variables representing the drone's orientation and throttle values. These values are then used in the run_simulation function to update the drone's position and orientation.

It's important to note that each key press corresponds to a specific modification in the global variables representing the drone's orientation and throttle values. For example, pressing the 'w' key increases the drone's throttle value, while pressing the 'a' key decreases the roll value and turns the drone to the left. Similarly, pressing the 's' key decreases the throttle value, while pressing the 'd' key increases the roll value and turns the drone to the right. To control the yaw of the drone, pressing ‘e’ increases the yaw value while pressing ‘q’ decreases the yaw value. Pressing the '[' key decreases the throttle value, while pressing the ']' key increases the roll value and turns the drone to the right. If z is pressed, the throttle, yaw, pitch, roll variables are all set back to zero. By understanding how each key press affects the drone's movement, users can easily control the drone's motion and adjust its position and orientation as needed.

Controller

The drone's behavior is controlled through a PID (Proportional-Integral-Derivative) controller, which takes the input from the drone's LIDAR (Light Detection and Ranging) sensor and outputs the desired roll, pitch, yaw, and throttle values.

The script imports several classes from other Python modules, including Drone_Class, Drone_Controller, Lidar_and_Wall_Simulator, and GUI. It also reads in the contents of a JSON configuration file, config.json, which contains information about how the simulation should be run.

The main function, run_simulation(), is responsible for running the simulation. It takes in several arguments, including a boolean use_gui, which indicates whether or not to display a GUI window, a drone_inst object representing the drone itself, a drone_controller_inst object representing the PID controller for the drone, a lidar_and_wall_sim_inst object representing the LIDAR and wall simulator, a list of walls for the simulator, and an optional GUI_inst object representing the GUI window.

The run_simulation() function contains a while loop that runs continuously until the run_program variable is set to False. Within the loop, the function first waits for a specified amount of time (timestep) and updates the drone's position based on its velocity and the elapsed time. It then reads in the new LIDAR readings and calculates the closest point to the drone. This information is then used by the PID controller to determine the desired roll, pitch, yaw, and throttle values for the drone.

In addition to controlling the drone's pitch, roll, yaw, and throttle, the drone is also using a PID controller to maintain a target distance from the wall it is following. The PID controller takes in the distance from the wall and calculates an error term based on the difference between the desired distance and the actual distance. The controller then adjusts the drone's forward velocity to maintain the desired distance from the wall.

This functionality can be seen in the get_target_drone_roll_pitch_yaw_thrust_pid method of the Drone_Controller class, which takes in the distance from the closest point on the wall and returns the target roll, pitch, yaw, and thrust values for the drone to follow the wall while maintaining the desired distance.

The function then adds any control values input by the user through the keyboard to the roll, pitch, yaw, and throttle values. It also sets upper and lower bounds for these values to ensure that they are within a reasonable range.

Finally, the function passes the roll, pitch, yaw, and throttle values to the drone object, which updates its behavior accordingly. If a GUI window is being used, the GUI is also updated with the new LIDAR readings and the drone's position.

The key_on_press() function handles any keypress events and updates the control values for the drone accordingly. The function takes in a single argument, event, which is a string

representing the key that was pressed. Depending on the key pressed, the function updates the pitch_ctrl, roll_ctrl, yaw_ctrl, or throttle_ctrl variables. If the ‘f’ key is pressed, then the drone will go in wall follow mode and orient itself automatically to face the wall.

The key_press_thread() function starts a new thread that listens for keypress events and calls the key_on_press() function whenever a key is pressed. This allows the simulation to continue running while also enabling user control of the drone through the keyboard.

Wrapper class around the MAVLink commands

The Sam4_Drone class is a subclass of Drone that represents a real-life drone. It inherits all the common methods and properties of Drone.

The class initializes a Drone_Realistic_Physics_Class object, which provides realistic physics simulation of a drone in the air. It also takes a target altitude as an argument to perform the takeoff. The takeoff method in this class overrides the takeoff method in Drone class and sets the drone's mode to TAKING_OFF, performs the takeoff by calling the takeoff method of the Drone_Realistic_Physics_Class object, and sets the drone's mode to KEYBOARD.

The set_attitude_setpoint method in this class overrides the same method in Drone and sets an attitude setpoint to the drone. This method takes target_roll, target_pitch, target_yaw, and hover_thrust as parameters, representing the desired roll, pitch, yaw, and thrust value, respectively. It also takes yaw_control_mode as an optional parameter to control the drone's yaw. This method sets the attitude setpoint to the drone by calling the set_attitude method of the Drone_Realistic_Physics_Class object if use_set_attitude is True; otherwise, it calls the set_yaw and set_velocity_body methods to set the drone's yaw and velocity.

GUI & Lidar Simulation
The overall function of this script is that it reads in wall positions as input, and simulates LIDAR readings to detect the distance from the drone to the walls.

The script defines three classes, namely LidarReading, Wall, and Lidar_and_Wall_Simulator. The LidarReading class defines a reading from the LIDAR sensor, with a particular angle and distance. The Wall class defines a wall in the environment, with start and end points in meters. The Lidar_and_Wall_Simulator class is the main class that simulates the LIDAR sensor and walls in the environment.

The script also defines a function called read_config that reads in a configuration file in JSON format. The configuration file contains a boolean value that determines whether to use a real LIDAR module or not. If use_real_lidar is True, the script imports the py_rplidar_sdk.s2lidar module, which is used to communicate with a real LIDAR module via serial communication.

The LidarReading class has a constructor that initializes the angle and distance of a LIDAR reading, as well as the roll and pitch angles of the drone. It also has a method called

update_relative_xy_distance that converts a LIDAR reading at a given angle and distance to a change in x and y coordinates relative to the drone's position.

The Wall class has a constructor that initializes the start and end points of a wall in meters, as well as the relative start and end points of the wall with respect to the drone's position. It also has a method called calculate_relative_walls_to_drone that rotates and translates the wall points to the drone's coordinate system.

The Lidar_and_Wall_Simulator class has a constructor that initializes the walls and the standard deviation of the LIDAR noise. It also has a method called read_new_lidar_readings_angle_deg_dist_m that simulates LIDAR readings at angles from 0 to 360 degrees, with a step size of self.lidar_angle_step_degrees. For each angle, the simulator calculates the distance to the closest wall, adds noise with a standard deviation of self.lidar_noise_meters_standard_dev, and returns an array of tuples containing the angle and distance values for the LIDAR readings.

Results and Analysis

The drone project aimed to develop a system that could follow walls in different modes. The team tested the drone in two different modes: keyboard input and wall follow mode. In the keyboard input mode, the user controlled the drone's speed and direction using the keys. The team found that the drone was responsive to the velocity commands, and the user could control the drone's movement easily. The wall follow mode was designed to allow the drone to autonomously follow a wall while maintaining a constant distance from it. The team successfully demonstrated that the drone could follow the wall in the wall follow mode without any additional user input, even through corners.

During the initial flight sessions, the team faced some challenges where the drone did not respond correctly to velocity commands. However, they were able to troubleshoot the issues over three flight sessions. The team worked for approximately 5 hours each session to identify and address the problem. Eventually, the team was able to get the drone to autonomously fly around the entire building successfully.

Despite the drone's success, the team's analysis revealed that their PID controller could use further tuning. During one of the flights, the drone overshot a corner by 0.3m, causing the propellers to hit the wall. The team hypothesized that this issue could be solved by increasing the derivative component of the PID controller from 0 to a higher value. By doing so, we hypothesize that it will improve the drone's ability to make smooth and precise turns.

Conclusion

In conclusion, the successful development of a drone that can follow walls using keyboard input and wall follow mode is a testament to the team's hard work and dedication. Despite encountering challenges during the flight testing phase, the team was able to overcome them and eventually autonomously fly the drone around an entire building. The team's analysis revealed that their PID controller could use more tuning, and they suggested increasing the derivative component of the controller to solve an overshoot problem. Overall, this project demonstrates the potential of drones to assist in various applications such as surveillance, monitoring, and inspection. It also highlights the importance of continued research and development to improve drone technology and expand their capabilities.

One important technology that was utilized in the successful implementation of the drone's wall-following capability was the LIDAR sensor. This advanced technology provided the drone with the ability to accurately measure the distance between itself and the walls, enabling it to maintain a consistent distance from the walls and maneuver through corners without crashing. The use of LIDAR technology is a testament to the importance of advanced sensors in the development of autonomous systems, and the successful integration of this technology in the drone's design is a significant accomplishment for the project team. Overall, the successful implementation of the drone's wall-following capability demonstrates the team's strong technical skills and their ability to troubleshoot and solve complex problems. The use of LIDAR and other advanced technologies paves the way for the development of even more sophisticated autonomous systems in the future.

Appendices

A.1 Slamtec RPLIDAR Public SDK for C++
A.2 Drone on a leash
A.3 FPS-FPV
A.4 Our py_rplidar_sdk Github (also linked as a submodule in our main Github)
A.5 Our Stack Inspector GitHub