Technical Manual

ALVN

Autonomous LiDAR and Vision based Navigator

By
Ciaran Maye
C00253212

Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

Table of Contents

App.py

```
import os
from time import sleep
from models.alvn import ALVN
import logging
import sys


def print_menu():
    os.system("clear")

    print("ALVN Menu")
    print("==========")
    print("Select an option:")
    print("1. Follow a path")
    print("2. Avoid obstacles")
    print("3. Follow path and avoid obstacles")
    print("4. Reset hardware")
    print("5. Quit")
    print()


def get_speed():
    while True:
        try:
            speed = int(input("Enter speed (0-100): "))
```

```
            if 0 <= speed <= 100:
                return speed
            else:
                print("Invalid input. Speed must be between 0 and 100.")
        except ValueError:
            print("Invalid input. Please enter a number between 0 and 100.")

def get_obstacle_avoider_type():
    while True:
        try:
            print("1. Polar")
            print("2. VFH (Not Functional)")
            type = int(input("Enter obstacle avoider type (1-2): "))
            if 1 <= type <= 2:
                return type
            else:
                print("Invalid input. Type must be between 1 and 2.")
        except ValueError:
            print("Invalid input. Please enter a number between 1 and 2.")

def reset_hardware(car):
    car.cleanup()
    sleep(1)
    print("Hardware reset completed.")
    sleep(2)


def handle_choice(car, choice):
    if choice == 1:
        car.drive_path(get_speed())
    elif choice == 2:
        type = get_obstacle_avoider_type()
        car.drive_obstacles(get_speed(), type)
    elif choice == 3:
        print("Not implemented yet :(")
    elif choice == 4:
        reset_hardware(car)
    elif choice == 5:
```

```
        print("Exiting...")
        car.cleanup()
        return False
    else:
        print("Invalid choice. Please try again.\n")
        sleep(2)
        return True

    return True


def main():
    logging.info(f"Starting ALVN, system info: {sys.version}")
    sleep(1)

    with ALVN() as car:
        while True:
            print_menu()

            try:
                choice = int(input("Enter your choice (1-5): "))
            except ValueError:
                print("Invalid input. Please enter a number between 1 and 5.\n")
                sleep(2)
                continue

            should_continue = handle_choice(car, choice)
            if not should_continue:
                break


if __name__ == "__main__":
    logging.basicConfig(level=logging.DEBUG)
    main()
```

ALVN.py

```
import logging
import cv2
import SunFounder_PiCar.picar as picar
from concurrent.futures import ThreadPoolExecutor
import queue

from models.path_follower import PathFollower
from models.obstacle_avoider import ObstacleAvoider
from models.VFH_obstacle_avoider import VFHObstacleAvoider

_SHOW_IMAGE = False

PORT_NAME = "/dev/ttyUSB0"


class ALVN(object):
    __SCREEN_WIDTH = 640
    __SCREEN_HEIGHT = 480

    def __init__(self):
        """Initialize the ALVN object."""
        logging.info("Creating a PiCar...")

        picar.setup()

        self.initialize_camera()

        self.initialize_pan_tilt_servos()

        self.back_wheels = self.setup_back_wheels()

        self.front_wheels = self.setup_front_wheels()
        self.executor = ThreadPoolExecutor(max_workers=4)
        self.frame_queue = queue.Queue()
        self.path_follower = PathFollower(self, self.executor)
        self.obstacle_avoider = ObstacleAvoider(self)
```

```
    self.vfh_obstacle_avoider = VFHObstacleAvoider(self)


def initialize_camera(self):
    """Initialize the camera."""
    logging.debug("Setting up camera")
    self.camera = cv2.VideoCapture(-1)
    self.camera.set(3, self.__SCREEN_WIDTH)
    self.camera.set(4, self.__SCREEN_HEIGHT)

def initialize_pan_tilt_servos(self):
    """Initialize the pan and tilt servos."""
    self.pan_servo = picar.Servo.Servo(1)
    self.pan_servo.offset = 5 # calibrate camera pan servo to center
    self.pan_servo.write(90)

    self.tilt_servo = picar.Servo.Servo(2)
    self.tilt_servo.offset = -30  # calibrate camera tilt servo to down
    self.tilt_servo.write(80)

def setup_back_wheels(self):
    """Set up the back wheels."""
    logging.debug("Set up back wheels")
    back_wheels = picar.back_wheels.Back_Wheels()
    back_wheels.calibration()
    back_wheels.cali_ok()
    back_wheels.ready()
    back_wheels.speed = 0
    return back_wheels

def setup_front_wheels(self):
    """Set up the front wheels."""
    logging.debug("Set up front wheels")
    front_wheels = picar.front_wheels.Front_Wheels()
    front_wheels.turning_offset = 0
    front_wheels.turn(90)
    return front_wheels
```

```
def __enter__(self):
    """Entering a with statement."""
    return self

def __exit__(self, _type, value, traceback):
    if traceback is not None:
        logging.error("Exiting with statement with exception %s" % traceback)
        self.cleanup()

def cleanup(self):
    """Reset the hardware."""
    logging.info("Stopping the car, resetting hardware.")
    self.back_wheels.speed = 0
    self.front_wheels.turn(90)
    self.camera.release()
    cv2.destroyAllWindows()

def drive_obstacles(self, speed, type):
    """Drive the car with obstacle avoidance."""
    logging.info(
        "Beginning to drive with obstacle avoidance at speed %s..." % speed
    )
    self.back_wheels.speed = speed
    if type == 1:
        self.obstacle_avoider.drive(speed)
    if type == 2:
        self.vfh_obstacle_avoider.drive(speed)

    if cv2.waitKey(1) & 0xFF == ord("q"):
        self.cleanup()


def drive_path(self, speed):
    logging.info("Beginning to drive with path following at speed %s..." % speed)
    self.back_wheels.speed = speed

    if not self.camera.isOpened():
        raise Exception("Could not open video device")
```

```
        while self.camera.isOpened():
            _, image_lane = self.camera.read()
            self.frame_queue.put(image_lane)

            if not self.frame_queue.empty():
                image_lane = self.frame_queue.get()
                image_lane = self.executor.submit(self.follow_path, image_lane).result()

            show_image("Path Line", image_lane)

            if cv2.waitKey(1) & 0xFF == ord("q"):
                self.cleanup()
                return


    def follow_path(self, image):
        image = self.path_follower.follow_path(image)
        return image


def show_image(title, frame, show=_SHOW_IMAGE):
    if show:
        cv2.imshow(title, frame)
```

path_follower.py

```
import cv2
import numpy as np

class PathFollower:
    def __init__(self, car, executor):
        self.car = car
        self.executor = executor
        self.buffer_size = 3
        self.steering_angles = []
        self.max_steering_delta = 20
        self.scaling_factor = 0.1

    def preprocess_image(self, img):
        hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

        lower_black = np.array([0, 0, 0])
        upper_black = np.array([180, 255, 80])

        mask = cv2.inRange(hsv, lower_black, upper_black)
        return mask


    def detect_lines(self, thresholded_img):
        rho = 1
        theta = np.pi / 180
        threshold = 40
        min_line_length = 20
        max_line_gap = 30

        lines = cv2.HoughLinesP(
            thresholded_img, rho, theta, threshold, np.array([]), min_line_length, max_line_gap
        )

        return lines

    def average_slope_intercept(self, lines):
```

```
    left_lines = []
    right_lines = []

    if lines is None:
        return None, None

    for line in lines:
        for x1, y1, x2, y2 in line:
            if x1 == x2:
                continue

            slope = (y2 - y1) / (x2 - x1)
            intercept = y1 - slope * x1
            if slope < 0:
                left_lines.append((slope, intercept))
            else:
                right_lines.append((slope, intercept))

    left_lane = np.mean(left_lines, axis=0) if left_lines else None
    right_lane = np.mean(right_lines, axis=0) if right_lines else None

    return left_lane, right_lane


def make_line_points(self, y1, y2, line):
    if line is None:
        return None

    slope, intercept = line

    # Check for infinite slopes and handle them
    if np.isinf(slope):
        if np.isinf(intercept):
            return None
        x1 = x2 = int(intercept)
    else:
        x1 = int((y1 - intercept) / (slope + 1e-8 ))
        x2 = int((y2 - intercept) / (slope + 1e-8 ))
```

```
    return (x1, y1), (x2, y2)



def find_line_center(self, left_line, right_line):
    if left_line is None or right_line is None:
        return 0, 0

    (left_x1, left_y1), (left_x2, left_y2) = left_line
    (right_x1, right_y1), (right_x2, right_y2) = right_line

    left_cx = (left_x1 + left_x2) // 2
    left_cy = (left_y1 + left_y2) // 2

    right_cx = (right_x1 + right_x2) // 2
    right_cy = (right_y1 + right_y2) // 2

    cx = (left_cx + right_cx) // 2
    cy = (left_cy + right_cy) // 2

    return cx, cy


def draw_center(self, img, cx, cy):
    cx, cy = self.clip_coordinates(img, cx, cy)
    cv2.circle(img, (cx, cy), 5, (0, 255, 0), -1)

def draw_lines(self, img, left_line, right_line, color=(0, 255, 0), thickness=5):
    left_line = self.clip_line_coordinates(img, left_line)
    right_line = self.clip_line_coordinates(img, right_line)

    if left_line is not None:
        cv2.line(img, left_line[0], left_line[1], color, thickness)
    if right_line is not None:
        cv2.line(img, right_line[0], right_line[1], color, thickness)


def follow_path(self, image):
```

```
        thresholded_img_future = self.executor.submit(self.preprocess_image, image)
        thresholded_img = thresholded_img_future.result()
        lines_future = self.executor.submit(self.detect_lines, thresholded_img)
        lines = lines_future.result()

        left_lane, right_lane = self.average_slope_intercept(lines)

        if left_lane is None and right_lane is None:
            return thresholded_img

        y1 = thresholded_img.shape[0]
        y2 = int(y1 * 0.6)

        left_line = self.make_line_points(y1, y2, left_lane)
        right_line = self.make_line_points(y1, y2, right_lane)

        if left_line is not None and right_line is not None:
            cx, cy = self.find_line_center(left_line, right_line)
            self.draw_center(thresholded_img, cx, cy)
        else:
            cx, cy = 0, 0

        self.draw_lines(thresholded_img, left_line, right_line)

        offset = cx - thresholded_img.shape[1] // 2
        steering_angle = self.calculate_steering_angle(offset)

        if self.steering_angles:
            last_angle = self.steering_angles[-1]
            steering_angle = np.clip(
                steering_angle,
                last_angle - self.max_steering_delta,
                last_angle + self.max_steering_delta,
            )

        self.steering_angles.append(steering_angle)
        if len(self.steering_angles) > self.buffer_size:
            self.steering_angles.pop(0)
```

```
    smoothed_angle = sum(self.steering_angles) / len(self.steering_angles)

    self.car.front_wheels.turn(smoothed_angle)
    print(f"Steering angle: {smoothed_angle:.2f}°")

    return thresholded_img


def clip_coordinates(self, img, cx, cy):
    max_x = img.shape[1] - 1
    max_y = img.shape[0] - 1

    cx = np.clip(cx, 0, max_x)
    cy = np.clip(cy, 0, max_y)

    return cx, cy

def clip_line_coordinates(self, img, line):
    if line is None:
        return None

    (x1, y1), (x2, y2) = line
    x1, y1 = self.clip_coordinates(img, x1, y1)
    x2, y2 = self.clip_coordinates(img, x2, y2)

    return (x1, y1), (x2, y2)


def calculate_steering_angle(self, offset):
    max_angle = 135
    min_angle = 45
    center_angle = 90

    angle = center_angle - offset * self.scaling_factor
    angle = np.clip(angle, min_angle, max_angle)

    return angle
```

obstacle_avoider.py

```
import logging
from rplidar import RPLidar, RPLidarException
from serial.serialutil import PortNotOpenError
import collections
import numpy as np
from time import sleep

PORT_NAME = "/dev/ttyUSB0"
lidar = RPLidar(PORT_NAME)
logging.basicConfig(level=logging.INFO)


class ObstacleAvoider(object):
    def __init__(self, car=None):
        logging.info("Initialising ObstacleAvoider...")
        self.car = car
        self.curr_steering_angle = 90
        self.window_size = 3
        self.angle_buffer = collections.deque(maxlen=self.window_size)
        self.max_steering_change = 30
        self.update_rate = 0.7
        self.speed = 0

    def drive(self, speed):
        """Drive the car forward while avoiding obstacles."""
        logging.info("Driving...")
        num_sectors = 5
        min_angle = 40
        max_angle = 140
        min_distance_threshold = 170
        self.speed = speed

        continue_avoiding = True

        while continue_avoiding:
            try:
```

```
            for measurement in lidar.iter_scans():
                sector_distances = self.get_sector_distances(
                    measurement, num_sectors, min_angle, max_angle
                )
                min_distance_sector = np.argmin(sector_distances)
                target_angle = (
                    min_angle
                    + (min_distance_sector + 0.5)
                    * (max_angle - min_angle)
                    / num_sectors
                )
                self.steer(target_angle)

                front_sector_index = num_sectors // 2
                front_distance = sector_distances[front_sector_index]

                # Check if the closest obstacle is too close and in front
                if front_distance < min_distance_threshold:
                    logging.info("Obstacle too close and in front, performing emergency maneuver")
                    self.emergency_maneuver()
                else:
                    continue_avoiding = False

            self.car.back_wheels.speed(self.speed)

        except RPLidarException as e:
            logging.warning("Incorrect descriptor starting bytes. Restarting the lidar device and trying again...")
            lidar.stop()
            lidar.stop_motor()
            lidar.disconnect()
            sleep(1)
            lidar.connect(PORT_NAME)
            lidar.start_motor()
            continue

        except KeyboardInterrupt:
            print("Keyboard Interrupt, Stopping the car")
            break
```

```
            except PortNotOpenError:
                logging.warning("Port Not Open Error, trying again...")
                lidar.stop()
                lidar.stop_motor()
                lidar.disconnect()
                sleep(1)
                lidar.connect(PORT_NAME)
                lidar.start_motor()
                continue

            except Exception as e:
                print(f"Stopping the car because of an error: {e}")
                break

            finally:
                self.cleanup()

        self.cleanup()

    def steer(self, angle):
        """Steer the car towards the target angle."""
        if self.car is not None:
            inverted_angle = 90 - (
                angle - 90
            )  # Invert the angle with respect to 90 degrees
            angle_diff = inverted_angle - self.curr_steering_angle
            clamped_diff = np.clip(
                angle_diff, -self.max_steering_change, self.max_steering_change
            )
            new_steering_angle = (
                self.curr_steering_angle + clamped_diff * self.update_rate
            )
            self.curr_steering_angle = new_steering_angle
            self.car.front_wheels.turn(self.curr_steering_angle)
            logging.info(
                f"Steering to {self.curr_steering_angle} degrees (angle: {angle}"
            )
```

```
def emergency_maneuver(self):
    """Perform emergency maneuver when an obstacle is too close."""
    logging.info("Emergency maneuver initiated")
    # Stop the car
    self.car.back_wheels.speed = 0
    sleep(1)

    # Reverse the car to a safe distance
    self.car.back_wheels.backward()
    self.car.back_wheels.speed = self.speed
    sleep(2)

    # Stop the car
    self.car.back_wheels.speed = 0
    sleep(1)

    # Continue driving forward
    self.car.back_wheels.forward()
    self.car.back_wheels.speed = self.speed
    logging.info("Emergency maneuver complete, resuming forward movement")


def get_sector_distances(self, measurement, num_sectors, min_angle, max_angle):
    """ Get the distances for each sector from the measurements. """
    sector_size = (max_angle - min_angle) / num_sectors
    sector_distances = np.zeros(num_sectors)
    sector_counts = np.zeros(num_sectors)

    valid_measurements = np.array(
        [m for m in measurement if min_angle <= m[1] <= max_angle and m[0] > 0]
    )

    if len(valid_measurements) > 0:
        angles = valid_measurements[:, 1]
        distances = valid_measurements[:, 2]
        sector_indices = ((angles - min_angle) / sector_size).astype(int)
        sector_indices = np.clip(sector_indices, 0, num_sectors - 1)
```

```
            np.add.at(sector_distances, sector_indices, distances)
            np.add.at(sector_counts, sector_indices, 1)
            sector_distances /= np.maximum(sector_counts, 1)

        return sector_distances

    def cleanup(self):
        """Reset the hardware."""
        logging.info("Stopping the car, resetting hardware.")
        if self.car is not None:
            self.car.back_wheels.speed = 0
            self.car.front_wheels.turn(90)
        lidar.stop()
        lidar.stop_motor()
        lidar.disconnect()

# Helper function to clear terminal, for debugging
def clear():
    print("\033c", end="")
```

VFH_obstacle_avoider.py

```
import logging
from rplidar import RPLidar, RPLidarException
import time
import collections
import numpy as np
from time import sleep
from serial.serialutil import PortNotOpenError

PORT_NAME = "/dev/ttyUSB0"
lidar = RPLidar(PORT_NAME)
logging.basicConfig(level=logging.INFO)

class VFHObstacleAvoider(object):
    def __init__(self, car=None):
        """Initialize VFHObstacleAvoider with a car object."""
        logging.info("Initialising VFHObstacleAvoider...")
        self.car = car
        self.curr_steering_angle = 90
        self.window_size = 5
        self.angle_buffer = collections.deque(maxlen=self.window_size)
        self.max_steering_change = 40
        self.update_rate = 0.6
        self.speed = 0
        self.num_sectors = 5
        self.min_angle = 30
        self.max_angle = 150
        self.min_distance_threshold = 120
        self.cell_resolution = 2
        self.histogram_threshold = 6
        self.vfh_max_angle = 40

    def drive(self, speed):
        """Drive the car using lidar obstacle avoidance."""
        logging.info(f"Beginning to drive with obstacle avoidance at speed {speed}...")
        self.speed = speed
```

```
        while True:
            try:
                continue_avoiding = True
                while continue_avoiding:
                    try:
                        for measurement in lidar.iter_scans(max_buf_meas=5000):
                            # Apply VFH algorithm to calculate the target angle
                            target_angle = self.calculate_vfh_target_angle(measurement)
                            self.steer(target_angle)

                            # Check if the closest obstacle is too close
                            if min(self.get_distances_within_angle_range(measurement, self.min_angle, self.max_angle)) <
self.min_distance_threshold:
                                logging.info("Obstacle too close, stopping the car")
                                break

                            else:
                                continue_avoiding = False
                        self.car.back_wheels.speed = self.speed

                    except RPLidarException as e:
                        if str(e) == "Incorrect descriptor starting bytes":
                            logging.warning("Incorrect descriptor starting bytes. Restarting the lidar device and trying again...")
                            lidar.stop()
                            lidar.stop_motor()
                            lidar.disconnect()
                            sleep(1)
                            lidar.connect()
                            lidar.start_motor()
                        if str(e) == "Check bit not equal to 1":
                            logging.warning("Check bit not equal to 1 Restarting the lidar device and trying again...")
                            lidar.stop()
                            lidar.stop_motor()
                            lidar.disconnect()
                            sleep(1)
                            lidar.connect()
                            lidar.start_motor()
```

```
                else:
                    raise e

            break

        except KeyboardInterrupt:
            print("Keyboard Interrupt, Stopping the car")
            break

        except PortNotOpenError:
            print("Port not open, waiting and trying again")
            time.sleep(2)  # Wait for 2 seconds before trying again

        except Exception as e:
            print(f"Stopping the car because of an error: {e}")
            break

        finally:
            self.cleanup()

def steer(self, angle):
    """Steer the car based on the given angle."""
    if self.car is not None:
        inverted_angle = 90 - (
            angle - 90
        )  # Invert the angle with respect to 90 degrees
        angle_diff = inverted_angle - self.curr_steering_angle
        clamped_diff = np.clip(
            angle_diff, -self.max_steering_change, self.max_steering_change
        )
        new_steering_angle = (
            self.curr_steering_angle + clamped_diff * self.update_rate
        )
        self.curr_steering_angle = new_steering_angle
        self.car.front_wheels.turn(self.curr_steering_angle)
        logging.info(
            f"Steering to {self.curr_steering_angle} degrees (angle: {angle})"
        )
```

```
def calculate_vfh_target_angle(self, measurement):
    """Calculate the target angle using the VFH algorithm."""
    histogram = self.build_histogram(measurement)
    candidate_angles = self.select_candidate_angles(histogram)
    return self.choose_best_angle(candidate_angles)

def build_histogram(self, measurement):
    """Build a histogram based on lidar measurements."""
    histogram = np.zeros(360)

    for m in measurement:
        if m[0] > 0:
            angle = int(m[1])
            distance = m[2]

            cell_size = int(distance / self.cell_resolution)
            for i in range(angle - cell_size, angle + cell_size + 1):
                        index = i % 360
                        histogram[index] += 1

    return histogram

def select_candidate_angles(self, histogram):
    """Select candidate angles from the histogram."""
    candidate_angles = []

    for i in range(self.min_angle, self.max_angle + 1):
        if histogram[i] < self.histogram_threshold:
            candidate_angles.append(i)

    logging.info(f"Candidate angles: {candidate_angles}")
    return candidate_angles

def choose_best_angle(self, candidate_angles):
    """Choose the best angle from the candidate angles."""
    if len(candidate_angles) == 0:
```

```
            return self.curr_steering_angle

        best_angle = candidate_angles[0]
        min_diff = abs(candidate_angles[0] - self.curr_steering_angle)

        for angle in candidate_angles:
            diff = abs(angle - self.curr_steering_angle)
            if diff < min_diff:
                min_diff = diff
                best_angle = angle

        # Limit steering angle change to vfh_max_angle
        angle_diff = best_angle - self.curr_steering_angle
        clamped_diff = np.clip(angle_diff, -self.vfh_max_angle, self.vfh_max_angle)
        best_angle = self.curr_steering_angle + clamped_diff

        return best_angle

    def get_distances_within_angle_range(self, measurement, min_angle, max_angle):
        """Get distances of the obstacles within the specified angle range."""
        distances = []

        for m in measurement:
            if min_angle <= m[1] <= max_angle and m[0] > 0:
                distances.append(m[2])

        if not distances:
            # If the list is empty, add a large value to avoid the min() error
            distances.append(float("inf"))

        return distances


    def cleanup(self):
        """Reset the hardware."""
        logging.info("Stopping the car, resetting hardware.")
        self.car.back_wheels.speed = 0
        self.car.front_wheels.turn(90)
```

```
    lidar.stop()
    lidar.stop_motor()
    lidar.disconnect()

def clear():
    print("\033c", end="")
```