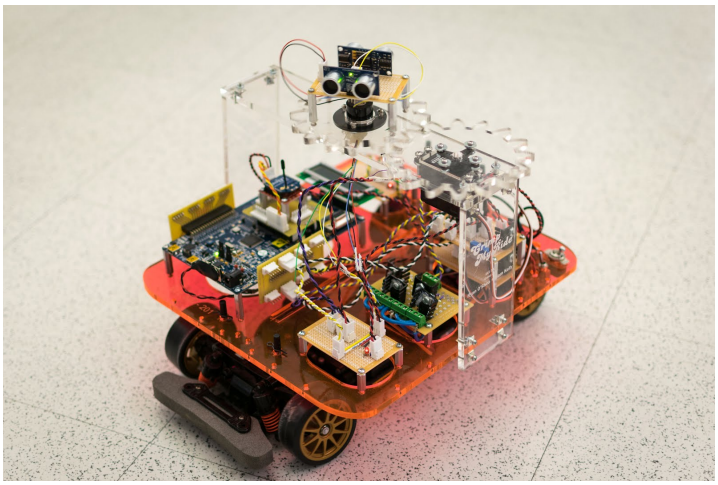
Ryan O'Shea + Liam Kelly
ELE 302
Stage 3: Independent Project
Group 201
May 15, 2015

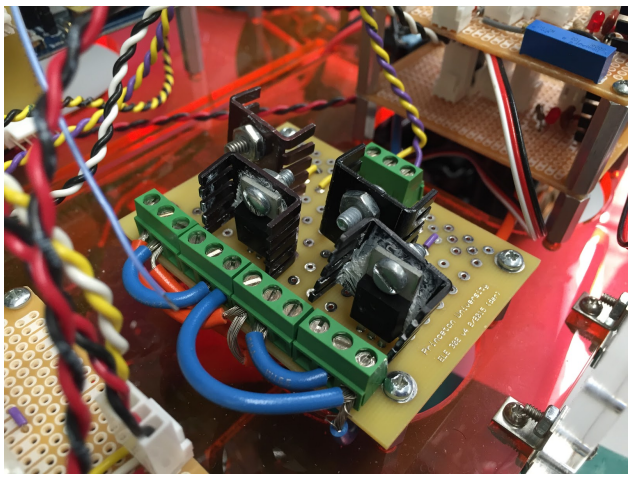
Overview



Our independent project was building crash avoidance into our car. The final car roamed around the room, avoiding static and moving obstacles whose locations were not pre-programmed. For obstacle detection, we mounted two ultrasonic rangefinders (3 meter range) facing opposite directions onto a gear assembly that rotated continuously at a fixed speed. The assembly rotated at 50rpm, giving us, via the two rangefinders, a fully refreshed view of the car's surroundings 100 times per minute. Using that information, we had the car perform two tasks while otherwise driving straight:

- steering or moving itself out of the way of objects it drives too close to (or come independently too close to the car of their own accord), and
- broadcasting the internal map of its surroundings that it uses to navigate to a computer, which graphed those detected surroundings in real time.

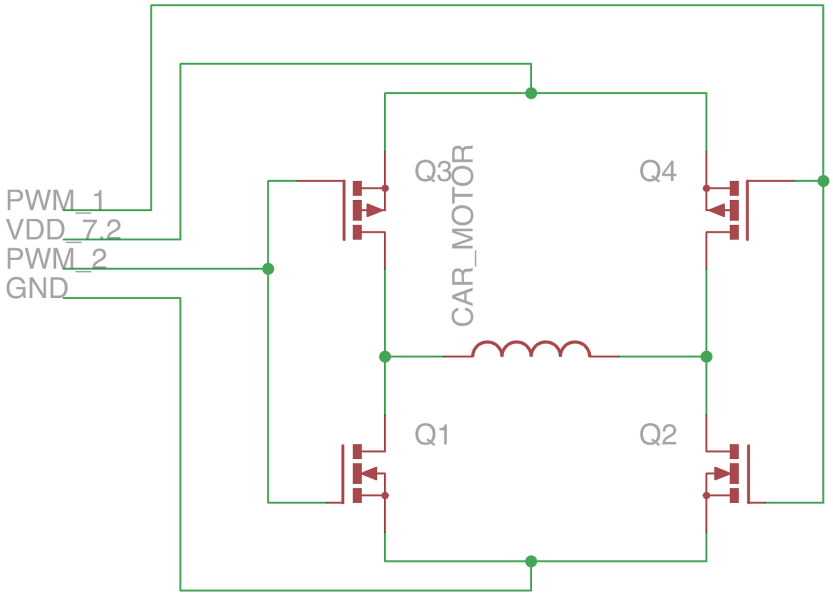
H-Bridge



To enable our car to move both forwards and backwards we built an H-Bridge on a MOSFET board. To simplify the design in the PSoC Creator, we tied the gates on both sides of the H-Bridge (see diagram, below). This helped us in two ways. First, it enables us to only use two signals from the PSoC in order to drive (one for direction, one for throttle). Second, it prevents us from shorting our battery because the transistors on the same side are never both "on", except for during transitioning from one direction to the other. The timescale of transition is

so negligible that we thought less complexity from this design outweighed the fact that our battery would need marginally more frequent trips to be charged.

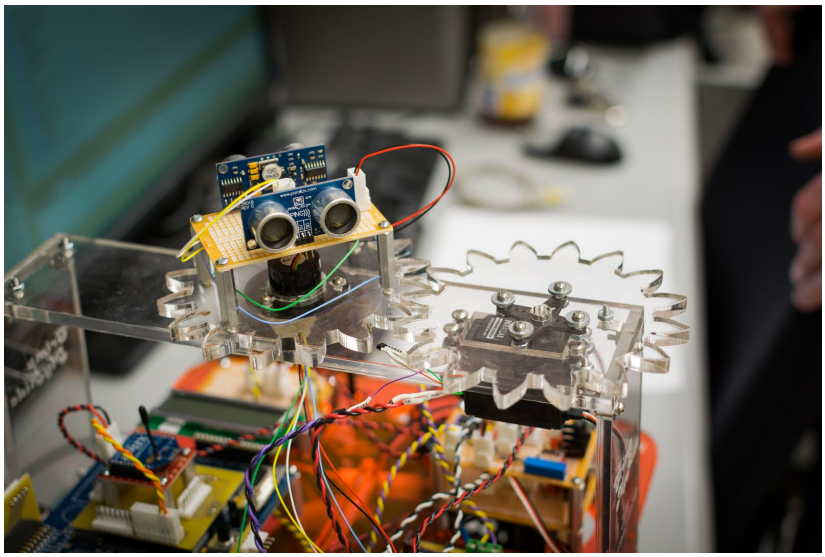
The PWM signals we used to control the motor are both generated on the PSoC. These signals are then passed through a comparator to step up the output voltage from 3.3 V to 7.2 V to ensure that the PMOS would be shut off during a logic "1" output for the PWM. To implement the bidirectional capability, we had one signal signify the direction of the car while the other signal causes movement in that direction. To get this to work properly, we XOR the direction output with the motor output, such that when the direction is a "1", we are inverting the PWM signal to drive the motor, causing similar behavior as with direction being "0" and then having the motor PWM be directly passed through as output. This enables us to not need to change our PWM parameters entirely when changing direction. The circuit schematic is below:



PING))) Sensor Board/Assembly

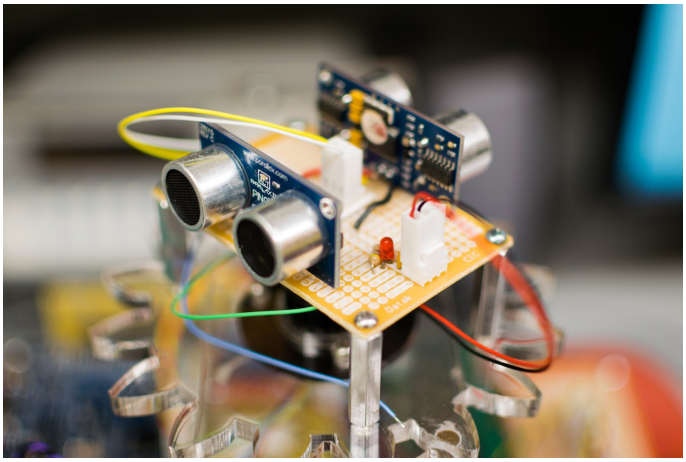
To accomplish the goal of spotting objects, we used Parallax PING))) sensors. One of our goals was being able to spot objects in any direction, not fixed angles based on the orientation of the car. To achieve this 360° vision, we built a rotating acrylic platform on which we placed two of the sensors pointing exactly opposite each other. A continuous rotation servo was used so that we could achieve constant rotational speeds of ~50 rpm to get updates as close to real time as possible without needing to change direction, which many other servo motor solutions would have required us to do. We noted that there would be a problem with wires twisting with each other given a continuous rotation, so we used a

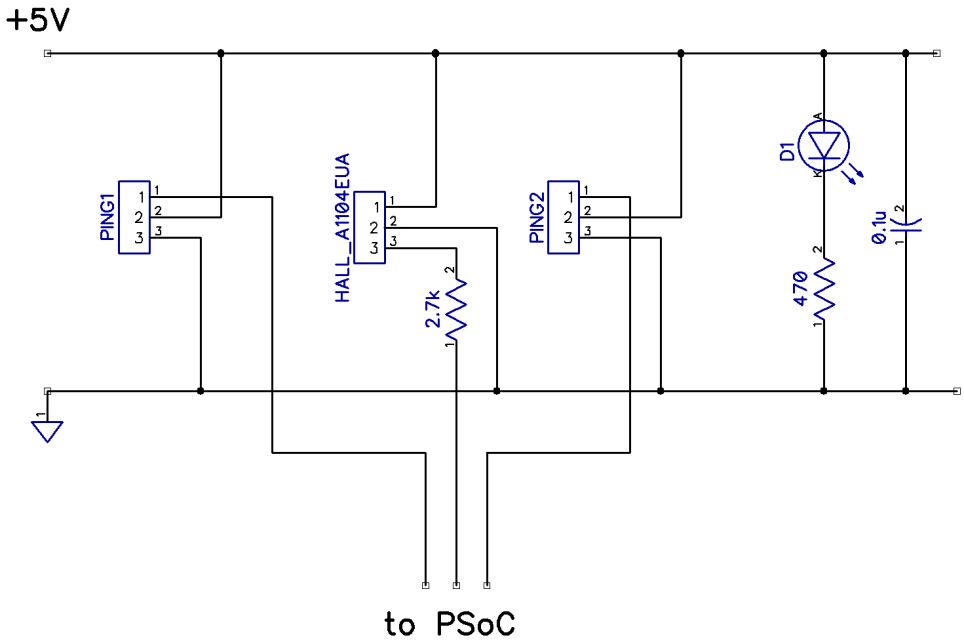
slip ring to allow the wires going to the platform to rotate freely while still connecting signals to the PSoC without any twisting of the wires occurring.



The sensor mount and rotation assembly

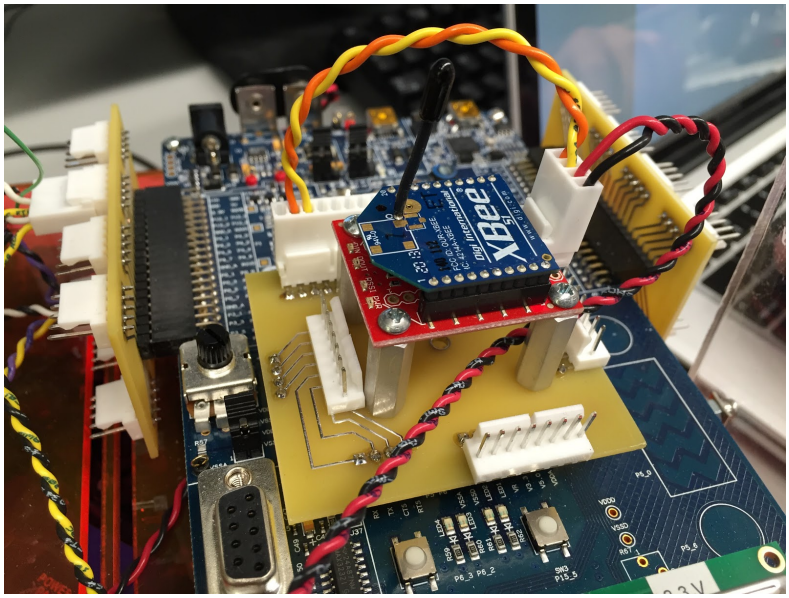
With regard to the PING))) sensors themselves, they are ultrasonic range finders that produce a 40kHz signal and report the amount of time before an echo response. We drove them almost as fast as possible to get the most data about our surroundings, firing the sensor every 20ms or at 50Hz. To know what angle the sensor is at when it is reading data, we attached a magnet to the underside of the rotating platform and a Hall effect sensor to read that magnet and reset a Timer in the PSoC. When we capture output from the PING))) sensor, we read off the value of the timer and do a direct conversion to angle offset from an arbitrarily defined 0°, which corresponds to the sensors firing in the direction of the front and back of the car. We differentiate between the sensors by knowing the different pins they are attached to so that 0° for one corresponds with 180° for the other. Angle proceeds clockwise from the front of the car, such that 0° is the front of the car, 90° is directly right of the car, 180° is the rear of the car, and 270° is directly left of the car. We then average values into one of 18 "bins", which each map to 20° subdivisions of the car's surroundings. This vector based view of the environment is then used in our control algorithm, discussed below.





Circuit schematic for the Ping))) sensors and platform Hall effect sensor

Wireless Surroundings Visualization

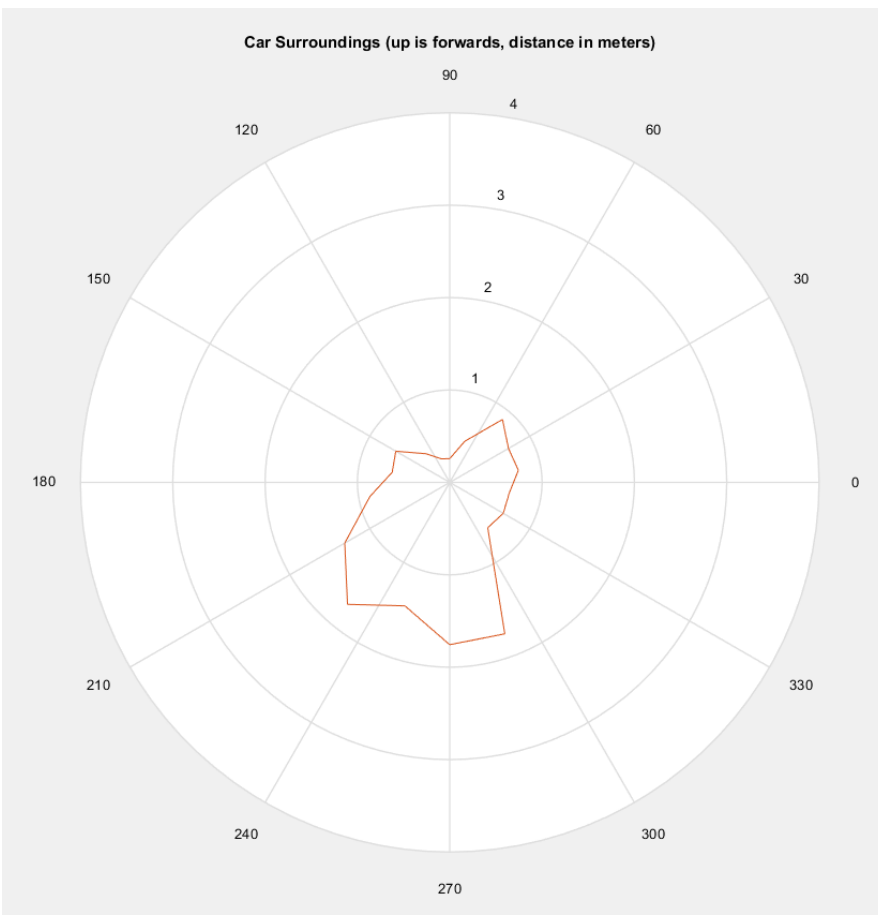


While our ultrasonic sensors created what is effectively a map of the car's local environment broken down into 18 vectors arranged radially around the car, the car was broadcasting that data (the 18 vectors) wirelessly to a laptop. There were a few steps in this process. First, we encoded the vectors in text using a schema of our own. First the vector number, between 0 and 17, followed by a comma, then the floating point distance recorded for that vector (rounded to 3 decimal places), then an equals sign, followed by the next 17 vectors encoded in the same way, then a newline character, which delimited each set of vectors. So the data was formatted like this:

```
0,0.613=1,2.138=2,1.213= … =17,0.321
0,0.454=1,0.899=2,2.288= … =17,1.746
…
```

We transmitted this data using the UART component of the PSoC to send it over the PSoC's RS-232 serial connection. To keep the car mobile, we used a pair of XBees to form a wireless bridge for the serial connection. One was mounted on the car, and one connected to the laptop. We increased the baud rate from 9600 to 57600 when we noticed that the performance of our car was compromised by transmitting information and doing control simultaneously. Stepping up the baud rate, however, meant that less processor time was spent transmitting data so the control algorithm would have more processor time to execute.

We wrote a MATLAB script (see Appendix B) to read the data coming in over the laptop's serial port, parse it using the schema we defined, and plot one set at a time. The data is broadcast every 600ms (100 times per minute, or the time it takes for the car to image its surroundings fully) from the car, and it is plotted every 600ms as well, but the plotted values are actually a weighted average of the last 3 sets of surroundings data, to achieve a more accurate picture of the car's surroundings. That visualization looks like this, with 90° corresponding to the front of the car and 0° being the right looking at the car from above:



Control Algorithm

Every 200ms, an interrupt was fired on the PSoC to update the steering of the car based on its environment. There is a response only if there is an object within 1 meter, otherwise the car will proceed straight at a speed of 2.5 feet per second. When the interrupt is fired, the car searches through each of its bins and stores the two closest objects and their bin numbers. The algorithm is such that adjacent bins are not seen as distinct objects (e.g. a wall is a wall even if it is picked up by more than one bin). If the second closest object is outside of the threshold, the car acts only on the first object, assuming it is within the threshold. If both objects are within the threshold, an "effective bin" is generated by a weighted average of the bin numbers, with the weights generated by the distances to the objects. The effective bin gives us a good idea of the general direction of objects in the car's view that it should respond to.

After the effective bin is generated, the car goes through a specified set of instructions to react to an object in that bin. Namely, it steers away from them. Specifically, objects behind the car lead to the car driving forward, directly away from the object (this includes steering the car to move directly away, regardless of the orientation of the object with respect to the car). If the object is near the front of the car but off to the side (within 1 meter), the wheels will point in a direction away from the obstacle in an attempt to steer out of the way of the object without changing the car's forward/backward direction. If that object moves closer, to within a secondary inner threshold (0.65 meters), the car will stop its forward motion, back up, turn away from the obstacle, and then continue on. Because steering out of the way doesn't do much to help with an object directly in front of the car, any object directly in front will result in the car reversing direction and turning the wheels in such a way as to put the object to its side. The car adapts its throttle when dealing with obstacles to move away from them effectively. We made many fine adjustments to how quickly the car should move in a variety of circumstances, but for example, the car moves away from objects substantially faster if they are closer than 0.4 meters, while it adopts a slower pace while avoiding objects farther away.

We noticed that our car is best used with static objects, although dynamic objects are also detectable and avoidable. With dynamic objects, it is important that they not be dropped within about 1.5 feet in front of the car if it is moving forward at 2.5 feet per second. Otherwise, it is on what we call a highway to the "danger zone", where behavior might result in the car running into the object since there may not be time to detect it with a maximal rotation speed of 50 rpm and two sensors. The car is able to detect and avoid objects of varying size, however, ranging from walls to the legs of chairs. The car seems to be able to detect nearly all surfaces, though it was unable to pick up soft foam balls, which absorb the ultrasound signals broadcast by the car. Because the car cannot see objects that are very low to the ground (below 3 inches in height), it sometimes collides with people's feet, since it can see the person's legs but not their protruding feet. However, in most cases, it responds well to human obstacles, even those with soft, loose clothing. During our demo, a professor stepped out onto the course about 2 feet in front of the car in motion, and it successfully

avoided a collision with her. The car's overall range is from about 3 meters on all sides of the car right up to directly touching the car. The sensors work down to about 3 cm, meaning that even objects that are just next to the car are well within the sensing range (although any object 3 cm from the sensor would have already collided with the car). It can see a vertical range from about 3 inches above the floor to about 3 feet above it, depending on the distance the object is from the sensor.

Overall, we were very pleased with the car's ability to avoid both static and moving obstacles both large and small. It *very* rarely came into contact with an obstacle during our testing, and we found its behavior to be accurate, responsive, and moving in an intuitive manner (i.e. after fine-tuning its behavior, the car seemed to move out of the way of obstacles it was presented with in a way that one would expect it to, if it could see its entire environment).

All code for the control algorithm is contained in Appendix A.

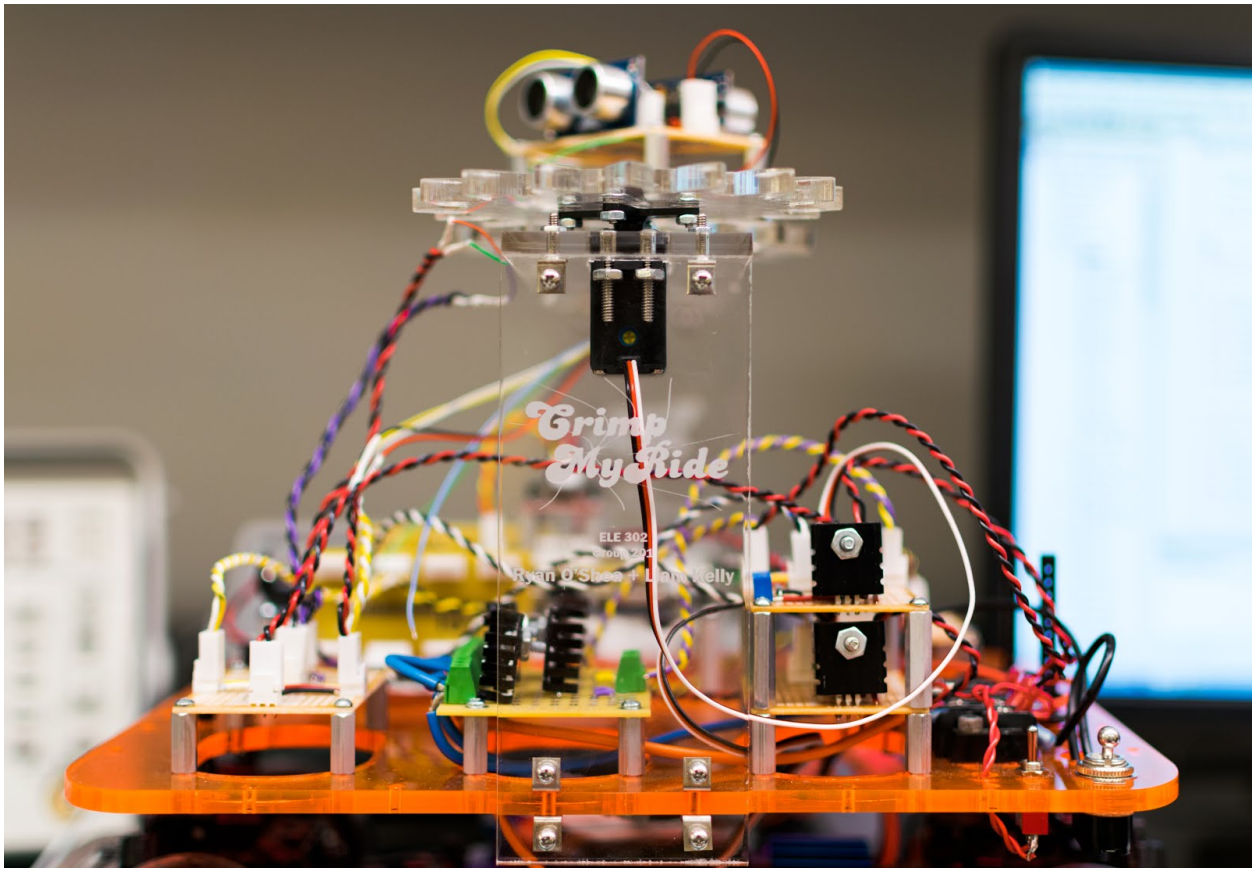
Conclusion

Our algorithm is designed to be quickly executing so the limiting factors of our system would be the accuracy and noise of the PING))) sensors, as well as the rotational speed of our platform. We found that averaging nearby measurements before putting them into bins was very helpful in reducing the noise of each measurement, at the cost of knowing exactly where the objects are. However, having a vector map of the surroundings with 18 vectors that have 20° of scope results in behavior that is still very responsive to any change in the environment.

By demo day, however, we were able to successfully demonstrate a car that can detect and avoid obstacles in real time through a vector mapping algorithm.

Acknowledgements

We would like to thank Professor Andrew Houck and Professor Antoine Kahn as well as our Lab Manager David Radcliff for all the work they put into the course and the encouragement we received in completing our assignments. We would also like to thank the TA's for the course, both graduate and undergraduate, for the time they helped us in debugging errors and helping our car function the way we envisioned.



The final car

Appendix A: main.c

```c
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include <device.h>
#include <stdio.h>
#include <stdlib.h>

/* Variables and Constants for Speed Control */
uint32 MAX_TIME = 65535; //Counter period
uint32 FREQ = 100; //Hz
uint32 lastTimeSeen[5];
uint16 FORWARD = 999;
uint16 BACKWARD = 0;
uint8  dir_flag;
float CIRCUMFERENCE = 0.66;
float SETPOINT = 3.5;
int currentMagnet;
float speed;

float epsilon = 0.01;
float Kp = 65;
float Ki = 10;
float Kd = .07;
float MAX_OUT = 700;
float MIN_OUT = 0;
float preError;
float integral;
float openloop;
float speed_eps = 0.05;
float battery_scale = 0.95;

float PIDCtrl(float wantedSpeed, float actualSpeed)/*, float dt)*/
{
    float error = wantedSpeed - actualSpeed;
    float derivative;
    float output;

    // Only integrate if error large enough
    float abserror;
    if (error >= 0) abserror = error;
    else abserror = -1*error;
    if (abserror > epsilon)
        integral += error;

    derivative = error - preError;
    output = openloop + Kp*error + Ki*integral + Kd*derivative;

    // Saturation filter
    if (output > MAX_OUT)
    {
        output = MAX_OUT;
```

```
    }
    if (output < MIN_OUT)
    {
        output = MIN_OUT;
    }


    preError = error;

    return output;
}

CY_ISR(magnet)
{
    uint32 nextTime = SpeedTimer_ReadCounter();
    uint32 prevTime = lastTimeSeen[currentMagnet];

    float dt = (prevTime - nextTime)/(float)FREQ;
    lastTimeSeen[currentMagnet] = nextTime;
    if (nextTime > prevTime)
    {
        nextTime = nextTime - MAX_TIME;
    }

    speed = (CIRCUMFERENCE / dt);

    lastTimeSeen[currentMagnet] = nextTime;
    currentMagnet++;
    if (currentMagnet == 5)
    {
        currentMagnet = 0;
    }
}

CY_ISR(stall)
{
    uint32 currentTime = SpeedTimer_ReadCounter();
    uint32 idx;
    uint32 oldTime;
    float pidOutput;
    if (currentMagnet == 0) idx = 5;
    else idx = currentMagnet;
    oldTime = lastTimeSeen[idx - 1];

    if ((oldTime - currentTime)/FREQ > 0.25) // Stalled
    {
        pidOutput = PIDCtrl(SETPOINT, (float)0);/*, .1);
        Motor_PWM_WriteCompare1((uint16) pidOutput);
    }
    else // Normal PWM update
    {
        pidOutput = PIDCtrl(SETPOINT, speed);/*, .1);
        Motor_PWM_WriteCompare1((uint16) pidOutput);
    }
}

/* Variables for Ping sensors */
float  SOUND_SPEED_IN_AIR = 346.5;
float  DIST_FREQ = 1000000.;
uint32 MAX_INT = 4294967295;
uint16 MAX_16 = 65535;
```

```
uint16 W_OFFSET = 1300;
uint16 Y_OFFSET = 19300;
uint16 CIRCLE = 360;
float  bins[20];
uint16 DIV_NUM = 20;
uint16 NUM_BINS = 18;
uint16 current_y_bin;
uint16 current_w_bin;
uint16 prev_y_bin;
uint16 prev_w_bin;
uint16 y_counts;
uint16 w_counts;
float  y_accum_dist;
float  w_accum_dist;

CY_ISR(ping1) /* White Sensor */
{
    uint16 platform = Platform_Timer_ReadCounter();
    uint32 junk = Ping1_Timer_ReadCapture();
    uint32 junk2 = Ping1_Timer_ReadCapture();
    uint32 frontEdgeTime = Ping1_Timer_ReadCapture();
    uint32 endEdgeTime = Ping1_Timer_ReadCapture();
    uint32 timeDiff = frontEdgeTime - endEdgeTime;
    float distance;
    char strbuffer[15];
    uint16 degree;

    distance = (SOUND_SPEED_IN_AIR * ((float) timeDiff / DIST_FREQ))/(float) 2;
    degree = MAX_16-platform+W_OFFSET;
    degree /= 100;
    if (degree > CIRCLE)
        degree -= CIRCLE;

    current_w_bin = degree/DIV_NUM;
    if (current_w_bin != prev_w_bin)
    {
        bins[prev_w_bin] = w_accum_dist/w_counts;
        prev_w_bin = current_w_bin;
        w_accum_dist = distance;
        w_counts = 1;
    }
    else
    {
        w_accum_dist += distance;
        w_counts++;
    }

    if (degree <= CIRCLE)
        bins[current_w_bin] = distance;
}

CY_ISR(ping2) /* Yellow Sensor */
{
    uint16 platform = Platform_Timer_ReadCounter();
    uint32 junk = Ping2_Timer_ReadCapture();
    uint32 junk2 = Ping2_Timer_ReadCapture();
    uint32 frontEdgeTime = Ping2_Timer_ReadCapture();
    uint32 endEdgeTime = Ping2_Timer_ReadCapture();
    uint32 timeDiff = frontEdgeTime - endEdgeTime;
    float distance;
    char strbuffer[15];
```

```
    uint16 degree;

    distance = (SOUND_SPEED_IN_AIR * ((float) timeDiff / DIST_FREQ))/(float) 2;
    degree = MAX_16-platform+Y_OFFSET;
    degree /= 100;
    if (degree > CIRCLE)
        degree -= CIRCLE;

    current_y_bin = degree/DIV_NUM;
    if (current_y_bin != prev_y_bin)
    {
        bins[prev_y_bin] = y_accum_dist/y_counts;
        prev_y_bin = current_y_bin;
        y_accum_dist = distance;
        y_counts = 1;
    }
    else
    {
        y_accum_dist += distance;
        y_counts++;
    }

    if (degree <= CIRCLE)
        bins[degree/DIV_NUM] = distance;
}

uint16 RIGHT = 190;
uint16 SLIGHT_RIGHT = 170;
uint16 SLIGHT_LEFT  = 130;
uint16 LEFT = 110;
uint16 MIDDLE = 153;
float  MIN_DISTANCE = 1.0;
float  MED_DISTANCE = 0.65;
float  TURN_DIST = 0.4;
float  ABSMIN_DISTANCE  = 0.4;
uint8  any_seen;

CY_ISR(platform)
{
    float closest_dist  = 10.0;
    float closest_dist2 = 10.0;
    uint16 closest_bin, closest_bin2;
    int i;
    char strbuffer[15];

    for (i = 0; i < NUM_BINS; i++)
    {
        if (bins[i] < closest_dist)
        {
            if ((i != closest_bin-1) && (i != closest_bin+1))
            {
                closest_dist2 = closest_dist;
                closest_bin2 = closest_bin;
            }
            closest_dist = bins[i];
            closest_bin  = i;
        }
    }
```

```
    /* Forward/Backward adjustment */
    if (closest_dist < MIN_DISTANCE)
    {
        uint16 effective_bin;
        any_seen = 1;

        if (closest_dist2 < MIN_DISTANCE)
        {
            float dist_sum = closest_dist + closest_dist2;
            if ((closest_bin >= 14) && (closest_bin <= 17)) // One point in left forward
quadrant
            {
                if ((closest_bin2 >= 0) && (closest_bin2 <= (closest_bin - 9))) // Second
point in (generally) front right
                {
                    effective_bin = ((uint16)(((closest_dist2/dist_sum)*(float)closest_bin +
(closest_dist/dist_sum)*(float)(closest_bin2 + 18)))) % 18;
                }
                else
                    effective_bin = (closest_bin + closest_bin2)/2;
            }
            else if ((closest_bin2 >= 14) && (closest_bin2 <= 17)) // One point in left
forward quadrant
            {
                if ((closest_bin >= 0) && (closest_bin <= (closest_bin2 - 9))) // Second point
in (generally) front right
                {
                    effective_bin = ((uint16)(((closest_dist/dist_sum)*(float)closest_bin2 +
(closest_dist2/dist_sum)*(float)(closest_bin + 18)))) % 18;
                }
                else
                    effective_bin = (closest_bin + closest_bin2)/2;
            }
            else if ((closest_bin >= 0) && (closest_bin <= 4)) // One point in right forward
quadrant
            {
                if (closest_bin2 >= (closest_bin+9)) // Second point in (generally) front left
                {
                    effective_bin =
((uint16)(((closest_dist2/dist_sum)*(float)(closest_bin+18) +
(closest_dist/dist_sum)*(float)closest_bin2))) % 18;
                }
                else
                    effective_bin = (closest_bin + closest_bin2)/2;
            }
            else if ((closest_bin2 >= 0) && (closest_bin2 <= 4)) // One point in right forward
quadrant
            {
                if (closest_bin >= (closest_bin2+9)) // Second point in (generally) front left
                {
                    effective_bin =
((uint16)(((closest_dist/dist_sum)*(float)(closest_bin2+18) +
(closest_dist2/dist_sum)*(float)closest_bin))) % 18;
                }
                else
                    effective_bin = (uint16)((closest_dist2/dist_sum)*(float)closest_bin +
(closest_dist/dist_sum)*(float)closest_bin2);
            }
            else
                effective_bin = (uint16)((closest_dist2/dist_sum)*(float)closest_bin +
(closest_dist/dist_sum)*(float)closest_bin2);
```

```
}
else
    effective_bin = closest_bin;
if (effective_bin >= NUM_BINS)
    effective_bin -= NUM_BINS;

if (closest_dist < ABSMIN_DISTANCE)
    SETPOINT = 3.5*battery_scale;
else if (closest_dist < MED_DISTANCE)
    SETPOINT = 3.0*battery_scale;
else
    SETPOINT = 2.5*battery_scale;
openloop = 40*SETPOINT;

switch(effective_bin)
{
    case 0:
        Steering_PWM_WriteCompare(RIGHT);
        Motor_PWM_WriteCompare2(BACKWARD);
        if (dir_flag == 0)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 1;
        }
        break;
    case 1:
        Steering_PWM_WriteCompare(RIGHT);
        Motor_PWM_WriteCompare2(BACKWARD);
        if (dir_flag == 0)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 1;
        }
        break;
    case 2:
        if (closest_dist < TURN_DIST)
        {
            Steering_PWM_WriteCompare(RIGHT);
            Motor_PWM_WriteCompare2(BACKWARD);
            if (dir_flag == 0)
            {
                SETPOINT = 3.0*battery_scale;
                openloop = 50*SETPOINT;
                preError = 0.0;
                integral = 2*SETPOINT;
                dir_flag = 1;
            }
        }
        else
        {
            Steering_PWM_WriteCompare(SLIGHT_LEFT);
            Motor_PWM_WriteCompare2(FORWARD);
            if (dir_flag)
            {
                SETPOINT = 3.0*battery_scale;
```

```
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 0;
        }
    }
    break;
case 3:
    if (closest_dist < TURN_DIST)
    {
        Steering_PWM_WriteCompare(RIGHT);
        Motor_PWM_WriteCompare2(BACKWARD);
        if (dir_flag == 0)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 1;
        }
    }
    else
    {
        Steering_PWM_WriteCompare(SLIGHT_LEFT);
        Motor_PWM_WriteCompare2(FORWARD);
        if (dir_flag)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 0;
        }
    }
    break;
case 4:
    Steering_PWM_WriteCompare(SLIGHT_LEFT);
    Motor_PWM_WriteCompare2(FORWARD);
    if (dir_flag)
    {
        SETPOINT = 3.0*battery_scale;
        openloop = 50*SETPOINT;
        preError = 0.0;
        integral = 2*SETPOINT;
        dir_flag = 0;
    }
    break;
case 5:
    Steering_PWM_WriteCompare(MIDDLE);
    Motor_PWM_WriteCompare2(FORWARD);
    if (dir_flag)
    {
        SETPOINT = 3.0*battery_scale;
        openloop = 50*SETPOINT;
        preError = 0.0;
        integral = 2*SETPOINT;
        dir_flag = 0;
    }
    break;
case 6:
    Steering_PWM_WriteCompare(MIDDLE);
```

```
        Motor_PWM_WriteCompare2(FORWARD);
        if (dir_flag)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 0;
        }
        break;
    case 7:
        Steering_PWM_WriteCompare(MIDDLE);
        Motor_PWM_WriteCompare2(FORWARD);
        if (dir_flag)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 0;
        }
        break;
    case 8:
        Steering_PWM_WriteCompare(MIDDLE);
        Motor_PWM_WriteCompare2(FORWARD);
        if (dir_flag)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 0;
        }
        break;
    case 9:
        Steering_PWM_WriteCompare(MIDDLE);
        Motor_PWM_WriteCompare2(FORWARD);
        if (dir_flag)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 0;
        }
        break;
    case 10:
        Steering_PWM_WriteCompare(MIDDLE);
        Motor_PWM_WriteCompare2(FORWARD);
        if (dir_flag)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 0;
        }
        break;
    case 11:
        Steering_PWM_WriteCompare(MIDDLE);
        Motor_PWM_WriteCompare2(FORWARD);
```

```
        if (dir_flag)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 0;
        }
        break;
    case 12:
        Steering_PWM_WriteCompare(MIDDLE);
        Motor_PWM_WriteCompare2(FORWARD);
        if (dir_flag)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 0;
        }
        break;
    case 13:
        Steering_PWM_WriteCompare(SLIGHT_RIGHT);
        Motor_PWM_WriteCompare2(FORWARD);
        if (dir_flag)
        {
            SETPOINT = 3.0*battery_scale;
            openloop = 50*SETPOINT;
            preError = 0.0;
            integral = 2*SETPOINT;
            dir_flag = 0;
        }
        break;
    case 14:
        if (closest_dist < TURN_DIST)
        {
            Steering_PWM_WriteCompare(LEFT);
            Motor_PWM_WriteCompare2(BACKWARD);
            if (dir_flag == 0)
            {
                SETPOINT = 3.0*battery_scale;
                openloop = 50*SETPOINT;
                preError = 0.0;
                integral = 2*SETPOINT;
                dir_flag = 1;
            }
        }
        else
        {
            Steering_PWM_WriteCompare(SLIGHT_RIGHT);
            Motor_PWM_WriteCompare2(FORWARD);
            if (dir_flag)
            {
                SETPOINT = 3.0*battery_scale;
                openloop = 50*SETPOINT;
                preError = 0.0;
                integral = 2*SETPOINT;
                dir_flag = 0;
            }
        }
        break;
```

```
        case 15:
            if (closest_dist < TURN_DIST)
            {
                Steering_PWM_WriteCompare(LEFT);
                Motor_PWM_WriteCompare2(BACKWARD);
                if (dir_flag == 0)
                {
                    SETPOINT = 3.0*battery_scale;
                    openloop = 50*SETPOINT;
                    preError = 0.0;
                    integral = 2*SETPOINT;
                    dir_flag = 1;
                }
            }
            else
            {
                Steering_PWM_WriteCompare(SLIGHT_RIGHT);
                Motor_PWM_WriteCompare2(FORWARD);
                if (dir_flag)
                {
                    SETPOINT = 3.0*battery_scale;
                    openloop = 50*SETPOINT;
                    preError = 0.0;
                    integral = 2*SETPOINT;
                    dir_flag = 0;
                }
            }
            break;
        case 16:
            Steering_PWM_WriteCompare(LEFT);
            Motor_PWM_WriteCompare2(BACKWARD);
            if (dir_flag == 0)
            {
                SETPOINT = 3.0*battery_scale;
                openloop = 50*SETPOINT;
                preError = 0.0;
                integral = 2*SETPOINT;
                dir_flag = 1;
            }
            break;
        case 17:
            Steering_PWM_WriteCompare(RIGHT);
            Motor_PWM_WriteCompare2(BACKWARD);
            if (dir_flag == 0)
            {
                SETPOINT = 3.0*battery_scale;
                openloop = 50*SETPOINT;
                preError = 0.0;
                integral = 2*SETPOINT;
                dir_flag = 1;
            }
            break;
    }
}

else if (any_seen)
{
    SETPOINT = 2.5*battery_scale;
    openloop = 50*SETPOINT;
    Steering_PWM_WriteCompare(MIDDLE);
    if (dir_flag)
```

```
        {
            preError = 0.0;
            integral = 2*SETPOINT;
            Motor_PWM_WriteCompare2(FORWARD);
            dir_flag = 0;
        }
        any_seen = 0;
    }
}

CY_ISR(tx)
{
    int i;
    char strbuffer[15];
    for (i = 0; i < NUM_BINS; i++)
    {
        sprintf(strbuffer, "%d,%04.3f=", i, bins[i]);
        UART_PutString(strbuffer);
    }
    sprintf(strbuffer, "\n");
    UART_PutString(strbuffer);
}

void main()
{
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */
    int i;
    float pidOutput;

    LCD_Start();

    /* Motor initializing */
    Motor_PWM_Start();
    SpeedTimer_Start();
    StallTimer_Start();
    CyGlobalIntEnable;
    inter_hall_Start();
    inter_hall_SetVector(magnet);
    inter_stall_Start();
    inter_stall_SetVector(stall);

    /* Steering initializing */
    Steering_PWM_Start();

    /* Distance initialization */
    Ping_PWM_Start();
    Ping1_Timer_Start();
    Ping2_Timer_Start();
    inter_ping1_Start();
    inter_ping1_SetVector(ping1);
    inter_ping2_Start();
    inter_ping2_SetVector(ping2);
    Ping_Motor_PWM_Start();
    Platform_Timer_Start();
    PlatInter_Counter_Start();
    inter_platform_Start();
    inter_platform_SetVector(platform);

    /* RS232 initialization */
    UART_Start();
```

```
    TX_Counter_Start();
    inter_tx_Start();
    inter_tx_SetVector(tx);

    speed = 0;
    currentMagnet = 0;
    preError = 0.0;
    integral = 0.0;
    /*MAX_OUT = Motor_PWM_ReadPeriod()+1;*/
    openloop = 50*SETPOINT;

    /* initialize the array */
    for (i = 0; i < 5; i++)
    {
        lastTimeSeen[i] = MAX_TIME;
    }

    // Calculate the speed pid output
    pidOutput = PIDCtrl(SETPOINT, 0);/*, .1);

    // Change the PWM to be the pid output
    Motor_PWM_WriteCompare1((uint16) pidOutput);
    Motor_PWM_WriteCompare2(FORWARD);
    dir_flag = 0;

    /* CyGlobalIntEnable; */ /* Uncomment this line to enable global interrupts. */
    for(;;)
    {
        /* Place your application code here. */
    }
}

/* [] END OF FILE */
```

Appendix B: mapping_script.m

```
s = serial('COM7');
set(s, 'InputBufferSize', 1024); %number of bytes in inout buffer
set(s, 'FlowControl', 'none');
set(s, 'BaudRate', 57600);
set(s, 'Parity', 'none');
set(s, 'DataBits', 8);
set(s, 'StopBit', 1);
set(s, 'Timeout',100);
fopen(s)

figure;

t = 0:.01:2*pi;
track = 3.5*ones(size(t));
%P = polar(t, track);
%set(P, 'Visible', 'off');

hold on;

current = zeros(19,1);
last1 = zeros(19,1);
last2 = zeros(19,1);
avgdist = zeros(19,1);

for n = 1:(18)
    bin(n) = n;
end

while 1
    clf;
    last2 = last1;
    last1 = current;
    line = fgetl(s);
    vals = strsplit(line,'=');
    vals = transpose(vals);
    bin  = zeros(size(vals));
    for n = 1:(size(vals)-1)
        split_vals = strsplit(vals{n},',');
        bin(n) = str2double(split_vals{1});
        current(n) = str2double(split_vals{2});
    end

    %for n = 1:(size(vals)-1)
    %    fprintf('bin: %d, dist: %1.3f\n',bin(n),dist(n));
    %end

    if (last2(7) ~= 0) && (last1(7) ~= 0)
        avgdist = 0.6*current + 0.3*last1 + 0.1*last2;
    elseif last1(7) ~= 0
        avgdist = 0.65*current + 0.35*last1;
    else
        avgdist = current;
    end

    angle = bin*-pi/9;
    angle = angle + pi/2;
    i=size(avgdist);
    avgdist(i(1)) = avgdist(1);
    %dist(i(1)) = dist(1);
```

```
    P = polar(t, track);
    set(P, 'Visible', 'off');
    hold on;
    polar(angle,avgdist);
    title('Car Surroundings (up is forwards, distance in meters)');
    drawnow;
end
```