Embedded Systems ENGR-UH 3530

Final Project Report

12/12/2023

Flavia Trotolo FT2063, Omar El Herraoui oe2015, Ahmad Fraij af3954

Introduction

The AI Robot kit is based on the Jetson Nano Developer Kit. Supports facial recognition, object tracking, auto line following or collision avoidance and so on. Support three 18650 batteries (not included), 7800mAh, up to 12.6V voltage output, and stronger motor power. The kit features an onboard S-8254AA + AO4407A lithium battery protection circuit, with anti-overcharge, anti-over-discharge, anti-over-current and short-circuit protection functions. The onboard APW7313 voltage regulator chip can provide a stable 5V voltage to the Jetson Nano. The onboard TB6612FNG dual H-bridge motor driver chip can drive the left and right motors to work. Onboard 0.91-inch 128 × 32 resolution OLED, real-time display of car IP address, memory, power, etc. The onboard ADS1115 AD acquisition chip is convenient for real-time monitoring of battery power. The kit was assembled using the following guidehttps://www.waveshare.com/wiki/JetBot_AI_Kit

AI Robot Kit based on Jetson Nano Developer Kit

Briefly, The autonomous robot should follow a black line while overcoming different problems in a modular Field formed by tiles with different patterns. The floor is white in color and the tiles are on different levels connected with ramps. The end of the field will be marked with a strip of reflective silver tape on the floor. Teams are not allowed to give their robot any advance information about the field as the robot is supposed to recognize the field by itself.

Problem Statement

The goal is to design a robot that can navigate a predetermined path by following lines and making decisions at intersections based on the position of green tape. The robot must be able to recognize and follow the curvature of a black line on white tiles, and it must choose whether to go straight, turn left, or turn right at intersections based on the position of the green tape. If the green tape is on the right side of the junction, the robot should turn right; if it is on the left side, the robot should turn left; and if it is on both sides, the robot should perform an about-turn. If the green tape is only on the opposite side of the intersection, the robot should go straight. The robot must also be able to stop if an obstacle appears in its path or if it reaches a reflective silver strip marking the end of the course.

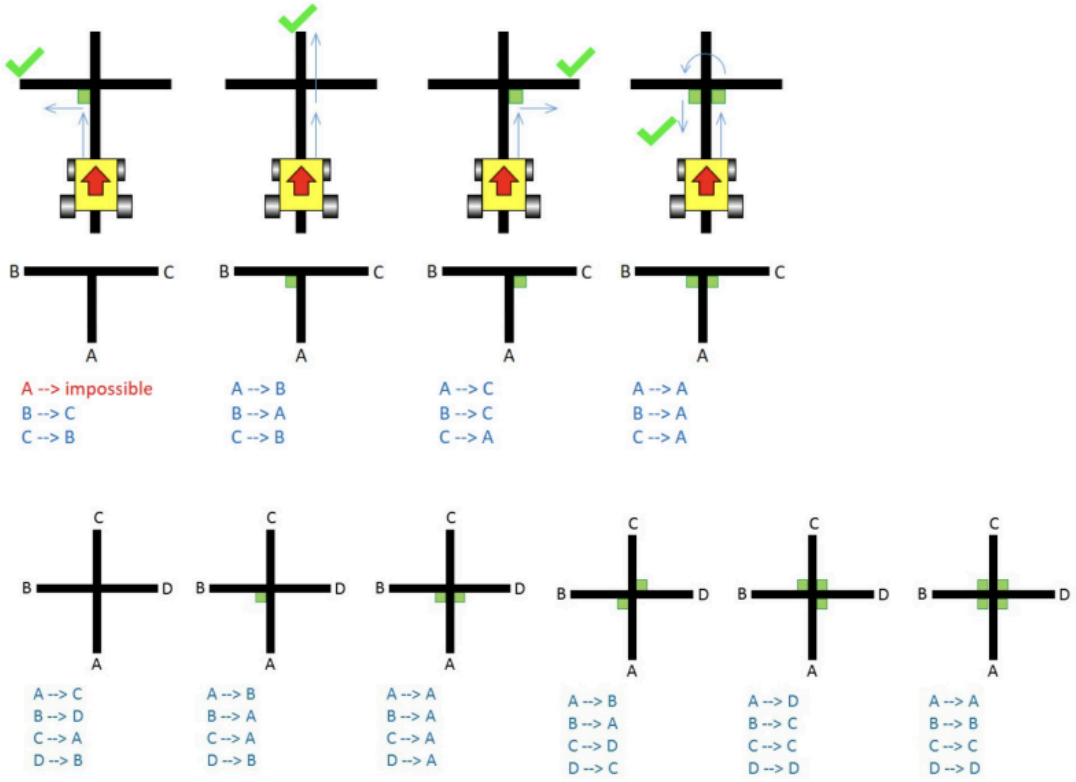

Figure 2: Figure showing examples of the directions to follow for each green marker pattern (From assignment sheet)

Our solution

Our code is designed to enable a robot to follow a black line on the ground while also detecting and responding to a green square along the line's path. It essentially captures frames from the camera, processes them to detect the black line and green square, and adjusts the robot's movement accordingly to follow the line and respond to the green square. Adjustments to parameters like thresholds, contour areas, or movement speeds can be made based on specifics of the ground used. The code is organized in the following parts:

- **Camera and Image Processing Setup**: The code initializes a camera instance and an image widget to display the camera feed. It uses OpenCV (cv2) and NumPy to process the images.
- **Finding the Black Line**: The find_black_object_center() function processes the image captured by the camera to detect the black line.
- **Detecting the Green Square**: The find_green_square() function detects a green square along the path and determines if the green square is on the left or right side of the image.
- **Main Update Function**: The update() function captures the image from the camera and crops it to focus on the center area, calls the black line and green square detection functions, and it controls the robot based on the detected line and green square: if a green square is detected, it stops the robot and instructs it to turn left or right accordingly but if If no green square is detected, it guides the robot to follow the black line by adjusting its movements based on the line's position relative to the cropped image.
- **Control Logic:** The robot's movements are controlled using the Robot class provided by the jetbot library. It uses commands like left(), right(), forward(), or stop() with varying speed values to maneuver the robot based on the line and green square detection.
- **Camera Observation and Display**: The camera's output is linked to the update function using traitlets.dlink and camera.observe. This enables continuous image processing and robot control based on the captured frames.
- **Sleep Time Between Frames**: there's a small delay of 0.01 seconds (time.sleep(0.01)) between frames to control the rate of image processing and robot movement.

We will explain the working of the individual function in detail:

1. Detection of black line

First, we used the following function find_black_object_center(), to locate and identify the center of a black object, presumed to be a line, within an image. Here's a breakdown of each step within the function:

- **Convert the image to grayscale**: The function starts by converting the input image (img) from the BGR color space to a grayscale image using cv2.cvtColor() function. This simplifies subsequent image processing by reducing it to a single channel representing the intensity.
- **Apply a threshold to create a binary image, isolating the black regions**: a thresholding operation is applied to the grayscale image using cv2.threshold(). This operation converts the grayscale image into a binary image where pixels are either white or black based on a specified threshold value. In this case, the threshold value is set to 50. Pixels with intensities below 50 become white (255) and those above become black (0), due to the cv2.THRESH_BINARY_INV flag used, which creates an inverted binary image.
- **Identifies contours in the binary image and finds the largest contour (assumed to be the black line)**: Then, cv2.findContours() is used to identify contours within the binary image. Contours are continuous curves that delineate the boundaries of objects in an image. The function returns a list of contours found in the image. If contours are found (if contours:), it proceeds to find the largest contour within the list of contours found using max(contours, key=cv2.contourArea). This finds the contour with the maximum area.
- **Calculates the center (cx, cy) of the detected contour**: Using cv2.moments() on the largest contour, it calculates various moments of the contour, including its centroid (center). If the calculated centroid moments (M['m00']) are not equal to zero (ensuring the contour has a valid area), it calculates the center (cx, cy) of the contour using the centroid moments. Finally, it returns a tuple (cx, cy) representing the coordinates of the center of the largest contour along with the thresholded image. If no contours are found or if the moments of the largest contour indicate a zero area (M['m00'] == 0), it returns None along with the thresholded image.

2. Detection of green squares on either side of the black line.

Then, the find_green_square() function is designed to detect a green square within an image and determine its position relative to the image's center. Here's an explanation of each step within this function:

- **Convert Image to HSV Color Space**: the function begins by converting the input image (img) from the BGR color space to the HSV (Hue, Saturation, Value) color space using cv2.cvtColor() function. This color space is often used for color-based segmentation and analysis.
- **Define Green Color Range in HSV**: Lower and upper bounds of the green color in the HSV color space are specified using numpy arrays (lower_green and upper_green). These values define a range of green hues, saturations, and values that the function will consider as green. Adjusting these values allows for tuning the green detection based on the specific shade or intensity of green desired.
- **Create a Mask for the Green Color**: using cv2.inRange(), a mask is created by thresholding the HSV image to isolate pixels that fall within the specified green range. The resulting mask will show areas where green color is present in the original image.
- **Find Contours in the Mask**: cv2.findContours() is applied to the mask to identify contours, which are continuous curves that outline regions of interest in the binary image. The contours detected represent the areas in the image that match the specified green color range.
- **Processing Detected Contours**: If contours are found (if contours:), the function proceeds to find the largest contour among the detected contours using max(contours, key=cv2.contourArea). This is assumed to be the contour representing the green square. The function calculates the bounding rectangle (x, y, w, h) of the largest contour and calculates its area (contour_area) using cv2.contourArea().
- **Checking for Valid Green Square**: A minimum area (min_area) threshold is set to filter out small contours that might not represent a significant green square. Adjusting this threshold allows for filtering based on the expected size of the green square.If the area of the largest contour exceeds the minimum area threshold, the function calculates the center (box_center) of the bounding box and the center of the image (image_center). It then determines whether the green square is on the left or right side of the image based on these centers. If the green square is detected and positioned either to the left or right of the image center, the function stops the robot's movement (robot.stop()) and returns either 'left' or 'right' accordingly.
  Return Handling:
- **Return Handling**: If no suitable green square is detected (either due to no contours found or not meeting the area threshold), the function returns None.

3. Update function

The update() function is a handler that processes the image captured by the camera to make decisions for controlling a robot based on the detected black line and green square. Let's break down the key steps within this function:

- **Global Variables**: It starts by declaring global variables (process_images, last_turn_time) that will be used to control the flow of image processing and robot actions.
    - **process_images**: This variable acts as a toggle, controlling the flow of image processing in the program. When set to True, it permits image analysis and processing. However, when set to False, it pauses the image processing operations, altering the behavior of the program based on its state.
    - **cooldown_time**: This variable represents a time duration measured in seconds. It establishes a cooldown period for specific actions in the program. For instance, it could define the minimum time gap required between consecutive turns, regulating the frequency of certain actions within the system.
    - **last_turn_time**: Serving as a timestamp, this variable records the time of the most recent turn action in seconds. It assists in managing the timing between successive turns. By comparing this timestamp with cooldown_time, the program can control the frequency of turn-related actions and prevent rapid or consecutive turns. This allows to achieve smooth turning
- **Processing Image**: Checks if the process_images flag is True. If it's False, the function exits without processing the image. This allows for pausing image processing under certain conditions, likely during a turn or specific action.
- **Image Cropping**: The captured image is cropped to focus on a particular region (crop_size = 175) using the dimensions specified. This cropped image is then processed further.
- **Finding Black Line and Green Square**: The cropped image is passed to the find_black_object_center() function to locate the black line. The result includes the center and a processed image highlighting the detected line. Additionally, it utilizes find_green_square() function to identify the green square's direction and generates a green mask for visual feedback on the processed image.
- **Displaying Processed Image**: the processed image (with line detection and green mask overlay) is displayed using the image_widget to provide visual feedback.
- **Handling Green Square Detection**: If a green square is detected (green_direction == 'left' or 'right'), it initiates a turn by adjusting the robot's motor speeds gradually to perform a smooth turn. During this turn, it sets process_images flag to False, effectively pausing the image processing. After the turn, it stops the robot's movement, updates last_turn_time, and resumes image processing by setting process_images back to True.

- **Following the Black Line**: If no green square is detected, it continues to follow the black line by adjusting the robot's movements based on the detected line's position within the cropped image.
- **Handling No Line Detection:** If no black line is detected, it makes the robot move forward to avoid getting stuck.

4. Design choices:

When designing the solution for the problem, we made specific choices to enhance the robot's ability to track green boxes and execute turns efficiently. These design choices encompassed:

- **Smooth Turning**: Initially, we attempted sharp turns, but the robot detected green boxes prematurely. Two potential design choices emerged:
    - Fix the size of the cropped image to delay green box detection.
    - Opt for smoother turns

  After testing both solutions on the track, we determined that smoother turns better suited our application.

- **Adding Cooldown Time after Each Turn**: Observing that the robot, while turning, continued to detect the green box, resulting in multiple turns, we introduced a brief cooldown period after each turn. This addition ensured that the robot did not continue turning after detecting the same green box multiple times.
- **Stop Processing Captured Images during Turning**: To prevent issues such as the robot following a black line or detecting additional green boxes during a turn, we implemented a solution. A global flag was introduced to control the processing of captured images. This flag was set to true only after completing the turn, and during the turn, it was set to false. This approach helped avoid undesired behaviors during turning.

5. Limitations:

U-turn

We were not able to write the code to detect two adjacent green squares to turn around. However, we tried the following solutions:

1. **If there were more than two valid contours or square which were within a threshold distance we would return 'Two adjacent green squares found' to the update function.**

```
valid_contours = []
  for contour in contours:
```

```
    x, y, w, h = cv2.boundingRect(contour)
    contour_area = cv2.contourArea(contour)
    min_area = 1000  # Adjust based on your requirements

    if contour_area > min_area:
        valid_contours.append((x, y, w, h))

#TURN AROUND
if len(valid_contours) >= 2:
    # Sort contours based on x-coordinate
    valid_contours.sort(key=lambda c: c[0])

    # Check if two green squares are adjacent horizontally
    first_x, _, first_w, _ = valid_contours[0]
    second_x, _, _, _ = valid_contours[1]
    distance_between = second_x - (first_x + first_w)

    if distance_between <= 10:  # Adjust threshold based on your scenario
        robot.stop()
        return 'Two adjacent green squares found'
```

2. **Our second approach was to do a U-turn if the detected contour area is greater than a certain threshold value but this did not work as contour area values for all turns whenever it detected green were constantly fluctuating. Here is the portion of code we tried to add to the detect green box function for this:**

```
if contour_area > min_area:
        # Calculate the center of the bounding box
        box_center = x + w / 2
        If contour_area>10:
            return ‘uturn’, mask

        # Determine if the box is on the left or right side of the black line
        if box_center < black_line_center:
            return 'left', mask
        else:
            return 'right', mask
```

5. Feedback from Camera Examples:

1. Real-time view from the JetBot's camera showing the detection of the black line. The robot's vision system processes this

image to identify the center of the black line, which guides its navigation.

2. Processed image from the JetBot's vision system highlighting the detected green square. This binary mask is used to determine the green square's position relative to the black line, influencing the robot's decision to turn left or right.

6. Completed Code Solution :

Link to Github: https://github.com/oe2015/Jetson_FC

```
import cv2
import numpy as np
from jetbot import Robot, Camera
import ipywidgets.widgets as widgets
from IPython.display import display
import traitlets
from jetbot import bgr8_to_jpeg
import time

# Create robot instance
robot = Robot()
```

```
# Create and display Image widget
image_widget = widgets.Image(format='jpeg', width=300, height=300)
display(image_widget)

# Create camera instance
camera = Camera.instance()

# Function to process the image and return the center of the black object
def find_black_object_center(img):
    # Convert to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Apply threshold to get binary image
    _, thresh = cv2.threshold(gray, 50, 255, cv2.THRESH_BINARY_INV)

    # Find contours
    contours, _ = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

    if contours:
        # Find the largest contour
        largest_contour = max(contours, key=cv2.contourArea)

        # Get the center of the contour
        M = cv2.moments(largest_contour)
        if M['m00'] != 0:
            cx = int(M['m10'] / M['m00'])
            cy = int(M['m01'] / M['m00'])
            return (cx, cy), thresh
    return None, thresh

def find_green_square(img, black_line_center):
    # Convert the image from BGR to HSV color space
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    # Define the range of green color in HSV
    # These values should be adjusted based on the green you want to detect
    lower_green = np.array([35, 100, 25])  # Lower end of the HSV range for green
    upper_green = np.array([80, 255, 255])  # Upper end of the HSV range for green
```

```
    # Create a mask for the green color
    mask = cv2.inRange(hsv, lower_green, upper_green)

    # Find contours in the mask
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    # Assume we're only interested in the largest contour
    if contours:
        largest_contour = max(contours, key=cv2.contourArea)
        x, y, w, h = cv2.boundingRect(largest_contour)
        contour_area = cv2.contourArea(largest_contour)

        # Define a minimum area for the contour to be considered a valid green square
        min_area = 0  # This is an arbitrary value; adjust based on your image

        if contour_area > min_area:
            # Calculate the center of the bounding box
            box_center = x + w / 2

            # Determine if the box is on the left or right side of the black line
            if box_center < black_line_center:
                return 'left', mask
            else:
                return 'right', mask

    return None, mask
# Add a flag to control image processing
process_images = True
cooldown_time = 3  # in seconds
last_turn_time = 0.0

def update(change):
    global process_images,last_turn_time  # Declare the flag as a global variable
    image = change['new']

     # Check if image processing is allowed
    if not process_images:
        return
```

```
    # Crop the image (for example, cropping the center 150x150 pixels)
    height, width = image.shape[:2]
    crop_size = 175
    crop_box = ((width - crop_size) // 2, (height - crop_size) // 2, (width + crop_size) // 2, (height
+ crop_size) // 2)
    cropped_image = image[crop_box[1]:crop_box[3], crop_box[0]:crop_box[2]]

    # Process the cropped image to find the black line and green square
    center, processed_image = find_black_object_center(cropped_image)

    # Ensure that cx has a valid value before using it
    if center is not None:
        cx, cy = center
        green_direction, green_mask = find_green_square(cropped_image, cx)

        # Overlay the green mask on the processed image for visual feedback
        processed_image = cv2.bitwise_or(processed_image, green_mask)
        # Set the flag to pause image processing during the turn


        # Display the processed image
        image_widget.value = cv2.imencode('.jpg', processed_image)[1].tobytes()

        if green_direction == 'left' or green_direction == 'right':
            print(green_direction)
            # Set the desired turn duration and motor speeds
            turn_duration = 0.3  # in seconds
            left_motor_speed = 0.2 if green_direction == 'left' else 0.5
            right_motor_speed = 0.3 if green_direction == 'left' else 0.2

            # Set the flag to pause image processing during the turn
            process_images = False

            # Calculate the time interval for each iteration of the loop
            time_interval = 0.01  # in seconds

            # Calculate the number of iterations based on the time interval
            num_iterations = int(turn_duration / time_interval)

            try:
```

```
            if time.time() - last_turn_time > cooldown_time:
                # Perform a smooth turn using a time-based loop
                for i in range(num_iterations):
                    progress = i / num_iterations
                    current_left_speed = left_motor_speed * (1 - progress)
                    current_right_speed = right_motor_speed * (1 - progress)
                    robot.set_motors(current_left_speed, current_right_speed)
                    time.sleep(time_interval)
                last_turn_time = time.time()

            finally:
                # Stop the robot after the turn is complete
                robot.stop()
                process_images = True



        else:
#             process_images = True
        # Process the image if enough time has passed since the last turn

            # If no green square is detected, follow the line
            if cx < int(crop_size * 0.2):  # Line is to the left (adjusted for cropped image size)
                robot.left(0.2)
            elif cx > int(crop_size * 0.6):  # Line is to the right (adjusted for cropped image size)
                robot.right(0.18)
            else:
                robot.forward(0.17)
    else:
        robot.forward(0.14)

# Link the camera output to the update function
camera_link = traitlets.dlink((camera, 'value'), (image_widget, 'value'), transform=bgr8_to_jpeg)
camera.observe(update, names='value')

# Add a delay between frames (adjust the duration based on your needs)
time.sleep(0.01)
```