ECEN 345/ECEN 3450-001

Mobile Robotics 1

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

Laboratory Assignment #7

Wall Following with Feedback Control

Written by
Hayden Ernst
Nelson Lefebvre

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

Due: 11/13/2022

Introduction

This lab utilizes the ultrasonic sensor implemented in Lab 6 to create a wall following behavior based on feedback control. The feedback control would utilize the distance from the wall as an input with proportional and derivative control parts to tune the behavior for following the wall.

This was done by removing many of the behaviors from the previous lab while keeping the sensing for the ultrasonic sensor. The highest priority behavior was set to the wall following behavior. In this behavior, feedback control methods were implemented.

This lab required roughly 6 hours of work. This was evenly distributed between partners. Two hours were spent working on the non-feedback control program while attempting to debug the ultrasonic sensor. Then four hours were spent working on the feedback control behaviors and tuning the constants. The main resources used in this lab were the lab handout as well as the lectures and notes. This gave a guide for how to program feedback control. 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 with feedback control. 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

The main information necessary to complete this lab is concerned with feedback control. With ‘normal’ or non-feedback control the actuators may be set a certain way, for example motors may be set to a desired speed, then the program assumes that the setting specified would be observed. However, this may not be true. In the case of motor speed, for example, when moving up an incline the motors may slow down. This can be avoided with feedback control which will have an input of the actual state of the actuator. The purpose of feedback control is to eliminate steady state error and achieve a desired transient response. Using this feedback input, the error between the desired and actual can be adjusted for, and

a better implementation of the actuation can be used. There are three types of feedback control, proportional, derivative, and integral control. These can be implemented separately or combined. The first and most common type is proportional control. Proportional control takes the error between desired and actual as an input and multiplies it as a proportional constant to get the control signal output to correct the error. Next is derivative control. This takes the derivative of the errors over time and multiplies it by a derivative constant and adds it to the output control signal. This is used so when the rate of change of the error is decreasing, the attempt to correct the error is also decreased. In this lab, the derivative was pseudo-calculated by subtracting the previous error from the current measured error. The final form of feedback control is integral control. This takes an integral of the error and multiplies it by an integral constant. This type of control helps to remove steady state error that may accumulate over time. This integral can be pseudo-calculated as well. This is done by keeping a sum of all the errors.

With feedback control, there are a few terms to be aware of. They are rise time, percent overshoot, and settling time. Rise time can be defined as the time required to get from 10% to 90% of the final value. Percent overshoot is the percent the amount of overshoot represents when compared to the final value. Overshoot occurs when the as the actual value approaches the desired value, it continues past the desired value as it could not stop on that value quick enough. Settling time is the time required for the output to settle within 2% of its final value. With this knowledge, the lab could be completed successfully.

Procedure

This lab used the same basic setup as Lab 6. It was therefore able to use the same sensing function to return the distance from the wall. The first step was to implement a wall following behavior without feedback control that would allow the CEENBoT to start at or under 20 inches from a wall to then follow the wall at a distance of 10 inches from the wall. Measurements were taken for how long it took the robot to reach the correct distance. Other observations were made about the efficiency of this program. These can be seen in the answers to the questions in the results section.

Next, the wall following behavior was modified to be a feedback controlled behavior with proportional control. The constant was tuned to a desired level and observations were made about the program's performance which can be seen in results.

Once the proportional part of the feedback control behavior was complete, the next step was to add a derivative element to create a PD controller. Again, the performance was observed to determine the increase in performance when adding each element of the PD controller. Last, code was implemented that would allow the CEENBoT to navigate an inside corner. This was done by utilizing the IR sensors to stop the bot before it collided with the wall. Then the servo was swept so the ultrasonic sensor could take the distance to the wall in front of the bot. Then the bot would back up to 10 inches from the wall and turn to follow it again. Finally, the program was demonstrated to show the wall following capabilities.

Source Code Discussion

The ultrasonic sensing part of the program was the same as Lab 6 as both the ultrasonic avoidance behavior used in Lab 6 and the wall following behaviors utilize the distance to an object. However, the wall following behavior is quite different for this lab. As this is the main behavior that is being examined in this lab, the rest were taken out of the program leaving only sensing and wall following.

As discussed in the background section, feedback control utilizes an error that stems from the difference between the desired value and actual value. For this lab the error is found by subtracting the current distance to the wall from the desired distance to the wall. This error is also used in the derivative calculation which is the difference in the errors over time. This is calculated in the program as the subtracting the last error from the current error. The integral is the error integrated over time. The program does this by adding the error to an integral value. Then the error, derivative, and integral are multiplied by their respective constants (kp, kd, and ki) and added to create a turn variable. It should be noted that integral control wasn't implemented so ki = 0. The turn variable was added to a target power (Tp) to get the speed for the right motor and subtracted from the target power to get the speed of the left motor. Finally, the last error is updated to the error from this time.

The values of the constants can be tuned to allow for quite good wall following and even to go around an external corner. However, issues can arise when confronted with an

internal corner. To resolve this, some code was written utilizing the IR sensors. If the IR sensors are triggered during the sensing function, the motors will be stopped so the bot doesn't continue into a collision with the object in front. Then the servo will be swept back to the front so the distance to the wall in front can be measured before turning the sensor back to the right. In the wall following behavior function, a check is again done to see if during this round of sensing the IR sensors were triggered. If so, the error is converted to inches so the number of steps to move that distance can be calculated. Then the bot is set to move to 10 inches from the wall in front. Finally it turns to the left 90 degrees so it can continue to follow the wall. The code can be seen in the source code section.

Results

Performance of wall-following without feedback control.

1. How long (in distance) does it take to get the robot to within 10 inches of the wall?

   It takes the robot roughly 2 feet until it is within 10 inches of the wall.

2. Once at this distance, how well does it follow a straight path along the wall?

   No, the robot continues to oscillate while overshooting the 10 inches.

3. In particular, notice how much oscillation (if any) you observe, and at what frequency (oscillations-per-distance)?

   The robot seems to oscillate once every two feet or so. This could also be written as 0.5 oscillations per foot.

4. Do the oscillations ever die out, or are they always present?

   These oscillations are always present as the non-feedback control allows for minimal corrections.

5. When do you notice the most overshoot?

   In this case, the most overshoot occurs at the beginning just before the robot makes its first turn away from the wall

6. Describe the performance of this wall-following controller in terms of overshoot, rise time, and settling time.

   This program causes a large amount of overshoot. It does have a fairly short rise time, but the settling time is also quite large. So the overall performance is not great

Performance of wall-following with proportional feedback control.

1. How long (in distance) does it take to get the robot to within 10 inches of the wall?

   The robot takes about 2 feet to get within 10 inches of the wall

2. Once at this distance, how well does it follow a straight path along the wall?

   The oscillations are somewhat less than without feedback control. The proportional control is quick to correct, but also quick to overshoot.

3. In particular, notice how much oscillation (if any) you observe, and at what frequency (oscillations-per-distance)?

   There is still oscillation, the robot oscillates every 10 inches.

4. Do the oscillations ever die out, or are they always present?

   It is still present but much less pronounced. Later on with the derivation it will be much less pronounced.

5. When do you notice the most overshoot?

   The most overshoot can be observed when the bot first moves in from 20 inches.

6. Describe the performance of this wall-following controller in terms of overshoot, rise time, and settling time.

   The performance of this program is better than that of the program with no feedback. It has less overshoot and a shorter rise time. It also settles much better. However, there is still a good amount of overshoot and settling time required as it is proportional control.

Performance of wall-following with proportional and derivative feedback control.

1. How long (in distance) does it take to get the robot to within 10 inches of the wall?

   The robot takes roughly 3 feet to get within 10 inches to the wall.

2. Once at this distance, how well does it follow a straight path along the wall?

   Once at the correct distance, the robot does follow a fairly straight path as the derivative control prevents large overshoot.

3. In particular, notice how much oscillation (if any) you observe, and at what frequency (oscillations-per-distance)?

   There is much less oscillation than the previous two programs, however it isn’t perfect. It appears to make one oscillation every four feet or 0.25 oscillations per foot.

4. Do the oscillations ever die out, or are they always present?

   These oscillations get close to dying out but they are still present.

5. When do you notice the most overshoot?

   The most overshoot can be observed when the robot is moving away from the wall. When it moves towards the wall, there is much less overshoot.

6. Describe the performance of this wall-following controller in terms of overshoot, rise time, and settling time.

   The overshoot is much less than the previous programs, the rise time is a little worse, and the settling time is much better. The PD controller therefore has the highest performance out of those tested.

Post-lab questions

1. Discuss what hurdles you had to overcome to get the program to work.
   We had some technical problems with the battery of our robot which was not working properly but from a practical point of view we also had the problem of the robot entering the corners. Indeed, when this one goes towards an interior corner it turns too quickly towards the wall and can sometimes enter in collision with this one.

2. What did you observe about servo-control and the P and D feedback controller implementation?
   We were able to see that the robot with the regulation of speed (velocity) and position of a motor based on a feedback signal make the robot move more efficiently to achieve the goal that is following the wall. Nevertheless, we were able to experiment with the trouble that this system can create. In fact, when the robot was near the

corner the distance between him and the wall decreased too harshly, and this made the robot go directly in the direction toward the wall.

3. How did specific values of Kp and Kd impact the robot's behavior in terms of time to reach the set-point (rise-time), the amount of overshoot, and the settling time of the overshoot?

   To start, this table shows the effects of increasing the gains.

Parameter	Rise Time	Overshoot	Settling time	Error at equilibrium
Kp	Decrease	Increase	Small change	Decrease
Ki	Decrease	Increase	Increase	Eliminate
Kd	Indefinite(small decrease or increase)	Decrease	Decrease	None

   As the table shows, there are several different effects for each gain value. This is also dependent on the target motor speed, so it could be different for each program.

   For this lab roughly 6 was found to be too great of a value for Kp. At this point, the overshoot would be too large to follow the wall. For Kd, a value of roughly twice the value of Kp was found to be too great as to make rise time too large.

4. Describe in detail how you implemented the P and D controllers and how you controlled motor speed and direction.

   The target motor speed was set to be 150 steps per second. The error was found to be the desired distance minus the current distance. The derivative was found to be the current error minus the last error. Then the variable turn was calculated by taking the Kp multiplied by the error and the Kd multiplied by the derivative. The turn was added to the target speed and set to the right motor speed and while it was subtracted from the target speed and set to the left motor speed. This allowed the robot to follow the wall.

5. What did you learn?

   Overall, we learned how feedback control can be a very effective method of robot control. We also learned the basics of implementing it. Our difficulties taught us the complexities of actually tuning the gains for the control.

Conclusion

In conclusion, this lab provided a further development ultrasonic distance sensor to create a wall follow behavior with the BBC skeleton code. This lab provided a good experience for future labs utilizing the PID controller for more applications, and a more complex robot program will be completed. We had some trouble creating this behavior for this lab nevertheless the robot can follow the wall but have some trouble with the wall corner.

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.
            HOMING, // 'Homing' state -- thef robot is moving
toward some stimuli
            WALL_FOLLOW

        } 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_PR; // Holds the value of the left
photoresistor
               float right_PL; // Holds the value of the right
photoresistor
               float left_PR_CAL;// Holds the light threshold for
the left photoresistor
               float right_PR_CAL;// Holds the light threshold for
the right photoresistor
               float US_distance_cm;
          } 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 calibrate(volatile SENSOR_DATA *pSensors);
          void wall_follow(volatile MOTOR_ACTION *pAction, 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( "Starting...\n"
); break;
                         case EXPLORING: LCD_printf(
"Exploring...\n" ); break;
                         case AVOIDING: LCD_printf( "Avoiding...\n"
); break;
                         case HOMING: LCD_printf( "Homing...\n" );
break;
                         case WALL_FOLLOW:
                              LCD_printf("Wall Following...\n");
                              LCD_printf("Distance cm: %f",
pSensors->US_distance_cm);
                              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()
          // Light and ultrasonic sensors calibration
          void calibrate(volatile SENSOR_DATA *pSensors)
          {
               // Calibrate the left light sensor
               ADC_set_channel(ADC_CHAN5);
               pSensors->left_PR_CAL = ADC_sample();
               // Calibrate the right light sensor
               ADC_set_channel(ADC_CHAN6);
               pSensors->right_PR_CAL = ADC_sample();
               // Set Ultrasonic heading
               ATTINY_set_RC_servo(0, 1400);
          } // end calibrate()

          // 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;
               unsigned long int usonic_time_us;
               SWTIME usonic_time_ticks;
               // 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 );
                          pSensors->left_IR = ATTINY_get_IR_state(
ATTINY_IR_LEFT);
                          pSensors->right_IR = ATTINY_get_IR_state(
ATTINY_IR_RIGHT);
                          // light sensor data
                          ADC_set_channel(ADC_CHAN5);
                          pSensors->left_PR = pSensors->left_PR_CAL -
ADC_sample();
                          // right light sensors
                          ADC_set_channel(ADC_CHAN6);
                          pSensors->right_PL = pSensors->right_PR_CAL
- ADC_sample();

                          ATTINY_set_RC_servo( RC_SERVO0, 500 );
                          if(pSensors->left_IR || pSensors-
>right_IR){
                               STEPPER_stop( STEPPER_BOTH,
STEPPER_BRK_OFF ); // Stop movement
                               ATTINY_set_RC_servo( RC_SERVO0, 1500
);           // turn the ultrasonic sensor to the front to measure
                               usonic_time_ticks = USONIC_ping();
                               // Change tick to time
                               usonic_time_us = 10 * (( unsigned long
int)(usonic_time_ticks));
                               // Change time to distance
```

```
                              pSensors->US_distance_cm = 0.006787 *
usonic_time_us;

                              ATTINY_set_RC_servo( RC_SERVO0, 500 );

                         }
                         else{
                              // Ping US
                              usonic_time_ticks = USONIC_ping();
                              // Change tick to time
                              usonic_time_us = 10 * (( unsigned long
int)(usonic_time_ticks));
                              // Change time to distance
                              pSensors->US_distance_cm = 0.006787 *
usonic_time_us;
                         }

                         // 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 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 values of light sensors for debugging
purposes
            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 );
// Stop motors
                    info_display(pAction, pSensors); // Refresh
display
                    // 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 );
// 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 wall_follow(volatile MOTOR_ACTION *pAction, volatile
SENSOR_DATA *pSensors)
     {
          // variables
          float error;
          float turn;
          float derivative;
          static float lasterror = 0;
          static float integral = 0;
          float inches;
```

```
                int steps;


                signed short int desired_distance = 15; //Target Wall
distance
                signed short int Tp = 150;                    //Target
Power Level
                const float kp = 1.25;
     //proportional control constant
                const float kd = 1.75;
     //derivative control constant
                const float ki = 0;
     //integral control constant


                if(pSensors->left_IR || pSensors->right_IR){
                     //calculate error and steps back to reach 10 in
                     error = desired_distance - pSensors-
>US_distance_cm;
                     inches = error/2.54;
                     steps = inches/0.05;
                     if(steps > 0){
                          STEPPER_move_stwt( STEPPER_BOTH,
                //back up to 10 in from wall
                          STEPPER_REV, steps, 200, 400,
STEPPER_BRK_OFF,
                          STEPPER_REV, steps, 200, 400,
STEPPER_BRK_OFF );
                     }
                     else{
                          STEPPER_move_stwt( STEPPER_BOTH,
                //move forwards to 10 in from wall
```

```
                        STEPPER_FWD, steps*(-1), 200, 400,
STEPPER_BRK_OFF,
                        STEPPER_FWD, steps*(-1), 200, 400,
STEPPER_BRK_OFF );
                    }

                    STEPPER_move_stwt( STEPPER_BOTH,
          //turn left 90 degrees
                    STEPPER_REV, DEG_90, 200, 400, STEPPER_BRK_OFF,
          //left
                    STEPPER_FWD, DEG_90, 200, 400, STEPPER_BRK_OFF
);   //right

                    //reset variables
                    derivative = 0;
                    lasterror = 0;
                    integral = 0;
                    return;

                }

                error = desired_distance - pSensors->US_distance_cm;
                derivative = error - lasterror;
                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;


            pAction->state = WALL_FOLLOW;

        }
        // ------------------------------------------- //
        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" );\
        //calibrate(&sensor_data);
        //ATTINY_set_RC_servo( RC_SERVO0, 500 );
        TMRSRVC_delay( TMR_SECS( 3 ) );
        LCD_clear();
        while( 1 )
        {
             sense( &sensor_data, 125 );
             // Behaviors
             cruise( &action );
             //light_follow(&action, &sensor_data);
             //light_observe(&action, &sensor_data);
             //US_avoid(&action, &sensor_data);
             wall_follow( &action, &sensor_data );
             //IR_avoid( &action, &sensor_data );
             act( &action );
             info_display( &action, &sensor_data );
        } // end while()
} // end CBOT_main()
```