1 System Architecture

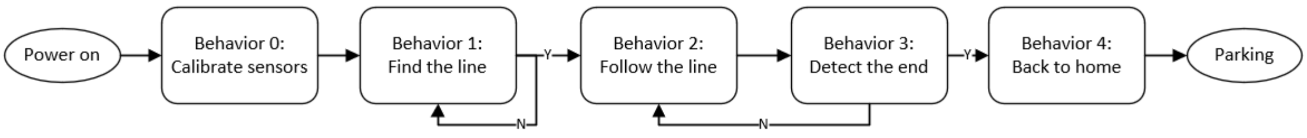
Power on
Behavior 0: Calibrate sensors
Behavior 1: Find the line
Y
N
Behavior 2: Follow the line
Behavior 3: Detect the end
Y
N
Behavior 4: Back to home
Parking

Figure 1: system architecture

Architecture description

The system architecture has been shown in figure 1 and figure 2, the robot has been designed as a finite state machine. Hence the architecture looks like 7 states, and the programme is made according to these 7 states. The initial state is “power on” state, once the power button on the robot car has been turned on, the robot will give a response on its signal light along with the default setting from their factory, then it will run the C code compiled into the Arduino board. That codes make the robot perform 5 basic behaviors to realize the line-following task. Next state after the “power on” state is “calibrate_sensors” state, at this state the inferred sensors will be calibrated according to the color value sampled from the “white area” of the map. The third state is “find_the_line” state, at this state the robot is expected to find the black line on the map via its calibrated IR sensors, the robot will keep going forward until finding the black line. The judgment condition of finding a black line is shown in the orange rhombus named “condition” in the figure 2. Once the sensor reading matches the condition, the robot will switch to the next state “follow_the_line”. The robot is expected to keep its center sensor on the black line at the same time go forward. Once the robot gets the end, it will keep counting the times that the sensor reading reaches the condition “off_line” (see behavior 3 in figure 2) until reach 1000 times. Then the robot will start “back_to_home” state, at this stage, the robot will first turn to the direction towards the original point according to its current orientation. And then it runs a certain distance in a straight line, the distance is calculated from its final coordinates when detecting the end. Finally, the robot stops at the original point and plays a melody via its buzzer, which is “parking” state. The details about the programme design are given as a flowchart in figure 2.

Task decomposition

The whole task in this project is that let a robot car track the black line to the end of the map, and then go back to the original point, it is a description of the functional expectation of the robot’s behavior. That task, or expectation, is given by human abstract language, which is not understandable by the Arduino board hence converting the abstract behavior into C programme is expected. Once the abstract task decomposed into small tasks, it will be straightforward to convert them into small code blocks like the blue, orange and red blocks showed in figure 2. Each smaller task is realized by a specific code block, hence it is easy to find the precise location of the bug when a certain state malfunction. Besides, decomposing a tough issue into several smaller issues is a common method in engineering. It helps people to figure out what the actual work they are going to finish step by step. Once the big issue is decomposed, the difficulty decreased at the same time. And the task decomposition may also benefit task arrangement in a teamwork scenario.

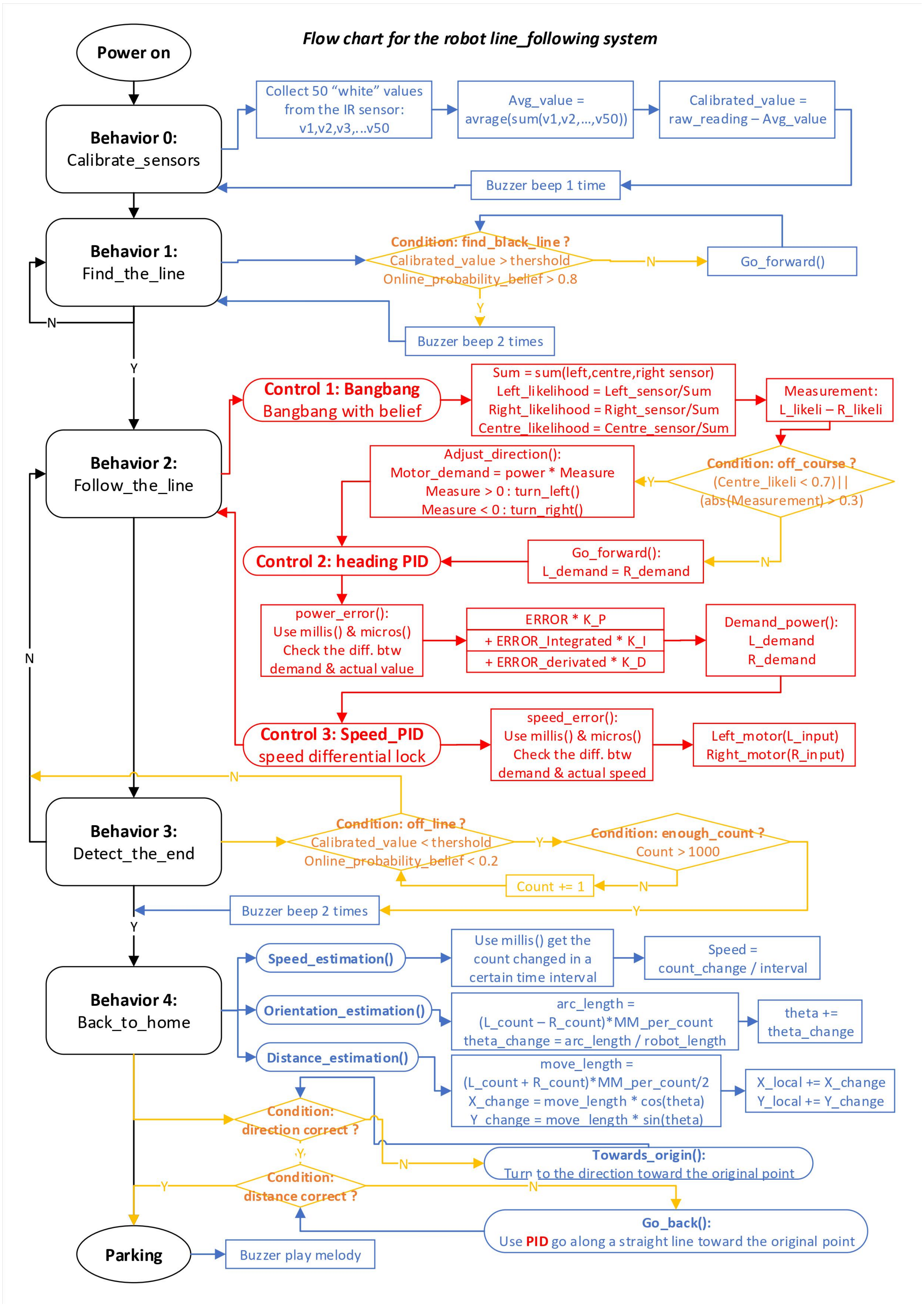
Flow chart for the robot line_following system
Power on
Behavior 0: Calibrate_sensors
Collect 50 "white" values from the IR sensor: v1,v2,v3,...v50
Avg_value = avrage(sum(v1,v2,...,v50))
Calibrated_value = raw_reading – Avg_value
Buzzer beep 1 time
Behavior 1: Find_the_line
Condition: find_black_line ? Calibrated_value > thershold Online_probability_belief > 0.8
N
Go_forward()
Y
Buzzer beep 2 times
N
Y
Behavior 2: Follow_the_line
Control 1: Bangbang Bangbang with belief
Sum = sum(left,centre,right sensor) Left_likelihood = Left_sensor/Sum Right_likelihood = Right_sensor/Sum Centre_likelihood = Centre_sensor/Sum
Measurement: L_likeli – R_likeli
Condition: off_course ? (Centre_likeli < 0.7) || (abs(Measurement) > 0.3)
Y
Adjust_direction(): Motor_demand = power * Measure Measure > 0 : turn_left() Measure < 0 : turn_right()
N
Go_forward(): L_demand = R_demand
Control 2: heading PID
power_error(): Use millis() & micros() Check the diff. btw demand & actual value
ERROR * K_P + ERROR_Integrated * K_I + ERROR_derivated * K_D
Demand_power(): L_demand R_demand
Control 3: Speed_PID speed differential lock
speed_error(): Use millis() & micros() Check the diff. btw demand & actual speed
Left_motor(L_input) Right_motor(R_input)
N
N
Behavior 3: Detect_the_end
Condition: off_line ? Calibrated_value < thershold Online_probability_belief < 0.2
Y
Condition: enough_count ? Count > 1000
N
Count += 1
Y
Buzzer beep 2 times
Y
Behavior 4: Back_to_home
Speed_estimation()
Use millis() get the count changed in a certain time interval
Speed = count_change / interval
Orientation_estimation()
arc_length = (L_count – R_count)*MM_per_count theta_change = arc_length / robot_length
theta += theta_change
Distance_estimation()
move_length = (L_count + R_count)*MM_per_count/2 X_change = move_length * cos(theta) Y_change = move_length * sin(theta)
X_local += X_change Y_local += Y_change
Condition: direction correct ?
N
Towards_origin(): Turn to the direction toward the original point
Y
Condition: distance correct ?
Y
N
Go_back(): Use PID go along a straight line toward the original point
Parking
Buzzer play melody

Figure 2: robot system flowchart

Deliberate system design

The simplest Bang-bang control makes the robot turn or goes straight according to the values from three sensors. And the output for this kind of Bang-bang control system is also quite simple because the final result only has three states which are turn left, turn right, and go straight at a fixed power input. However, this kind of Bang-bang control can be developed with the agent belief. As control 1 in figure 2 shows, once adding agent belief (probability) into the Bang-bang control, the final state of the demanded motor power will become unlimited. Then the robot can easily handle different types of curves like figure 4 shows. And also the turning action will become continuous and smooth.

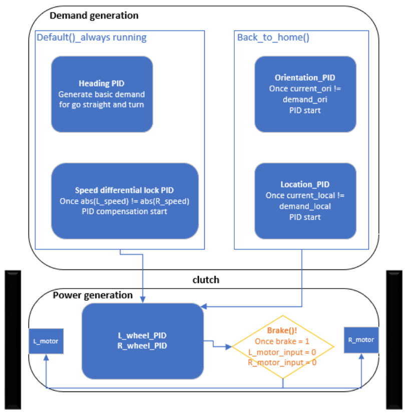

Figure 3: hierarchy clutch motor control system

Another deliberate design is a hierarchy motor control system (see figure 3) inspired by the clutch of the real vehicle. The higher layer uses four PID controllers to calculate a property power input demand for the motor and then the demand will be sent to the lower layer. The lower layer uses two PID controllers with a brake block directly control the motors. This model is investigated to give smoother acceleration progress in as short as possible time, at the same time, use a lower layer as a buffer to protect the mechanical disruption on the transmission gear. According to the practical experience, this system is designed as the clutch structure in this robot. The higher layer is running at the period of 10000 microseconds while the lower is running at the period of 5000 microseconds, once a brake is required, the higher layer block will change the variable "brake" to 1, then the lower layer will give a null input to the motor, which makes the robot performs an emergency brake or parking. Besides, an anti-locked brake can be performed via a null demand given by the higher layer instead of changing the brake variable.

Time-consuming design

PID parameter configure

The three parameters in the PID controller are required a lot of time to be investigated. P is the coefficient for giving an initial demand according to the error, I is for increasing the demand according to the accumulated error, and D is for buffering the abrupt changes in the demand (see control 2 in figure 2). Due to 6 sorts of PID controllers are implemented in this robot system, it is a challenge to work out a proper configuration on so many PID parameters. However, the common method to adjust these parameters is making multiple experiments, which is time-consuming.

Practical suggestion

Debug code block

Arduino programming has an inconvenient problem, some bugs in the programme are invisible unless the programme is running onboard the chip. To improve the debug efficiency, a debug code block could be useful. All critical variables in the programme should be included in the block within the printf() function and make this block a function structure, then attach the function name in the place where the variables are expected to be checked. And those unexpected variables can be hidden via simply comment the printf() line in the debug function. By this method, the onboard debug is much easier with the serial monitor in Arduino software.

2 Challenge & solution

Challenge technology

The most interesting and challenging part of the algorithm could be the kinematic estimation part. It is interesting because using this technology a simple SLAM work could be done without computer vision and Markov model. Meanwhile, it is a challenge because it accumulates the systematic errors which might cause a big distortion at the "back_to_home" state. In this technology, all the estimations related to the motor encoders. By counting the pause changes on the encoders, the speed, direction, and location can be worked out (details in the figure 2 behavior 4). And the trigger method of the pulse counting function is also an interesting technology, which uses the interrupt mechanism.

Behavior 4 in figure 2 shows that the kinematic issues include the speed estimation, the orientation estimation, and the distance estimation. And each estimation has many parameters to be set up, for example, the distance traveled by wheels after one count on encoder pulse change, MM_per_count, which is calculated from the perimeter of the wheel and the counts per wheel rotation. And the dimension of the robot car, robot_length, is also an important parameter. However, the parameters found from the user handbook from the official website are unconvincing due to the manufacturing error in different elements, hence a manual measurement of these parameters is required. And it might be error happening in the measurement process, which brings a systematical error into the back_to_home behavior. This kind of systematic error can be wiped out by calibrating the parameters according to the average value of many times' measurement, or adjust the condition (see the orange block of behavior 3 in figure 2) according to experiment. Both of these methods consume a lot of time to wipe the systematic error.

- Advantage

  Use the kinematic estimation by counting the pulse change on the encoders is a useful and easy way to collect the actual move action of the robot. According to the pulse changes, a probable location change and orientation change can be calculated out. Though it is just an estimation, this method is accurate enough to be implemented conveniently in the short-term moving task. And the utility of the inner interrupt mechanism could make sure as much as possible pulse changes to be recorded.

- Disadvantage

  However, this kind of kinematic estimation can hardly handle the long-term moving task and high speed moving task. Due to the accumulated error, the robot might have a big error after a long work period. And the RAM of the robot is limited, which means it unable to store a long-time count of the pulse change. In the high speed moving task, a robot might suffer friction on the contact area between the wheels and ground, which means the moving distance of a point on the wheel is unequal to the moving distance of a point on the encoder axle. Hence in high-speed situation counting the pulse change on the encoders might be useless.

Solution

There are two basic methods to wipe out the systematic error in the kinematic estimation process. One is measuring and calibrating the physical parameters as accurately as possible, which means decrease the error at the source. Here in this project one of the physical parameters is MM_per_count, it equals the perimeter of the wheel divide by the encoder counts after the wheel making one revolution. To calibrate the default values, attach a small stick on the radial direction toward the wheel center, put the wheel on a white paper then use a pen to record the current location. After rotating the wheel for 10 revolutions, record the reading of the encoder count from the serial monitor in Arduino. Do 3 sort of this action and take the average count, then the count divide by 10 will be a quite accurate MM_per_count value. Use two values separately for the left and right wheel will also decrease the error. Another method is adjusting the judgment condition in the algorithm to perform an equivalent error-free result. The judgment conditions are presented in orange blocks in figure 2, for example, in "back_to_home" state it appears an obvious overmuch turning angular, which can be defused by multiply a coefficient less than 1, or vice versa. However, to find out the proper coefficients for the orientation and distance judgment condition, massive experiments are required.

Extra consideration

The robust of the functions in the whole line_following system is depended on the condition blocks in the system flowchart (figure 2). The sensor reading could have a big difference when facing different environments, which means the judgment condition should have some extent of tolerance to the environment change. For example, in the condition block "find_black_line" not only the sensor reading is treated as a judgment condition, but the agent belief (probability) is also used as an alternative condition, which improves the robustness of the whole robot system. The condition black "off_line" has a similar insurance mechanism, and actually, all the condition blocks in the robot system should have some tolerance towards the environment change (see figure 4).

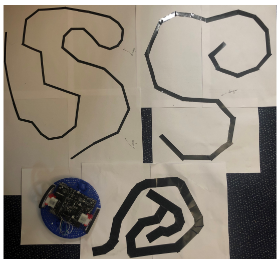

Figure 4: challenge map for line-following

3 Future investigate

Further investigating problem

System robust insurance

According to a large number of experiment results, sometimes the robot might get rid of the expected track. As mentioned in the "extra consideration" part, the judgment condition is a critical element, but how to make sure the condition blocks have a tolerance to the different environment is a problem. Here the system robust is insured by creating extreme situation tests, by using maps having different configurations to adjust the coefficients in the condition block (4). Once the coefficient configuration makes the robot pass all the tests, it is to be believed the system has robust towards its task environment, more extreme situation tests should be designed for testing the robustness of the line following system. In the future, use the conditional statements make the robot distinguish what kind of brunch or curve it located on is an interesting study, for example, solve a maze in figure 5.

Speed control

The moving speed of the robot is an important issue. Due to the moving action and line detecting action onboard are execute asynchronous (though the fast calculation speed of the chips makes their execution seems synchronous), the higher moving speed at a curve requires faster line detection and speed control. High moving speed might cause detail while low speed drags the task efficiency, hence how to make a compromise is a problem. However, the PID control system here is a good example to handle the speed control (see figure 3), which illustrates that the feedback control is a good solution to do speed control in line following task. Hence more complex feedback control system like figure 3 is worthy to be further investigated.

Future study design

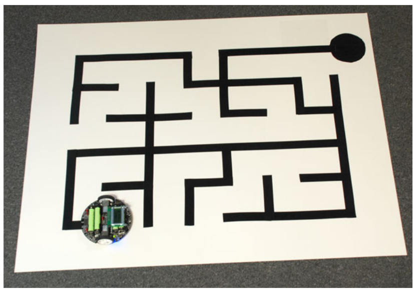

Figure 5: Robot solve a maze. *Retrieved from: https://www.pololu.com/docs/0J21/8.a*

Aim	Technology	Measurement
From the start point, track the black line and try each road brunch, finally find the end point	PID control, millis(), interrupt, kinematic estimation, tree searching algorithm	average time consume, test results in different lighting condition, probability of find the right end

Learnt from Romi case

PID control, usage of millis() & interrupt, count encoder, kinematic estimation, state machine, judgment condition, Arduino programme, task decompose. Details see figure 2.