ECEN 345

Mobile Robotics I

Prof. A. Gilmore (Instructor)
Kashfia Nabila (TA)

Laboratory Assignment #8

Line Following with Feedback Control

Member(s)

Nga Pham

Department of Electrical and
Computer Engineering(ECE)
University of Nebraska - Lincoln

Due: 04/28/2024

Contents

1 Overview

This lab focused on the development and implementation of a feedback-controlled system for line-following behavior in a robot programmed within a behavior-based control structure. The primary objective was to achieve a successful line following on a black line with the aid of proportional (P), integral (I), and derivative (D) control techniques.

The task was completed individually by Nga. The robot encountered challenges in executing sharp, narrow turns quickly. Resources used included the document provided on the STEMBOT website, "The CEENBoT-API: Programmer's Reference" by Jose Santos. A total of 20 hours were invested both in the lab and at home in completing it.

2 Background

Behavior-based control is a robotics programming technique where behaviors or functions are implemented as independent modules. This allows for flexible and some sort of parallel control of the robot's behavioral response to the environment. This lab builds upon previous knowledge of sensor integration and motor control to create a robot capable of following the line when the line is detected. This worked by employing two QRE1113 infrared (IR) sensors to detect the presence of a black line on a light-colored surface.

2.1 The Line-Following Sensors

The QRE1113 sensor operates by constantly emitting IR light through an LED. When this light reflects off a surface with high reflectivity (like a light-colored floor), it biases a phototransistor, causing it to conduct current. This results in a relatively low voltage output at the sensor's output terminal (OUT). Conversely, when the IR light encounters a surface with low reflectivity (like a black line), the phototransistor doesn't receive enough reflected light to conduct properly, leading to a high voltage output at the OUT terminal.

The robot utilizes two IR sensors positioned at user-defined separations. The voltage readings from each sensor's OUT terminal are fed into the robot's controller through Analog-to-Digital Converters (ADCs). By analyzing these voltage values, the robot's control program can determine its relative position to the line. A high voltage from both sensors indicates the robot is centered over the line, this is the target position. A high voltage from one sensor and a low voltage from the other sensor indicate the robot is positioned off-center, either to the left or right of the line, which means some correction is required.

2.2 Robot Programming

The robot's behavior is controlled using a modified BBC (Behavior-Based Control) program. A new behavior, named "Line_Follow," is implemented specifically for this task. The program incorporates a corresponding "Line_Sense" function to interpret the sensor data and determine the robot's position relative to the line. For this demonstration, only the line-following behavior is essential. Other behaviors might be temporarily disabled to prioritize line following.

3 Procedure

The first part of this experiment was to install the sensor. Mount the two IR reflectance sensors on the front of the CEENBoT using the provided L-brackets and standoffs. Connect the sensors' power and ground wires to the +5V and GND pins on RC servo ports 3 and 4 (J7) on the controller board. Connect the OUT terminals of the sensors to two chosen ADC channel inputs on header J3.

The second part was to develop a line following behavior. Create a new behavior named "Line_Follow" within the BBC program. This behavior will incorporate a corresponding "Line_Sense" function responsible for interpreting sensor data and determining the robot's position relative to the line. Design the "Line_Follow" behavior to utilize feedback control mechanisms. At a minimum, implement proportional (P) control. To enhance performance, consider incorporating derivative (D) and integral (I) control as well. Program the "Line_Follow" behavior to keep the sensors centered on the black line when the robot encounters it. The robot should continuously adjust its motor movements to maintain this position and follow the line.

The third part was to test the robot to determine if it functions correctly. Start by running the robot with your initial implementation of the "Line_Follow" behavior. Observe its performance on the the designated path. Evaluate the robot's ability to stay centered on the line. Look for any deviations or wobbliness in its movement. Make adjustments to the proportional gain (K_p), derivative gain (K_d), and integral gain (K_i) based on your observations.

4 Source Code Discussion

The code defines various structures:

- `MOTOR_ACTION`: Stores information about the robot's current state (startup, line follow) and motor actions (speed, acceleration for left and right motors).
- `SENSOR_DATA`: Stores data from the left and right line sensors.
- `ROBOT_STATE`: An enumerated type representing the robot's different states.

Different functions also existed throughout the code:

- `LINE_sense`: This function gathers data from the IR sensors at a specified interval using a timer.
- `act`: This function compares the desired motor action with the previous action. If there's a change, it executes the new action by calling the `__MOTOR_ACTION` function, which likely controls the stepper motors.
- `info_display` : This function displays information about the robot's current state (startup, line follow) on the LCD but only if the state has changed to prevent excessive flickering.
- `compare_actions` : This function compares two `MOTOR_ACTION` structures and returns true if they are identical.
- `LINE_Follow`: This is the core function for line following. It calculates an adjustment value based on a Proportional-Integral-Derivative (PID) controller and the sensor data. It then sets the left and right motor speeds based on the calculated adjustment value.
- `PID`: This function calculates a PID control output based on a setpoint (desired sensor value) and the measured sensor value.
- `CBOT_Main`: This function initializes various modules (LED, LCD, stepper motor, ADC) and sets the voltage reference. It then enters a loop that continuously performs the following:

  1. Calls `LINE_sense` to (simulate) reading sensor data.

  2. Calls `LINE_Follow` to determine motor actions based on (simulated) sensor data.

  3. Calls `act` to execute the desired motor actions if they have changed from the previous action.

  4. Calls `info_display` to update the LCD with the robot's current state.

5 Results

The line-following behavior was implemented in an iterative manner.A basic proportional (P) control was implemented first. Sensor readings were obtained (simulated for now) and compared to a threshold value representing the ideal sensor value for detecting the line center. The difference (error) between the actual and desired sensor value was used to adjust the motor speeds. A positive error indicated the robot was deviating left, so the right motor speed was increased, and vice versa. The initial P gain value was adjusted through trial and error to achieve a balance between responsiveness and stability. A high gain resulted in faster corrections but could lead to overshooting and oscillations. To improve responsiveness, particularly during turns, a derivative (D) control term was added. The D term considers the rate of change of the sensor readings, allowing the robot to react faster to approaching turns. The P and D gains were then fine-tuned together to achieve optimal performance. While P and D terms address immediate errors and their rates of change, persistent errors over time (drifting from the line) might not be fully corrected. To address this, integral (I) control was explored in simulations. The I term accumulates the error over time, gradually adjusting the motor speeds to counteract persistent deviations. However, implementing and tuning the I term requires further experimentation due to potential windup issues.

For the feedback control mechanism, the P control adjusts the motor speeds proportionally to the error (difference) between the desired sensor reading (center of the line) and the actual sensor reading. A higher P gain results in larger adjustments for larger errors but needs careful tuning to avoid instability. The D control considers the rate of change of the error signal. When the robot approaches a turn, the sensor readings change rapidly. The D term helps the robot react faster to these changes and adjust motor speeds accordingly, leading to smoother turns. The line-following behavior achieved moderate success in simulations with the implemented P, D, and I control. The robot was able to follow straight lines and make gentle turns with proper adjustments to the P, D, and I gain. The D control noticeably improved handling during turns.

Some hurdles overcome were fine-tuning the gain and the limited turning radius. Balancing responsiveness and stability through gain adjustments proved to be an iterative process. A highly responsive robot might overshoot the line, requiring gain reduction. The robot's physical design makes it more difficult to move around a tight turn.

For this lab, it assumed the robot was already placed on the line. The "trigger" for engaging the line-following behavior worked by continuously calling the `LINE_sense` to simulate reading sensor data. Regardless of actual sensor data (since it's simulated), the `LINE_Follow` function is always executed. This suggests the current implementation assumes the robot starts on the line and continuously attempts to follow it. An ideal voltage for the line was chosen, which was 4.5V. Then, reading from both line sensors will be compare to this value and an adjustment will be made after computing using the `PID()`. This approach was fairly effective in triggering the line-following behavior. The robot was able to follow straight lines and make gentle turns with proper gain adjustments. This indicates the core line-following functionality is working as intended.

6 Conclusion

This lab experiment provided valuable insights into implementing line-following behavior in robots using sensor feedback and proportional-integral-derivative (PID) control. By developing a modular structure for the line-following behavior and utilizing PID control, the robot successfully navigated its environment and followed a line when detected. This experience enhanced our understanding of sensor-based control and its role in achieving autonomous robot behaviors.

7 Appendix

7.1 Experiment source code

```c
/* Author : Nga Pham
 * Date: 4/15/2024
 * Course: ECEN345
 * Lab#: Lab#6
 * Desc: Provides a C program structure that emulates multi-tasking and
 * modularity for Behavior-based control with easy scalability
 */

#include "capi324v221.h"

// Behavior-Based Control Skeleton code.

// ---------------------- Defines:

#define DEG_90 150      /* Number of steps for a 90-degree (in place) turn. */
#define BASE_SPEED 100  // Starting speed
#define KP 0.3          // Proportional gain
#define KI 0.7          // Integral gain
#define KD 0.03         // Derivative gain

// Desc: This macro-function can be used to reset a motor-action structure
//       easily.  It is a helper macro-function.
#define __RESET_ACTION(motor_action) \
  do {                               \
    (motor_action).speed_L = 0;      \
    (motor_action).speed_R = 0;      \
    (motor_action).accel_L = 0;      \
    (motor_action).accel_R = 0;      \
    (motor_action).state = STARTUP;  \
  } while (0) /* end __RESET_ACTION() */

// Desc: This macro-fuction translates action to motion -- it is a helper
//       macro-function.
#define __MOTOR_ACTION(motor_action) \
  do { \
    STEPPER_set_accel2((motor_action).accel_L, (motor_action).accel_R); \
    STEPPER_runn((motor_action).speed_L, (motor_action).speed_R); \
  } while (0) /* end __MOTOR_ACTION() */

// Desc: This macro-function is used to set the action, in a more natural
//       manner (as if it was a function).

```

```
// ---------------------- Type Declarations:

// Desc: The following custom enumerated type can be used to specify the
//       current state of the robot.  This parameter can be expanded upon
//       as complexity grows without intefering with the 'act()' function.
//     It is a new type which can take the values of 0, 1, or 2 using
//     the SYMBOLIC representations of STARTUP, EXPLORING, etc.
typedef enum ROBOT_STATE_TYPE {

  STARTUP = 0,  // 'Startup' state -- initial state upon RESET.
  EXPLORING,    // 'Exploring' state -- the robot is 'roaming around'.
  LINE_FOLLOW   // 'Line Follow' state -- the robot is following a black (dark) line.

} ROBOT_STATE;

// Desc: Structure encapsulates a 'motor' action. It contains parameters that
//       controls the motors 'down the line' with information depicting the
//       current state of the robot.  The 'state' variable is useful to
//       'print' information on the LCD based on the current 'state', for
//       example.
typedef struct MOTOR_ACTION_TYPE {
  ROBOT_STATE state;           // Holds the current STATE of the robot.
  signed short int speed_L;    // SPEED for LEFT  motor.
  signed short int speed_R;    // SPEED for RIGHT motor.
  unsigned short int accel_L;  // ACCELERATION for LEFT  motor.
  unsigned short int accel_R;  // ACCELERATION for RIGHT motor.

} MOTOR_ACTION;

// Desc: Structure encapsulates 'sensed' data.  Right now that only consists
//       of the state of the left & right IR sensors when queried.  You can
//       expand this structure and add additional custom fields as needed.
typedef struct SENSOR_DATA_TYPE {
  float left_line;   // Holds the storage for right line sensor reading.
  float right_line;  // Holds the storage for left line sensor reading.

} SENSOR_DATA;

// ---------------------- Globals:
volatile MOTOR_ACTION action;  // This variable holds parameters that determine
```

```c
// the current action that is taking place.
// Here, a structure named "action" of type
// MOTOR_ACTION is declared.
float integral = 0;
float last_error = 0;

// ---------------------- Prototypes:
void LINE_sense(volatile SENSOR_DATA *pSensors, TIMER16 interval_ms);
void act(volatile MOTOR_ACTION *pAction);
void info_display(volatile MOTOR_ACTION *pAction);
BOOL compare_actions(volatile MOTOR_ACTION *a, volatile MOTOR_ACTION *b);
void LINE_Follow(volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA *pSensors);

// ---------------------- Convenience Functions:
void info_display(volatile MOTOR_ACTION *pAction) {
  // NOTE:  We keep track of the 'previous' state to prevent the LCD
  //        display from being needlessly written, if there's nothing
  //        new to display.  Otherwise, the screen will 'flicker' from
  //        too many writes.
  static ROBOT_STATE previous_state = STARTUP;

  if ((pAction->state != previous_state) || (pAction->state == STARTUP)) {
    LCD_clear();

    //  Display information based on the current 'ROBOT STATE'.
    switch (pAction->state) {
      case STARTUP:

        // Nofify program is about to start.
        LCD_printf("Starting...\n");

        break;

      case LINE_FOLLOW:
        LCD_printf("Following line...\n");

        break;

      default:

        LCD_printf("Unknown state!\n");

    }  // end switch()

    // Note the new state in effect.
    previous_state = pAction->state;

  }  // end if()

}  // end info_display()
```

```

// ------------------------------------------------------ //
BOOL compare_actions(volatile MOTOR_ACTION *a, volatile MOTOR_ACTION *b) {
  // NOTE:  The 'sole' purpose of this function is to
  //        compare the 'elements' of MOTOR_ACTION structures
  //        'a' and 'b' and see if 'any' differ.

  // Assume these actions are equal.
  BOOL rval = TRUE;

  if ((a->state != b->state) || (a->speed_L != b->speed_L) || (a->speed_R != b->speed_R) || (a->accel_L != b->accel_L) || (a->accel_R != b->accel_R))

    rval = FALSE;

  // Return comparison result.
  return rval;

}  // end compare_actions()

// ---------------------- Top-Level Behaviorals:
void LINE_sense(volatile SENSOR_DATA *pSensors, TIMER16 interval_ms) {
  // Sense must know if it's already sensing.
  //
  // NOTE: 'BOOL' is a custom data type offered by the CEENBoT API.
  //
  static BOOL timer_started = FALSE;

  // The 'sense' timer is used to control how often gathering sensor
  // data takes place.  The pace at which this happens needs to be
  // controlled.  So we're forced to use TIMER OBJECTS along with the
  // TIMER SERVICE.  It must be 'static' because the timer object must remain
  // 'alive' even when it is out of scope -- otherwise the program will crash.
  static TIMEROBJ sense_timer;

  // If this is the FIRST time that sense() is running, we need to start the
  // sense timer.  We do this ONLY ONCE!
  if (timer_started == FALSE) {
    // Start the 'sense timer' to tick on every 'interval_ms'.
    //
    // NOTE:  You can adjust the delay value to suit your needs.
    //
    TMRSRVC_new(&sense_timer, TMRFLG_NOTIFY_FLAG, TMRTCM_RESTART, interval_ms);

    // Mark that the timer has already been started.
    timer_started = TRUE;

  }  // end if()
```

```c

  // Otherwise, just do the usual thing and just 'sense'.
  else {
    // Only read the sensors when it is time to do so (e.g., every
    // 125ms).  Otherwise, do nothing.
    if (TIMER_ALARM(sense_timer)) {
      ADC_SAMPLE sample_line_left;
      ADC_SAMPLE sample_line_right;
      LCD_printf("read line sensor...\n");

      // NOTE: Just as a 'debugging' feature, let's also toggle the green LED
      //       to know that this is working for sure.  The LED will only
      //       toggle when 'it's time'.
      LED_toggle(LED_Green);

      // Read the left and right sensors, and store this
      // data in the 'SENSOR_DATA' structure.

      ADC_set_channel(ADC_CHAN3);
      sample_line_left = ADC_sample();
      pSensors->left_line = (sample_line_left * 5.0f) / 1024.0f;

      ADC_set_channel(ADC_CHAN4);
      sample_line_right = ADC_sample();
      pSensors->right_line = (sample_line_right * 5.0f) / 1024.0f;

      // Snooze the alarm so it can trigger again.
      TIMER_SNOOZE(sense_timer);

    }  // end if()

  }  // end else.

}  // end sense()
// -------------------------------------------- //
float PID(float setpoint, float measured_value) {
  float error = setpoint - measured_value;
  integral += error;
  float derivative = error - last_error;
  last_error = error;
  return (KP * error) + (KI * integral) + (KD * derivative);
}  // end PID()
// -------------------------------------------- //
void LINE_Follow(volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA *pSensors) {
  float left_sensor = pSensors->left_line;
  float right_sensor = pSensors->right_line;
  float position;
  LCD_clear();

  if ((right_sensor <= 4.3) || (right_sensor >= 4.7)) {
    position = pSensors->right_line;
  } else {
    position = pSensors->left_line;
  }
```

```

pAction->state = LINE_FOLLOW;  // Change state to LINE_FOLLOW

if (position == right_sensor) {
  float adjustment = PID(4.5, position);

  if ((position <= 4.3))

  {
    pAction->speed_L = BASE_SPEED - adjustment;
    pAction->speed_R = BASE_SPEED + adjustment;
    if (pAction->speed_L <= 0) {
      pAction->speed_L = 3;
    }
    LCD_printf_RC(3, 0, "shift left R\n");
  } else if (position >= 4.7) {
    pAction->speed_L = BASE_SPEED + adjustment;
    pAction->speed_R = BASE_SPEED - adjustment;
    if (pAction->speed_R <= 0) {
      pAction->speed_R = 3;
    }
    LCD_printf_RC(3, 0, "shift right\n");
  }
}
if (position == left_sensor) {
  if (position <= 4.3 || position >= 4.7) {
    float adjustment = PID(4.5, position);

    if ((position <= 4.3))

    {
      pAction->speed_L = BASE_SPEED + adjustment;
      pAction->speed_R = BASE_SPEED - adjustment;
      if (pAction->speed_R <= 0) {
        pAction->speed_R = 5;
      }
      LCD_printf_RC(3, 0, "shift right AL\n");
    }

    else if ((position >= 4.7))

    {
      pAction->speed_L = BASE_SPEED - adjustment;
      pAction->speed_R = BASE_SPEED + adjustment;
      if (pAction->speed_L <= 0) {
        pAction->speed_L = 5;
      }
      LCD_printf_RC(3, 0, "shift left AL\n");
    } else {
      pAction->speed_L = 50;
      pAction->speed_R = 50;
    }
  } else {
    LCD_printf_RC(3, 0, "straight\n");
    pAction->speed_L = 30;
    pAction->speed_R = 30;
  }
```

```
  }

  __MOTOR_ACTION(*pAction);

}  // end LINE_Follow()
// ------------------------------------------- //

void act(volatile MOTOR_ACTION *pAction) {
  // 'act()' always keeps track of the PREVIOUS action to determine
  // if a new action must be executed, and to execute such action ONLY
  // if any parameters in the 'MOTOR_ACTION' structure have changed.
  // This is necessary to prevent motor 'jitter'.
  static MOTOR_ACTION previous_action = {

      STARTUP, 0, 0, 0, 0

  };

  if (compare_actions(pAction, &previous_action) == FALSE) {
    // Perform the action.  Just call the 'free-running' version
    // of stepper move function and feed these same parameters.
    __MOTOR_ACTION(*pAction);

    // Save the previous action.
    previous_action = *pAction;

  }  // end if()

}  // end act()
// ---------------------- CBOT Main:
void CBOT_main(void) {
  volatile SENSOR_DATA sensor_data;

  // ** Open the needed modules.
  LED_open();       // Open the LED subsystem module.
  LCD_open();       // Open the LCD subsystem module.
  STEPPER_open();   // Open the STEPPER subsyste module.
  ADC_open();       // Open the ADC subsystem module.

  // Set the voltage reference (we want 5V reference).
  ADC_set_VREF(ADC_VREF_AVCC);

  // Reset the current motor action.
  __RESET_ACTION(action);

  // Wait 3 seconds or so.
  TMRSRVC_delay(TMR_SECS(3));

  // Clear the screen and enter the arbitration loop.
  LCD_clear();

  // Enter the 'arbitration' while() loop -- it is important that NONE
  // of the behavior functions listed in the arbitration loop BLOCK!
```

```c
  // Behaviors are listed in increasing order of priority, with the last
  // behavior having the greatest priority (because it has the last 'say'
  // regarding motor action (or any action)).
  while (1) {
    // Sense must always happen first.
    // (IR sense happens every 125ms).
    LINE_sense(&sensor_data, 125);

    // Behaviors.
    LINE_Follow(&action, &sensor_data);

    // Perform the action of highest priority.
    act(&action);

    // Real-time display info, should happen last, if possible (
    // except for 'ballistic' behaviors).  Technically this is sort of
    // 'optional' as it does not constitute a 'behavior'.
    info_display(&action);

  }  // end while()

}  // end CBOT_main()
```