ECEN 345/ECEN 3450-001

Mobile Robotics 1

Prof. A. Gilmore (Instructor)
Carl Endrulat (TA)

Laboratory Assignment #8

Line Following with Feedback Control

Written by
Hayden Ernst
Nelson Lefebvre

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

Due: 11/20/2022

Introduction

The purpose of this lab was to create a line following behavior utilizing a pair of IR reflectance sensors with a PID feedback control loop. The feedback would utilize the difference between the sensor values as an input to the feedback control to keep them on the same type of surface, the line. In order to do this, IR reflectance sensors were experimented with so the correct method of control could be found. Some experimentation was also done with the PID feedback control system in order to find the correct method of control.

This lab required roughly 5 hours of work. This was evenly distributed between partners. Roughly one hour was spent working with the IR reflectance sensors working to get them to function correctly with the CEENBoT. Another hour was spent creating the PID controller and integrating it with the sensors. A final three hours were spent tuning the PID controller to work well for line following. The main resources used in this lab were the lab handout as well as the lectures and notes. The IR reflectance sensor manual was another important resource. These resources gave a guide for how to mount and make use of the sensors as well as how to program with a feedback controller. Atmel Studio 7 and the CEENBoT Utility Tool were used to program the CEENBoT.

The background section will next go over some background information needed to complete this lab such the IR reflectance sensor operation. The procedure section will explain how each step in the lab was completed. A source code discussion section explains the program in more detail. The results section will discuss the outcomes of the experiments and answer questions found in the lab handout. The conclusion will summarize the report and offer some personal thoughts. The source code will be posted at the end of the document.

Background

This lab utilizes feedback control, which was first experimented with in the previous lab, lab 7. The information regarding feedback control can be found in the Lab 7 Report. This background section will instead cover the sensor utilized in this lab.

The IR reflectance sensors are simple three-terminal devices. These consist of a VCC, GND, and an OUT signal. ‘OUT’ is an analog signal which is inversely proportional to the amount of reflectivity from IR light.

VCC is connected to the +5V output pin on the CEENBoT while GND is connected to the ground. The OUT terminal is an analog output that can be connected to one of the ADC channels to get a voltage value from the sensor. As these are IR reflectance sensors that emit their own IR light, then measure the reflectance, they work better when they are mounted relatively close to the surface. Therefore, the mounting can make use of ultrasonic sensor mount as well as standoffs to position the sensors very close to the ground for precise operation.

With this information, the lab could be completed successfully.

Procedure

First, the two IR reflectance sensors were mounted on the front of the CEENBoT as discussed in the background section which was explained in the lab handout. Once this was done, some experimentation was done to get the expected values from the sensors. After that, the line following behavior was programmed with a proportional control element. This was tested with different speeds and control constant values. Some satisfactory values were discovered during the testing. These values will be explained further in the source code discussion section. In an effort to increase the speed of the robot, a derivative control element as well as a proportional control element were added. After more experimentation with these, it was determined that the extra control elements were unneeded. The main issue that was found was the difference between the sensor values on different surfaces. In order to overcome this difficulty, two different constants were utilized. One was used for when the left sensor goes off the line and one for when the right goes off the line. Once these values were tuned, the demonstration was done.

Source Code Discussion

The full source code can be seen in the source code section below. First, as the line following behavior would be the only one, the others were removed or commented out. The sensor info retrieving function was edited so the voltage values from the sensors would be

retrieved from the correct ADC channels. The code for this is as follows. Note that the voltage from the left reflectance sensor has 0.9 subtracted from it before it is saved to the sensor values struct. This is due to the difference in values the sensors returned on the white surface of the line. If this subtraction was not done the bot would not follow a straight line when both sensors were on the same surface. This could also cause issues with the error between sensors when one goes onto the black part of the line.

```
ADC_set_channel(ADC_CHAN5);
pSensors->right_Reflect = ((ADC_sample() * 5.0f) / 1024);

ADC_set_channel(ADC_CHAN4);
pSensors->left_Reflect = ((ADC_sample() * 5.0f) / 1024) - 0.9;
```

If the subtraction were not there, the difference in sensors would cause the bot to continuously veer slightly to the right in a straight line. Then, when the right sensor goes off the line, the error would not be as large as it should be causing the turn to be less than it should be as well. This could then cause an issue while navigating turns in the line. However, due to the subtraction, the left sensor would never be able to return as close a value to the full 5 volts as the right sensor can. This causes an issue when the left sensor goes off the line. In this case the error again won't be as great as it should be causing sharp turns to suffer again. With either case, there is an uneven error in the sensors. However, with subtraction, the issue will only be seen in the turns, otherwise, there would be an issue during both the turns and the straights. Therefore, the decision was made to utilize subtraction to equalize the values.

This then required the use of two different proportional constants. One for when the left reflectance was greater than the right and vice versa. As can be seen in the source code section, the target power (Tp) was set to 125 steps/second. The proportional constant (kp) for when the left sensor is greater than the right sensor was set to 100 and the kp for when the right sensor is greater than the left was set to 60. The error was calculated as the right sensor value minus the left. The turn was set to the constant multiplied by the error. Then the speed of each motor was set to the Tp minus or plus the turn. The left motor was set to

equal Tp minus turn and the right was set to Tp plus the turn. With this, the program was complete.

Results/Questions

1. Talk about how you implemented your line-following behavior.

   The line-following behavior was created as explained in the background section. It utilized the difference in values between the sensors as the error. This error was multiplied by a different proportional constant depending on which sensor value was higher than the other. This created the 'turn' value. This was then added or subtracted to the motor speed depending on which side it was. As can be seen in the source code section, the target power (Tp) was set to 125 steps/second. The proportional constant (kp) for when the left sensor is greater than the right sensor was set to 100 and the kp for when the right sensor is greater than the left was set to 60. The error was calculated as the right sensor value minus the left. The turn was set to the constant multiplied by the error. Then the speed of each motor was set to the Tp minus or plus the turn. The left motor was set to equal Tp minus turn and the right was set to Tp plus the turn.

2. Explain in detail the feedback control mechanisms you used in achieving the line-following tracking tasks and the performance improvements achieved by adding each.

   The only control mechanism we used in this program was proportional control. This was implemented as explained in the question above. Some testing was done with the other two control methods. One with the derivative control where the derivative was set to the current error minus the last error. We couldn't get this to have any meaningful impact on the line following. In fact, it seemed to produce a worse effect. Another experiment was done with a form of integral control. If the error was less than one, the integral would be continued to be set to zero. Otherwise, if the error is greater than one, the error would be added to the integral. This was designed to increase the rate of turn the longer the bot is going off the line. This way the proportional constant could remain small while still being able to

go around tight corners. This didn't seem to make a visible difference when experimented with either, so the idea was abandoned. Upon further reflection though, the threshold may have been too large for the integral control to ever begin. Regardless, satisfactory performance was achieved while only utilizing proportional control.

3. Discuss the overall performance of your line follow behavior, and the hurdles you found necessary to overcome.

This implementation of the line following behavior worked well. It was able to follow the line during straights, slight bends, and sharp corners. The main difficulty that was overcome was as discussed in the source code discussion section, the difference between sensor readings. Due to this, the proportional constant had to be very large for the bot to be able to turn to the right when the left sensor got off the line. Due to this, the turn to the left when the right sensor leaves the line was very extreme causing overcorrections among other issues. Several attempts to fix this were made while trying to utilize differential and proportional control. The conventional differential control method didn't seem to have very much of an effect on the control as observed. A method of integral control was also attempted as discussed above. This didn't seem to have a large effect either. As stated before, upon further reflection this could have been due to an error in the implementation of it. This idea was thus abandoned. At this point, the simple method of utilizing a different constant for each direction was implemented and found to work very well.

4. Finally, talk about how you implemented the "trigger" for engaging the line-following behavior.

The 'trigger' condition was not actually necessary for the completion of this lab as the robot was placed on the line for the line following behavior to begin.

Conclusion

In conclusion, this lab provided another experience with feedback control in the form of a line following behavior. This lab allowed for hands on work with a different type of sensor input for a PID controller and allowed for more experimentation with different challenges and ways of implementing feedback control. This will be good experience for future applications where feedback control can be used. As feedback control is a useful tool in any sort of robotics it may be encountered many times and the experience with different implementations of it is quite valuable.

Source Code

```
// Behavior-Based Control Skeleton code.
//
// Desc: Provides a C program structure that emulates multi-tasking and
//       modularity for Behavior-based control with easy scalability.
//
// Supplied for: Students of Mobile Robotics I, Fall 2013.
// University of Nebraska-Lincoln Dept. of Computer & Electronics Engineering
// Alisa N. Gilmore, P.E., Instructor, Course Developer.  Jose Santos, T.A.
// Version 1.3  Updated 10/11/2011.
// Version 1.4  Updated 12/2/2013
//
//      - Updated __MOTOR_ACTION() macro-function to invoke new functions
//        added to the API: `STEPPER_set_accel2()' and `STEPPER_run2()'.
//        In particular `STEPPER_run2()' now takes positive or negative
//        speed values to imply the direction of each wheel.
//
// Version 1.5  Updated 2/25/2015
//
//      - Fixed an error in __MOTOR_ACTION() and __RESET_ACTION() macros
//        where there was an extra curly brace ({) that should have been
//        removed.
//
// Version 1.6  Updated 2/24/2016
//
//      - In the 'IR_sense()' function, we now make use of the 'TIMER_ALARM()'
//        and 'TIMER_SNOOZE()' macros that were introduced in API version 2.x,
//        which makes usage of the timer object clear.   Before, students
//        manipulated the 'tc' flag inside the timer object directly, but this
```

```
//        always caused confusion, the 'TIMER_ALARM()' and 'TIMER_SNOOZE()'
//        macros achieve the same thing, but transparently.
//
//      - Also fixed __MOTOR_ACTION() macro, which previously invoked
//        'STEPPER_run2()', but the API has been modified so that this same
//        effect is now achieved with the function 'STEPPER_runn()', which
//        means to run the stepper motors and allow negative values to mean
//        reverse motion (that's what the second 'n' in 'runn' stands for).
//        This might change again in the future, because for consistency in the
//        API, it should have been called 'STEPPER_runn2()' since it takes two
//        parameters and not just one.
//
//        LAB 6
//
#include "capi324v221.h"
// ---------------------- Defines:
#define DEG_90 150 /* Number of steps for a 90-degree (in place) turn. */


// 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() */

              // Behavior value
              typedef enum ROBOT_STATE_TYPE {
                     STARTUP = 0, // 'Startup' state -- initial state upon RESET
                     EXPLORING, // 'Exploring' state -- the robot is 'roaming around'.
                     AVOIDING, // 'Avoiding' state -- the robot is avoiding a collision.
```

```
		LINE_FOLLOW // 'Line Following' state -- the robot is following a 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;


	typedef struct SENSOR_DATA_TYPE {
		BOOL left_IR; // Holds the state of the left IR.
		BOOL right_IR; // Holds the state of the right IR.
		float left_Reflect; // Holds the value of the left reflectance sensor
		float right_Reflect; // Holds the value of the right reflectance sensor
	} SENSOR_DATA;

	volatile MOTOR_ACTION action; // Holds parameters that determine the current
action that is taking place.


	void sense( volatile SENSOR_DATA *pSensors, TIMER16 interval_ms );
	void cruise( volatile MOTOR_ACTION *pAction );
	void line_follow(volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA
*pSensors);
	void IR_avoid( volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA *pSensors
);
	void act( volatile MOTOR_ACTION *pAction );
	void info_display( volatile MOTOR_ACTION *pAction, volatile
SENSOR_DATA*pSensors);
	BOOL compare_actions( volatile MOTOR_ACTION *a, volatile MOTOR_ACTION *b );
	// ---------------------- Convenience Functions:
	void info_display( volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA
*pSensors)
	{
```

```
                static ROBOT_STATE previous_state = STARTUP;
                if ( ( pAction->state != previous_state ) || ( pAction->state ==
STARTUP
                ) )
                {
                        LCD_clear();
                        // Display robot behavior
                        switch( pAction->state )
                        {
                                case STARTUP: LCD_printf( "Starting...\n" ); break;
                                case EXPLORING: LCD_printf( "Exploring...\n" ); break;
                                case LINE_FOLLOW: LCD_printf("Line Following...\n");
break;
                                default: LCD_printf( "Unknown state!\n" );
                        } // end switch()
                        // Return the state
                        previous_state = pAction->state;
                } // end if()
        }
        // ------------------------------------------------------ //
        BOOL compare_actions( volatile MOTOR_ACTION *a, volatile MOTOR_ACTION *b )
        {
                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 Behaviors:
        void 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;
```

```
                    // Sense timer is here to control the frequency of information about
sensors. The rate at which this happens must be controlled. We are therefore required to
use TIMER objects with the TIMER SERVICE. It must be "static" because the timer object must
remain even when it is out of range, 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 timer activate in every interval, we use then is this
variable interval_ms
                          TMRSRVC_new( &sense_timer, TMRFLG_NOTIFY_FLAG, TMRTCM_RESTART,
                          interval_ms );
                          // Make timer start
                          timer_started = TRUE;
                    } // end if()
                    // Otherwise just sense
                    else
                    {
                          // Ativate sensor when need
                          if ( TIMER_ALARM( sense_timer ) )
                          {
                                LED_toggle( LED_Green );
                                // light sensor data
                                ADC_set_channel(ADC_CHAN5);
                                pSensors->right_Reflect = ((ADC_sample() * 5.0f) / 1024);
                                // right light sensors
                                ADC_set_channel(ADC_CHAN4);
                                pSensors->left_Reflect = ((ADC_sample() * 5.0f) / 1024) -
0.9;

                                // Snooze the alarm, to make it trigger again.
                                TIMER_SNOOZE( sense_timer );
                          } // end if()
                    } // end else.
              } // end sense()
              // ------------------------------------------ //
              void cruise( volatile MOTOR_ACTION *pAction )
              {
                    // Nothing really changing from the other lab for the cruise function,
the most important action will be done by act
```

```
		pAction->state = EXPLORING;
		pAction->speed_L = 200;
		pAction->speed_R = 200;
		pAction->accel_L = 400;
		pAction->accel_R = 400;
	} // end explore()
	// ------------------------------------------- //
	void line_follow(volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA
*pSensors)
	{
		// variables
		float error;
		float turn;
		float derivative;
		static float lasterror = 0;
		static float integral = 0;


		signed short int Tp = 125;			//Target Power Level
		float kp = 0;					//proportional
control constant
		const float kd = 0;				//derivative
control constant
		const float ki = 0;				//integral
control constant


		if (pSensors->left_Reflect > pSensors->right_Reflect)
			kp = 100;
		else if(pSensors->left_Reflect < pSensors->right_Reflect)
			kp = 60;

		error = pSensors->right_Reflect - pSensors->left_Reflect;
		derivative = error - lasterror;
		if(error < 1)
			integral = 0;
		else
			integral += error;
		turn = (kp * error) + (kd * derivative) + (ki * integral);

		pAction->speed_L = Tp - turn;
		pAction->speed_R = Tp + turn;
```

```
		pAction->accel_L = 400;
		pAction->accel_R = 400;

		lasterror = error;

		/*LCD_clear();
		LCD_printf("Line Following...\n");
		LCD_printf("Right: %f\n", pSensors->right_Reflect);
		LCD_printf("Left: %f", pSensors->left_Reflect);
		*/
		pAction->state = LINE_FOLLOW;

	} //end line follow
	// ------------------------------------------- //
	// If one of the IR  sensor is triggered then avoid or run away the obstacle
	void IR_avoid( volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA *pSensors
)
	{
		if ((pSensors->left_IR == TRUE || pSensors->right_IR == TRUE))
		{
			STEPPER_stop( STEPPER_BOTH, STEPPER_BRK_OFF ); // Stop movement
			pAction->state = AVOIDING; // Mode : avoiding
			info_display(pAction, pSensors);
			STEPPER_move_stwt( STEPPER_BOTH,
			STEPPER_REV, 200, 200, 400, STEPPER_BRK_OFF,
			STEPPER_REV, 200, 200, 400, STEPPER_BRK_OFF );
			// If 2 IR sensors are triggered
			if (pSensors->left_IR == TRUE && pSensors->right_IR == TRUE)
			{
				// Turn RIGHT 180*
				STEPPER_move_stwt( STEPPER_BOTH,
				STEPPER_FWD, DEG_90 * 2, 200, 400, STEPPER_BRK_OFF,
				STEPPER_REV, DEG_90 * 2, 200, 400, STEPPER_BRK_OFF );
				// If left IR sensors is triggered
			} else if( pSensors->left_IR == TRUE )
			{
				// Turn RIGHT 90*
				STEPPER_move_stwt( STEPPER_BOTH,
				STEPPER_FWD, DEG_90, 200, 400, STEPPER_BRK_OFF,
				STEPPER_REV, DEG_90, 200, 400, STEPPER_BRK_OFF );
				// If right sensor is triggered
			} else if (pSensors->right_IR == TRUE)
```

```
                        {
                                // Turn LEFT 90*
                                STEPPER_move_stwt( STEPPER_BOTH,
                                STEPPER_REV, DEG_90, 200, 400, STEPPER_BRK_OFF,
                                STEPPER_FWD, DEG_90, 200, 400, STEPPER_BRK_OFF );
                        } // end if()
                        // Resume normal movement
                        pAction->speed_L = 200;
                        pAction->speed_R = 200;
                        pAction->accel_L = 400;
                        pAction->accel_R = 400;
                } // end if()
        } // end avoid()
        // ------------------------------------------- //
        void act( volatile MOTOR_ACTION *pAction )
        {
                // act track of the past action to determine if a new action must be
execute . change only if MOTOR ACTION chg
                static MOTOR_ACTION previous_action = {
                        STARTUP, 0, 0, 0, 0
                };
                if( compare_actions( pAction, &previous_action ) == FALSE )
                {
                        __MOTOR_ACTION( *pAction );
                        previous_action = *pAction;
                } // end if()
        } // end act()
        // ---------------------- CBOT Main:
        void CBOT_main( void )
        {
                volatile SENSOR_DATA sensor_data;
                LED_open();
                LCD_open();
                STEPPER_open();
                ADC_open();
                USONIC_open();
                ADC_set_VREF(ADC_VREF_AVCC);
                __RESET_ACTION( action );
                LCD_printf( "Starting...\n" );
                TMRSRVC_delay( TMR_SECS( 2 ) );
                LCD_clear();
                while( 1 )
```

```
        {
                sense( &sensor_data, 50 );
                // Behaviors
                //cruise( &action );
                line_follow( &action, &sensor_data );
                //IR_avoid( &action, &sensor_data );
                act( &action );
                info_display( &action, &sensor_data );
        } // end while()
} // end CBOT_main()
```