ECEN 345/ECEN 3450-001

Mobile Robotics 1

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

Laboratory Assignment #5

Homing on a Light Source Using Behavior Based Control

Written by
Hayden Ernst
Nelson Lefebvre

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

Due: 10/23/2022

Introduction

This lab utilized the photo-resistors from the previous lab to implement a robot that would utilize a Behavior-based control (BBC) to cruise around, home on a light source, and avoid obstacles. A bbc skeleton program was given to be adapted and improved for the lab. This puts together the past labs in which these cruise, avoidance, and homing behaviors were implemented separately. A CRUISE, IR_AVOID, and LIGHT_FOLLOW functions were made for each behavior and implemented into the bbc skeleton program to emulate the BBC in c code. The complete implementation will be explained in more detail in the rest of the report.

This lab required roughly two hours of work. This was roughly evenly split between planning, programming, testing, and debugging code. The actual programming was fairly simple as it involved bringing together the code from previous labs and implementing it into a BBC structure. The main resources used were from the lab handout as well as the provided bbc skeleton code where an overview of the program requirements and an outline of the program were found. Both the lab handout and the 'CEENBoT-API: Programming Fundamentals' document held information about getting data from the ADC as well as functions for locomotion. 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 as sensor and bot control methods. 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 worked with BBC using the given bbc skeleton code. BBC is different from the reactive control used in previous labs. BBC is a method of control that grew out of reactive control. BBC involves the use of "behaviors" as modules for control. Each behavior is utilized to achieve a single goal which is more complex than an action. For example, this lab has a cruise, avoid, and light follow behavior. Behaviors can then take inputs from sensors and can send outputs to other actuators.

Behaviors must also work in parallel, so each behavior could act on an opportunity. The c programming language used to program the CEENBoT does not inherently allow for parallel operations. So, the bbc skeleton code given provides an implementation to allow each behavior to act as well as to allow for priorities to be upheld.

The given code (shown in the source code section with changes for this lab) polls sensors by calling sensing functions and saving the data retrieved to several variables. This data is used in other behavior functions to determine whether the behavior should be executed or not. These functions are called in order of least priority to highest priority. They set a behavior that should be acted on with the last behavior function having the final opportunity to set the behavior variable and tell the program which behavior should be acted upon. Then a function takes which action to do as an input and acts on it.

This information should give a good understanding of the lab with the explanations given in the rest of the report.

Procedure

First, the bbc skeleton program was edited for the avoidance and cruise behaviors. These were already in the skeleton program, so the avoid was edited to be completer and more like that which was implemented in a previous lab. These were then called in the main function with the cruise/explore first and the avoid last so if the IR sensors detect an object, the highest priority will be avoiding it. The implementation of the sensing allows for different intervals between polling of the sensors. The length of this interval was experimented with, and different values were tested. The results will be discussed in the results section. The avoid behavior must be executed ballistically. Therefore, the motor commands are done inside of the function before returning to the main function. This implementation was then tested in the lab to check its function.

Next, a photo sensing functionality and light following behavior function were added for this behavior. The photo sensing function worked on the same interval as the IR sensing at 125ms as this was found to be sufficient for sensing light while not causing issues with the change in values for the motor speeds. Next a function for following the light was implemented. This behavior would trigger when the average of both light sensors is above a minimum value. Then the difference in the values of the left and right light sensors would be taken. This value multiplied by some gain would then be subtracted from the left and added to the right velocities. So, when the value is more negative, the left velocity will increase, and the right will decrease turning towards the right. The opposite is true when the value is positive. The light sensors were calibrated on startup with a calibration function that would take the difference between the two sensors and allow for it during the light sensing function. However, this could be seen to cause issues during operation if the sensors were seeing different light during startup. Next, the arbitration order of cruise, light follow, and avoid was setup in the main function with that order as how the functions were called. Finally, a debugging display was done with the info display function and each mode of operation. When an action was acted upon, it would be displayed with the info display function. Additionally, the light following function would display right and left sensor values to help understand that process. Finally, the implementation was tested and demonstrated.

Source Code Discussion

After having explained and implemented the code base provided, `Light_Follow` and `Light_Observe` (along with calibrate and motor copy) were also added. Also added was a state for the triggered sensor. The states are provided and allow the hierarchy of behaviors, from the most important to the least important (IR_avoid, light_observe, light_follow and cruise). If it is triggered to the left then the robot will turn to the left (90 degree) and vice versa for the right. Light_Follow allows to detect the light intensity with the photosensors (with the differential control seen in class). While the light observe function is triggered when the light intensity is high and then flees the light source. The function `Calibrate` function measured the light intensity seen by the light sensors and saved it to the limit.

Results

Several of the questions from the handout, specifically those related to discussing implementations, can be found in the procedure and source code discussion sections.

The interval for sensing on both the IR sensors and photo sensors was experimented with and varied a bit. It was found that with longer intervals, either the object or light could be missed due to the sensors not checking in time. If the interval is very short, then the values could change very rapidly causing other issues. The default time of 125ms was found to be adequate so it was kept.

Other than this, the demonstration went well with only minor limitations due to sensor and program abilities. One issue that can be commented on is a bug that was found with the motor speed during the demonstration. This issue could have been due to the calibration function at startup. This function could cause bugs when each light sensor sees a very different amount of light during startup, and this could have happened accidentally during the demonstration. From here, it would probably be better to hardcode a calibration using the measurements from the previous lab using the photo sensors.

Conclusion

In conclusion, this lab provided a learning experience with combining different previous labs into a behavior-based control scheme using a skeleton program. This lab provided a good experience for future labs where more behaviors can be implemented, and a more complex robot program will be completed. There was a little confusion in partners being chosen/assigned for the labs going forwards, so there were some parts of the program that could be optimized if we were building it together. However, the lab still turned out well and it taught us a lot.

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 5
//
#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.
HOMING, // 'Homing' state -- the robot is moving toward some stimuli
OBSERVE
} ROBOT_STATE;
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; // State of the left IR.
BOOL right_IR; // State of the right IR.
float left_PR; // Value of the left photoresistor
float right_PL; // Value of the right photoresistor
float left_PR_CAL; // Light calculated for the left photoresistor
float right_PR_CAL; // Light calculated for the right photoresistor
} 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 IR_avoid( volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA *pSensors);
void light_follow(volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA*pSensors);
void light_observe(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 );
void copy_motor( volatile MOTOR_ACTION *a, volatile MOTOR_ACTION *b );
void calibrate(volatile SENSOR_DATA *pSensors);
// ---------------------- 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( "STARTUP...\n" ); break;
case EXPLORING: LCD_printf( "EXPLORING...\n" ); break;
case AVOIDING: LCD_printf( "AVOIDING...\n" ); break;
case HOMING: LCD_printf( "HOMING...\n" ); break;
case OBSERVE: LCD_printf("OBSERVE...\n"); break;
default: LCD_printf( "DEFAULT\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 rval;

// Return comparison result.

} // end compare_actions()
void copy_motor( volatile MOTOR_ACTION *a, volatile MOTOR_ACTION *b )
{
if(!compare_actions(a, b)) // Compare to see if they are already the same motor state
{
a->accel_L = b->accel_L;
a->accel_R = b->accel_R;
a->speed_L = b->speed_L;
a->speed_R = b->speed_R;
```

```
a->state = b->state;
} // end if()
} // end copy_motor

// Light sensors calibration
void calibrate(volatile SENSOR_DATA *pSensors)
{
ADC_set_channel(ADC_CHAN3);
pSensors->left_PR_CAL = ADC_sample();
ADC_set_channel(ADC_CHAN4);
pSensors->right_PR_CAL = ADC_sample();
} // end calibrate()

// Top-Level Behaviors:
void sense( volatile SENSOR_DATA *pSensors, TIMER16 interval_ms )
{
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 ) )
{
pSensors->left_IR = ATTINY_get_IR_state( ATTINY_IR_LEFT );
pSensors->right_IR = ATTINY_get_IR_state( ATTINY_IR_RIGHT );
ADC_set_channel(ADC_CHAN3);
pSensors->left_PR = pSensors->left_PR_CAL - ADC_sample();
ADC_set_channel(ADC_CHAN4);
pSensors->right_PL = pSensors->right_PR_CAL - ADC_sample();
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 fonction, the most impotant 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()
```

```
// With the light value determine if he should home
void light_follow(volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA*pSensors)
{
// set up variables
float error;
float p = 0.0;
const float kp = 0.35;
// If light sensed > value -> homing
if ((pSensors->left_PR + pSensors->right_PL) / 2 > 30)
{
// Motors -> homing
pAction->state = HOMING;
// Difference between the two light sensors
error = (pSensors->left_PR - pSensors->right_PL);
// Multiply by constant to use it
p = error * kp;
// Change speed
pAction->speed_L = pAction->speed_L - p;
pAction->speed_R = pAction->speed_R + p;
// More information in the "Mobile Robotics I Module 8 In Class.pdf"
} // end if()
// Print in LCD screen the values of the sensors
LCD_printf_RC(1, 1, "Left: %f", pSensors->left_PR);
LCD_printf_RC(2, 1, "Right: %f", pSensors->right_PL);
} // end light_follow()


// We add that if the value are really back from before then turn the robot
void light_observe(volatile MOTOR_ACTION *pAction, volatile SENSOR_DATA*pSensors)
{
// If the sensor are under the value
```

```
if ((pSensors->left_PR > 800) && (pSensors->right_PL > 800))
{
// Mode : OBSERVE
pAction->state = OBSERVE;
// Make the motors stop moving
STEPPER_stop( STEPPER_BOTH, STEPPER_BRK_OFF );
info_display(pAction, pSensors);
// Back !
STEPPER_move_stwt( STEPPER_BOTH,
STEPPER_REV, 200, 200, 400, STEPPER_BRK_OFF,
STEPPER_REV, 200, 200, 400, STEPPER_BRK_OFF );
// 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 );
// Normal movement
pAction->speed_L = 200;
pAction->speed_R = 200;
pAction->accel_L = 400;
pAction->accel_R = 400;
}
} // end light_observe()


// 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 );
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 IR sensors 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() looking at the past action and determine wich new action must be executed. But to
execute such action only if the parameters in  'MOTOR_ACTION' have changed.
static MOTOR_ACTION previous_action = {
STARTUP, 0, 0, 0, 0
};
if( compare_actions( pAction, &previous_action ) == FALSE )
{
// Make the stepper free run
__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;
// Initiate the needed module
LED_open();
LCD_open();
STEPPER_open();
ADC_open();
ADC_set_VREF(ADC_VREF_AVCC);
```

```
// Reset motor action
__RESET_ACTION( action );
LCD_printf( "STARTUP...\n" );\
// Calibrate sensors
calibrate(&sensor_data);
TMRSRVC_delay( TMR_SECS( 3 ) );
LCD_clear();
while( 1 )
{
// Sense 1rst to feel is environment
// IR sense every 125ms
sense( &sensor_data, 125 );
// Then Behaviors
cruise( &action );
light_follow(&action, &sensor_data);
// Then Ballistic Behaviors
light_observe(&action, &sensor_data);
IR_avoid( &action, &sensor_data );
// Action with the highest priority
act( &action );
// Display the newest info
info_display( &action, &sensor_data );
} // end while()
} // end CBOT_main()
```